Python

Zwięzłe kompendium dla programisty

Helion

David M. Beazley

Tytuł oryginału: Python Distilled

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-283-9020-1

Authorized translation from the English language edition, entitled Python Distilled, 1st Edition by David M Beazley, published by Pearson Education, Inc, publishing as Addison Wesley Professional, Copyright © 2022 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Polish language edition published by Helion S.A., Copyright © 2022.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiejkolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz wydawca dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz wydawca nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
https://helion.pl/user/opinie/pyzwko_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

- Poleć książkę na Facebook.com
- Kup w wersji papierowej
- Oceń książkę

- Księgarnia internetowa
- Lubię to! » Nasza społeczność

Spis treści

Przedmowa **11**

Rozdział 1. Podstawy Pythona **13**

1.1. Uruchamianie Pythona 13
1.2. Programy Pythona 14
1.3. Prymitywy, zmienne i wyrażenia 15
1.4. Operatory arytmetyczne 16
1.5. Warunki i sterowanie przepływem programu 19
1.6. Ciągi tekstowe 20
1.7. Operacje na plikach 23
1.8. Listy 24
1.9. Krotki 26
1.10. Zbiory 28
1.11. Słowniki 29
1.12. Iteracja i pętle 31
1.13. Funkcje 32
1.14. Wyjątki 34
1.15. Zakończenie programu 35
1.16. Obiekty i klasy 36
1.17. Moduły 39
1.18. Pisanie skryptów 41
1.19. Pakiety 42
1.20. Strukturyzacja aplikacji 43
1.21. Zarządzanie pakietami stron trzecich 44
1.22. Python pasuje do Twojego mózgu 45

Rozdział 2. Operatory, wyrażenia i manipulacja danymi 46
2.1. Literały 46
2.2. Wyrażenia i lokalizacje 47
2.3. Standardowe operatory 48
2.4. Modyfikacje w miejscu 49
2.5. Porównywanie obiektów 50
2.6. Operatory porównania porządkowego 51
2.7. Wyrażenia logiczne i wartości prawdziwe 52
2.8. Wyrażenia warunkowe 53
2.9. Operacje obejmujące elementy iterowalne 53
2.10. Operacje na sekwencjach 55
2.11. Operacje na mutowalnych obiektach sekwencyjnych 57
2.12. Operacje na zbiorach 58
2.13. Operacje na mapowaniach 59
2.14. Lista, zbiór i słownik 60
2.15. Wyrażenia generujące 62
2.16. Operator atrybutu (.) 63
2.17. Operator wywołania funkcji () 63
2.18. Kolejność liczenia 64
2.19. Podsumowanie: sekretne życie danych 65

Rozdział 3. Struktura i kontrola przepływu programu 66
3.1. Struktura i wykonanie programu 66
3.2. Wykonanie warunkowe 67
3.3. Pętle i iteracje 67
3.4. Wyjątki 70
3.4.1. Hierarchia wyjątków 73
3.4.2. Wyjątki i kontrola przepływu 75
3.4.3. Definiowanie nowych wyjątków 75
3.4.4. Powiązane wyjątki 76
3.4.5. Śledzenie wyjątków 78
3.4.6. Wskazówka dotycząca obsługi wyjątków 79
3.5. Menedżery kontekstu i instrukcja with 80
3.6. Asercje i __debug__ 82
3.7. Podsumowanie 83

Rozdział 4. Obiekty, typy i protokoły 85
4.1. Podstawowe pojęcia 85
4.2. Tożsamość i typ obiektu 86
4.3. Zliczanie referencji i odśmiecanie pamięci 87

4.4. Referencje i kopie 89
4.5. Reprezentacja obiektu i wyświetlanie 90
4.6. Obiekty pierwszoklasowe 91
4.7. Używanie wartości None dla opcjonalnych lub brakujących danych 92
4.8. Protokoły obiektu i abstrakcja danych 93
4.9. Protokół obiektu 94
4.10. Protokół liczbowy 95
4.11. Protokół porównania 98
4.12. Protokoły konwersji 99
4.13. Protokół kontenera 100
4.14. Protokół iteracji 102
4.15. Protokół atrybutów 103
4.16. Protokół funkcji 103
4.17. Protokół menedżera kontekstu 104
4.18. Podsumowanie: pythoniczność 104

Rozdział 5. Funkcje 106

5.1. Definicje funkcji 106
5.2. Argumenty domyślne 106
5.3. Argumenty wariadyczne (zmienna liczba argumentów) 107
5.4. Argumenty słów kluczowych 108
5.5. Wariadyczne argumenty słów kluczowych 109
5.6. Funkcje akceptujące wszystkie dane wejściowe 109
5.7. Argumenty tylko pozycyjne 110
5.8. Nazwy, wpisy dokumentacyjne i wskazówki dotyczące typów 111
5.9. Zastosowanie funkcji i przekazywanie parametrów 112
5.10. Zwracane wartości 113
5.11. Obsługa błędów 114
5.12. Zasady określania zakresu 115
5.13. Rekurencja 118
5.14. Wyrażenie lambda 118
5.15. Funkcje wyższego rzędu 119
5.16. Przekazywanie argumentów w funkcjach zwrotnych 121
5.17. Zwracanie wyników z wywołań zwrotnych 125
5.18. Dekoratory 127
5.19. Funkcje map, filter i reduce 130
5.20. Przegląd funkcji, atrybutów i sygnatur 131
5.21. Inspekcja środowiska 133
5.22. Dynamiczne wykonywanie i tworzenie kodu 135

5.23. Funkcje asynchroniczne i await ... 136
5.24. Podsumowanie: przemyślenia na temat funkcji i kompozycji ... 139

Rozdział 6. Generatory ... 140
6.1. Generatory i yield ... 140
6.2. Generatory z możliwością ponownego uruchomienia ... 143
6.3. Delegowanie generatora ... 143
6.4. Używanie generatorów w praktyce ... 144
6.5. Ulepszone generatory i wyrażenia yield ... 147
6.6. Zastosowania ulepszonych generatorów ... 148
6.7. Generatory i obsługa await ... 151
6.8. Podsumowanie: krótka historia generatorów i patrzenie w przyszłość ... 152

Rozdział 7. Klasy i programowanie obiektowe ... 153
7.1. Obiekty ... 153
7.2. Wyrażenie class ... 154
7.3. Instancje ... 155
7.4. Dostęp do atrybutów ... 156
7.5. Zasady ustalania zakresu ... 157
7.6. Przeciążanie operatora i protokoły ... 158
7.7. Dziedziczenie ... 159
7.8. Unikanie dziedziczenia poprzez kompozycję ... 162
7.9. Unikanie dziedziczenia poprzez funkcje ... 164
7.10. Wiązanie dynamiczne i technika kaczego typowania ... 165
7.11. Niebezpieczeństwo dziedziczenia po typach wbudowanych ... 166
7.12. Zmienne i metody klasy ... 167
7.13. Metody statyczne ... 170
7.14. Słowo na temat wzorców projektowych ... 173
7.15. Enkapsulacja danych i atrybuty prywatne ... 174
7.16. Wskazówka typu ... 176
7.17. Właściwości ... 177
7.18. Typy, interfejsy i klasy abstrakcyjne ... 180
7.19. Wielokrotne dziedziczenie, interfejsy i domieszki ... 183
7.20. Dyspozycja oparta na typie ... 188
7.21. Dekoratory klas ... 189
7.22. Nadzorowane dziedziczenie ... 192
7.23. Cykl życia obiektu i zarządzanie pamięcią ... 193
7.24. Słabe referencje ... 197
7.25. Wewnętrzna reprezentacja obiektów i wiązanie atrybutu ... 199
7.26. Proxy, wrapper i delegacje ... 201

7.27. Zmniejszenie wykorzystania pamięci za pomocą __slots__ 203
7.28. Deskryptory 204
7.29. Proces definicji klasy 207
7.30. Dynamiczne tworzenie klas 208
7.31. Metaklasy 209
7.32. Obiekty wbudowane dla instancji i klas 213
7.33. Podsumowanie: zachowaj prostotę 214

Rozdział 8. Moduły i pakiety 215

8.1. Moduły i wyrażenie import 215
8.2. Buforowanie modułów 217
8.3. Importowanie wybranych nazw z modułu 218
8.4. Importy cykliczne 220
8.5. Ponowne ładowanie i zwolnienie modułu 221
8.6. Kompilacja modułów 222
8.7. Ścieżka wyszukiwania modułów 223
8.8. Wykonanie jako program główny 224
8.9. Pakiety 225
8.10. Import wewnątrz pakietu 226
8.11. Uruchamianie podmodułu pakietu jako skryptu 227
8.12. Kontrolowanie przestrzeni nazw pakietu 228
8.13. Kontrolowanie eksportu pakietów 229
8.14. Dane pakietu 230
8.15. Obiekty modułu 231
8.16. Wdrażanie pakietów Pythona 232
8.17. Przedostatnie słowo: zacznij od pakietu 233
8.18. Podsumowanie: zachowaj prostotę 234

Rozdział 9. Obsługa operacji wejścia-wyjścia 235

9.1. Reprezentacja danych 235
9.2. Kodowanie i dekodowanie tekstu 236
9.3. Formatowanie tekstu i bajtów 238
9.4. Czytanie opcji wiersza poleceń 241
9.5. Zmienne środowiskowe 243
9.6. Pliki i obiekty plików 243
9.6.1. Nazwy plików 244
9.6.2. Tryby plików 245
9.6.3. Buforowanie operacji wejścia-wyjścia 245
9.6.4. Kodowanie w trybie tekstowym 246
9.6.5. Obsługa wiersza w trybie tekstowym 247

9.7. Warstwy abstrakcyjne wejścia-wyjścia ... 247
9.7.1. Metody plików ... 248
9.8. Standardowe wejście, wyjście i błąd ... 250
9.9. Katalogi ... 251
9.10. Funkcja print() ... 252
9.11. Generowanie wyjścia ... 252
9.12. Pobieranie danych wejściowych ... 253
9.13. Serializacja obiektów ... 254
9.14. Operacje blokujące i współbieżność ... 255
9.14.1. Nieblokujące operacje wejścia-wyjścia ... 256
9.14.2. Odpytywanie operacji wejścia-wyjścia ... 257
9.14.3. Wątki ... 257
9.14.4. Równoczesne wykonywanie z asyncio ... 258
9.15. Standardowe moduły biblioteczne ... 259
9.15.1. Moduł asyncio ... 259
9.15.2. Moduł binascii ... 260
9.15.3. Moduł cgi ... 261
9.15.4. Moduł configparser ... 261
9.15.5. Moduł csv ... 262
9.15.6. Moduł errno ... 263
9.15.7. Moduł fcntl ... 263
9.15.8. Moduł hashlib ... 264
9.15.9. Pakiet http ... 264
9.15.10. Moduł io ... 265
9.15.11. Moduł json ... 265
9.15.12. Moduł logging ... 266
9.15.13. Moduł os ... 266
9.15.14. Moduł os.path ... 267
9.15.15. Moduł pathlib ... 267
9.15.16. Moduł re ... 268
9.15.17. Moduł shutil ... 269
9.15.18. Moduł select ... 269
9.15.19. Moduł smtplib ... 270
9.15.20. Moduł socket ... 270
9.15.21. Moduł struct ... 272
9.15.22. Moduł subprocess ... 273
9.15.23. Moduł tempfile ... 273
9.15.24. Moduł textwrap ... 274
9.15.25. Moduł threading ... 275
9.15.26. Moduł time ... 276

9.15.27. Pakiet urllib 277
9.15.28. Moduł unicodedata 278
9.15.29. Pakiet xml 279
9.16. Podsumowanie 279

Rozdział 10. Funkcje wbudowane i biblioteka standardowa 281

10.1. Funkcje wbudowane 281
10.2. Wyjątki wbudowane 297
10.2.1. Klasy bazowe wyjątków 297
10.2.2. Atrybuty wyjątków 298
10.2.3. Predefiniowane klasy wyjątków 299
10.3. Biblioteka standardowa 301
10.3.1. Moduł collections 302
10.3.2. Moduł datetime 302
10.3.3. Moduł itertools 302
10.3.4. Moduł inspect 302
10.3.5. Moduł math 302
10.3.6. Moduł os 302
10.3.7. Moduł random 302
10.3.8. Moduł re 302
10.3.9. Moduł shutil 303
10.3.10. Moduł statistics 303
10.3.11. Moduł sys 303
10.3.12. Moduł time 303
10.3.13. Moduł turtle 303
10.3.14. Moduł unittest 303
10.4. Podsumowanie: korzystaj z wbudowanych elementów 303

Przedmowa

Minęło ponad 20 lat, odkąd napisałem książkę *Python Essential Reference*. W tamtym czasie Python był znacznie mniejszym językiem i zawierał w swojej standardowej bibliotece zestaw przydatnych elementów. To było coś, co w większości mogło zmieścić się w Twojej głowie. *Essential Reference* odzwierciedlało tę epokę — była to mała książka, którą można było zabrać ze sobą, aby napisać kod Pythona na bezludnej wyspie lub w tajnym skarbcu. Poprzez trzy kolejne wydania, *Essential Reference* trzymał się wizji zwartego, ale kompletnego opisu języka —jeśli zamierzałeś programować w Pythonie na wakacjach, dlaczego miałeś nie wykorzystać go w całości?

Dziś, ponad dekadę od publikacji ostatniego wydania *Python Essential Reference*, świat Pythona jest zupełnie inny. Nie jest już językiem niszowym — Python stał się jednym z najpopularniejszych języków programowania na świecie. Programiści Pythona mają również pod ręką bogactwo informacji w postaci zaawansowanych edytorów, środowisk IDE, notatników, stron internetowych i nie tylko. W rzeczywistości prawdopodobnie nie ma potrzeby zaglądać do opisu języka, ponieważ prawie każdą poszukiwaną informację możesz wyczarować, aby pojawiła się przed Twoimi oczami, za dotknięciem kilku klawiszy.

Łatwość wyszukiwania informacji i skala wszechświata Pythona stanowią za to inne wyzwanie. Jeśli dopiero zaczynasz się uczyć lub musisz rozwiązać nowy problem, kwestia, od czego zacząć, może być przytłaczająca. Może być również trudno oddzielić funkcje różnych narzędzi od samego rdzenia języka. Tego rodzaju problemy są powodem powstania tej książki.

Python. Zwięzłe kompendium programisty to książka o programowaniu w Pythonie. Nie próbuje udokumentować wszystkiego, co można zrobić lub co zostało zrobione w Pythonie. Koncentruje się na przedstawieniu nowoczesnego, ale wyselekcjonowanego rdzenia języka. Zostały w niej zebrane całe lata uczenia Pythona naukowców, inżynierów i specjalistów od oprogramowania, a także tworzenia bibliotek oprogramowania, przesuwania granic tego, co sprawia, że Python działa i odkrywania, co jest najbardziej przydatne.

W większości, książka skupia się na samym programowaniu w Pythonie. Obejmuje to techniki abstrakcji, strukturę programu, dane, funkcje, obiekty, moduły i tak dalej — tematy, które będą przydatne dla programistów pracujących nad projektami Pythona dowolnej wielkości.

Czysty opis języka, który można łatwo uzyskać za pomocą IDE (taki jak listy funkcji, nazwy poleceń, argumenty itp.) jest generalnie pomijany. Podjąłem również świadomy wybór, aby nie opisywać szybko zmieniającego się świata narzędzi Pythona — edytorów, środowisk IDE, wdrażania i spraw pokrewnych.

Być może kontrowersyjnie, generalnie nie skupiam się na funkcjach językowych związanych z zarządzaniem dużymi projektami programistycznymi. Python jest czasem używany do wielkich i poważnych rzeczy — składających się z milionów wierszy kodu. Takie aplikacje wymagają specjalistycznych narzędzi, projektu i funkcji, a także komisji, spotkań i decyzji, które należy podjąć w bardzo ważnych sprawach. To za dużo jak na tę małą książeczkę. Ale może szczera odpowiedź jest taka, że nie używam Pythona do pisania takich aplikacji — i ty też nie powinieneś. Przynajmniej nie jako hobby.

W trakcie pisania książki, funkcje języka stale ewoluują. Ta książka została napisana w erze Pythona 3.9. W związku z tym nie obejmuje niektórych głównych dodatków planowanych w późniejszych wydaniach — na przykład dopasowania wzorców strukturalnych.

To temat na inny czas i miejsce.

Na koniec uważam, że ważne jest, aby programowanie sprawiało przyjemność. Mam nadzieję, że moja książka nie tylko pomoże Ci stać się produktywnym programistą Pythona, ale także uchwyci trochę magii, która zainspirowała ludzi do używania Pythona do badania gwiazd, latania helikopterami na Marsie i spryskiwania wiewiórek armatkami wodnymi na podwórku.

Podziękowanie

Chciałbym podziękować recenzentom merytorycznym, Shawnowi Brownowi, Sophie Tabac i Pete Feinowi za ich pomocne komentarze. Chciałbym również podziękować mojej wieloletniej redaktorce Debrze Williams Cauley za jej pracę nad tym i poprzednimi projektami. Wielu studentów, którzy wzięli udział w moich zajęciach, miało znaczący, choć pośredni, wpływ na tematy poruszane w tej książce. Na koniec chciałbym podziękować Pauli, Thomasowi i Lewisowi za ich wsparcie i miłość.

O autorze

David Beazley jest autorem podręcznika *Python Essential Reference*, wydanie czwarte (Addison-Wesley, 2010) i książki *Python Cookbook*, wydanie trzecie (polskie wydanie: *Python. Receptury. Wydanie III*, Helion, 2013). Obecnie prowadzi zaawansowane kursy informatyki za pośrednictwem swojej firmy Dabeaz LLC (*www.dabeaz.com*). Używa, pisze, mówi i naucza o Pythonie od 1996 roku.

1

Podstawy Pythona

Ten rozdział omawia podstawy języka Python. Opisuje zmienne, typy danych, wyrażenia, sterowanie przepływem programu, funkcje, klasy i operacje wejścia-wyjścia. Rozdział kończy się omówieniem modułów, skryptów, pakietów i kilkoma wskazówkami dotyczącymi tworzenia większych programów. Ten rozdział nie jest próbą dostarczenia wyczerpujących informacji dla każdej funkcji ani nie zajmuje się wszystkimi narzędziami, które mogą współtworzyć większy projekt w Pythonie. Doświadczeni programiści powinni być jednak w stanie wydobyć z tego materiału to, co potrzebne, aby pisać bardziej zaawansowane programy. Nowicjuszy zachęca się do wypróbowania przykładów w prostym środowisku, takim jak okno terminala i podstawowy edytor tekstu.

1.1. Uruchamianie Pythona

Programy Pythona są wykonywane przez interpreter. Istnieje wiele różnych środowisk, w których może działać interpreter Pythona: IDE, przeglądarka lub okno terminala. Rdzeń interpretera stanowi jednak aplikacja tekstowa, którą można uruchomić, wpisując polecenie `python` w powłoce systemowej, takiej jak bash. Ponieważ Python w wersjach 2 i 3 może być zainstalowany na tej samej maszynie, może być konieczne wpisanie komend `python2` lub `python3` (dla odpowiedniej wersji). Informacje zawarte w książce opierają się na wersji Pythona 3.8 lub nowszej.

Po uruchomieniu interpretera pojawia się znak zachęty, po którym można tworzyć programy w środowisku REPL (ang. *read-evaluation-print-loop*), czyli użytkownik może wprowadzać polecenia, które zostaną wykonane, a ich wynik wypisany na ekran. Na przykład w poniższym przykładzie interpreter wyświetla komunikat o prawach autorskich i przedstawia użytkownikowi znak zachęty >>>, po którym użytkownik wpisuje popularny program „Hello World”:

```
Python 3.8.0 (default, Feb 3 2019, 05:53:21)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Witaj, świecie')
Witaj, świecie
>>>
```

Niektóre środowiska mogą wyświetlać inny znak zachęty. Poniższe dane wyjściowe pochodzą z *ipython* (alternatywna powłoka dla Pythona):

```
Python 3.8.0 (default, Feb 4, 2019, 07:39:16)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.5.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print('Witaj, świecie')
Witaj, świecie

In [2]:
```

Bez względu na uzyskany wynik wyjściowy podstawowa zasada jest taka sama. Wpisujesz polecenie, które od razu się uruchamia, i natychmiast widzisz dane wyjściowe.

Tryb interaktywny Pythona jest jedną z jego najbardziej użytecznych funkcji, ponieważ możesz wpisać dowolną poprawną instrukcję i natychmiast zobaczyć wynik. Jest to przydatne do debugowania i eksperymentowania. Wiele osób, w tym autor tej książki, używa interaktywnego Pythona jako kalkulatora. Na przykład:

```
>>> 6000 + 4523.50 + 134.25
10657.75
>>> _ + 8192.75
18850.5
>>>
```

Kiedy korzystasz z Pythona w sposób interaktywny, zmienna _ przechowuje wynik ostatniej operacji. Jest to przydatne, jeśli chcesz użyć tego wyniku w kolejnych instrukcjach. Ta zmienna jest definiowana tylko podczas pracy interaktywnej, więc nie używaj jej w programach zapisanych w pliku.

Możesz wyjść z interaktywnego interpretera, wpisując polecenie `quit()` lub znak `EOF` (koniec pliku). W systemie UNIX EOF uzyskujemy za pomocą skrótu *Ctrl+D*; w systemie Windows — poprzez skrót *Ctrl+Z*.

1.2. Programy Pythona

Jeśli chcesz stworzyć program, który można uruchamiać wielokrotnie, umieść instrukcje w pliku tekstowym.

Na przykład:

```
# hello.py
print('Witaj, świecie')
```

Pliki źródłowe Pythona to zakodowane w UTF-8 pliki tekstowe, które zwykle mają rozszerzenie *.py*. Znak # oznacza komentarz, który obejmuje cały wiersz. Znaki międzynarodowe (Unicode) mogą być swobodnie używane w kodzie źródłowym, o ile korzystasz z kodowania UTF-8 (jest to domyślne ustawienie w większości edytorów, ale nigdy nie zaszkodzi sprawdzić, jakie są ustawienia edytora, jeśli nie masz pewności).

Aby uruchomić plik *hello.py*, podaj nazwę pliku w interpreterze w następujący sposób:

```
shell % python3 hello.py
Witaj, Świecie
shell %
```

W pierwszym wierszu programu powszechnie używa się znaków `#!` — aby określić rodzaj interpretera:

```
#!/usr/bin/env python3
print('Witaj, Świecie')
```

W systemie UNIX, jeśli nadasz temu plikowi uprawnienia do wykonywania (na przykład poprzez polecenie `chmod +x hello.py`), możesz uruchomić program, wpisując `hello.py` w swojej powłoce.

W systemie Windows, aby uruchomić plik Pythona, możesz dwukrotnie kliknąć plik *.py* lub wpisać nazwę programu w wierszu poleceń w menu *Start* systemu Windows. Znaki `#!`, jeśli zostały podane, służą do wybrania wersji interpretera (Python 2 lub 3). Wykonywanie programu może mieć miejsce w oknie konsoli, które znika natychmiast po zakończeniu programu — często zanim będzie można odczytać jego dane wyjściowe. Do debugowania lepiej uruchomić program w środowisku programistycznym Pythona.

Interpreter uruchamia instrukcje w kolejności, aż dotrze do końca pliku wejściowego. W tym momencie program oraz Python kończą działanie.

1.3. Prymitywy, zmienne i wyrażenia

Python dostarcza zbiór typów prymitywnych, takich jak liczby całkowite, zmiennoprzecinkowe i łańcuchy:

```
42            # int
4.2           # float
'forty-two'   # str
True          # bool
```

Zmienna to nazwa, która odwołuje się do wartości. Wartość reprezentuje obiekt pewnego typu:

```
x = 42
```

Czasami możesz zobaczyć typ wyraźnie dołączony do nazwy. Na przykład:

```
x: int = 42
```

Typ jest jedynie wskazówką, aby poprawić czytelność kodu. Może być używany przez narzędzia do sprawdzania kodu innych firm. W przeciwnym razie jest całkowicie ignorowany. Nie uniemożliwia to późniejszego przypisania innego rodzaju wartości.

Poniższe wyrażenie jest kombinacją prymitywów, nazw oraz operatorów i zwraca wartość:

```
2 + 3 * 4 # -> 14
```

Poniższy program używa zmiennych i wyrażeń do obliczania odsetek złożonych:

```
# interest.py

principal = 1000 # Kwota początkowa
rate = 0.05      # Stopa procentowa
```

```
numyears = 5      # Liczba lat
year = 1
while year <= numyears:
    principal = principal * (1 + rate)
    print(year, principal)
    year += 1
```

Po wykonaniu program generuje następujące dane wyjściowe:

```
1 1050.0
2 1102.5
3 1157.625
4 1215.5062500000001
5 1276.2815625000003
```

Instrukcja `while` testuje wyrażenie warunkowe. Jeśli testowany warunek jest prawdziwy, wykonywana jest zawartość instrukcji `while`. Warunek jest następnie ponownie testowany, a zawartość jest wykonywana do czasu, aż warunek stanie się fałszywy. Treść pętli jest oznaczona wcięciem. W ten sposób trzy instrukcje zawarte w pętli `while` w pliku *interest.py* są wykonywane w każdej iteracji. Python nie określa liczby znaków dla wcięcia, o ile są one spójne w obrębie bloku. Najczęściej używa się czterech spacji na każdy poziom wcięcia.

Jednym z problemów z programem *interest.py* jest to, że dane wyjściowe nie są zbyt ładne. Aby było lepiej, możesz wyrównać kolumny do prawej i ograniczyć dokładność kwoty do dwóch cyfr po przecinku. Zmień funkcję `print()` tak, aby używała tak zwanego *f-stringu* w następujący sposób:

```
print(f'{year:>3d} {principal:0.2f}')
```

W f-stringu nazwy zmiennych i wyrażenia mogą być obliczone poprzez umieszczenie ich w nawiasach klamrowych. Opcjonalnie każde podstawienie może mieć dołączony specyfikator formatowania. `'>3d'` oznacza trzycyfrową liczbę dziesiętną wyrównaną do prawej. `'0.2f'` oznacza liczbę zmiennoprzecinkową z dokładnością do dwóch miejsc po przecinku. Więcej informacji na temat kodów formatowania można znaleźć w rozdziale 9.

Teraz wynik wyjściowy programu wygląda tak:

```
1 1050.00
2 1102.50
3 1157.62
4 1215.51
5 1276.28
```

1.4. Operatory arytmetyczne

Python ma standardowy zestaw operatorów matematycznych, przedstawiony w tabeli 1.1. Operatory te mają to samo znaczenie co w większości innych języków programowania.

Tabela 1.1. *Operatory arytmetyczne*

Operacja	Opis
`x + y`	Dodawanie
`x – y`	Odejmowanie
`x * y`	Mnożenie
`x / y`	Dzielenie
`x // y`	Dzielenie całkowite
`x ** y`	Potęgowanie (`x` do potęgi `y`)
`x % y`	Modulo (`x` modulo `y`). Reszta
`–x`	Jednoargumentowe odejmowanie
`+x`	Jednoargumentowe dodawanie

Operator dzielenia (/) tworzy liczbę zmiennoprzecinkową, gdy argumentami są liczby całkowite. Dlatego `7/4` równa się `1.75`. Operator // obcina wynik do liczby całkowitej i działa zarówno z liczbami całkowitymi, jak i zmiennoprzecinkowymi. Operator modulo zwraca resztę z dzielenia `x // y`. Na przykład `7 % 4` to 3. W przypadku liczb zmiennoprzecinkowych operator modulo zwraca zmiennoprzecinkową resztę z działania `x // y`, czyli `x – (x // y) * y`.

Ponadto wbudowane funkcje z tabeli 1.2 udostępniają kilka częściej używanych operacji numerycznych.

Tabela 1.2. *Popularne funkcje matematyczne*

Funkcja	Opis
`abs(x)`	Wartość bezwzględna
`divmod(x, y)`	Zwraca `(x // y, x % y)`
`pow(x, y [,modulo])`	Zwraca `(x ** y) % modulo`
`round(x [,n])`	Zaokrągla `x` z precyzją `n`

Funkcja `round()` implementuje „zaokrąglanie bankiera”, czyli zaokrągla do najbliższej parzystej wartości (na przykład `0.5` jest zaokrąglane do `0.0`, a `1.5` do `2.0`).

Liczby całkowite mają kilka dodatkowych operatorów do manipulacji bitami, jak pokazano w tabeli 1.3.

Tabela 1.3. *Operatory manipulacji bitami*

Operacja	Opis
`x << y`	Przesunięcie w lewo
`x >> y`	Przesunięcie w prawo
`x & y`	Iloczyn bitowy
`x | y`	Suma bitowa
`x ^ y`	Bitowa różnica symetryczna (xor)
`~x`	Negacja bitowa

Zwykle używa się ich razem z binarnymi liczbami całkowitymi. Na przykład:

```
a = 0b11001001
mask = 0b11110000
x = (a & mask) >> 4 # x = 0b1100 (12)
```

W tym przykładzie `0b11001001` to sposób zapisu liczby całkowitej w formacie binarnym. Mogłeś zapisać ją w postaci dziesiętnej `201` lub szesnastkowej `0xc9`, ale jeśli bawisz się bitami, to reprezentacja binarna ułatwia wizualizację tego, co robisz.

Semantyka operatorów bitowych zakłada, że liczby całkowite wykorzystują kod uzupełnień do dwóch i że znak określony jest przez pierwszy bit z lewej. Jeśli pracujesz z wzorcami bitów, które są przeznaczone do mapowania na natywne liczby całkowite na sprzęcie, wymagana jest pewna ostrożność. Dzieje się tak, ponieważ Python nie obcina bitów ani nie pozwala na przepełnienie wartości — zamiast tego wynik będzie dowolnie duży. Od Ciebie zależy, czy wynik jest odpowiednio zwymiarowany lub w razie potrzeby skrócony.

Aby porównać liczby, użyj operatorów porównania z tabeli 1.4.

Tabela 1.4. *Operatory porównania*

Operacja	Opis
`x == y`	Równe
`x != y`	Nie jest równe
`x < y`	Mniejsze niż
`x > y`	Większe niż
`x >= y`	Większe lub równe
`x <= y`	Mniejsze lub równe

Wynikiem porównania jest wartość logiczna `True` (prawda) lub `False` (fałsz).

Operatory `and`, `or` i `not` (nie mylić z powyższymi operatorami manipulacji bitami) mogą tworzyć bardziej złożone wyrażenia logiczne. Zachowanie tych operatorów pokazano w tabeli 1.5.

Wartość jest uważana za fałszywą, jeśli ma wartość `False`, `None`, zero lub jest pusta. W przeciwnym razie ma wartość `True`.

Tabela 1.5. *Operatory logiczne*

Operator	Opis
`x or y`	Jeśli `x` ma wartość `False`, zwróć `y`; w przeciwnym razie zwróć `x`.
`x and y`	Jeśli `x` ma wartość `False`, zwróć `x`; w przeciwnym razie zwróć `y`.
`not x`	Jeśli `x` ma wartość `False`, zwróć `True`; w przeciwnym razie zwróć `False`.

Często pisze się wyrażenie, które aktualizuje wartość. Na przykład:

```
x = x + 1
y = y * n
```

Zamiast tego możesz napisać następującą skróconą wersję:

```
x += 1
y *= n
```

Ta skrócona forma może być używana z dowolnymi operatorami: +, –, *, **, /, //, %, &, |, ^, <<, >>. Python nie ma operatorów inkrementacji (++) ani dekrementacji (—) występujących w niektórych innych językach.

1.5. Warunki i sterowanie przepływem programu

Instrukcje `while`, `if` i `else` służą do wykonywania pętli oraz warunkowego wykonywania kodu.

Oto przykład:

```
if a < b:
    print('Komputer mówi Tak')
else:
    print('Komputer mówi Nie')
```

Treść klauzul `if` i `else` oznaczono przez wcięcie. Klauzula `else` jest opcjonalna. Aby utworzyć pustą klauzulę, użyj instrukcji `pass` w następujący sposób:

```
if a < b:
    pass # Nic nie robi
else:
    print('Komputer mówi Nie')
```

Aby obsłużyć przypadki z wieloma testami, użyj instrukcji `elif`:

```
if suffix == '.htm':
    content = 'text/html'
elif suffix == '.jpg':
    content = 'image/jpeg'
elif suffix == '.png':
    content = 'image/png'
else:
    raise RuntimeError(f'Nieznany rodzaj treści {suffix!r}')
```

Jeśli przypisujesz wartość w połączeniu z testem, użyj wyrażenia warunkowego:

```
maxval = a if a > b else b
```

Dłuższy zapis tego samego:

```
if a > b:
    maxval = a
else:
    maxval = b
```

Czasami możesz zobaczyć przypisanie zmiennej i warunku połączone za pomocą operatora `:=`. Jest to wyrażenie przypisania (lub, bardziej potocznie, „operator morsa”, ponieważ `:=` wygląda jak przewrócony na bok mors). Na przykład:

```
x = 0
while (x := x + 1) < 10: # Wyświetla 1, 2, 3, ..., 9
    print(x)
```

Nawiasy, używane do otaczania wyrażenia przypisania, są zawsze wymagane.

Instrukcja `break` może służyć do wcześniejszego przerwania pętli. Dotyczy to tylko najbardziej wewnętrznej pętli. Na przykład:

```
x = 0
while x < 10:
    if x == 5:
        break # Zatrzymuje pętlę. Przechodzi do instrukcji Done
    print(x)
    x += 1

print('Zrobione')
```

Instrukcja `continue` pomija resztę treści pętli i wraca na jej początek. Na przykład:

```
x = 0
while x < 10:
    x += 1
    if x == 5:
       continue # Pomija print(x). Wraca na początek pętli
    print(x)

print('Zrobione')
```

1.6. Ciągi tekstowe

Aby zdefiniować literał ciągu, ujmij go w pojedynczy, podwójny lub potrójny cudzysłów w następujący sposób:

```
a = 'Witaj, świecie'
b = "Python jest fajny"
c = '''Komputer mówi Nie.'''
d = """Komputer nadal mówi Nie."""
```

Ten sam typ cudzysłowu użyty do rozpoczęcia łańcucha musi być użyty do jego zakończenia. Ciągi w potrójnym cudzysłowie przechwytują cały tekst aż do końcowego potrójnego cudzysłowu — w przeciwieństwie do ciągów w pojedynczym i podwójnym cudzysłowie, które muszą być określone w jednym wierszu logicznym. Ciągi w potrójnym cudzysłowie są przydatne, gdy zawartość literału ciągu obejmuje wiele wierszy tekstu:

```
print('''Content-type: text/html

<h1> Witaj, świecie </h1>
Kliknij <a href="http://www.python.org">tutaj</a>.
''')
```

Bezpośrednio sąsiadujące literały ciągów są łączone w jeden ciąg. Powyższy przykład można zatem zapisać również jako:

```
print(
'Content-type: text/html\n'
'\n'
'<h1> Witaj, świecie </h1>\n'
'Kliknij <a href="http://www.python.org">tutaj</a>\n'
)
```

Jeśli znak cudzysłowu otwierającego ciągu jest poprzedzony literą `f`, to znaczy, że znaki ucieczki są interpretowane dosłownie. Na przykład we wcześniejszych przykładach do wyświetlenia wartości wyrażenia użyto następującej instrukcji:

```
print(f'{year:>3d} {principal:0.2f}')
```

Chociaż używa się tylko prostych nazw zmiennych, może się pojawić dowolne prawidłowe wyrażenie. Na przykład:

```
base_year = 2020
...
print(f'{base_year + year:>4d} {principal:0.2f}')
```

Jako alternatywę dla ciągów z `f` można zastosować metodę `format()` i operator `%`. Na przykład:

```
print('{0:>3d} {1:0.2f}'.format(year, principal))
print('%3d %0.2f' % (year, principal))
```

Więcej informacji na temat formatowania ciągów można znaleźć w rozdziale 9.

Łańcuchy są przechowywane jako sekwencje znaków Unicode indeksowane liczbami całkowitymi, zaczynając od zera. Ujemne indeksy liczone są od końca ciągu. Długość ciągu `s` jest obliczana za pomocą `len(s)`. Aby wyodrębnić pojedynczy znak, użyj operatora indeksowania `s[i]`, gdzie `i` jest indeksem.

```
a = 'Hello World'
print(len(a))       # 11
b = a[4]            # b = 'o'
c = a[-1]           # c = 'd'
```

Aby wyodrębnić podciąg, użyj operatora wycinania `s[i:j]`. Wycina wszystkie znaki z `s`, których indeks `k` należy do zakresu `i <= k < j`. Jeśli któryś z indeksów zostanie pominięty, zakłada się odpowiednio początek lub koniec ciągu:

```
c = a[:5]           # c = 'Hello'
d = a[6:]           # d = 'World'
e = a[3:8]          # e = 'lo Wo'
f = a[-5:]          # f = 'World'
```

Łańcuchy mają różne metody manipulowania ich zawartością. Na przykład metoda `replace()` wykonuje prostą zamianę tekstu:

```
g = a.replace('Hello', 'Hello Cruel') # f = 'Hello Cruel World'
```

Tabela 1.6 przedstawia kilka popularnych metod operacji na łańcuchach. Tu i wszędzie indziej argumenty ujęte w nawiasy kwadratowe są opcjonalne.

Łańcuchy są łączone za pomocą operatora plus (+):

```
g = a + 'ly' # g = 'Hello Worldly'
```

Python nigdy nie interpretuje zawartości ciągu jako danych liczbowych, więc znak + zawsze łączy ciągi:

```
x = '37'
y = '42'
z = x + y       # z = '3742' (konkatenacja ciągów)
```

Tabela 1.6. *Popularne metody ciągów*

Metoda	Opis
`s.endswith(prefix [,start [,end]])`	Sprawdza, czy łańcuch kończy się znakiem `prefix`.
`s.find(sub [,start [,end]])`	Znajduje pierwsze wystąpienie określonego podłańcucha lub –1, jeśli nie znaleziono.
`s.lower()`	Konwertuje na małe litery.
`s.replace(old, new [,maxreplace])`	Zastępuje podciąg.
`s.split([sep [,maxsplit]])`	Dzieli łańcuch, używając `sep` jako separatora. `maxsplit` to maksymalna liczba podziałów do wykonania.
`s.startswith(prefix [,start [,end]])`	Sprawdza, czy łańcuch zaczyna się od `prefix`.
`s.strip([chrs])`	Usuwa początkowe i końcowe białe znaki lub znaki podane w `chrs`.
`s.upper()`	Konwertuje ciąg na wielkie litery.

Aby wykonać obliczenia matematyczne, łańcuch znaków musi najpierw zostać przekonwertowany na wartość liczbową za pomocą funkcji takiej jak `int()` lub `float()`. Na przykład:

```
z = int(x) + int(y)       # z = 79 (dodawanie liczb całkowitych)
```

Wartości niebędące łańcuchami można przekonwertować na reprezentację łańcuchową za pomocą funkcji `str()`, `repr()` lub `format()`. Oto przykład:

```
s = 'Wartość x to ' + str(x)
s = 'Wartość x to ' + repr(x)
s = 'Wartość x to ' + format(x, '4d')
```

Chociaż zarówno `str()`, jak i `repr()` tworzą łańcuchy, ich dane wyjściowe są często różne. `str()` tworzy dane wyjściowe, które otrzymujesz, gdy używasz funkcji `print()`, podczas gdy `repr()` tworzy ciąg, który wpisujesz do programu, aby dokładnie reprezentować wartość obiektu.

Na przykład:

```
>>> s = 'hello\nworld'
>>> print(str(s))
hello
world
>>> print(repr(s))
'hello\nworld'
>>>
```

Podczas debugowania skorzystaj z funkcji `repr(s)` do wygenerowania danych wyjściowych, ponieważ pokazuje więcej informacji o wartości i jej typie.

Funkcja `format()` służy do konwersji pojedynczej wartości na ciąg z określonym formatowaniem. Na przykład:

```
>>> x = 12.34567
>>> format(x, '0.2f')
'12.35'
>>>
```

Kod formatu podany funkcji `format()` jest tym samym kodem, którego użyłbyś z f-stringiem podczas tworzenia sformatowanego wyjścia. Na przykład powyższy kod można zastąpić następującym:

```
>>> f'{x:0.2f}'
'12.35'
>>>
```

1.7. Operacje na plikach

Poniższy program otwiera plik i odczytuje jego zawartość, wiersz po wierszu, jako ciągi tekstowe:

```
with open('data.txt') as file:
    for line in file:
        print(line, end='') # end='' pomija dodatkowy znak nowej linii
```

Funkcja `open()` zwraca nowy obiekt pliku. Poprzedzająca ją instrukcja `with` deklaruje blok instrukcji (lub kontekstu), w którym plik (`file`) będzie używany. Gdy program opuści ten blok, plik jest automatycznie zamykany. Jeśli nie użyjesz instrukcji `with`, kod będzie wyglądał tak:

```
file = open('data.txt')
for line in file:
    print(line, end='') # end='' pomija dodatkowy znak nowej linii
file.close()
```

Łatwo zapomnieć o funkcji `close()`, więc lepiej skorzystać z instrukcji `with`, gdzie plik jest zamykany za Ciebie. Pętla `for` iteruje po kolejnych wierszach w pliku, dopóki nie będzie dostępnych więcej danych.

Jeśli chcesz przeczytać plik w całości jako ciąg, użyj metody `read()` w następujący sposób:

```
with open('data.txt') as file:
    data = file.read()
```

Jeśli chcesz czytać duży plik porcjami, podaj określony rozmiar do metody `read()` w następujący sposób:

```
with open('data.txt') as file:
    while (chunk := file.read(10000)):
        print(chunk, end='')
```

Operator `:=` użyty w tym przykładzie, przypisuje wartość do zmiennej i zwraca jej wartość, aby mogła zostać przetestowana przez pętlę `while`. Po osiągnięciu końca pliku `read()` zwraca pusty ciąg. Alternatywnym sposobem napisania powyższej funkcji jest użycie `break`:

```
with open('data.txt') as file:
    while True:
        chunk = file.read(10000)
        if not chunk:
            break
        print(chunk, end='')
```

Aby dane wyjściowe programu trafiły do pliku, podaj argument do funkcji `print()`:

```
with open('out.txt', 'wt') as out:
    while year <= numyears:
        principal = principal * (1 + rate)
        print(f'{year:>3d} {principal:0.2f}', file=out)
        year += 1
```

Ponadto obiekty plikowe obsługują metodę `write()`, której można używać do zapisywania danych łańcuchowych. Na przykład funkcja `print()` w poprzednim przykładzie mogła zostać napisana w ten sposób:

```
out.write(f'{year:3d} {principal:0.2f}\n')
```

Domyślnie pliki zawierają tekst zakodowany jako UTF-8. Jeśli pracujesz z innym kodowaniem tekstu, użyj dodatkowego argumentu kodowania podczas otwierania pliku. Na przykład:

```
with open('data.txt', encoding='latin-1') as file:
    data = file.read()
```

Czasami możesz chcieć odczytać dane wpisane interaktywnie w konsoli. Aby to zrobić, użyj funkcji `input()`. Na przykład:

```
name = input('Podaj swoje imię : ')
print('Witaj, ', name)
```

Funkcja `input()` zwraca cały wpisany tekst aż do znaku nowej linii, który nie jest uwzględniony.

1.8. Listy

Listy to uporządkowana kolekcja dowolnych obiektów. Utwórz listę, umieszczając wartości w nawiasach kwadratowych:

```
names = ['Dave', 'Paula', 'Thomas', 'Lewis']
```

Listy są indeksowane liczbami całkowitymi, zaczynając od zera. Użyj operatora indeksowania, aby uzyskać dostęp do poszczególnych elementów listy i je modyfikować:

```
a = names[2]        # Zwraca trzeci element listy, 'Thomas'
names[2] = 'Tom'    # Zmienia trzeci element na 'Tom'
print(names[-1])    # Wyświetla ostatni element ('Lewis')
```

Aby dodać nowe pozycje na końcu listy, użyj metody `append()`:

```
name.append('Alex')
```

Aby wstawić element w liście w określonej pozycji, użyj metody `insert()`:

```
name.insert(2, 'Aya')
```

Aby wykonać iterację po elementach listy, użyj pętli `for`:

```
for name in names:
    print(name)
```

Możesz wyodrębnić lub zmienić przypisanie części listy, używając operatora wycinania:

```
b = names[0:2]      # b -> ['Dave', 'Paula']
c = names[2:]       # c -> ['Aya', 'Tom', 'Lewis', 'Alex']
names[1] = 'Becky'  # Zastępuje ciąg 'Paula' ciągiem 'Becky'
names[0:2] = ['Dave', 'Mark', 'Jeff'] # Zastąp pierwsze dwa elementy
                                      # na ['Dave','Mark','Jeff']
```

Użyj operatora plus (+), aby połączyć listy:

```
a = ['x','y'] + ['z','z','y'] # Wynik to ['x','y','z','z','y']
```

Pustą listę tworzy się na dwa sposoby:

```
names = []          # Pusta lista
names = list()      # Pusta lista
```

Określenie [] dla pustej listy jest bardziej idiomatyczne. `list` to nazwa klasy skojarzonej z typem listy. Częściej używa się jej podczas konwersji danych na listę. Na przykład:

```
letters = list('Dave')    # letters = ['D', 'a', 'v', 'e']
```

W większości przypadków wszystkie pozycje na liście są tego samego typu (na przykład lista liczb lub lista ciągów). Jednak listy mogą zawierać dowolną mieszankę obiektów Pythona, w tym inne listy, jak w poniższym przykładzie:

```
a = [1, 'Dave', 3.14, ['Mark', 7, 9, [100, 101]], 10]
```

Dostęp do elementów zawartych na listach zagnieżdżonych można uzyskać, stosując więcej niż jedną operację indeksowania:

```
a[1]          # Zwraca 'Dave'
a[3][2]       # Zwraca 9
a[3][3][1]    # Zwraca 101
```

Poniższy program *pcost.py* ilustruje sposób wczytywania danych do listy i wykonywania prostych obliczeń. W tym przykładzie przyjmuje się, że wiersze zawierają wartości oddzielone przecinkami.

Program oblicza sumę iloczynu dwóch kolumn.

```
# pcost.py
#
# Czyta wiersze wejściowe w formacie 'NAZWA,UDZIAŁY,CENA'.
# Na przykład:
#
#   SYM,123,456,78

import sys
if len(sys.argv) != 2:
    raise SystemExit(f'Przykład użycia: {sys.argv[0]} nazwa_pliku')

rows = []
with open(sys.argv[1], 'rt') as file:
    for line in file:
        rows.append(line.split(','))
```

```
# Wiersze zawierają listę w poniższym formacie
# [
# ['SYM', '123', '456.78']
# ...
# ]

total = sum([int(row[1]) * float(row[2]) for row in rows])
print(f'Całkowity koszt: {total:0.2f}')
```

Pierwsza linia tego programu używa instrukcji `import` do załadowania modułu `sys` z biblioteki Pythona. Ten moduł służy do uzyskiwania argumentów wiersza poleceń, które znajdują się na liście `sys.argv`. Wstępne sprawdzenie zapewnia, że podana została nazwa pliku. Jeśli nie, zgłaszany jest wyjątek `SystemExit` z komunikatem o błędzie. W tej wiadomości `sys.argv[0]` wstawia nazwę uruchomionego programu.

Funkcja `open()` używa nazwy pliku określonej w wierszu poleceń. W pętli `for line in file` czytany jest plik, linia po linii. Każda linia jest podzielona na małą listę, a przecinek jest separatorem. Ta lista jest dołączona do wierszy. Wynik końcowy, `rows`, to lista list — pamiętaj, że lista może zawierać wszystko, w tym inne listy.

Wyrażenie `[int(row[1]) * float(row[2]) for row in rows]` tworzy nową listę, przeglądając wszystkie listy w wierszach i obliczając iloczyn drugiego i trzeciego elementu. Ta użyteczna technika konstruowania listy jest znana jako *listy składane* (ang. *list comprehension*). To samo obliczenie można by wyrazić bardziej szczegółowo w następujący sposób:

```
values = []
for row in rows:
    values.append(int(row[1]) * float(row[2]))
total = sum(values)
```

Co do zasady, listy składane są preferowaną techniką wykonywania prostych obliczeń. Wbudowana funkcja `sum()` oblicza sumę dla wszystkich elementów w sekwencji.

1.9. Krotki

Aby utworzyć proste struktury danych, możesz spakować kolekcję wartości do niezmiennego obiektu znanego jako *krotka* (ang. *tuple*). Aby utworzyć krotkę, umieść grupę wartości w nawiasach:

```
holding = ('GOOG', 100, 490.10)
address = ('www.python.org', 80)
```

Można również zdefiniować krotki 0- i 1-elementowe, ale mają specjalną składnię:

```
a = ()          # 0-krotka (pusta krotka)
b = (item,)     # 1-krotka (zwróć uwagę na końcowy przecinek)
```

Wartości w krotce można wyodrębnić za pomocą indeksu liczbowego, podobnie jak listę. Jednak bardziej powszechne jest rozpakowywanie krotek do zestawu zmiennych, na przykład:

```
name, shares, price = holding
host, port = address
```

Chociaż krotki obsługują większość tych samych operacji co listy (takie jak indeksowanie, wycinanie i łączenie), elementów krotki nie można zmienić po utworzeniu — to znaczy, że nie można zastępować, usuwać ani dołączać nowych elementów do istniejącej krotki. Krotkę najlepiej postrzegać jako pojedynczy niezmienny obiekt, który składa się z kilku części, a nie jako zbiór odrębnych obiektów, takich jak lista.

Krotki i listy są często używane razem do reprezentowania danych. Na przykład ten program pokazuje, jak można odczytać plik zawierający kolumny danych oddzielone przecinkami:

```
# Plik zawierający wiersze w postaci „nazwa, udziały, cena”
filename = 'portfolio.csv'

portfolio = []
with open(filename) as file:
    for line in file:
        row = line.split(',')
        name = row[0]
        shares = int(row[1])
        price = float(row[2])
        holding = (name, shares, price)
        portfolio.append(holding)
```

Wynikowa lista `portfolio` stworzona przez ten program wygląda jak dwuwymiarowa tablica wierszy i kolumn. Każdy wiersz jest reprezentowany przez krotkę i można uzyskać do niego dostęp w następujący sposób:

```
>>> portfolio[0]
('AA', 100, 32.2)
>>> portfolio[1]
('IBM', 50, 91.1)
>>>
```

Dostęp do poszczególnych pozycji danych można uzyskać w następujący sposób:

```
>>> portfolio[1][1]
50
>>> portfolio[1] [2]
91.1
>>>
```

Oto jak w pętli rozpakować pola rekordów do zbioru zmiennych:

```
total = 0.0
for name, shares, price in portfolio:
    total += shares * price
```

Alternatywnie możesz skorzystać z list składanych:

```
total = sum([shares * price for _, shares, price in portfolio])
```

Podczas iteracji po krotkach zmienna _ może służyć do wskazania odrzuconej wartości. W powyższym wyliczeniu oznacza to, że ignorujemy pierwszy element (nazwę).

1.10. Zbiory

Zbiór to nieuporządkowana kolekcja unikalnych obiektów. Zbiory służą do znajdowania odrębnych wartości lub rozwiązywania problemów z członkostwem. Aby utworzyć zbiór, umieść kolekcję wartości w nawiasach klamrowych lub podaj istniejącą kolekcję elementów do `set()`. Na przykład:

```
names1 = {'IBM', 'MSFT', 'AA'}
names2 = set(['IBM', 'MSFT', 'HPE', 'IBM', 'CAT'])
```

Elementy zbioru są zazwyczaj ograniczone do obiektów niezmiennych. Na przykład możesz utworzyć zbiór liczb, ciągów lub krotek, ale nie możesz stworzyć zbioru zawierającego listy. Jednak większość popularnych obiektów prawdopodobnie będzie współpracować ze zbiorami — jeśli masz wątpliwości, spróbuj.

W przeciwieństwie do list i krotek zbiory są nieuporządkowane i nie mogą być indeksowane za pomocą liczb. Co więcej, elementy zbioru nigdy nie są powielane. Jeśli na przykład sprawdzisz wartość `names2` z poprzedniego kodu, otrzymasz następujące informacje:

```
>>> names2
{'CAT', 'IBM', 'MSFT', 'HPE'}
>>>
```

Zauważ, że `'IBM'` pojawia się tylko raz. Nie można również przewidzieć kolejności elementów; wynik może się różnić od pokazanego. Kolejność może się nawet zmieniać w zależności od interpretera na tym samym komputerze.

Jeśli pracujesz z istniejącymi danymi, możesz również utworzyć zbiór za pomocą list składanych. Na przykład to wyrażenie zamienia wszystkie nazwy udziałów dla danych z poprzedniej sekcji w zbiór:

```
names = {s[0] for s in portfolio}
```

Aby utworzyć pusty zbiór, użyj `set()` bez argumentów:

```
r = set()      # Początkowo pusty zbiór
```

Zbiory obsługują standardowe zbiory operacji, w tym sumę, przecięcie, różnicę i różnicę symetryczną. Oto przykład:

```
a = t | s      # Suma {'MSFT', 'CAT', 'HPE', 'AA', 'IBM'}
b = t & s      # Przecięcie {'IBM', 'MSFT'}
c = t - s      # Różnica {'CAT', 'HPE'}
d = s - t      # Różnica {'AA'}
e = t ^ s      # Różnica symetryczna {'CAT', 'HPE', 'AA'}
```

Operacja różnicy `s – t` daje elementy w `s`, które nie są w `t`. Symetryczna różnica `s ^ t` daje elementy, które są w `s` lub `t`, ale nie w obu.

Nowe elementy można dodawać do zbioru za pomocą `add()` lub `update()`:

```
t.add('DIS')                    # Dodaj pojedynczy element
s.update({'JJ', 'GE', 'ACME'})  # Dodaje wiele elementów do s
```

Element można usunąć za pomocą `remove()` lub `discard()`:

```
t.remove('IBM')      # Usuń 'IBM' lub zgłoś wyjątek KeyError, jeśli go nie ma
s.discard('SCOX')    # Usuń 'SCOX', jeśli istnieje
```

Różnica między `remove()` a `discard()` polega na tym, że `discard()` nie zgłasza wyjątku, jeśli element nie występuje.

1.11. Słowniki

Słownik tworzy mapowanie pomiędzy kluczami a wartościami. Słownik tworzy się, umieszczając pary klucz-wartość, oddzielone dwukropkiem, w nawiasach klamrowych ({ }), w ten sposób:

```
s = {
     'name': 'GOOG',
     'shares': 100,
     'price': 490.10
    }
```

Aby uzyskać dostęp do elementów słownika, użyj operatora indeksowania w następujący sposób:

```
name = s['name']
cost = s['shares'] * s['price']
```

Wstawianie lub modyfikowanie obiektów działa tak:

```
s['shares'] = 75
s['date'] = '2007-06-07'
```

Słownik jest użytecznym sposobem tworzenia obiektu składającego się ze zdefiniowanych pól. Jednak słowniki są również powszechnie używane jako mapowanie do wykonywania szybkich wyszukiwań danych nieuporządkowanych. Na przykład oto słownik cen akcji:

```
prices = {
'GOOG': 490.1,
'AAPL': 123.5,
'IBM': 91.5,
'MSFT': 52.13
}
```

Mając taki słownik, możesz sprawdzić cenę:

```
p = prices['IBM']
```

Członkostwo w słowniku jest testowane za pomocą operatora `in`:

```
if 'IBM' in prices:
    p = prices['IBM']
else:
    p = 0.0
```

Ta konkretna sekwencja kroków może być również wykonana w bardziej zwięzły sposób za pomocą metody `get()`:

```
p = prices.get('IBM', 0.0) # prices['IBM'], jeśli istnieje; w przeciwnym razie 0.0
```

Użyj instrukcji `del`, aby usunąć element słownika:

```
del prices['GOOG']
```

Chociaż łańcuchy są najpopularniejszym typem klucza, możesz używać wielu innych obiektów Pythona, w tym liczb i krotek. Na przykład krotki są często używane do konstruowania kluczy złożonych lub wieloczęściowych:

```
prices = { }
prices[('IBM', '2015-02-03')] = 91.23
prices['IBM', '2015-02-04'] = 91.42      # Pominięto nawiasy
```

W słowniku można umieścić dowolny obiekt, w tym inne słowniki. Jednak mutowalne struktury danych, takie jak listy, zbiory i słowniki, nie mogą być używane jako klucze.

Słowniki są często wykorzystywane jako bloki konstrukcyjne dla różnych algorytmów i problemów z obsługą danych. Jednym z takich problemów jest zestawienie. Oto przykład tego, jak możesz policzyć całkowitą liczbę udziałów dla każdej nazwy akcji dla poprzednich danych:

```
portfolio = [
('ACME', 50, 92.34),
('IBM', 75, 102.25),
('PHP', 40, 74.50),
('IBM', 50, 124.75)
]

total_shares = {s[0]: 0 for s in portfolio}
for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_shares = {'IBM': 125, 'ACME': 50, 'PHP': 40}
```

`{s[0]: 0 for s in portfolio}` jest przykładem słownika składanego. Tworzy słownik par klucz-wartość z innego zbioru danych. W tym przypadku tworzy początkowe mapowanie słownika, mapując nazwę akcji na 0. Pętla `for` iteruje po słowniku i sumuje wszystkie posiadane udziały dla każdego symbolu giełdowego.

Wiele typowych zadań przetwarzania danych, takich jak to, zostało już zaimplementowanych przez moduły biblioteczne. Na przykład moduł `collections` ma obiekt `Counter`, którego można użyć do tego zadania:

```
from collections import Counter

total_shares = Counter()
for name, shares, _ in portfolio:
    total_shares[name] += shares

# total_shares = Counter({'IBM': 125, 'ACME': 50, 'PHP': 40})
```

Pusty słownik tworzony jest na dwa sposoby:

```
prices = {}       # Pusty słownik
prices = dict()   # Pusty słownik
```

Bardziej idiomatyczne jest użycie {} dla pustego słownika — chociaż wymagana jest ostrożność, ponieważ może wyglądać tak, jakbyś próbował utworzyć pusty zbiór (zamiast tego użyj `set()`). Funkcja `dict()` jest powszechnie używana do tworzenia słowników na podstawie wartości klucz-wartość. Na przykład:

```
pairs = [('IBM', 125), ('ACME', 50), ('PHP', 40)]
d = dict(pairs)
```

Aby uzyskać listę kluczy słownika, przekonwertuj słownik na listę:

```
syms = list(prices)      # syms = ['AAPL', 'MSFT', 'IBM', 'GOOG']
```

Alternatywnie możesz uzyskać klucze za pomocą `dict.keys()`:

```
syms = prices.keys()
```

Różnica między tymi dwiema metodami polega na tym, że `keys()` zwracają specjalny „widok kluczy", który jest dołączony do słownika i aktywnie odzwierciedla zmiany wprowadzone w słowniku. Na przykład:

```
>>> d = {'x': 2, 'y': 3}
>>> k = d.keys()
>>> k
dict_keys(['x', 'y'])
>>> d['z'] = 4
>>> k
dict_keys(['x', 'y', 'z'])
>>>
```

Klucze zawsze pojawiają się w tej samej kolejności, w jakiej pozycje zostały początkowo wstawione do słownika. Powyższa konwersja listy zachowa tę kolejność. Może to być przydatne, gdy słowniki są używane do reprezentowania danych klucz-wartość odczytywanych z plików i innych źródeł danych. Słownik zachowa kolejność wprowadzania. Może to pomóc w czytelności i debugowaniu. Przyda się również, jeśli chcesz zapisać dane z powrotem do pliku. Jednak przed Pythonem 3.6 ta kolejność nie była gwarantowana, więc nie można na niej polegać, jeśli wymagana jest zgodność ze starszymi wersjami Pythona. Kolejność nie jest również gwarantowana, jeśli miało miejsce wielokrotne usuwanie i wstawianie.

Aby uzyskać wartości przechowywane w słowniku, użyj metody `dict.values()`. Aby uzyskać pary klucz-wartość, użyj `dict.items()`. Oto jak wykonać iterację całej zawartości słownika jako pary klucz-wartość:

```
for sym, price in prices.items():
    print(f'{sym} = {price}')
```

1.12. Iteracja i pętle

Najczęściej stosowaną konstrukcją pętli jest instrukcja `for`, która iteruje po kolekcji elementów. Jedną z popularnych form iteracji jest wykonanie pętli przez wszystkie elementy sekwencji — takie jak ciąg, lista lub krotka. Oto przykład:

```
for n in [1, 2, 3, 4, 5, 6, 7, 8, 9]:
    print(f'2 do potęgi {n} to {2**n}')
```

W tym przykładzie zmiennej n będą przypisywane kolejne pozycje z listy [1, 2, 3, 4, ..., 9] w każdej iteracji. Ponieważ zapętlanie zakresów liczb całkowitych jest dość powszechne, istnieje skrót:

```
for n in range(1, 10):
    print(f'2 do potęgi {n} to {2**n}')
```

Funkcja range(i, j [,step]) tworzy obiekt, który reprezentuje zakres liczb całkowitych o wartościach od i w górę, ale nie włączając j. Jeśli wartość początkowa zostanie pominięta, przyjmuje się, że wynosi zero. Opcjonalny krok iteracji może być również podany jako trzeci argument. Oto kilka przykładów:

```
a = range(5)          # a = 0, 1, 2, 3, 4
b = range(1, 8)       # b = 1, 2, 3, 4, 5, 6, 7
c = range(0, 14, 3)   # c = 0, 3, 6, 9, 12
d = range(8, 1, -1)   # d = 8, 7, 6, 5, 4, 3, 2
```

Obiekt utworzony przez range() wylicza wartości na żądanie. Dzięki temu jest wydajny w użyciu nawet dla dużego zakresu liczb.

Instrukcja for nie ogranicza się do sekwencji liczb całkowitych. Może być używana do iteracji wielu rodzajów obiektów, w tym ciągów, list, słowników i plików. Oto przykład:

```
message = 'Hello World'
# Wyświetl poszczególne znaki wiadomości
for c in message:
    print(c)

names = ['Dave', 'Mark', 'Ann', 'Phil']
# Wyświetl elementy listy
for name in names:
    print(name)

prices = {'GOOG': 490.10, 'IBM': 91.50, 'AAPL': 123.15}
# Wyświetl wszystkie elementy słownika
for key in prices:
    print(key, '=', prices[key])

# Wyświetl wszystkie linie pliku
with open('foo.txt') as file:
    for line in file:
        print(line, end='')
```

Pętla for jest jedną z najpotężniejszych funkcji językowych Pythona, ponieważ możesz tworzyć niestandardowe obiekty iteratorów i funkcje generatora, które dostarczają mu sekwencje wartości. Więcej szczegółów na temat iteratorów i generatorów można znaleźć w rozdziale 6.

1.13. Funkcje

Użyj instrukcji def, aby zdefiniować funkcję:

```
def remainder(a, b):
    q = a // b      # // dzieli całkowicie.
    r = a - q * b
    return r
```

Aby wywołać funkcję, użyj jej nazwy oraz podaj argumenty w nawiasach, na przykład `result = remainder(37, 15)`.

Powszechną praktyką jest zamieszczanie opisu funkcji w pierwszej linii. Ten opis zwraca polecenie `help()` i może być używany przez IDE oraz inne narzędzia programistyczne do wspomagania programisty. Na przykład:

```
def remainder(a, b):
    '''
    Oblicza resztę z dzielenia a przez b
    '''
    q = a // b
    r = a - q * b
    return r
```

Jeśli dane wejściowe i wyjściowe funkcji nie są jasno określone za pomocą nazw, mogą być oznaczone typami:

```
def remainder(a: int, b: int) -> int:
    '''
    Oblicza resztę z dzielenia a przez b
    '''
    q = a // b
    r = a - q * b
    return r
```

Takie adnotacje mają jedynie charakter informacyjny i nie są wymuszane w czasie wykonywania. Nadal można wywołać powyższą funkcję z wartościami niebędącymi liczbami całkowitymi, takimi jak `result = remainder(37.5, 3.2)`.

Użyj krotki, aby zwrócić wiele wartości z funkcji:

```
def divide(a, b):
    q = a // b # Jeśli a i b są liczbami całkowitymi, q jest liczbą całkowitą
    r = a - q * b
    return (q, r)
```

Gdy w krotce zwracanych jest wiele wartości, można je rozpakować do oddzielnych zmiennych w następujący sposób:

```
quotient, remainder = divide(1456, 33)
```

Aby przypisać wartość domyślną do parametru funkcji, użyj zapisu:

```
def connect(hostname, port, timeout=300):
    # Ciało funkcji
    ...
```

Gdy w definicji funkcji podane są wartości domyślne, można je pominąć w kolejnych wywołaniach funkcji. Pominięty argument przyjmie podaną wartość domyślną.

Oto przykład:

```
connect('www.python.org', 80)
connect('www.python.org', 80, 500)
```

Argumenty domyślne są często używane dla opcjonalnych właściwości. Jeśli jest wiele takich argumentów, może ucierpieć czytelność programu. Dlatego zaleca się ich określanie za pomocą argumentów słów kluczowych. Na przykład:

```
connect('www.python.org', 80, timeout=500)
```

Jeśli znasz nazwy argumentów, wszystkie z nich można nazwać podczas wywoływania funkcji. Po nazwaniu argumentów kolejność, w jakiej są podane, nie ma znaczenia. Na przykład poprawny jest taki zapis:

```
connect(port=80, hostname='www.python.org')
```

Gdy zmienne są tworzone lub przypisywane wewnątrz funkcji, ich zakres jest lokalny. Oznacza to, że zmienna jest zdefiniowana tylko w treści funkcji i jest niszczona, gdy funkcja zwraca wynik. Funkcje mogą również uzyskiwać dostęp do zmiennych zdefiniowanych poza funkcją, o ile są one zdefiniowane w tym samym pliku. Na przykład:

```
debug = True       # Zmienna globalna

def read_data(filename):
   if debug:
       print('Odczyt', filename)
    ...
```

Zasady ustalania zakresu opisano bardziej szczegółowo w rozdziale 5.

1.14. Wyjątki

Jeśli w programie wystąpi błąd, zgłaszany jest wyjątek i pojawia się komunikat dotyczący ostatniego wywołania:

```
Traceback (most recent call last):
  File "readport.py", line 9, in <module>
    shares = int(row[1])
ValueError: invalid literal for int() with base 10: 'N/A'
```

Powyższy komunikat wskazuje typ błędu, który wystąpił, wraz z jego lokalizacją. Zwykle błędy powodują zakończenie działania programu. Możesz jednak przechwycić i obsłużyć wyjątki za pomocą instrukcji `try-except`, tak jak poniżej:

```
portfolio = []
with open('portfolio.csv') as file:
    for line in file:
        row = line.split(',')
        try:
            name = row[0]
            shares = int(row[1])
            price = float(row[2])
            holding = (name, shares, price)
            portfolio.append(holding)
        except ValueError as err:
            print('Błędny wiersz:', row)
            print('Powód:', err)
```

W tym kodzie, jeśli wystąpi wyjątek `ValueError`, szczegóły dotyczące przyczyny błędu są umieszczane w zmiennej `err`, a program przechodzi do bloku z obsługą wyjątku. Jeśli zostanie zgłoszony inny rodzaj wyjątku, program zawiesza się jak zwykle. Jeśli nie wystąpią żadne błędy, blok z obsługą wyjątku jest ignorowany. Gdy obsługiwany jest wyjątek, wykonywanie programu jest wznawiane z instrukcją, która następuje bezpośrednio po ostatnim bloku z obsługą wyjątku. Program nie powraca do lokalizacji, w której wystąpił wyjątek.

Instrukcja `raise` służy do sygnalizowania wyjątku. Musisz podać nazwę wyjątku. Oto przykład, jak wyrzucić wyjątek `RuntimeError` (wyjątek wbudowany):

```
raise RuntimeError('Komputer mówi Nie')
```

Właściwe zarządzanie zasobami systemowymi, takimi jak blokady, pliki i połączenia sieciowe, jest często trudne w połączeniu z obsługą wyjątków. Czasami występują działania, które należy wykonać bez względu na to, co się stanie. W tym celu użyj `try-finally`. Oto przykład dotyczący blokady, którą należy zwolnić, aby uniknąć zakleszczenia:

```
import threading
lock = threading.Lock()
...
lock.acquire()
# Jeśli blokada zostanie nałożona, musi zostać zwolniona
try:
    ...
    instrukcje
    ...
finally:
    lock.release() # Zawsze działa
```

Aby uprościć takie programowanie, większość obiektów, które obejmują zarządzanie zasobami, obsługuje również instrukcję `with`. Oto zmodyfikowana wersja powyższego kodu:

```
with lock:
    ...
    instrukcje
    ...
```

W tym przykładzie obiekt `lock` jest blokowany automatycznie po wykonaniu instrukcji `with`. Gdy wykonanie opuszcza kontekst bloku `with`, blokada na `lock` jest automatycznie zwalniana. Odbywa się to niezależnie od tego, co dzieje się w bloku `with`. Jeśli na przykład wystąpi wyjątek, blokada zostanie zwolniona, gdy program opuści kontekst bloku.

Instrukcja `with` jest zwykle zgodna tylko z obiektami związanymi z zasobami systemowymi lub środowiskiem wykonawczym — takimi jak pliki, połączenia i blokady. Jednak obiekty zdefiniowane przez użytkownika mogą mieć własne przetwarzanie niestandardowe, co opisano w dalszej części rozdziału 3.

1.15. Zakończenie programu

Program kończy się, gdy nie ma więcej instrukcji do wykonania w programie lub gdy zostanie zgłoszony nieobsługiwany wyjątek `SystemExit`. Jeśli chcesz wymusić zamknięcie programu, zobacz poniższe instrukcje:

```
raise SystemExit()                    # Wyjdź bez komunikatu o błędzie
raise SystemExit("Coś jest nie tak")  # Wyjdź z błędem
```

Przy wyjściu interpreter dokłada wszelkich starań, aby zwolnić z pamięci wszystkie aktywne obiekty.

Jeśli jednak musisz wykonać określoną akcję czyszczenia (usunąć pliki, zamknąć połączenie), możesz zarejestrować ją w module `atexit` w następujący sposób:

```
import atexit

# Przykład
connection = open_connection("deaddot.com")

def cleanup():
    print "Wyjście..."
    close_connection(connection)

atexit.register(cleanup)
```

1.16. Obiekty i klasy

Wszystkie wartości użyte w programie są obiektami. Obiekt składa się z wewnętrznych danych i metod, które wykonują różnego rodzaju operacje na tych danych. Używałeś już obiektów i metod podczas pracy z wbudowanymi typami, takimi jak ciągi i listy. Na przykład:

```
items = [37, 42]    # Utwórz obiekt listy
items.append(73)    # Wywołaj metodę append()
```

Funkcja `dir()` wyświetla listę metod dostępnych dla obiektu. Jest to przydatne narzędzie do interaktywnych eksperymentów, gdy nie jest dostępne żadne bardziej zaawansowane IDE. Na przykład:

```
>>> items = [37, 42]
>>> dir(items)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
...
'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
>>>
```

Podczas sprawdzania obiektów zobaczysz na liście znajome metody, takie jak `append()` i `insert()`. Zobaczysz również metody specjalne, których nazwy zaczynają się i kończą podwójnym podkreśleniem. Te metody implementują różne operatory. Na przykład metoda `__add__()` służy do implementacji operatora +. Metody te są wyjaśnione bardziej szczegółowo w kolejnych rozdziałach.

```
>>> items.__add__([73, 101])
[37, 42, 73, 101]
>>>
```

Instrukcja `class` służy do definiowania nowych typów obiektów i programowania obiektowego. Na przykład poniższa klasa definiuje stos z operacjami `push()` i `pop()`:

```
class Stack:
    def __init__(self): # Inicjalizacja stosu
        self._items = [ ]

    def push(self, item):
        self._items.append(item)

    def pop(self):
        return self._items.pop()

    def __repr__(self):
        return f'<{type(self).__name__} at 0x{id(self):x}, size={len(self)}>'

    def __len__(self):
        return len(self._items)
```

Wewnątrz definicji klasy metody są definiowane za pomocą instrukcji `def`. Pierwszy argument w każdej metodzie zawsze odnosi się do samego obiektu. Zgodnie z konwencją argument ten ma nazwę `self`. Wszystkie operacje obejmujące atrybuty obiektu muszą się jawnie odnosić do zmiennej `self`.

Metody z podwójnymi znakami podkreślenia na początku i na końcu są metodami specjalnymi. Na przykład `__init__` służy do inicjalizacji obiektu. W tym przypadku `__init__` tworzy wewnętrzną listę do przechowywania danych stosu.

Aby użyć klasy, stwórz kod taki jak ten:

```
s = Stack()       # Utwórz stos
s.push('Dave')    # Włóż do niego kilka rzeczy
s.push(42)
p.push([3, 4, 5])
x = s.pop()       # x przyjmuje wartości [3,4,5]
y = s.pop()       # y przyjmuje wartość 42
```

W definicji klasy zauważysz, że metody używają wewnętrznej zmiennej `_items`. Python nie ma żadnego mechanizmu ukrywania ani ochrony danych. Istnieje jednak konwencja programowania, w której nazwy poprzedzone pojedynczym podkreśleniem są uważane za „prywatne”. W tym przykładzie element `_items` powinien być traktowany (przez Ciebie) jako implementacja wewnętrzna i nie może być używany poza samą klasą `Stack`. Pamiętaj, że ta konwencja nie jest faktycznie egzekwowana — jeśli chcesz uzyskać dostęp do `_items`, możesz to zrobić w dowolnym momencie. Po prostu będziesz musiał o tym powiedzieć swoim współpracownikom, gdy będą sprawdzać Twój kod.

Metody `__repr__()` i `__len__()` są po to, aby obiekt dobrze współgrał z resztą środowiska. W tym przypadku `__len__()` powoduje, że stos działa z wbudowaną funkcją `len()`, a `__repr__()` zmienia sposób wyświetlania stosu. Dobrym pomysłem jest zawsze definiowanie `__repr__()`, ponieważ może to uprościć debugowanie.

```
>>> s = Stack()
>>> s.push('Dave')
>>> s.push(42)
>>> len(s)
2
>>> s
<Stack at 0x10108c1d0, size=2>
>>>
```

Główną cechą obiektów jest to, że możesz dodawać lub przedefiniować możliwości istniejących klas poprzez dziedziczenie.

Załóżmy, że chcesz dodać metodę zamiany dwóch najwyższych elementów na stosie. Możesz napisać taką klasę:

```
class MyStack(Stack):
    def swap(self):
        a = self.pop()
        b = self.pop()
        self.push(a)
        self.push(b)
```

Klasa `MyStack` jest identyczna jak `Stack` — z tą różnicą, że ma nową metodę, `swap()`.

```
>>> s = MyStack()
>>> s.push('Dave')
>>> s.push(42)
>>> s.swap()
>>> s.pop()
'Dave'
>>> s.pop()
42
>>>
```

Dziedziczenie można również wykorzystać do zmiany zachowania istniejącej metody. Załóżmy, że chcesz ograniczyć stos do przechowywania tylko danych liczbowych. Napisz taką klasę:

```
class NumericStack(Stack):
    def push(self, item):
        if not isinstance(item, (int, float)):
            raise TypeError('Oczekiwano liczby lub wartości zmiennoprzecinkowej')
        super().push(item)
```

W tym przykładzie metoda `push()` została przedefiniowana, aby dodać dodatkowe sprawdzanie. Operacja `super()` jest sposobem na wywołanie poprzednio zdefiniowanej metody `push()`. Oto jak ta klasa będzie działać:

```
>>> s = NumericStack()
>>> s.push(42)
>>> s.push('Dave')
Traceback (most recent call last):
...
TypeError: Expected an int or float
>>>
```

Często dziedziczenie nie jest najlepszym rozwiązaniem. Załóżmy, że chcesz zdefiniować prosty, czterofunkcyjny kalkulator oparty na stosie, który działałby w ten sposób:

```
>>> # Oblicz 2 + 3 * 4
>>> calc = Calculator()
>>> calc.push(2)
>>> calc.push(3)
>>> calc.push(4)
>>> calc.mul()
```

```
>>> calc.add()
>>> calc.pop()
14
>>>
```

Możesz spojrzeć na ten kod, sprawdzić wykorzystanie metod `push()` oraz `pop()` i pomyśleć, że klasę `Calculator` można zdefiniować poprzez dziedziczenie z klasy `Stack`. Mimo że takie rozwiązanie by zadziałało, prawdopodobnie lepiej zdefiniować `Calculator` jako całkowicie odrębną klasę:

```
class Calculator:
    def __init__(self):
        self._stack = Stack()

    def push(self, item):
        self._stack.push(item)

    def pop(self):
        return self._stack.pop()

    def add(self):
        self.push(self.pop() + self.pop())

    def mul(self):
        self.push(self.pop() * self.pop())

    def sub(self):
        right = self.pop()
        self.push(self.pop() - right)

    def div(self):
        right = self.pop()
        self.push(self.pop() / right)
```

W tej implementacji klasa `Calculator` zawiera `Stack` jako wewnętrzny szczegół implementacji. To przykład *kompozycji*. Metody `push()` i `pop()` delegują do wewnętrznej klasy `Stack`. Głównym powodem takiego podejścia jest to, że tak naprawdę nie myślisz o implementacji kalkulatora jako o stosie. To osobna koncepcja — inny rodzaj obiektu. Analogicznie Twój telefon zawiera jednostkę centralną (CPU), ale zwykle nie myślisz o telefonie jako o rodzaju procesora.

1.17. Moduły

Gdy Twoje programy będą się powiększać, będziesz chciał podzielić je na wiele plików, aby ułatwić ich ewentualną poprawę. Aby to zrobić, użyj instrukcji `import`. Aby utworzyć moduł, umieść odpowiednie instrukcje i definicje w pliku z rozszerzeniem *.py* z taką samą nazwą jak nazwa modułu.

Oto przykład:

```
# readport.py
#
# Odczytuje plik z danymi 'NAZWA,UDZIAŁY,CENA'
```

```
def read_portfolio(filename):
    portfolio = []
    with open(filename) as file:
        for line in file:
            row = line.split(',')
            try:
                name = row[0]
                shares = int(row[1])
                price = float(row[2])
                holding = (name, shares, price)
                portfolio.append(holding)
            except ValueError as err:
                print('Błędny wiersz:', row)
                print('Powód:', err)
    return portfolio
```

Aby skorzystać ze swojego modułu w innych plikach, użyj instrukcji `import`. Oto przykład modułu *pcost.py*, który używa powyższej funkcji `read_portfolio()`:

```
# pcost.py

import readport

def portfolio_cost(filename):
    '''
    Oblicz łączną cenę akcji*cenę portfela
    '''
    port = readport.read_portfolio(filename)
    return sum(shares * price for _, shares, price in port)
```

Instrukcja `import` tworzy nową przestrzeń nazw (lub środowisko) i wykonuje wszystkie instrukcje w skojarzonym pliku *.py* w tej przestrzeni nazw. Aby uzyskać dostęp do zawartości przestrzeni nazw po zaimportowaniu, użyj nazwy modułu jako przedrostka, tak jak w `readport.read_portfolio()` w poprzednim przykładzie.

Jeśli instrukcja `import` nie powiedzie się z powodu wyjątku `ImportError`, musisz sprawdzić kilka rzeczy w swoim środowisku. Najpierw upewnij się, że utworzyłeś plik o nazwie *readport.py*. Następnie sprawdź katalogi wymienione w *sys.path*. Jeśli Twój plik nie jest zapisany w jednym z tych katalogów, Python nie będzie w stanie go znaleźć.

Jeśli chcesz zaimportować moduł pod inną nazwą, podaj w instrukcji `import` opcjonalny kwalifikator:

```
import readport as rp
port = rp.read_portfolio('portfolio.dat')
```

Aby zaimportować określone definicje do bieżącej przestrzeni nazw, użyj instrukcji `from`:

```
from readport import read_portfolio
port = read_portfolio('portfolio.dat')
```

Podobnie jak w przypadku obiektów, funkcja `dir()` wyświetla zawartość modułu. Jest to przydatne narzędzie do interaktywnych eksperymentów.

```
>>> import readport
>>> dir(readport)
```

```
['__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', 'read_portfolio']
...
>>>
```

Python zapewnia dużą standardową bibliotekę modułów, które upraszczają część zadań programistycznych. Na przykład moduł `csv` jest standardową biblioteką do obsługi plików o wartościach oddzielonych przecinkami. Możesz go użyć w swoim programie w następujący sposób:

```
# readport.py
#
# Odczytuje plik z danymi 'NAZWA,UDZIAŁY,CENA'

import csv

def read_portfolio(filename):
    portfolio = []
    with open(filename) as file:
        rows = csv.reader(file)
        for row in rows:
            try:
                name = row[0]
                shares = int(row[1])
                price = float(row[2])
                holding = (name, shares, price)
                portfolio.append(holding)
            except ValueError as err:
                print('Błędny wiersz:', row)
                print('Powód:', err)
    return portfolio
```

Python posiada również ogromną liczbę modułów innych firm, które można zainstalować w celu rozwiązania niemal każdego zadania, jakie można sobie wyobrazić (w tym odczytywanie plików CSV). Zobacz *https://pypi.org*.

1.18. Pisanie skryptów

Każdy plik można wykonać jako skrypt lub jako bibliotekę zaimportowaną za pomocą instrukcji `import`. Do lepszego wsparcia importów kod skryptu jest często dołączany razem ze sprawdzeniem nazwy modułu:

```
# readport.py
#
# Odczytuje plik z danymi 'NAZWA,UDZIAŁY,CENA'

import csv

def read_portfolio(filename):
    ...
def main():
    portfolio = read_portfolio('portfolio.csv')
```

```
    for name, shares, price in portfolio:
        print(f'{name:>10s} {shares:10d} {price:10.2f}')

if __name__ == '__main__':
    main()
```

`__name__` jest wbudowaną zmienną, która zawsze zawiera nazwę załączanego modułu. Jeśli program jest uruchamiany jako główny skrypt za pomocą polecenia `python readport.py`, zmienna `__name__` jest ustawiana na `'__main__'`. W przeciwnym razie, jeśli kod jest importowany za pomocą instrukcji, takiej jak `import readport`, zmienna `__name__` jest ustawiana na `'readport'`.

Jak pokazano, program na stałe korzysta z pliku o nazwie *portfolio.csv*. Możesz także zapytać użytkownika o nazwę pliku lub zaakceptować nazwę pliku jako argument wiersza poleceń. Aby to zrobić, użyj wbudowanej funkcji `input()` lub listy `sys.argv`. Oto zmodyfikowana wersja funkcji `main()`:

```
def main(argv):
    if len(argv) == 1:
        filename = input('Podaj nazwę pliku: ')
    elif len(argv) == 2:
        filename = argv[1]
    else:
        raise SystemExit(f'Przykład użycia: {argv[0]} [ nazwa_pliku ]')
    portfolio = read_portfolio(filename)
    for name, shares, price in portfolio:
        print(f'{name:>10s} {shares:10d} {price:10.2f}')

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

Ten program można uruchomić na dwa różne sposoby z wiersza poleceń:

```
bash % python readport.py
Podaj nazwę pliku: portfolio.csv
...
bash % python readport.py portfolio.csv
...
bash % python readport.py a b c
Przykład użycia: readport.py [ nazwa_pliku ]
bash %
```

W przypadku bardzo prostych programów często wystarczy przetworzyć argumenty w `sys.argv`, jak pokazano. Dla bardziej zaawansowanych zastosowań można skorzystać ze standardowej biblioteki `argparse`.

1.19. Pakiety

W dużych aplikacjach powszechne jest organizowanie kodu w pakiety. Pakiet to hierarchiczna kolekcja modułów. W systemie plików umieść swój kod jako kolekcję plików w katalogu takim jak ten:

```
tutorial/
    __init__.py
    readport.py
    pcost.py
    stack.py
    ...
```

Katalog powinien zawierać plik *__init__.py*, który może być pusty. Teraz powinieneś być w stanie tworzyć zagnieżdżone instrukcje importu. Na przykład:

```
import tutorial.readport
port = tutorial.readport.read_portfolio('portfolio.dat')
```

Jeśli nie lubisz długich nazw, możesz skrócić powyższy zapis:

```
from tutorial.readport import read_portfolio
port = read_portfolio('portfolio.dat')
```

Jednym z problemów w trakcie pracy z pakietami jest wykonywanie importu między plikami w ramach tego samego pakietu. We wcześniejszym przykładzie pokazano moduł *pcost.py*, który rozpoczynał się od importu:

```
# pcost.py
import readport
...
```

Jeśli pliki *pcost.py* i *readport.py* zostaną przeniesione do pakietu, ta instrukcja importu zostanie przerwana. Aby to naprawić, musisz użyć w pełni kwalifikowanej ścieżki dla importu modułu:

```
# pcost.py
from tutorial import readport
...
```

Alternatywnie możesz użyć importu zależnego od pakietu w ten sposób:

```
# pcost.py
from . import readport
...
```

Ta ostatnia forma ma tę zaletę, że nazwa pakietu nie jest zakodowana na sztywno. Ułatwia to późniejszą zmianę nazwy pakietu lub przenoszenie go w ramach projektu.

Inne subtelne szczegóły dotyczące pakietów zostaną omówione później (patrz rozdział 8.).

1.20. Strukturyzacja aplikacji

Gdy zaczniesz pisać więcej kodu w Pythonie, może się okazać, że pracujesz nad większymi aplikacjami, które zawierają mieszankę własnego kodu oraz zależnego od innych firm. Zarządzanie tym wszystkim to złożony temat, który wciąż ewoluuje. Istnieje również wiele sprzecznych opinii na temat tego, co stanowi najlepszą praktykę. Jest jednak kilka istotnych aspektów, o których powinieneś wiedzieć.

Przede wszystkim standardową praktyką jest organizowanie dużych fragmentów kodu w pakiety (tj. katalogi plików *.py*, które zawierają specjalny plik *__init__.py*). Robiąc to, wybierz

unikalną nazwę pakietu dla nazwy katalogu najwyższego poziomu. Głównym celem katalogu pakietów jest zarządzanie instrukcjami importu i przestrzeniami nazw modułów używanych podczas programowania. Chcesz, aby Twój kod był odizolowany od kodu innych osób.

Oprócz głównego kodu źródłowego projektu możesz dodatkowo posiadać testy, przykłady, skrypty i dokumentację. Ten dodatkowy materiał zwykle znajduje się w oddzielnym zestawie katalogów, innym niż pakiet zawierający kod źródłowy. Dlatego często tworzy się katalog najwyższego poziomu dla Twojego projektu i umieszcza się w nim całą swoją pracę. Przykładowa organizacja projektu może wyglądać tak:

```
tutorial-project/
    tutorial/
        __init__.py
        readport.py
        pcost.py
        stack.py
        ...
    tests/
        test_stack.py
        test_pcost.py
        ...
examples/
    sample.py
    ...
doc/
    tutorial.txt
    ...
```

Pamiętaj, że jest więcej niż jeden sposób. Charakter problemu, który rozwiązujesz, może mieć inną strukturę. Niemniej dopóki główny zestaw plików z kodem źródłowym znajduje się w odpowiednim pakiecie (w katalogu z plikiem *__init__.py*), powinno być dobrze.

1.21. Zarządzanie pakietami stron trzecich

Python ma dużą bibliotekę pakietów, które można znaleźć w Python Package Index (*https://pypi.org*). Być może będziesz musiał polegać na niektórych z tych pakietów w swoim własnym kodzie. Aby zainstalować pakiet innej firmy, użyj polecenia takiego jak `pip`:

```
bash % python3 -m pip install pakiet
```

Zainstalowane pakiety są umieszczane w specjalnym katalogu *site-packages*, który można znaleźć, sprawdzając wartość *sys.path*. Na przykład na maszynie UNIX pakiety mogą być umieszczone w */usr/local/lib/python3.8/site-packages*. Jeśli zastanawiasz się, skąd pochodzi pakiet, sprawdź atrybut `__file__` pakietu po zaimportowaniu go do interpretera:

```
>>> import pandas
>>> pandas.__file__
'/usr/local/lib/python3.8/site-packages/pandas/__init__.py'
>>>
```

Jednym z potencjalnych problemów z instalacją pakietu jest to, że możesz nie mieć uprawnień do zmiany lokalnie zainstalowanej wersji Pythona. Nawet gdybyś miał uprawnienia, może to nie być dobry pomysł. Na przykład wiele systemów ma już zainstalowany język Python, który jest wykorzystywany przez różne narzędzia systemowe. Zmiana instalacji tej wersji Pythona jest często złym rozwiązaniem.

Aby stworzyć piaskownicę, w której możesz instalować pakiety i pracować bez martwienia się, że cokolwiek zepsujesz, utwórz wirtualne środowisko za pomocą polecenia:

```
bash % python3 -m venv myproject
```

Spowoduje to skonfigurowanie właściwej instalacji Pythona w katalogu o nazwie *myproject/*. W tym katalogu znajdziesz plik wykonywalny interpretera oraz bibliotekę i będziesz tam mógł bezpiecznie instalować pakiety. Jeśli na przykład uruchomisz *myproject/bin/python3*, otrzymasz interpreter skonfigurowany do użytku osobistego. Możesz instalować pakiety w tym interpreterze, nie martwiąc się o uszkodzenie jakiejkolwiek części domyślnej instalacji Pythona. Aby zainstalować pakiet, podobnie jak poprzednio użyj polecenia `pip`, ale upewnij się, że podałeś poprawny interpreter:

```
bash % ./myproject/bin/python3 -m pip install pakiet
```

Istnieją różne narzędzia, które mają na celu uproszczenie korzystania z `pip` i `venv`. Takie sprawy mogą być również obsługiwane automagicznie przez Twoje IDE. Ponieważ jest to płynna i stale ewoluująca część Pythona, nie podano tutaj dalszych porad.

1.22. Python pasuje do Twojego mózgu

We wczesnym rozwoju języka Python powszechne motto brzmiało: „Pasuje do Twojego mózgu". Nawet dzisiaj rdzeń Pythona stanowi mały język programowania wraz z użytecznym zbiorem wbudowanych obiektów: list, zbiorów i słowników. Wiele praktycznych problemów można rozwiązać za pomocą jedynie podstawowych funkcji przedstawionych w tym rozdziale. Warto o tym pamiętać, gdy zaczynasz swoją przygodę z Pythonem — chociaż zawsze istnieją bardziej skomplikowane sposoby rozwiązania problemu, może istnieć również prosty sposób na jego rozwiązanie za pomocą podstawowych funkcji Pythona. W razie wątpliwości prawdopodobnie podziękujesz swojemu dawnemu „ja" za to, że tak zrobiłeś.

2

Operatory, wyrażenia i manipulacja danymi

Ten rozdział opisuje wyrażenia, operatory oraz sposoby operowania na danych. Wyrażenia są podstawą wykonywania użytecznych obliczeń. Co więcej, biblioteki innych firm mogą dostosować zachowanie Pythona, aby zapewnić lepsze wrażenia użytkownika. Ten rozdział opisuje wyrażenia wysokiego poziomu. Rozdział 3. omawia podstawowe protokoły, których można użyć do dostosowania zachowania interpretera.

2.1. Literały

Literał to wartość wpisana bezpośrednio do programu, taka jak `42`, `4.2` lub `'czterdziesci-dwa'`.

Literały całkowite reprezentują liczbę całkowitą ze znakiem o dowolnym rozmiarze. Możliwe jest określenie liczb całkowitych w formacie binarnym, ósemkowym lub szesnastkowym:

```
42          # Dziesiętna liczba całkowita
0b101010    # Binarna liczba całkowita
0o52        # Ósemkowa liczba całkowita
0x2a        # Szesnastkowa liczba całkowita
```

Baza nie jest przechowywana jako część wartości całkowitej. Wszystkie powyższe literały, jeśli zostaną wyświetlone, będą wyświetlane jako liczba 42. Możesz użyć wbudowanych funkcji bin(x), oct(x) lub hex(x) do konwersji liczby całkowitej na łańcuch reprezentujący jej wartość w różnych podstawach.

Liczby zmiennoprzecinkowe można zapisać, dodając kropkę dziesiętną lub używając notacji naukowej, gdzie `e` lub `E` określa wykładnik. Wszystkie poniższe są liczbami zmiennoprzecinkowymi:

```
4.2
42.
0,42
```

```
4.2e+2
4.2E2
-4.2e-2
```

Wewnętrznie liczby zmiennoprzecinkowe są przechowywane jako wartości podwójnej precyzji IEEE 754 (64-bitowe).

W literałach numerycznych pojedynczy znak podkreślenia (_) może służyć jako wizualny separator między cyframi. Na przykład:

```
123_456_789
0x1234_5678
0b111_00_101
123.789_012
```

Separator cyfr nie jest przechowywany jako część liczby — służy tylko do ułatwienia odczytywania dużych literałów numerycznych w kodzie źródłowym.

Literały logiczne są zapisywane jako `True` i `False`.

Literały ciągów są zapisywane przez umieszczanie znaków w pojedynczych, podwójnych lub potrójnych cudzysłowach. Ciągi w cudzysłowie pojedynczym i podwójnym muszą się znajdować w tym samym wierszu. Ciągi w potrójnym cudzysłowie mogą obejmować wiele wierszy. Na przykład:

```
'Witaj, świecie'
"Witaj, świecie"
'''Witaj, świecie'''
"""Witaj, świecie"""
```

Literały krotki, listy, zbioru i słownika są zapisywane w następujący sposób:

```
(1, 2, 3)               # krotka
[1, 2, 3]               # lista
{1, 2, 3}               # zbiór
{'x':1, 'y':2, 'z':3}   # słownik
```

2.2. Wyrażenia i lokalizacje

Wyrażenie reprezentuje obliczenie, którego wynikiem jest konkretna wartość. Składa się z kombinacji literałów, nazw, operatorów i wywołań funkcji lub metod. Wyrażenie może się zawsze pojawić po prawej stronie instrukcji przypisania, może być używane jako operand w operacjach w innych wyrażeniach lub przekazywane jako argument funkcji. Na przykład:

```
value = 2 + 3 * 5 + sqrt(6+7)
```

Operatory, takie jak + (dodawanie) lub * (mnożenie), reprezentują operację wykonywaną na obiektach dostarczonych jako operandy. `sqrt()` to funkcja stosowana do argumentów wejściowych.

Lewa strona przypisania reprezentuje lokalizację, w której przechowywane jest odniesienie do obiektu. Ta lokalizacja, jak pokazano w poprzednim przykładzie, może być prostym identyfikatorem, takim jak `value`. Może to być również atrybut obiektu lub indeks w kontenerze. Przykład:

```
a = 4 + 2
b[1] = 4 + 2
c['key'] = 4 + 2
d.value = 4 + 2
```

Odczytywanie wartości z lokalizacji jest również wyrażeniem. Na przykład:

```
value = a + b[1] + c['key']
```

Przypisanie wartości i obliczanie wyrażenia to odrębne pojęcia. Nie możesz dołączyć operatora przypisania jako części wyrażenia:

```
while line=file.readline(): # Błąd składni
    print(line)
```

Jednak operator przypisania (`:=`) może być użyty do wykonania tej połączonej akcji obliczania wyrażenia i przypisania. Na przykład:

```
while (line:=file.readline()):
    print(line)
```

Operator `:=` jest zwykle używany w połączeniu z wyrażeniami takimi jak `if` i `while`.

W rzeczywistości użycie go jako normalnego operatora przypisania skutkuje błędem składni, chyba że umieścisz go w nawiasach.

2.3. Standardowe operatory

Obiekty Pythona mogą działać z dowolnymi operatorami z tabeli 2.1.

Zwykle mają one interpretację numeryczną. Istnieją jednak godne uwagi przypadki szczególne. Na przykład operator `+` jest również używany do łączenia sekwencji, operator `*` replikuje sekwencje, `-` jest używany do znajdowania różnic w zbiorach, a `%` wykonuje formatowanie ciągu:

```
[1,2,3] + [4,5]       # [1,2,3,4,5]
[1,2,3] * 4           # [1,2,3,1,2,3,1,2,3,1,2,3]
'%s ma %d wiadomości' % ('Dave', 37)
```

Sprawdzanie operatorów jest procesem dynamicznym. Operacje obejmujące mieszane typy danych często „działają”, jeśli istnieje intuicyjny sens działania. Na przykład możesz dodać liczby całkowite i ułamki:

```
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = 5
>>> a + b
Fraction(17, 3)
>>>
```

Tabela 2.1. *Operatory standardowe*

Operacja	Opis
`x + y`	Dodawanie
`x – y`	Odejmowanie
`x * y`	Mnożenie
`x / y`	Dzielenie
`x // y`	Dzielenie całkowite
`x @ y`	Mnożenie macierzy
`x ** y`	Potęga (`x` do potęgi `y`)
`x % y`	Modulo (`x` modulo `y`). Reszta
`x << y`	Przesunięcie w lewo
`x >> y`	Przesunięcie w prawo
`x & y`	Bitowe i
`x | y`	Bitowe lub
`x ^ y`	Bitowe xor (alternatywa wykluczająca)
`~x`	Negacja bitowa
`–x`	Jednoargumentowy minus
`+x`	Jednoargumentowy plus
`abs(x)`	Wartość bezwzględna
`divmod(x, y)`	Zwraca `(x // y, x % y)`
`pow(x, y [, modulo])`	Zwraca `(x ** y) % modulo`
`round(x [, n])`	Zaokrągla do `n` miejsc po przecinku

Takie operacje jednak nie zawsze działają — na przykład nie działają na ułamkach dziesiętnych.

```
>>> from decimal import Decimal
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Decimal('5')
>>> a + b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Fraction' and 'decimal.Decimal'
>>>
```

Jednak w przypadku większości kombinacji liczb Python stosuje standardową hierarchię wartości logicznych, liczb całkowitych, ułamków, liczb zmiennoprzecinkowych i liczb zespolonych. Operacje typu mieszanego po prostu zadziałają — nie musisz się tym martwić.

2.4. Modyfikacje w miejscu

Python udostępnia operacje modyfikacji „w miejscu” (ang. *in-place*) lub „rozszerzone”, wymienione w tabeli 2.2.

Tabela 2.2. *Rozszerzone operatory przypisania*

Operacja	Opis
`x += y`	`x = x + y`
`x –= y`	`x = x – y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x //= y`	`x = x // y`
`x **= y`	`x = x ** y`
`x %= y`	`x = x % y`
`x @= y`	`x = x @ y`
`x &= y`	`x = x & y`
`x |= y`	`x = x | y`
`x ^= y`	`x = x ^ y`
`x >>= y`	`x = x >> y`
`x <<= y`	`x = x << y`

Nie są one uważane za wyrażenia. Zamiast tego służą do aktualizacji wartości w miejscu. Na przykład:

```
a = 3
a = a + 1    # a = 4
a += 1       # a = 5
```

Obiekty mutowalne mogą używać tych operatorów do wykonywania mutacji danych w miejscu jako optymalizacji. Rozważ ten przykład:

```
>>> a = [1, 2, 3]
>>> b = a                  # Tworzy nowe odniesienie do a
>>> a += [4, 5]            # Aktualizacja w miejscu (nie tworzy nowej listy)
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
>>>
```

W tym przykładzie `a` i `b` są odwołaniami do tej samej listy. Gdy wykonywane jest `a += [4, 5]`, przeprowadzana jest aktualizacja obiektu listy w miejscu bez tworzenia nowej listy. W związku z tym `b` także widzi tę aktualizację. To często zaskakujące.

2.5. Porównywanie obiektów

Operator równości (`x == y`) testuje wartości `x` i `y` pod kątem równości. W przypadku list i krotek muszą one mieć taki sam rozmiar, posiadać takie same elementy i muszą być one umieszczone w tej samej kolejności. W przypadku słowników wartość `True` jest zwracana tylko wtedy, gdy `x` i `y` mają ten sam zestaw kluczy, a wszystkie obiekty z tym samym kluczem mają równe wartości. Dwa zbiory są równe, jeśli mają te same elementy.

Sprawdzenie równości między obiektami niezgodnych typów, takich jak plik i liczba zmiennoprzecinkowa, nie powoduje błędu, ale zwracana jest wartość `False`. Jednak czasami porównanie obiektów różnych typów da wynik `True`. Na przykład porównanie liczby całkowitej i liczby zmiennoprzecinkowej o tej samej wartości:

```
>>> 2 == 2.0
True
>>>
```

Operatory tożsamości (`x is y` i `x is not y`) testują dwie wartości, aby sprawdzić, czy odnoszą się one do tego samego obiektu w pamięci (np. `id(x) == id(y)`). Ogólnie może być tak, że `x == y`, ale `x is not y`. Na przykład:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>> a == b
True
>>>
```

W praktyce porównywanie obiektów za pomocą operatora `is` prawie nigdy nie jest tym, co chcesz uzyskać. Użyj operatora == dla wszystkich porównań, chyba że masz dobry powód, aby oczekiwać, że oba obiekty będą tożsame.

2.6. Operatory porównania porządkowego

Operatory porównania porządkowego z tabeli 2.3 mają standardową interpretację matematyczną dla liczb. Zwracają wartość logiczną.

Tabela 2.3. *Operatory porównania porządkowego*

Operacja	Opis
`x < y`	Mniejsze niż
`x > y`	Większe niż
`x >= y`	Większe lub równe
`x <= y`	Mniejsze lub równe

W przypadku zbiorów operacja `x < y` sprawdza, czy `x` jest ścisłym podzbiorem `y` (tzn. ma mniej elementów, ale nie jest równe `y`).

Podczas porównywania dwóch sekwencji porównywane są pierwsze elementy każdej z nich. Jeśli się różnią, to decyduje o wyniku całej operacji. Jeśli są takie same, porównanie przenosi się do drugiego elementu każdej sekwencji. Proces ten trwa, dopóki nie zostaną znalezione dwa różne elementy lub nie ma więcej elementów w żadnej z sekwencji. Jeżeli osiągnięto koniec obu sekwencji, sekwencje uważa się za równe. Jeśli `a` jest podciągiem `b`, to `a < b`.

Łańcuchy i bajty są porównywane za pomocą porządkowania leksykograficznego. Każdemu znakowi przypisany jest unikalny indeks numeryczny określony przez zbiór znaków (np. ASCII lub Unicode). Znak jest mniejszy niż inny znak, jeśli jego indeks jest mniejszy.

Nie wszystkie typy obsługują porównania porządkowe. Na przykład próba użycia < w słownikach jest niezdefiniowana i skutkuje błędem TypeError. Podobnie zastosowanie takich porównań do niezgodnych typów (takich jak ciąg i liczba) spowoduje wystąpienie TypeError.

2.7. Wyrażenia logiczne i wartości prawdziwe

Operatory and, or i not mogą tworzyć złożone wyrażenia logiczne. Zachowanie tych operatorów pokazano w tabeli 2.4.

Tabela 2.4. *Operatory logiczne*

Operator	Opis
x or y	Jeśli x ma wartość False, zwróć y; w przeciwnym razie zwróć x.
x and y	Jeśli x ma wartość False, zwróć x; w przeciwnym razie zwróć y.
not x	Jeśli x ma wartość False, zwróć True; w przeciwnym razie zwróć False.

Gdy używasz wyrażenia do określenia wartości True lub False, wartości True, dowolna niezerowa liczba, niepusty ciąg, lista, krotka lub słownik są uważane za wartość True. Wartości False, zero, None, puste listy, krotki i słowniki są oceniane jako wartość False.

Wyrażenia logiczne są liczone od lewej do prawej i wykorzystują prawy operand tylko wtedy, gdy jest to konieczne do określenia końcowej wartości. Na przykład a and b sprawdza wartość b tylko wtedy, gdy a ma wartość True. Ta technika nazywa się *skróconym wartościowaniem* (ang. *short-circuit evaluation*). Przydatne może być uproszczenie kodu obejmującego test i późniejszą operację. Na przykład:

```
if y != 0:
    result = x / y
else:
    result = 0

# Alternatywnie
result = y and x / y
```

W drugiej wersji podział x / y jest wykonywany tylko wtedy, gdy y jest niezerowe.

Poleganie na niejawnej „prawdziwości" obiektów może prowadzić do trudnych do znalezienia błędów. Rozważmy na przykład tę funkcję:

```
def f(x, items=None):
    if not items:
        items = []
    items.append(x)
    return items
```

Ta funkcja ma opcjonalny argument, który — jeśli nie zostanie podany — spowoduje utworzenie i zwrócenie nowej listy. Na przykład:

```
>>> foo(4)
[4]
>>>
```

Jednak funkcja zachowuje się naprawdę dziwnie, jeśli jako argument podasz jej istniejącą pustą listę:

```
>>> a = []
>>> foo(3, a)
[3]
>>> a  # Zwróć uwagę, że a NIE zostało zaktualizowane
[]
>>>
```

To błąd oparty na sprawdzaniu prawdziwości. Puste listy mają wartość `False`, więc kod utworzył nową listę, zamiast używać tej (`a`), która została przekazana jako argument. Aby to naprawić, musisz być bardziej precyzyjny w sprawdzaniu wartości `None`:

```
def f(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

Dobrą praktyką jest zawsze precyzyjne stosowanie sprawdzeń warunkowych.

2.8. Wyrażenia warunkowe

Typowym wzorcem programowania jest warunkowe przypisanie wartości na podstawie wyniku wyrażenia. Na przykład:

```
if a <= b:
    minvalue = a
else:
    minvalue = b
```

Ten kod można skrócić za pomocą wyrażenia warunkowego. Na przykład:

```
minvalue = a if a <= b else b
```

W takich wyrażeniach najpierw oceniany jest warunek znajdujący się w środku. Jeśli wynikiem jest `True`, brane jest pod uwagę wyrażenie po lewej stronie `if`. W przeciwnym razie liczone jest wyrażenie po `else`. Klauzula `else` jest zawsze wymagana.

2.9. Operacje obejmujące elementy iterowalne

Iteracja jest ważną funkcją Pythona obsługiwaną przez wszystkie kontenery Pythona (listy, krotki, słowniki itd.), pliki, a także funkcje generatora. Operacje z tabeli 2.5 można zastosować do dowolnych iterowalnych obiektów.

Najważniejszą operacją na obiekcie iterowalnym jest pętla `for`. W ten sposób przechodzisz przez kolejne wartości. Wszystkie inne operacje opierają się na tym działaniu.

Tabela 2.5. *Operacje na obiektach iteracyjnych*

Operacja	Opis
`for vars in s:`	Iteracja
`v1, v2, ... = s`	Rozpakowywanie zmiennych
`x in s, x not in s`	Członkostwo
`[a, *s, b], (a, *s, b), {a, *s, b}`	Rozszerzanie list, krotek lub zbiorów

Operator `x in s` sprawdza, czy obiekt `x` pojawia się jako jeden z elementów generowanych przez iterowalne `s`, i zwraca `True` lub `False`. Operator `x not in s` jest tym samym co `not (x in s)`. W przypadku ciągów operatory `in` i `not in` akceptują podciągi. Na przykład `'hello' in 'hello world'` daje wartość `True`. Należy zauważyć, że operator `in` nie obsługuje symboli wieloznacznych ani żadnego rodzaju dopasowywania wzorców.

Każdy obiekt obsługujący iterację może mieć swoje wartości rozpakowane w serii lokalizacji. Na przykład:

```
items = [3, 4, 5]
x, y, z = items       # x = 3, y = 4, z = 5

letters = "abc"
x, y, z = letters     # x = 'a', y = 'b', z = 'c'
```

Lokalizacje po lewej stronie nie muszą być prostymi nazwami zmiennych. Dopuszczalna jest każda prawidłowa lokalizacja, która może się pojawić po lewej stronie znaku równości. Możesz więc napisać kod w ten sposób:

```
items = [3, 4, 5]
d = { }
d['x'], d['y'], d['z'] = items
```

Podczas rozpakowywania wartości do lokalizacji liczba lokalizacji po lewej stronie musi dokładnie odpowiadać liczbie elementów w iterowalnym obiekcie po prawej stronie. W przypadku zagnieżdżonych struktur danych dopasuj lokalizacje i dane, stosując ten sam wzorzec strukturalny. Rozważmy następujący przykład rozpakowywania dwóch zagnieżdżonych krotek:

```
datetime = ((5, 19, 2008), (10, 30, "am"))
(month, day, year), (hour, minute, am_pm) = datetime
```

Czasami zmienna _ jest używana do wskazania wartości do pominięcia podczas rozpakowywania. Jeśli na przykład zależy Ci tylko na dniu i godzinie, możesz użyć:

```
(_, day, _), (hour, _, _) = datetime
```

Jeśli liczba elementów do rozpakowania nie jest znana, możesz użyć rozszerzonej formy rozpakowywania, dołączając zmienną oznaczoną gwiazdką, taką jak `*extra` w poniższym przykładzie:

```
items = [1, 2, 3, 4, 5]
a, b, *extra = items        # a = 1, b = 2, extra = [3,4,5]
*extra, a, b                # extra = [1,2,3], a = 4, b = 5
a, *extra, b                # a = 1, extra = [2,3,4], b = 5
```

W tym przykładzie *extra otrzymuje wszystkie dodatkowe elementy. To zawsze jest lista. Podczas rozpakowywania pojedynczego elementu iteracyjnego można użyć nie więcej niż jednej zmiennej oznaczonej gwiazdką. Jednak podczas rozpakowywania bardziej złożonych struktur danych obejmujących różne iterowalne obiekty można użyć wielu zmiennych oznaczonych gwiazdką. Na przykład:

```
datetime = ((5, 19, 2008), (10, 30, "am"))
(month, *_), (hour, *_) = datetime
```

Dowolny element iteracyjny można rozwinąć podczas tworzenia list, krotek i zbiorów. Odbywa się to również za pomocą gwiazdki (*). Na przykład:

```
items = [1, 2, 3]
a = [10, *items, 11]          # a = [10, 1, 2, 3, 11] (lista)
b = (*items, 10, *items)      # b = [1, 2, 3, 10, 1, 2, 3] (krotka)
c = {10, 11, *items}          # c = {1, 2, 3, 10, 11} (zbiór)
```

W tym przykładzie zawartość elementów jest po prostu wklejana do listy, krotki lub tworzonego zbioru, tak jakbyś wpisywał ją w tym miejscu. Ta ekspansja określana jest jako *splatting*. Podczas definiowania literału możesz uwzględnić dowolną liczbę rozszerzeń *. Jednak wiele obiektów iterowalnych (takich jak pliki lub generatory) obsługuje tylko jednorazową iterację. Jeśli użyjesz *, zawartość zostanie zużyta, a element iteracyjny nie będzie generował więcej wartości w kolejnych iteracjach.

Wiele funkcji wbudowanych akceptuje jako dane wejściowe dowolne iterowalne dane. Tabela 2.6 przedstawia niektóre z tych operacji.

Tabela 2.6. *Funkcje operujące na danych iterowalnych*

Funkcja	Opis
`list(s)`	Utwórz listę z s.
`tuple(s)`	Utwórz krotkę z s.
`set(s)`	Utwórz zbiór z s.
`min(s [, key])`	Minimalna pozycja w s.
`max(s [, key])`	Maksymalna pozycja w s.
`any(s)`	Zwraca `True`, jeśli jakikolwiek element w s ma wartość `True`.
`all(s)`	Zwraca `True`, jeśli wszystkie elementy w s mają wartość `True`.
`sum(s [, initial])`	Suma pozycji z opcjonalną wartością początkową.
`sorted(s [, key])`	Tworzy posortowaną listę.

Dotyczy to również wielu innych funkcji bibliotecznych — na przykład funkcji w module `statistics`.

2.10. Operacje na sekwencjach

Sekwencja to iterowalny kontener, który ma rozmiar i umożliwia dostęp do elementów za pomocą indeksu liczb całkowitych zaczynającego się od 0. Przykładami są ciągi, listy i krotki. Oprócz wszystkich operacji związanych z iteracją operatory z tabeli 2.7 można zastosować do sekwencji.

Tabela 2.7. *Operacje na sekwencjach*

Operacja	Opis
`s + r`	Złączenie.
`s * n, n * s`	Tworzy `n` kopii `s`, gdzie `n` jest liczbą całkowitą.
`s[i]`	Indeksowanie.
`s[i:j]`	Wycinanie.
`s[i:j:step]`	Zaawansowane wycinanie.
`len(s)`	Długość.

Operator + łączy dwie sekwencje tego samego typu. Na przykład:

```
>>> a = [3, 4, 5]
>>> b = [6, 7]
>>> a + b
[3, 4, 5, 6, 7]
>>>
```

Operator `s * n` tworzy *n* kopii sekwencji. Są to jednak płytkie kopie, które replikują elementy tylko przez odniesienie. Spójrz na poniższy kod:

```
>>> a = [3, 4, 5]
>>> b = [a]
>>> c = 4 * b
>>> c
[[3, 4, 5], [3, 4, 5], [3, 4, 5], [3, 4, 5]]
>>> a[0] = -7
>>> c
[[-7, 4, 5], [-7, 4, 5], [-7, 4, 5], [-7, 4, 5]]
>>>
```

Zwróć uwagę, jak zmiana elementu `a` modyfikuje każdy element listy `c`. W tym przypadku odwołanie do listy `a` zostało umieszczone w liście `b`. Kiedy `b` zostało zreplikowane, stworzono cztery dodatkowe odniesienia do `a`. Wreszcie, gdy zmodyfikowano `a`, ta zmiana została rozesłana do wszystkich pozostałych kopii `a`. Takie zachowanie mnożenia sekwencji często nie jest intencją programisty. Jednym ze sposobów obejścia tego problemu jest ręczne skonstruowanie zreplikowanej sekwencji poprzez zduplikowanie zawartości `a`. Oto przykład:

```
a = [3, 4, 5]
c = [list(a) for _ in range(4)] # list() tworzy kopię listy
```

Operator indeksowania `s[n]` zwraca *n*-ty obiekt z sekwencji; `s[0]` to pierwszy obiekt. Ujemne indeksy mogą służyć do pobierania znaków z końca sekwencji. Na przykład `s[-1]` zwraca ostatni element. W przeciwnym razie próby uzyskania dostępu do elementów, które są poza zakresem, powodują wystąpienie wyjątku `IndexError`.

Operator wycinania `s[i:j]` wyodrębnia podciąg z `s`, składający się z elementów o indeksie `k`, gdzie `i <= k < j`. Zarówno `i`, jak i `j` muszą być liczbami całkowitymi. Jeżeli indeks początkowy lub końcowy zostanie pominięty, zakłada się odpowiednio początek lub koniec sekwencji. Ujemne indeksy są dozwolone i zakłada się, że odnoszą się do końca ciągu.

Operatorowi wycinania można nadać opcjonalny krok *step*: `s[i:j:step]`, który powoduje, że podczas wycinania zostaną pominięte niektóre elementy. Jednak zachowanie jest nieco bardziej subtelne. Jeśli podano krok, `i` jest indeksem początkowym, `j` jest indeksem końcowym, a otrzymany podciąg to elementy `s[i]`, `s[i+step]`, `s[i+2*step]` i tak dalej, aż do osiągnięcia indeksu `j` (który nie jest uwzględniony). Krok może być również ujemny. Jeśli początkowy indeks `i` zostanie pominięty, to jest ustawiany na początku (jeśli krok jest dodatni) lub na końcu sekwencji (jeśli krok jest ujemny). Jeśli indeks końcowy `j` zostanie pominięty, ustawiany jest na koniec (jeśli krok jest dodatni) lub początek sekwencji (jeśli krok jest ujemny). Oto kilka przykładów:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

a[2:5]     # [2, 3, 4]
a[:3]      # [0, 1, 2]
a[-3:]     # [7, 8, 9]
a[::2]     # [0, 2, 4, 6, 8]
a[::-2]    # [9, 7, 5, 3, 1]
a[0:5:2]   # [0, 2, 4]
a[5:0:-2]  # [5, 3, 1]
a[:5:1]    # [0, 1, 2, 3, 4]
a[:5:-1]   # [9, 8, 7, 6]
a[5::1]    # [5, 6, 7, 8, 9]
a[5::-1]   # [5, 4, 3, 2, 1, 0]
a[5:0:-1]  # [5, 4, 3, 2, 1]
```

Skomplikowane wycinanie ciągu może skutkować kodem, który będzie później trudny do zrozumienia. W związku z tym zachowanie pewnego rozsądku jest uzasadnione. Wycinane podciągi można ponazywać za pomocą metody `slice()`. Na przykład:

```
firstfive = slice(0, 5)
s = 'hello world'
print(s[firstfive]) # Wyświetla 'hello'
```

2.11. Operacje na mutowalnych obiektach sekwencyjnych

Ciągi i krotki są niezmienne (ang. *immutable*) i nie można ich modyfikować po utworzeniu. Zawartość listy lub innej sekwencji mutowalnej można modyfikować w miejscu za pomocą operatorów z tabeli 2.8.

Operator `s[i] = x` zmienia element `i` sekwencji tak, aby odnosił się do obiektu `x`, zwiększając liczbę odwołań `x`. Indeksy ujemne są określane względem końca listy, a próba przypisania wartości do indeksu spoza zakresu powoduje wystąpienie wyjątku `IndexError`. Operator przypisania wycinka `s[i:j] = r` zastępuje elementy `k`, gdzie `i <= k < j`, elementami z sekwencji `r`. Indeksy mają takie samo znaczenie jak przy wycinaniu. W razie potrzeby sekwencja `s` może zostać rozszerzona lub zmniejszona, aby pomieścić wszystkie elementy w `r`. Oto przykład:

Tabela 2.8. *Operacje na mutowalnych obiektach sekwencyjnych*

Operacja	Opis
`s[i] = x`	Przypisanie indeksu
`s[i:j] = r`	Przypisanie wycinka
`s[i:j:step] = r`	Zaawansowane przypisanie wycinka
`del s[i]`	Usuwanie elementu
`del s[i:j]`	Usuwanie wycinka
`del s[i:j:step]`	Zaawansowane usuwanie wycinka

```
a = [1, 2, 3, 4, 5]
a[1] = 6               # a = [1, 6, 3, 4, 5]
a[2:4] = [10, 11]      # a = [1, 6, 10, 11, 5]
a[3:4] = [-1, -2, -3]  # a = [1, 6, 10, –1, –2, –3, 5]
a[2:] = [0]            # a = [1, 6, 0]
```

Przypisanie wycinka może być wykonane z opcjonalnym argumentem określającym krok. Zachowanie jest wtedy jednak bardziej ograniczone, ponieważ argument po prawej stronie musi mieć dokładnie taką samą liczbę elementów jak zastępowany wycinek. Oto przykład:

```
a = [1, 2, 3, 4, 5]
a[1::2] = [10, 11]        # a = [1, 10, 3, 11, 5]
a[1::2] = [30, 40, 50]    # ValueError. Tylko dwa elementy w wycinku po lewej stronie.
```

Operator `del s[i]` usuwa element `i` z sekwencji i zmniejsza jego liczbę odwołań. `del s[i:j]` usuwa wszystkie elementy z wycinka. Można również podać krok, jak w `del s[i:j:step]`.

Opisana tutaj semantyka dotyczy wbudowanego typu listy. Operacje obejmujące wycinanie sekwencji to bogaty obszar do dostosowywania w pakietach innych firm. Może się okazać, że operacje wycinania na obiektach spoza listy mają określone różne zasady dotyczące przypisywania, usuwania i współdzielenia obiektów. Na przykład popularny pakiet `numpy` ma inną semantykę wycinania niż listy Pythona.

2.12. Operacje na zbiorach

Zbiór to nieuporządkowana kolekcja unikalnych wartości. Na zbiorach można wykonywać operacje z tabeli 2.9.

Oto kilka przykładów:

```
>>> a = {'a', 'b', 'c'}
>>> b = {'c', 'd'}
>>> a | b
{'a', 'b', 'c', 'd'}
>>> a & b
>>> {'c'}
>>> a - b
{'a', 'b'}
>>> b - a
{'d'}
>>> a ^ b
{'a', 'b', 'd'}
>>>
```

Tabela 2.9. *Operacje na zbiorach*

Operacja	Opis
`s | t`	Złączenie `s` i `t`
`s & t`	Przecięcie `s` i `t`
`s – t`	Różnica zbioru (elementy w `s`, nie w `t`)
`s ^ t`	Różnica symetryczna (elementy niezawarte w `s` lub `t`)
`len(s)`	Liczba elementów w zbiorze
`item in s, item not in s`	Test członkostwa elementu
`s.add(item)`	Dodawanie elementu do zbioru `s`
`s.remove(item)`	Usuwanie elementu z `s`, jeśli istnieje (w przeciwnym razie błąd)
`s.discard(item)`	Odrzucanie elementu z `s`, jeśli istnieje

Operacje na zbiorach działają również na obiektach słownikowych kluczy i elementów. Aby na przykład dowiedzieć się, które klucze są wspólne dla dwóch słowników, wykonaj następujące czynności:

```
>>> a = {'x': 1, 'y': 2, 'z': 3}
>>> b = {'z': 3, 'w': 4, 'q': 5}
>>> a.keys() & b.keys()
{'z'}
>>>
```

2.13. Operacje na mapowaniach

Mapowanie to powiązanie kluczy i wartości. Przykładem jest wbudowany typ `dict`. Do mapowań można zastosować operacje z tabeli 2.10.

Tabela 2.10. *Operacje na mapowaniach*

Operacja	Opis
`x = m[k]`	Indeksowanie kluczem
`m[k] = x`	Przypisanie do klucza
`del m[k]`	Usuwanie elementu klucza
`k in m`	Testowanie członkostwa
`len(m)`	Liczba elementów w mapowaniu
`m.keys()`	Zwraca klucze
`m.values()`	Zwraca wartości
`m.items()`	Zwraca pary (`klucz, wartość`)

Wartości kluczy mogą być dowolnymi niezmiennymi obiektami, takimi jak ciągi, liczby i krotki. Używając krotki jako klucza, możesz pominąć nawiasy i wpisać wartości oddzielone przecinkami w następujący sposób:

```
d = { }
d[1,2,3] = "foo"
d[1,0,3] = "bar"
```

W takim przypadku wartości klucza reprezentują krotkę, dzięki czemu te przypisania są równoważne z następującymi:

```
d[(1,2,3)] = "foo"
d[(1,0,3)] = "bar"
```

Używanie krotki jako klucza jest powszechną techniką tworzenia kluczy złożonych w mapowaniu. Na przykład klucz może się składać z „imienia" i „nazwiska".

2.14. Lista, zbiór i słownik

Jedną z najczęstszych operacji na danych jest przekształcenie zbioru danych w inną strukturę danych. Na przykład tutaj bierzemy wszystkie elementy listy, stosujemy operację i tworzymy nową listę:

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    nums.append(n * n)
```

Ponieważ ten rodzaj operacji jest bardzo powszechny, jest dostępny jako operator zwany listą składaną. Oto bardziej kompaktowa wersja tego kodu:

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
```

Możliwe jest również zastosowanie filtra do operacji:

```
squares = [n * n for n in nums if n > 2] # [9, 16, 25]
```

Ogólna składnia list składanych jest następująca:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN]
```

Ta składnia jest równoważna z następującym kodem:

```
result = []
for item1 in iterable1:
    if condition1:
        for item2 in iterable2:
            if condition2:
                ...
                for itemN in iterableN:
                    if conditionN:
                        result.append(expression)
```

Listy składane są bardzo przydatnym sposobem przetwarzania danych list w różnych formach. Oto kilka praktycznych przykładów:

```
# Niektóre dane (lista słowników)
portfolio = [
  {'name': 'IBM', 'shares': 100, 'price': 91.1},
```

```
  {'name': 'MSFT', 'shares': 50, 'price': 45.67},
  {'name': 'HPE', 'shares': 75, 'price': 34.51},
  {'name': 'CAT', 'shares': 60, 'price': 67.89},
  {'name': 'IBM', 'shares': 200, 'price': 95.25}
]

# Zbierz wszystkie nazwy ['IBM', 'MSFT', 'HPE', 'CAT', 'IBM']
name = [s['name'] for s in portfolio]

# Znajdź wszystkie wpisy z ponad 100 udziałami ['IBM']
more100 = [s['name'] for s in portfolio if s['shares'] > 100]

# Znajdź łączną wartość udziały*cena
cost = sum([s['shares']*s['price'] for s in portfolio])

# Zbierz krotki (nazwisko, udziały)
name_shares = [(s['name'], s['shares']) for s in portfolio]
```

Wszystkie zmienne użyte wewnątrz listy składanej są prywatne dla tej listy. Nie musisz się martwić, że takie zmienne nadpiszą inne zmienne o tej samej nazwie. Na przykład:

```
>>> x = 42
>>> squares = [x*x for x in [1,2,3]]
>>> squares
[1, 4, 9]
>>> x
42
>>>
```

Zamiast tworzyć listę, możesz również utworzyć zbiór, zmieniając nawiasy w nawiasy klamrowe. Nazywa się to zbiorem składanym. Zbiór składany da Ci zestaw odrębnych wartości. Na przykład:

```
# Zbiór składany
names = {s['name'] for s in portfolio}
# names = {'IBM', 'MSFT', 'HPE', 'CAT'}
```

Jeśli określisz pary klucz-wartość, zamiast tego utworzysz słownik. Jest to słownik składany. Na przykład:

```
prices = {s['name']:s['price'] for s in portfolio}
# prices = {'IBM': 95.25, 'MSFT': 45.67, 'HPE': 34.51, 'CAT': 67.89}
```

Tworząc zbiory i słowniki, należy pamiętać, że późniejsze wpisy mogą nadpisać wcześniejsze. Na przykład w słowniku `prices` otrzymujesz ostatnią cenę za `'IBM'`. Pierwsza cena przepada.

W ramach składania nie można obsłużyć wyjątków. Jeśli jest to problem, rozważ opakowanie obsługi wyjątków za pomocą funkcji, jak pokazano tutaj:

```
def toint(x):
    try:
        return int(x)
    except ValueError:
        return None

values = ['1', '2', '-4', 'n/a', '-3', '5']
```

```
data1 = [toint(x) for x in values]
# data1 = [1, 2, –4, None, –3, 5]

data2 = [toint(x) for x in values if toint(x) is not None]
# data2 = [1, 2, –4, –3, 5]
```

W ostatnim przykładzie można uniknąć podwójnego liczenia `toint(x)`, używając operatora `:=`. Na przykład:

```
data3 = [v for x in values if (v:=toint(x)) is not None]
# data3 = [1, 2, –4, –3, 5]
data4 = [v for x in values if (v:=toint(x)) is not None and v >= 0]
# data4 = [1, 2, 5]
```

2.15. Wyrażenia generujące

Wyrażenie generujące to obiekt, który wykonuje te same obliczenia co lista składana, ale iteracyjnie generuje wynik. Składnia jest taka sama jak w przypadku list składanych, z tym wyjątkiem, że zamiast nawiasów kwadratowych używa się nawiasów okrągłych. Oto przykład:

```
nums = [1,2,3,4]
squares = (x*x for x in nums)
```

W przeciwieństwie do list składanych wyrażenie generujące w rzeczywistości nie tworzy listy ani natychmiast nie liczy wyrażenia w nawiasach. Zamiast tego tworzy obiekt generatora, który generuje wartości na żądanie za pomocą iteracji. Jeśli spojrzysz na wynik powyższego przykładu, zobaczysz, co następuje:

```
>>> squares
<generator object at 0x590a8>
>>> next(squares)
1
>>> next(squares)
4
...
>>> for n in squares:
... print(n)
9
16
>>>
```

Wyrażenia generującego można użyć tylko raz. Jeśli spróbujesz powtórzyć iterację po raz drugi, nic nie otrzymasz:

```
>>> for n in squares:
... print(n)
...
>>>
```

Różnica między listą składaną a wyrażeniem generującym jest ważna, ale subtelna. Dzięki listom składanym Python faktycznie tworzy listę, która zawiera dane wynikowe. Za pomocą wyrażenia generującego Python tworzy generator, który jedynie wie, jak generować dane na żądanie. W niektórych aplikacjach może to znacznie poprawić wydajność i wykorzystanie pamięci. Oto przykład:

```
# Przeczytaj plik
f = open('data.txt')                        # Otwórz plik
lines = (t.strip() for t in f)              # Czytaj linie, usuń końcowe/początkowe białe znaki
comments = (t for t in lines if t[0] == '#') # Wszystkie komentarze
for c in comments:
    print(c)
```

W tym przykładzie wyrażenie generujące, które wyodrębnia wiersze z pliku i usuwa białe znaki, w rzeczywistości nie odczytuje i nie przechowuje całego pliku w pamięci. To samo dotyczy wyrażenia, które wyodrębnia komentarze. Zamiast tego wiersze pliku są odczytywane jeden po drugim, gdy program rozpoczyna iterację w pętli `for`. Podczas tej iteracji wiersze pliku są tworzone na żądanie i odpowiednio filtrowane. W rzeczywistości cały plik nie zostanie załadowany do pamięci podczas tego procesu. Jest to zatem bardzo wydajny sposób na odczytanie komentarzy z olbrzymiego pliku źródłowego Pythona.

W przeciwieństwie do list składanych wyrażenie generujące nie tworzy obiektu, który działa jak sekwencja. Nie może być indeksowany i żadna ze zwykłych operacji na listach (takich jak `append()`) nie zadziała. Jednak elementy tworzone przez wyrażenie generujące można przekonwertować na listę za pomocą `list()`:

```
clist = list(comments)
```

W przypadku przekazania pojedynczego argumentu funkcji jeden zestaw nawiasów może zostać usunięty. Na przykład następujące wyrażenia są równoważne:

```
sum((x*x for x in values))
sum(x*x for x in values)      # Usunięto dodatkowe nawiasy
```

W obu przypadkach tworzony jest generator (`x*x for x in values`), który jest przekazywany do funkcji `sum()`.

2.16. Operator atrybutu (.)

Operator kropki (`.`) służy do uzyskiwania dostępu do atrybutów obiektu. Oto przykład:

```
foo.x = 3
print(foo.y)
a = foo.bar(3,4,5)
```

W jednym wyrażeniu może się pojawić więcej niż jeden operator kropki, na przykład `foo.y.a.b`. Operator kropki można również zastosować do pośrednich wyników funkcji, na przykład `a = foo.bar(3,4,5).spam`. Takie długie łańcuchy pobrań atrybutów nie są jednak popularne.

2.17. Operator wywołania funkcji ()

Operator `f(args)` służy do wywołania funkcji `f`. Każdy argument funkcji jest wyrażeniem. Przed wywołaniem funkcji wszystkie wyrażenia argumentów są liczone od lewej do prawej. Określa się to jako aplikacyjny porządek liczenia. Więcej informacji o funkcjach można znaleźć w rozdziale 5.

2.18. Kolejność liczenia

Tabela 2.11 przedstawia kolejność działania (reguły pierwszeństwa) dla operatorów Pythona. Wszystkie operatory z wyjątkiem operatora potęgi (**) są liczone od lewej do prawej i są wymienione w tabeli od najwyższego do najniższego priorytetu. Oznacza to, że operatory wymienione jako pierwsze w tabeli są liczone przed operatorami wymienionymi później. Operatory zawarte razem w podsekcjach, takie jak `x * y`, `x / y`, `x // y`, `x @ y` i `x % y`, mają równy priorytet.

Kolejność liczenia z tabeli 2.11 nie zależy od typów `x` i `y`. Tak więc mimo że obiekty zdefiniowane przez użytkownika mogą przedefiniować poszczególne operatory, dostosowanie podstawowej kolejności liczenia, pierwszeństwa i reguł asocjacji nie jest możliwe.

Tabela 2.11. *Kolejność liczenia (od najwyższego priorytetu do najniższego)*

Operator	Nazwa
`(...), [...], {...}`	Tworzenie krotek, list i słowników
`s[i], s[i:j]`	Indeksowanie i wycinanie
`s.attr`	Wyszukiwanie atrybutów
`f(...)`	Wywołania funkcji
`+x, -x, ~x`	Operatory jednoargumentowe
`x ** y`	Potęga (liczenie od prawej strony)
`x * y, x / y, x // y, x % y, x @ y`	Mnożenie, dzielenie, dzielenie całkowite, modulo, mnożenie macierzy
`x + y, x - y`	Dodawanie, odejmowanie
`x << y, x >> y`	Przesunięcie bitowe
`x & y`	Bitowe i
`x ^ y`	Bitowa różnica symetryczna
`x | y`	Bitowe lub
`x < y, x <= y, x > y, x >= y, x == y, x !=y, x is y, x is not y, x in y, x not in y`	Porównanie, tożsamość, testy członkostwa
`not x`	Logiczna negacja
`x and y`	Logiczne i
`x or y`	Logiczne lub
`lambda args: expr`	Funkcja anonimowa
`expr if expr else expr`	Wyrażenie warunkowe
`x := expr`	Wyrażenie przypisania

Częstą pomyłką dotyczącą reguł pierwszeństwa jest sytuacja, gdy operatory bitowe i (`&`) i bitowe lub (`|`) są używane jak operatory logiczne `and` i `or`. Na przykład:

```
>>> a = 10
>>> a <= 10 and 1 < a
True
>>> a <= 10 & 1 < a
False
>>>
```

To ostatnie wyrażenie jest liczone jako `a <= (10 & 1) < a` lub `a <= 0 < a`. Możesz to naprawić, dodając nawiasy:

```
>>> (a <= 10) & (1 < a)
True
>>>
```

Takie operacje mogą się wydawać dziwnym przypadkiem brzegowym, ale pojawiają się z pewną częstotliwością w pakietach zorientowanych na dane, takich jak `numpy` i `pandas`. Operatory logiczne `and` lub `or` nie mogą być przerabiane do własnych potrzeb, więc zamiast nich używane są operatory bitowe — nawet jeśli mają wyższy poziom pierwszeństwa i dają inny wynik, gdy są wykorzystywane w relacjach logicznych.

2.19. Podsumowanie: sekretne życie danych

Jednym z najczęstszych zastosowań Pythona jest używanie go w aplikacjach związanych z manipulacją i analizą danych. W tych dziedzinach Python zapewnia rodzaj „języka dedykowanego" do rozwiązywania Twoich problemów. Wbudowane operatory i wyrażenia są rdzeniem tego języka i cała reszta jest „budowana" wokół nich. Tak więc kiedy kiedy już zdobędziesz orientację we wbudowanych obiektach i operacjach Pythona, przekonasz się, że możesz je stosować wszędzie.

Załóżmy na przykład, że pracujesz z bazą danych i chcesz iterować po rekordach zwróconych przez zapytanie. Są szanse, że użyjesz do tego instrukcji `for`. Lub załóżmy, że pracujesz z tablicami liczbowymi i chcesz wykonać operacje na tablicach, element po elemencie. Możesz pomyśleć, że standardowe operatory matematyczne zadziałają — Twoje założenia byłyby właściwe. Albo załóżmy, że używasz biblioteki do pobierania danych przez HTTP i chcesz uzyskać dostęp do zawartości nagłówków HTTP. Istnieje duża szansa, że dane będą prezentowane w sposób przypominający słownik.

Więcej informacji o wewnętrznych protokołach Pythona i sposobach ich dostosowywania znajduje się w rozdziale 4.

3

Struktura i kontrola przepływu programu

Ten rozdział zawiera szczegóły dotyczące struktury i kontroli przepływu programu. Tematy obejmują warunki, pętle, wyjątki i menedżery kontekstu.

3.1. Struktura i wykonanie programu

Programy Pythona mają strukturę sekwencji instrukcji. Wszystkie cechy języka, w tym przypisanie zmiennych, wyrażenia, definicje funkcji, klasy i importy modułów, są instrukcjami, które mają taki sam status jak wszystkie inne instrukcje — co oznacza, że każda instrukcja może być umieszczona prawie w dowolnym miejscu w programie (chociaż niektóre instrukcje, takie jak `return`, mogą się pojawić tylko wewnątrz funkcji). Na przykład ten kod definiuje dwie różne wersje funkcji wewnątrz warunku:

```
if debug:
    def square(x):
        if not isinstance(x,float):
            raise TypeError('Spodziewany typ float')
        return x * x
else:
    def square(x):
        return x * x
```

Podczas ładowania plików źródłowych interpreter wykonuje instrukcje w kolejności, w jakiej się pojawiają, dopóki nie będzie więcej instrukcji do wykonania. Ten model wykonywania dotyczy zarówno plików uruchamianych jako program główny, jak i plików bibliotek ładowanych przez `import`.

3.2. Wykonanie warunkowe

Instrukcje `if`, `else` i `elif` kontrolują warunkowe wykonanie kodu. Ogólny format instrukcji warunkowej to:

```
if expression:
    instrukcje
elif expression:
    instrukcje
elif expression:
    instrukcje
...
else:
    instrukcje
```

Jeśli nie ma być podjęte żadne działanie, możesz pominąć klauzule `else` i `elif` warunku. Użyj instrukcji `pass`, jeśli nie istnieją żadne instrukcje dla określonej klauzuli:

```
if expression:
    pass  # To do: zaimplementuj
else:
    instrukcje
```

3.3. Pętle i iteracje

Pętle implementujesz za pomocą instrukcji `for` i `while`. Oto przykład:

```
while expression:
    instrukcje

for i in s:
    instrukcje
```

Instrukcja `while` wykonuje instrukcje, dopóki skojarzone wyrażenie nie zostanie ocenione jako `False`. Instrukcja `for` iteruje po wszystkich elementach `s` do czasu, aż nie będzie już więcej dostępnych elementów. Instrukcja `for` działa z dowolnym obiektem obsługującym iterację. Obejmuje to wbudowane typy sekwencji, takie jak listy, krotki i ciągi, ale także dowolny obiekt, który implementuje protokół iteratora.

W instrukcji `for i in s` zmienna `i` jest nazywana zmienną iteracyjną. W każdej iteracji pętli otrzymuje nową wartość od `s`. Zakres zmiennej iteracji nie jest prywatny dla instrukcji `for`. Jeśli wcześniej zdefiniowana zmienna ma taką samą nazwę, to wartość zostanie nadpisana. Co więcej, zmienna iteracji zachowuje ostatnią wartość po zakończeniu pętli.

Jeśli elementy utworzone przez iterację są iteracjami o identycznym rozmiarze, możesz rozpakować ich wartości do oddzielnych zmiennych iteracyjnych za pomocą instrukcji takiej jak ta:

```
s = [(1, 2, 3), (4, 5, 6)]
for x, y, z in s:
    instrukcje
```

W tym przykładzie s musi być obiektem iterowalnym zawierającym trzy elementy. W każdej iteracji zawartości zmiennych x, y i z są przypisywane elementy odpowiadające elementom iterowalnym. Chociaż najczęściej używa się tej operacji, gdy s jest sekwencją krotek, rozpakowywanie działa, gdy elementy w s są dowolnymi rodzajami iteracji, w tym listami, generatorami i łańcuchami.

Czasami podczas rozpakowywania używana jest zmienna jednorazowa, taka jak _. Na przykład:

```
for x, _, z in s:
    instrukcje
```

W tym przykładzie wartość nadal jest umieszczana w zmiennej _, ale nazwa zmiennej sugeruje, że nie jest ona interesująca ani przydatna w dalszej części kodu (`statements`).

Jeśli elementy wytwarzane przez element iteracyjny mają różne rozmiary, możesz w trakcie rozpakowywania skorzystać z symboli wieloznacznych, aby umieścić wiele wartości w zmiennej. Na przykład:

```
s = [(1, 2), (3, 4, 5), (6, 7, 8, 9)]

for x, y, *extra in s:
    instrukcje          # x = 1, y = 2, extra = []
                        # x = 3, y = 4, extra = [5]
                        # x = 6, y = 7, extra = [8, 9]
                        # ...
```

W tym przykładzie wymagane są co najmniej dwie wartości x i y, ale `*extra` otrzymuje wszelkie dodatkowe wartości, które również mogą być obecne. Te wartości są zawsze umieszczane na liście. W jednym rozpakowaniu może się pojawić co najwyżej jedna zmienna oznaczona gwiazdką, ale za to może się ona znajdować w dowolnej pozycji. Oba te warianty są zatem poprawne:

```
for *first, x, y in s:
...
for x, *middle, y in s:
...
```

Podczas wykonywania pętli czasami przydatne jest śledzenie indeksu liczbowego oprócz wartości danych. Oto przykład:

```
i = 0
for x in s:
    instrukcje
    i += 1
```

Python zapewnia wbudowaną funkcję `enumerate()`, której można użyć do uproszczenia tego kodu:

```
for i, x in enumerate(s):
    instrukcje
```

`enumerate(s)` tworzy iterator, który generuje krotki `(0, s[0])`, `(1, s[1])`, `(2, s[2])` i tak dalej. Inną wartość początkową dla licznika można podać z argumentem słowa kluczowego `start` dla `enumerate()`:

```
for i, x in enumerate(s, start=100):
    instrukcje
```

W takim przypadku zostaną utworzone krotki w postaci `(100, s[0])`, `(101, s[1])` itd.

Innym powszechnym problemem związanym z pętlami jest iteracja równoległa na dwóch lub większej liczbie iterowalnych elementów, na przykład pisanie pętli, w której w każdej iteracji bierzesz elementy z różnych sekwencji:

```
# s i t to dwie sekwencje
i = 0
while i < len(s) and i < len(t):
    x = s[i] # Bierze element z s
    y = t[i] # Bierze element z t
    instrukcje
    i += 1
```

Ten kod można uprościć za pomocą funkcji `zip()`. Na przykład:

```
# s i t to dwie sekwencje
for x, y in zip(s, t):
    instrukcje
```

`zip(s, t)` łączy iterowalne `s` i `t` w iterowalne krotki `(s[0], t[0])`, `(s[1], t[1])`, `(s[2], t[2])` i tak dalej, zatrzymując się na najkrótszej z `s` lub `t`, jeśli mają nierówną długość. Wynikiem funkcji `zip()` jest iterator, który podczas iteracji generuje wyniki. Jeśli chcesz przekonwertować wynik na listę, użyj `list(zip(s, t))`.

Aby przerwać pętlę, użyj instrukcji `break`. Poniższy kod odczytuje wiersze tekstu z pliku do czasu, aż napotka pusty wiersz tekstu:

```
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            break # Pusta linia, przestań czytać
        # Przetwarzanie wyciętej linii
         ...
```

Aby przejść do następnej iteracji pętli (pomijając resztę ciała pętli), użyj instrukcji `continue`. Ta operacja jest przydatna, w przypadku gdy przejście do kolejnego poziomu pętli spowodowałoby, że program byłby zbyt głęboko zagnieżdżony lub niepotrzebnie skomplikowany.

W poniższym przykładzie pętla pomija wszystkie puste wiersze w pliku:

```
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            continue # Pomiń pustą linię
        # Przetwarzanie wyciętej linii
         ...
```

Instrukcje `break` i `continue` dotyczą tylko najbardziej wewnętrznej pętli. Jeśli chcesz przerwać wykonywanie programu z głęboko zagnieżdżonej struktury pętli, możesz użyć wyjątku. Python nie zapewnia instrukcji `goto`. Możesz również dołączyć instrukcję `else` do konstrukcji pętli, jak w poniższym przykładzie:

```
# for-else
with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
           break
        # Przetwarzanie wyciętej linii
        ...
    else:
        raise RuntimeError('Brak separatora sekcji')
```

Klauzula `else` pętli jest wykonywana tylko wtedy, gdy pętla wykona się do końca. Dzieje się to albo natychmiast (jeśli pętla w ogóle się nie wykona), albo po ostatniej iteracji. Jeśli pętla zostanie wcześniej zakończona za pomocą instrukcji `break`, klauzula `else` jest pomijana.

Podstawowym przypadkiem użycia klauzuli pętli `else` jest kod, który iteruje po danych, ale musi ustawić lub sprawdzić jakiś rodzaj flagi bądź warunku, jeśli pętla zostanie przedwcześnie zerwana. Jeśli na przykład nie użyłeś instrukcji `else`, poprzedni kod musi skorzystać z dodatkowej flagi:

```
found_separator = False

with open('foo.txt') as file:
    for line in file:
        stripped = line.strip()
        if not stripped:
            found_separator = True
            break
        # Przetwarzanie wyciętej linii
        ...
    if not found_separator:
        raise RuntimeError('Brak separatora sekcji')
```

3.4. Wyjątki

Wyjątki wskazują błędy i wyłamują się z normalnego przepływu programu. Wyjątek jest zgłaszany za pomocą instrukcji `raise`. Ogólny format instrukcji `raise` to `raise Exception([value])`, gdzie `Exception` to typ wyjątku, a `value` to argument opcjonalny, podający szczegółowe informacje na temat wyjątku. Oto przykład:

```
raise RuntimeError('Błąd nie do naprawienia')
```

Aby przechwycić wyjątek, użyj instrukcji `try` i `except`, jak pokazano tutaj:

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError as e:
    instrukcje
```

Gdy wystąpi wyjątek, interpreter zatrzymuje wykonywanie instrukcji w bloku `try` i szuka klauzuli `except` pasującej do typu wyjątku, który wystąpił. Jeśli zostanie znaleziony, kontrola jest przekazywana do pierwszej instrukcji w klauzuli `except`. Po wykonaniu klauzuli `except` program kontynuuje wykonywanie pierwszej instrukcji, która pojawia się po całym bloku `try-except`.

Nie jest konieczne, aby instrukcja `try` pasowała do wszystkich możliwych wyjątków, które mogą wystąpić. Jeśli nie można znaleźć pasującej klauzuli `except`, wyjątek jest nadal propagowany i może zostać przechwycony w innym bloku `try-except`, który faktycznie może obsłużyć wyjątek w innym miejscu. Jeśli chodzi o styl programowania, powinieneś przechwytywać tylko te wyjątki, które Twój kod może poprawnie obsłużyć — bez szkody dla działania programu. W przeciwnym wypadku często lepiej pozwolić na propagację wyjątku.

Jeśli wyjątek dociera do najwyższego poziomu programu bez przechwycenia, interpreter przerywa działanie z komunikatem o błędzie.

Jeśli instrukcja `raise` jest używana samodzielnie, ostatni wygenerowany wyjątek jest zgłaszany ponownie. Działa to tylko podczas obsługi wcześniej zgłoszonego wyjątku. Na przykład:

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError:
    print("Cóż, to nie zadziałało.")
    raise # Ponownie zgłasza bieżący wyjątek
```

Każda klauzula `except` może być używana z modyfikatorem `as var` podającym nazwę zmiennej, w której umieszczane jest wystąpienie typu wyjątku, jeśli ten wystąpi. Kod obsługi wyjątków może sprawdzić tę wartość, aby dowiedzieć się więcej o przyczynie wyjątku. Na przykład można użyć instrukcji `isinstance()` do sprawdzenia typu wyjątku.

Wyjątki mają kilka standardowych atrybutów, przydatnych w kodzie, który wymaga wykonania dalszych działań w odpowiedzi na błąd:

`e.args`

: Krotka argumentów otrzymywana podczas zgłaszania wyjątku. W większości przypadków jest to krotka jednoelementowa z łańcuchem opisującym błąd. W przypadku wyjątków `OSError` wartość jest krotką dwu- lub trzyelementową, zawierającą całkowity numer błędu, komunikat o błędzie i opcjonalną nazwę pliku.

`e.__cause__`

: Poprzedni wyjątek, jeśli wyjątek został celowo zgłoszony w odpowiedzi na obsługę innego wyjątku. Zobacz późniejszą sekcję o powiązanych wyjątkach.

`e.__context__`

: Poprzedni wyjątek, jeśli został on zgłoszony podczas obsługi innego wyjątku.

`e.__traceback__`

: Obiekt śledzenia stosu skojarzony z wyjątkiem.

Zmienna używana do przechowywania wartości wyjątku jest dostępna tylko wewnątrz powiązanego bloku `except`. Gdy program opuszcza blok, zmienna staje się niezdefiniowana. Na przykład:

```
try:
int('N/A') # Zgłasza wyjątek ValueError
except ValueError as e:
print('Niepowodzenie:', e)

print(e) # Niepowodzenie -> NameError. 'e' nie jest zdefiniowane.
```

Wiele bloków obsługi wyjątków można określić za pomocą wielu klauzul `except`:

```
try:
    jakieś operacje
except TypeError as e:
    # Obsługa błędu
    ...
except ValueError as e:
    # Obsługa błędu
    ...
```

Pojedyncza klauzula obsługi wyjątku może przechwycić wiele typów wyjątków, na przykład:

```
try:
    jakieś operacje
except (TypeError, ValueError) as e:
    # Obsługa jednego lub drugiego błędu
    ...
```

Aby zignorować wyjątek, użyj instrukcji `pass` w następujący sposób:

```
try:
    jakieś operacje
except ValueError:
    pass # Nic nie rób
```

Ciche ignorowanie błędów jest często niebezpieczne i jest źródłem trudnych do znalezienia błędów. Nawet jeśli błąd zostanie zignorowany, często rozsądnie jest go zapisać w dzienniku zdarzeń lub w innym miejscu, w którym można go później sprawdzić.

Aby przechwycić wszystkie wyjątki poza tymi związanymi z zakończeniem programu, użyj `Exception` w następujący sposób:

```
try:
    instrukcje
except Exception as e:
    print(f'Wystąpił błąd: {e!r}')
```

Podczas przechwytywania wszystkich wyjątków należy bardzo uważać, aby przekazać użytkownikowi dokładne informacje o błędach. Na przykład w poprzednim kodzie drukowany jest komunikat o błędzie i powiązana z nim wartość wyjątku. Jeśli nie podasz żadnych informacji o wartości wyjątku, może to bardzo utrudnić debugowanie kodu, który kończy się niepowodzeniem z nieoczekiwanych powodów.

Instrukcja `try` obsługuje również klauzulę `else`, która musi następować po ostatniej klauzuli `except`. Ten kod jest wykonywany, jeśli kod w bloku `try` nie zgłasza wyjątku. Oto przykład:

```
try:
    file = open('foo.txt', 'rt')
except FileNotFoundError as e:
    print(f'Nie mogę otworzyć pliku foo : {e}')
    data = ''
else:
    data = file.read()
    file.close()
```

Instrukcja `finally` definiuje akcję czyszczenia, która musi zostać wykonana niezależnie od tego, co dzieje się w bloku `try-except`. Oto przykład:

```
file = open('foo.txt', 'rt')
try:
    # Jakieś operacje
    ...
finally:
    file.close()
    # Plik zamknięty niezależnie od tego, co się stało
```

Klauzula `finally` nie służy do wyłapywania błędów. Jest raczej używana w kodzie, który musi być zawsze wykonywany, niezależnie od tego, czy wystąpi błąd. Jeśli nie zostanie zgłoszony żaden wyjątek, kod w klauzuli `finally` jest wykonywany bezpośrednio po kodzie w bloku `try`. Jeśli wyjątek wystąpi, pasujący blok `except` (jeśli istnieje) jest wykonywany jako pierwszy, a następnie kontrola jest przekazywana do pierwszej instrukcji klauzuli `finally`. Jeśli po wykonaniu tego kodu wyjątek nadal oczekuje, jest on ponownie zgłaszany, aby został przechwycony przez inny kod obsługi wyjątków.

3.4.1. Hierarchia wyjątków

Jednym z wyzwań związanych z pracą z wyjątkami jest zarządzanie ogromną liczbą wyjątków, które mogą potencjalnie wystąpić w programie. Na przykład istnieje ponad 60 wbudowanych wyjątków. Uwzględniając resztę biblioteki standardowej, można się doszukać setek możliwych wyjątków. Co więcej, często nie ma możliwości określenia z góry, jakie wyjątki może wywołać jakakolwiek część kodu. Wyjątki nie są rejestrowane jako część sygnatury wywołania funkcji ani nie istnieje żaden rodzaj kompilatora, który weryfikuje poprawność obsługi wyjątków w kodzie. W rezultacie obsługa wyjątków może się czasami wydawać przypadkowa i niezorganizowana.

Pomaga to w uświadomieniu sobie, że wyjątki są zorganizowane w hierarchię poprzez dziedziczenie. Zamiast skupiać się na konkretnych błędach, łatwiej jest skoncentrować się na bardziej ogólnych kategoriach błędów. Rozważmy na przykład różne błędy, które mogą się pojawić podczas wyszukiwania wartości w kontenerze:

```
try:
    item = items[index]
except IndexError: # Wyjątek zgłaszany, jeśli elementy są sekwencją
    ...
except KeyError:   # Wyjątek zgłaszany, jeśli elementy są mapowaniem
    ...
```

Zamiast pisać kod do obsługi dwóch bardzo specyficznych wyjątków, łatwiej będzie to zrobić tak:

```
try:
    item = items[index]
except LookupError:
    ...
```

`LookupError` to klasa, która reprezentuje grupę wyjątków wyższego poziomu. `IndexError` i `KeyError` dziedziczą po `LookupError`, więc klauzula `except` przechwyci jedną z nich. Jednak `LookupError` nie obejmuje błędów niezwiązanych z wyszukiwaniem.

Tabela 3.1 opisuje najczęstsze kategorie wbudowanych wyjątków.

Tabela 3.1. *Kategorie wyjątków*

Klasa wyjątku	Opis
`BaseException`	Klasa główna dla wszystkich wyjątków
`Exception`	Klasa bazowa dla wszystkich błędów związanych z programem
`ArithmeticError`	Klasa bazowa dla wszystkich błędów matematycznych
`ImportError`	Klasa bazowa dla błędów związanych z importem
`LookupError`	Klasa bazowa dla wszystkich błędów wyszukiwania kontenera
`OSError`	Klasa bazowa dla wszystkich błędów systemowych. `IOError` i `EnvironmentError` to aliasy
`ValueError`	Klasa bazowa dla błędów związanych z wartością, w tym Unicode
`UnicodeError`	Klasa bazowa dla błędów związanych z kodowaniem łańcucha Unicode

Klasa `BaseException` jest rzadko używana bezpośrednio w obsłudze wyjątków, ponieważ pasuje do wszystkich możliwych wyjątków. Obejmuje ona specjalne wyjątki, które wpływają na kontrolę sterowania, takie jak `SystemExit`, `KeyboardInterrupt` i `StopIteration`. Przechwytywanie ich rzadko jest tym, czego chcesz. Zamiast tego wszystkie normalne błędy związane z programem dziedziczą po `Exception`. `ArithmeticError` jest podstawą wszystkich błędów matematycznych, takich jak `ZeroDivisionError`, `FloatingPointError` i `OverflowError`. `ImportError` jest bazą dla wszystkich błędów związanych z importem. `LookupError` jest bazą dla wszystkich błędów związanych z wyszukiwaniem kontenerów. `OSError` jest klasą bazową dla wszystkich błędów pochodzących z systemu operacyjnego i środowiska. `OSError` obejmuje szeroki zakres wyjątków związanych z plikami, połączeniami sieciowymi, uprawnieniami, potokami, limitami czasu i nie tylko. Wyjątek `ValueError` jest często zgłaszany, gdy do operacji zostanie podana zła wartość wejściowa. `UnicodeError` to podklasa `ValueError` grupująca wszystkie błędy kodowania i dekodowania związane z Unicode.

Tabela 3.2 przedstawia niektóre typowe wbudowane wyjątki, które dziedziczą bezpośrednio po `Exception`, ale nie są częścią większej grupy wyjątków.

Tabela 3.2. *Inne wbudowane wyjątki*

Klasa wyjątku	Opis
`AssertionError`	Nieudane użycie instrukcji `assert`
`AttributeError`	Złe wyszukiwanie atrybutów w obiekcie
`EOFError`	Koniec pliku
`MemoryError`	Naprawialny błąd braku pamięci
`NameError`	Nie znaleziono nazwy w lokalnej lub globalnej przestrzeni nazw
`NotImplementedError`	Niezaimplementowana funkcjonalność
`RuntimeError`	Ogólny błąd świadczący o tym, że stało się coś złego
`TypeError`	Operacja zastosowana do obiektu niewłaściwego typu
`UnboundLocalError`	Użycie zmiennej lokalnej przed przypisaniem wartości

3.4.2. Wyjątki i kontrola przepływu

Zwykle wyjątki są zarezerwowane do obsługi błędów. Kilka wyjątków jest jednak używanych do zmiany kontroli przepływu programu. Te wyjątki, pokazane w tabeli 3.3, dziedziczą bezpośrednio po `BaseException`.

Tabela 3.3. *Wyjątki stosowane w celu kontroli przepływu programu*

Klasa wyjątku	Opis
`SystemExit`	Zgłaszany, aby wskazać zakończenie programu
`KeyboardInterrupt`	Zgłaszany, gdy program zostanie przerwany przez *Ctrl+C*
`StopIteration`	Zgłaszany, aby zasygnalizować koniec iteracji

Wyjątek `SystemExit` służy do celowego zakończenia programu. Jako argument możesz podać całkowity kod wyjścia lub wiadomość. Jeśli podano łańcuch, jest on wypisywany do `sys.stderr`, a program kończy się kodem błędu równym 1.

Oto typowy przykład:

```
import sys

if len(sys.argv) != 2:
    raise SystemExit(f'Przykład użycia: {sys.argv[0]} nazwa_pliku')

filename = sys.argv[1]
```

Wyjątek `KeyboardInterrupt` jest zgłaszany, gdy program odbiera sygnał `SIGINT` (zazwyczaj przez naciśnięcie *Ctrl+C* w terminalu). Ten wyjątek jest nieco nietypowy, ponieważ jest asynchroniczny — co oznacza, że może wystąpić w dowolnym momencie i w dowolnej instrukcji w programie. Gdy to się stanie, domyślnym zachowaniem Pythona jest po prostu zakończenie działania programu. Jeśli chcesz sterować dostarczaniem sygnału `SIGINT`, możesz użyć modułu biblioteki sygnałów (patrz rozdział 9.).

Wyjątek `StopIteration` jest częścią protokołu iteracji i sygnalizuje jej koniec.

3.4.3. Definiowanie nowych wyjątków

Wszystkie wbudowane wyjątki są zdefiniowane w kategoriach klas. Aby utworzyć nowy wyjątek, zdefiniuj nową klasę, która dziedziczy po wyjątku, na przykład:

```
class NetworkError(Exception):
    pass
```

Aby użyć nowego wyjątku, skorzystaj z instrukcji `raise` w następujący sposób:

```
raise NetworkError('Nie można znaleźć hosta')
```

Podczas zgłaszania wyjątku opcjonalne wartości podane w instrukcji `raise` są używane jako argumenty konstruktora klasy wyjątku. W większości przypadków jest to ciąg zawierający jakiś komunikat o błędzie. Wyjątki zdefiniowane przez użytkownika można jednak zapisać tak, aby przyjmowały jedną lub więcej wartości, jak pokazano w tym przykładzie:

```
class DeviceError(Exception):
    def __init__(self, errno, msg):
        self.args = (errno, msg)
        self.errno = errno
        self.errmsg = msg

# Zgłasza wyjątek (wiele argumentów)
raise DeviceError(1, 'Brak odpowiedzi')
```

Kiedy tworzysz niestandardową klasę wyjątku, która redefiniuje `__init__()`, ważne jest, aby przypisać krotkę zawierającą argumenty `__init__()` do atrybutu `self.args`, jak pokazano wyżej. Ten atrybut jest używany podczas wyświetlania komunikatów śledzenia wyjątków. Jeśli pozostanie niezdefiniowany, użytkownicy nie będą mogli zobaczyć żadnych przydatnych informacji o wyjątku, gdy wystąpi błąd.

Wyjątki można zorganizować w hierarchię za pomocą dziedziczenia. Na przykład zdefiniowany wcześniej wyjątek `NetworkError` może służyć jako klasa bazowa dla wielu bardziej szczegółowych błędów. Oto przykład:

```
class HostnameError(NetworkError):
    pass

class TimeoutError(NetworkError):
    pass

def error1():
    raise HostnameError('Nieznany host')

def error2():
    raise TimeoutError('Przekroczony czas')

try:
    error1()
except NetworkError as e:
    if type(e) is HostnameError:
        # Wykonaj specjalne działania dla tego rodzaju błędu
        ...
```

W takim przypadku klauzula `except NetworkError` przechwytuje wszelkie wyjątki pochodzące od `NetworkError`. Aby znaleźć konkretny typ błędu, który został zgłoszony, sprawdź typ wartości wykonania za pomocą `type()`.

3.4.4. Powiązane wyjątki

Czasami w odpowiedzi na wyjątek możesz chcieć zgłosić inny wyjątek. Aby to zrobić, wywołaj powiązany wyjątek:

```
class ApplicationError(Exception):
    pass

def do_something():
    x = int('N/A') # Wywołuje ValueError
```

```
def spam():
    try:
        do_something()
    except Exception as e:
        raise ApplicationError('Błąd') from e
```

Jeśli wystąpi nieprzechwycony wyjątek `ApplicationError`, otrzymasz komunikat zawierający oba wyjątki. Na przykład:

```
>>> spam()
Traceback (most recent call last):
  File "c.py", line 9, in spam
    do_something()
  File "c.py", line 5, in do_something
    x = int('N/A')
ValueError: invalid literal for int() with base 10: 'N/A'
```

Powyższy wyjątek był bezpośrednią przyczyną następującego wyjątku:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "c.py", line 11, in spam
    raise ApplicationError('Tu wystąpił błąd') from e
__main__.ApplicationError: Tu wystąpił błąd
>>>
```

Jeśli przechwycisz wyjątek `ApplicationError`, atrybut `__cause__` wynikowego wyjątku będzie zawierał drugi wyjątek. Na przykład:

```
try:
    spam()
except ApplicationError as e:
    print('Wystąpił błąd. Powód:', e.__cause__)
```

Jeśli chcesz zgłosić nowy wyjątek bez dołączania powiązania innych wyjątków, zgłoś błąd w powiązaniu z `None` w następujący sposób:

```
def spam():
    try:
        do_something()
    except Exception as e:
       raise ApplicationError('Wystąpił błąd') from None
```

Błąd programistyczny, który pojawia się w bloku `except`, spowoduje również powstanie powiązanego wyjątku, ale jego obsługa działa w nieco inny sposób. Załóżmy na przykład, że masz jakiś błędny kod, taki jak ten:

```
def spam():
    try:
        do_something()
    except Exception as e:
        print('Wystąpił błąd:', err) # Błąd niezdefiniowany (literówka)
```

Wynikowy komunikat śledzenia wyjątków jest nieco inny:

```
>>> spam()
Traceback (most recent call last):
```

```
  File "d.py", line 9, in spam
    do_something()
  File "d.py", line 5, in do_something
    x = int('N/A')
ValueError: invalid literal for int() with base 10: 'N/A'
```

Podczas obsługi powyższego wyjątku wystąpił kolejny wyjątek:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "d.py", line 11, in spam
    print('Wystąpił błąd. Powód:', err)
NameError: name 'err' is not defined
>>>
```

Jeśli podczas obsługi innego wyjątku zostanie zgłoszony nieoczekiwany wyjątek, atrybut `__context__` (zamiast `__cause__`) przechowuje informacje o wyjątku, który był obsługiwany w momencie wystąpienia błędu. Na przykład:

```
try:
    spam()
except Exception as e:
    print('Wystąpił błąd. Powód:', e)
    if e.__context__:
        print('Podczas obsługi:', e.__context__)
```

Istnieje ważne rozróżnienie między oczekiwanymi i nieoczekiwanymi wyjątkami w powiązanych wyjątkach. W pierwszym przykładzie kod został napisany tak, aby przewidzieć możliwość wystąpienia wyjątku. Na przykład kod został jawnie opakowany w blok `try-except`:

```
try:
    do_something()
except Exception as e:
    raise ApplicationError('Wystąpił błąd') from e
```

W drugim przypadku wystąpił błąd programistyczny w bloku `except`:

```
try:
    do_something()
except Exception as e:
    print('Wystąpił błąd:', err) # Niezdefiniowany błąd
```

Różnica między tymi dwoma przypadkami jest subtelna, ale ważna. Dlatego informacje o powiązanych wyjątkach są umieszczane w atrybucie `__cause__` lub `__context__`.

Atrybut `__cause__` jest zarezerwowany na wypadek, gdy spodziewasz się możliwości niepowodzenia.

Atrybut `__context__` jest ustawiony w obu przypadkach, ale byłby jedynym źródłem informacji o nieoczekiwanym wyjątku zgłoszonym podczas obsługi innego wyjątku.

3.4.5. Śledzenie wyjątków

Z wyjątkami skojarzone jest śledzenie stosu (ang. *stack traceback*), które dostarcza informacji o tym, gdzie wystąpił błąd. Jest ono przechowywane w atrybucie `__traceback__` wyjątku. Na potrzeby raportowania lub debugowania możesz chcieć samodzielnie wygenerować komunikat śledzenia. W tym celu można użyć modułu `traceback`. Na przykład:

```
import traceback

try:
    spam()
except Exception as e:
    tblines = traceback.format_exception(type(e), e, e.__traceback__)
    tbmsg = ''.join(tblines)
    print('Wystąpił błąd:')
    print(tbmsg)
```

W tym kodzie `format_exception()` tworzy listę ciągów zawierających dane wyjściowe, które Python normalnie wygeneruje w komunikacie śledzenia. Jako dane wejściowe podajesz typ wyjątku, wartość i atrybut śledzenia wyjątku.

3.4.6. Wskazówka dotycząca obsługi wyjątków

W większych programach obsługa wyjątków jest jednym z najtrudniejszych zadań do wykonania. Istnieje jednak kilka praktycznych zasad, które to ułatwiają.

Pierwszą zasadą jest nieprzechwytywanie wyjątków, których nie można obsłużyć w konkretnej lokalizacji w kodzie. Przyjrzyj się następującej funkcji:

```
def read_data(filename):
    with open(filename, 'rt') as file:
        rows = []
        for line in file:
            row = line.split()
            rows.append((row[0], int(row[1]), float(row[2]))
    return rows
```

Załóżmy, że funkcja `open()` nie działa z powodu złej nazwy pliku. Czy jest to błąd, który powinien zostać przechwycony za pomocą instrukcji `try-except` w tej funkcji? Prawdopodobnie nie. Jeśli w wywołaniu tej funkcji zostanie podana zła nazwa pliku, nie ma sensownego sposobu na odzyskanie sprawności działania programu. Nie ma pliku do otwarcia, żadnych danych do odczytania i nic więcej nie można zrobić. Lepiej pozwolić, aby operacja się nie powiodła, i zgłosić wyjątek. Unikanie sprawdzania błędów w `read_data()` nie oznacza, że wyjątek nigdzie nie byłby obsługiwany — oznacza po prostu, że nie jest to rolą funkcji `read_data()`. Być może kod, który prosił użytkownika o podanie nazwy pliku, obsłuży ten wyjątek.

Ta rada może się wydawać sprzeczna z doświadczeniem programistów przyzwyczajonych do języków, które opierają się na specjalnych kodach błędów lub opakowanych typach wynikowych. W tych językach dokładamy wszelkich starań, aby zawsze sprawdzać kody powrotu pod kątem błędów we wszystkich operacjach. Podczas pracy z Pythonem tak się nie postępuje. Jeśli operacja może się nie powieść i nie możesz nic zrobić, aby przywrócić jej sprawność, lepiej po prostu pozwolić jej się niepoprawnie wykonać. Wyjątek rozprzestrzeni się na wyższe poziomy programu, gdzie zwykle za jego obsługę odpowiada inny kod.

Z drugiej strony funkcja może być w stanie odzyskać sprawność po wystąpieniu błędu. Na przykład:

```
def read_data(filename):
    with open(filename, 'rt') as file:
```

```
        rows = []
        for line in file:
            row = line.split()
            try:
                rows.append((row[0], int(row[1]), float(row[2]))
            except ValueError as e:
                print('Błędny wiersz:', row)
                print('Powód:', e)
    return rows
```

Przechwytując błędy, postaraj się, aby klauzule `except` były rozsądnie małe. Powyższy kod mógł zostać napisany w celu wychwytywania wszystkich błędów przy instrukcji `except Exception`. Jednak zrobienie tego spowodowałoby, że kod wychwyciłby wszystkie błędy programistyczne, których prawdopodobnie nie należy ignorować. Nie rób tego — utrudni to debugowanie.

Na koniec, jeśli jawnie zgłaszasz wyjątek, rozważ utworzenie własnych typów wyjątków. Na przykład:

```
class ApplicationError(Exception):
    pass

class UnauthorizedUserError(ApplicationError):
    pass

def spam():
   ...
    raise UnauthorizedUserError('Wyjście')
   ...
```

Jednym z trudniejszych problemów związanych z dużymi aplikacjami jest szukanie winnych (w przypadku wystąpienia błędu programu), którzy byli odpowiedzialni za poszczególne fragmenty kodu. Jeśli tworzysz własne wyjątki, będziesz w stanie lepiej odróżnić celowo obsługiwane błędy od błędów programistycznych. Jeśli Twój program ulegnie awarii z jakimś rodzajem błędu aplikacji zdefiniowanego powyżej, od razu będziesz wiedział, dlaczego ten błąd został zgłoszony — ponieważ napisałeś kod, który to zrobi. Z drugiej strony, jeśli program ulegnie awarii i zostanie wywołany jeden z wbudowanych wyjątków Pythona (taki jak `TypeError` lub `ValueError`), może to wskazywać na poważniejszy problem.

3.5. Menedżery kontekstu i instrukcja with

Właściwe zarządzanie zasobami systemowymi, takimi jak pliki, blokady i połączenia, jest często trudnym zadaniem występującym w połączeniu z obsługą wyjątków Na przykład zgłoszony wyjątek może spowodować, że kontrola przepływu programu ominie instrukcje (takie jak blokady) odpowiedzialne za zwalnianie krytycznych zasobów.

Instrukcja `with` umożliwia wykonanie serii instrukcji w kontekście środowiska wykonawczego, który jest kontrolowany przez obiekt służący jako menedżer kontekstu. Oto przykład:

```
with open('debuglog', 'wt') as file:
    file.write('Debugowanie\n')
    instrukcje
    file.write('Zrobione\n')
```

```
import threading
lock = threading.Lock()
with lock:
    # Sekcja krytyczna
    instrukcje
    # Koniec sekcji krytycznej
```

W pierwszym przykładzie instrukcja `with` powoduje automatyczne zamknięcie otwartego pliku, gdy kontrola przepływu programu opuści blok następnych instrukcji. W drugim przykładzie instrukcja `with` automatycznie nakłada i zwalnia blokadę, gdy kontrola wchodzi i opuszcza blok instrukcji, które następują.

Instrukcja `with obj` umożliwia obiektowi `obj` zarządzanie tym, co się dzieje, gdy program wchodzi i wychodzi z powiązanego bloku instrukcji, który po niej następuje. Kiedy instrukcja `with obj` jest wykonywana, wywołuje metodę `obj.__enter__()`, aby zasygnalizować, że wprowadzany jest nowy kontekst. Gdy program opuszcza kontekst, wykonywana jest metoda `obj.__exit__(type, value, traceback)`. Jeśli nie zgłoszono żadnego wyjątku, wszystkie trzy argumenty `__exit__()` są ustawione na `None`. W przeciwnym razie zawierają typ, wartość i śledzenie skojarzone z wyjątkiem, który spowodował, że kontrola przepływu programu opuściła kontekst. Jeśli metoda `__exit__()` zwraca `True`, oznacza to, że zgłoszony wyjątek został obsłużony i nie powinien być dłużej propagowany. Zwrócenie wartości `None` lub `False` spowoduje propagację wyjątku.

Instrukcja `with obj` akceptuje opcjonalny specyfikator `as var`. Jeśli jest podany, wartość zwrócona przez `obj.__enter__()` jest umieszczana w `var`. Ta wartość jest zwykle taka sama jak `obj`, ponieważ umożliwia to konstruowanie obiektu i używanie go jako menedżera kontekstu w tym samym kroku. Rozważmy na przykład tę klasę:

```
class Manager:
    def __init__(self, x):
        self.x = x

    def yow(self):
        pass

    def __enter__(self):
        return self

    def __exit__(self, ty, val, tb):
        pass
```

Dzięki instrukcji `with` możesz w jednym kroku utworzyć i używać instancji jako menedżera kontekstu:

```
with Manager(42) as m:
    m.yow()
```

Oto ciekawszy przykład dotyczący transakcji listowych:

```
class ListTransaction:
    def __init__(self,thelist):
        self.thelist = thelist
```

```
    def __enter__(self):
        self.workingcopy = list(self.thelist)
        return self.workingcopy

    def __exit__(self, type, value, tb):
        if type is None:
            self.thelist[:] = self.workingcopy
        return False
```

Ta klasa umożliwia wykonanie sekwencji modyfikacji istniejącej listy. Modyfikacje zaczną jednak obowiązywać tylko wtedy, gdy nie wystąpią żadne wyjątki. W przeciwnym razie oryginalna lista pozostaje niezmieniona. Na przykład:

```
items = [1,2,3]

with ListTransaction(items) as working:
    working.append(4)
    working.append(5)
print(items) # Produkuje [1,2,3,4,5]

try:
    with ListTransaction(items) as working:
        working.append(6)
        working.append(7)
        raise RuntimeError("Mamy kłopot!")
except RuntimeError:
    pass

print(items) # Produkuje [1,2,3,4,5]
```

Moduł standardowej biblioteki `contextlib` zawiera funkcjonalność związaną z bardziej zaawansowanymi zastosowaniami menedżerów kontekstu. Jeśli regularnie tworzysz menedżery kontekstu, warto do niego zajrzeć.

3.6. Asercje i __debug__

Instrukcja `assert` może wprowadzić do programu kod debugujący. Ogólną formą `assert` jest:

```
assertz test [, msg]
```

gdzie `test` jest wyrażeniem, którego wartością powinno być `True` lub `False`. Jeśli `test` ma wartość `False`, `assert` zgłasza wyjątek `AssertionError` z opcjonalnym komunikatem `msg` dostarczonym do instrukcji `assert`. Oto przykład:

```
def write_data(file, data):
    assert file, 'zapis danych: plik nie został zdefiniowany!'
    ...
```

Instrukcja `assert` nie powinna być używana w kodzie, który musi zostać poprawnie wykonany, ponieważ program się nie uruchomi, jeśli Python będzie pracował w trybie zoptymalizowanym (określonym za pomocą opcji *-O* interpretera). Błędem jest zwłaszcza użycie instrukcji `assert` do sprawdzenia danych wejściowych użytkownika lub powodzenia

jakiejś ważnej operacji. Instrukcje `assert` używane są do sprawdzenia niezmiennych wartości, to znaczy wartości, które zawsze powinny być prawdziwe; jeśli któraś z nich będzie miała inną wartość, oznacza to błąd w programie, a nie błąd użytkownika.

Jeśli przykładowo funkcja `write_data()`, pokazana wcześniej, była przeznaczona do użytku przez użytkownika końcowego, instrukcję `assert` należy zastąpić konwencjonalną instrukcją `if` i żądaną obsługą błędów.

Powszechnym zastosowaniem asercji jest testowanie. Na przykład możesz jej użyć do włączenia minimalnego testu funkcji:

```
def factorial(n):
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result

assert factorial(5) == 120
```

Celem takiego testu nie jest to, aby był wyczerpujący, ale aby służył jako rodzaj „testu dymnego". Jeśli w funkcji zostanie zepsute coś oczywistego, kod natychmiastowo zakończy swoje działanie z nieudaną asercją.

Asercje mogą być również przydatne przy określaniu rodzaju umowy programistycznej dotyczącej oczekiwanych danych wejściowych i wyjściowych. Na przykład:

```
def factorial(n):
    assert n > 0, "musi być dodatnia wartość"
    result = 1
    while n > 1:
        result *= n
        n -= 1
    return result
```

Ponownie nie chodzi tu o sprawdzenie danych wejściowych użytkownika. To raczej sprawdzenie wewnętrznej spójności programu. Jeśli jakiś inny kod spróbuje obliczyć silnie dla liczby ujemnej, asercja zakończy się niepowodzeniem i wskaże niewłaściwy kod, aby można było go debugować.

3.7. Podsumowanie

Chociaż Python obsługuje wiele różnych stylów programowania obejmujących funkcje i obiekty, podstawowym modelem wykonywania programu jest programowanie imperatywne. Oznacza to, że programy składają się z instrukcji, które są wykonywane jedna po drugiej w kolejności, w jakiej pojawiają się w pliku źródłowym. Istnieją tylko trzy podstawowe konstrukcje kontroli sterowania programem: instrukcja `if`, pętla `while` i pętla `for`. Jeśli chcesz zrozumieć, w jaki sposób Python wykonuje Twój kod, zapoznaj się z poniższymi uwagami.

Zdecydowanie najbardziej skomplikowaną i potencjalnie podatną na błędy funkcją są wyjątki. W rzeczywistości większość tego rozdziału skupiała się na tym, jak poprawnie

implementować obsługę wyjątków. Nawet jeśli zastosujesz się do tej rady, wyjątki pozostają delikatną częścią projektowania bibliotek, frameworków i interfejsów API. Wyjątki mogą również wywoływać spustoszenie w prawidłowym zarządzaniu zasobami — jest to problem rozwiązywany za pomocą menedżerów kontekstu i instrukcji `with`.

W tym rozdziale nie omówiono technik, których można użyć do dostosowania prawie każdej właściwości języka Python — w tym wbudowanych operatorów, a nawet aspektów kontroli programu. Chociaż struktura programu w Pythonie pozornie jest prosta, zaskakująco dużo funkcjonalności może często działać pod spodem. Wiele informacji na ten temat podano w następnym rozdziale.

4

Obiekty, typy i protokoły

Programy Pythona pracują na obiektach różnych typów. Istnieje wiele wbudowanych typów, takich jak liczby, ciągi, listy, zbiory i słowniki. Ponadto możesz tworzyć własne typy za pomocą klas. Ten rozdział opisuje podstawowy model obiektów Pythona i mechanizmy, dzięki którym wszystkie obiekty działają. Szczególną uwagę zwraca się na protokoły, które definiują podstawowe zachowanie różnych obiektów.

4.1. Podstawowe pojęcia

Każda część danych przechowywana w programie jest obiektem. Każdy obiekt ma tożsamość, typ (nazywany również jego klasą) i wartość. Kiedy piszesz na przykład `a = 42`, tworzony jest obiekt typu `integer` o wartości 42. Tożsamość obiektu to liczba reprezentująca jego położenie w pamięci; `a` jest etykietą, która odnosi się do tej konkretnej lokalizacji, chociaż etykieta nie jest częścią samego obiektu.

Typ obiektu, nazywany również klasą obiektu, definiuje wewnętrzną reprezentację danych obiektu, a także obsługiwane metody. Kiedy tworzony jest obiekt określonego typu, ten obiekt jest nazywany *instancją* tego typu. Po utworzeniu instancji jej tożsamość nie ulega zmianie. Jeśli wartość obiektu można zmodyfikować, mówi się, że obiekt jest zmienny. Jeśli wartość nie może zostać zmodyfikowana, obiekt jest niezmienny. Obiekt, który zawiera odniesienia do innych obiektów, jest nazywany kontenerem.

Obiekty charakteryzują się swoimi atrybutami. Atrybut to wartość powiązana z obiektem, do którego dostęp uzyskuje się za pomocą operatora kropki (`.`). Atrybut może być prostą wartością danych, taką jak liczba. Jednak atrybutem może być również funkcja, która jest wywoływana w celu wykonania jakiejś operacji. Takie funkcje nazywane są metodami. Poniższy przykład ilustruje dostęp do atrybutów:

```
a = 34              # Tworzy liczbę całkowitą
n = a.numerator     # Pobiera licznik (atrybut)
b = [1, 2, 3]       # Tworzy listę
b.append(7)         # Dodaje nowy element za pomocą metody append
```

Obiekty mogą również implementować różne operatory, takie jak operator +. Na przykład:

```
c = a + 10        # c = 34 + 10
d = b + [4, 5]    # d = [1, 2, 3, 7, 4, 5]
```

Chociaż w przypadku operatorów stosowana jest inna składnia, ostatecznie są mapowane do metod. Na przykład napisanie `a + 10` powoduje wykonanie metody `a.__add__(10)`.

4.2. Tożsamość i typ obiektu

Wbudowana funkcja `id()` zwraca tożsamość obiektu. Tożsamość jest liczbą całkowitą, która zwykle odpowiada lokalizacji obiektu w pamięci. Operatory `is` i `is not` porównują tożsamości dwóch obiektów. `type()` zwraca typ obiektu. Oto przykład różnych sposobów porównywania dwóch obiektów:

```
# Porównaj dwa obiekty
def compare(a, b):
    if a is b:
        print('ten sam obiekt')
    if a == b:
        print('ta sama wartość')
    if type(a) is type(b):
        print('ten sam typ')
```

Oto jak działa ta funkcja:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> compare(a, a)
ten sam obiekt
ta sama wartość
ten sam typ
>>> compare(a, b)
ta sama wartość
ten sam typ
>>> compare(a, [4,5,6])
ten sam typ
>>>
```

Typ obiektu sam w sobie jest obiektem zwanym klasą obiektu. Ten obiekt jest jednoznacznie zdefiniowany i jest zawsze taki sam dla wszystkich instancji danego typu. Klasy mają zwykle nazwy (`list`, `int`, `dict` itd.), których można używać do tworzenia instancji, sprawdzania typu i dostarczania wskazówek dotyczących typu. Na przykład:

```
items = list()

if isinstance(items, list):
    items.append(item)

def removeall(items: list, item) -> list:
    return [i for i in items if i != item]
```

Podtyp to typ zdefiniowany przez dziedziczenie. Zawiera wszystkie cechy oryginalnego typu oraz dodatkowe i (lub) przedefiniowane metody. Dziedziczenie omówiono bardziej szczegółowo w rozdziale 7. Poniżej przedstawiono przykład definiowania podtypu listy z dodaną do niego nową metodą:

```
class mylist(list):
    def removeall(self, val):
        return [i for i in self if i != val]

# Przykład
items = mylist([5, 8, 2, 7, 2, 13, 9])
x = items.removeall(2)
print(x) # [5, 8, 7, 13, 9]
```

Funkcja `isinstance(instance, type)` jest preferowanym sposobem sprawdzania wartości z typem, ponieważ rozpoznaje podtypy. Może również sprawdzać wiele możliwych typów.

Na przykład:

```
if isinstance(items, (list, tuple)):
    maxval = max(items)
```

Chociaż do programu można dodać kontrolę typu, często nie jest to tak przydatne, jak można by sobie wyobrażać. Po pierwsze, nadmierne sprawdzanie wpływa na wydajność. Po drugie, programy nie zawsze definiują obiekty, które pasują do hierarchii typów. Jeśli na przykład celem instrukcji `isinstance(items, list)` jest sprawdzenie, czy elementy są podobne do listy, nie będzie ona działać z obiektami, które mają ten sam interfejs programistyczny co lista, ale nie dziedziczą bezpośrednio po wbudowanym typie listy (przykładem jest `deque` z modułu `collections`).

4.3. Zliczanie referencji i odśmiecanie pamięci

Python zarządza obiektami poprzez automatyczne usuwanie śmieci. Wszystkie obiekty mają zliczane referencje. Licznik odwołań do obiektu jest zwiększany za każdym razem, gdy zostanie przypisany do nowej nazwy lub jest umieszczony w kontenerze, takim jak lista, krotka bądź słownik:

```
a = 37       # Tworzy obiekt o wartości 37
b = a        # Zwiększa licznik referencji do 37
c = []
c.append(b)  # Zwiększa licznik referencji do 37
```

Ten przykład tworzy pojedynczy obiekt zawierający wartość 37. `a` to nazwa, która początkowo odnosi się do nowo utworzonego obiektu. Gdy `b` jest przypisane do `a`, `b` staje się nową nazwą dla tego samego obiektu, a liczba odwołań do obiektu wzrasta. Gdy umieścisz `b` na liście, liczba odwołań do obiektu ponownie wzrośnie. W całym przykładzie tylko jeden obiekt odpowiada wartości 37. Wszystkie inne operacje tworzą odniesienia do tego obiektu.

Liczba odwołań do obiektu jest zmniejszana przez instrukcję `del` lub za każdym razem, gdy odwołanie wykracza poza zakres bądź jest ponownie przypisywane. Oto przykład:

```
del a        # Zmniejsza liczbę referencji do 37
b = 42       # Zmniejsza liczbę referencji do 37
c[0] = 2.0   # Zmniejsza liczbę referencji do 37
```

Bieżącą liczbę odwołań obiektu można uzyskać za pomocą funkcji `sys.getrefcount()`. Na przykład:

```
>>> a = 37
>>> import sys
>>> sys.getrefcount(a)
7
>>>
```

Liczba referencji jest często znacznie wyższa, niż można by się spodziewać. W przypadku danych niezmiennych, takich jak liczby i łańcuchy, interpreter agresywnie dzieli obiekty między różne części programu w celu oszczędzania pamięci. Po prostu tego nie zauważasz, ponieważ obiekty są niezmienne.

Gdy liczba odwołań do obiektu osiągnie zero, jest on usuwany z pamięci. Jednak w niektórych przypadkach w kolekcji obiektów, które nie są już używane, może istnieć zależność cykliczna. Oto przykład:

```
a = { }
b = { }
a['b'] = b      # a zawiera odniesienie do b
b['a'] = a      # b zawiera odniesienie do a
del a
del b
```

W tym przykładzie instrukcje `del` zmniejszają liczbę odwołań `a` i `b` oraz niszczą nazwy używane do odwoływania się do podstawowych obiektów. Ponieważ jednak każdy obiekt zawiera odwołanie do drugiego, liczba odwołań nie spada do zera, a obiekty pozostają w pamięci. Interpreter nie spowoduje wycieku pamięci, ale zniszczenie obiektów zostanie opóźnione, dopóki detektor cyklu nie wykona operacji, aby znaleźć i usunąć niedostępne obiekty. Algorytm wykrywania cykli działa okresowo, gdy interpreter alokuje coraz więcej pamięci podczas wykonywania. Dokładne zachowanie można dostroić i kontrolować za pomocą funkcji w module biblioteki standardowej `gc`. Funkcja `gc.collect()` może być wykorzystana do natychmiastowego wywołania cyklicznego odśmiecania pamięci.

W większości programów odśmiecanie jest przeprowadzane automatycznie co zwalnia programistę z konieczności zastanawiania się nad tym. Możesz się jednak spotkać z sytuacjami, w których ręczne usuwanie obiektów może mieć sens. Jeden z takich scenariuszy pojawia się podczas pracy z gigantycznymi strukturami danych. Zerknij na przykład na ten kod:

```
def some_calculation():
    data = create_giant_data_structure()
    # Użyj danych do jakiejś części obliczeń
    ...
    # Zwolnij pamięć danych
    del data

    # Obliczenia trwają
    ...
```

W tym kodzie użycie instrukcji `del data` wskazuje, że zmienna `data` nie jest już potrzebna. Jeśli spowoduje to, że licznik odwołań osiągnie `0`, pamięć obiektu jest w tym momencie zwalniana.

Bez instrukcji `del` obiekt utrzymuje się przez nieokreślony czas, aż zmienna `data` wyjdzie poza zakres na końcu funkcji. Możesz to zauważyć tylko wtedy, gdy próbujesz się dowiedzieć, dlaczego Twój program używa więcej pamięci, niż powinien.

4.4. Referencje i kopie

Kiedy program wykonuje przypisanie, takie jak `b = a`, tworzone jest nowe odniesienie do `a`. W przypadku niezmiennych obiektów, takich jak liczby i ciągi, to przypisanie wydaje się tworzyć kopię `a` (nawet jeśli tak nie jest). Jednak dla zmiennych obiektów, takich jak listy i słowniki, wygląda to zupełnie inaczej. Oto przykład:

```
>>> a = [1,2,3,4]
>>> b = a               # b jest odniesieniem do a
>>> b is a
True
>>> b[2] = -100         # Zmień element w b
>>> a                   # Zwróć uwagę, że zmienił się również element w a
[1, 2, -100, 4]
>>>
```

Ponieważ w tym przykładzie `a` i `b` odnoszą się do tego samego obiektu, zmiana dokonana w jednej ze zmiennych jest odzwierciedlana w drugiej. Aby tego uniknąć, musisz utworzyć kopię obiektu, a nie nową referencję.

Do obiektów kontenerów, takich jak listy i słowniki, stosowane są dwa typy operacji kopiowania: kopia płytka i kopia głęboka. Płytka kopia tworzy nowy obiekt, ale wypełnia go odniesieniami do elementów zawartych w oryginalnym obiekcie. Oto przykład:

```
>>> a = [1, 2, [3, 4]]
>>> b = list(a)           # Utwórz płytką kopię a
>>> b is a
False
>>> b.append(100)         # Dołącz element do b
>>> b
[1, 2, [3, 4], 100]
>>> a                     # Zauważ, że a jest niezmienione
[1, 2, [3, 4]]
>>> b[2][0] = -100        # Modyfikacja elementu wewnątrz b
>>> b
[1, 2, [-100, 4], 100]
>>> a                     # Zwróć uwagę na zmianę wewnątrz a
[1, 2, [-100, 4]]
>>>
```

W tym przypadku `a` i `b` są oddzielnymi obiektami listy, ale elementy, które one zawierają, są wspólne. Dlatego zmiana jednego z elementów `a` modyfikuje również element `b`, jak pokazano.

Głęboka kopia tworzy nowy obiekt i rekursywnie kopiuje wszystkie zawarte w nim obiekty. Nie ma wbudowanego operatora do tworzenia głębokich kopii obiektów, ale możesz użyć funkcji `copy.deepcopy()` ze standardowej biblioteki:

```
>>> import copy
>>> a = [1, 2, [3, 4]]
>>> b = copy.deepcopy(a)
>>> b[2][0] = -100
>>> b
[1, 2, [-100, 4]]
>>> a                    # Zauważ, że a jest niezmienione
[1, 2, [3, 4]]
>>>
```

Korzystanie z `deepcopy()` jest odradzane w większości programów. Kopiowanie obiektu jest powolne i często niepotrzebne. Zarezerwuj `deepcopy()` dla sytuacji, w których faktycznie potrzebujesz kopii, ponieważ masz zamiar zmienić dane i nie chcesz, aby te zmiany wpłynęły na oryginalny obiekt. Należy również pamiętać, że funkcja `deepcopy()` zawiedzie w przypadku pracy z obiektami, które obejmują stan systemu lub środowiska wykonawczego (takimi jak otwarte pliki, połączenia sieciowe, wątki, generatory itd.).

4.5. Reprezentacja obiektu i wyświetlanie

Programy często muszą wyświetlać obiekty, na przykład aby pokazać dane użytkownikowi lub wyświetlić je w celu debugowania. Jeśli dostarczysz obiekt `x` do funkcji `print(x)` lub przekonwertujesz go na ciąg znaków za pomocą `str(x)`, otrzymasz „ładną”, czytelną dla człowieka reprezentację wartości obiektu. Rozważmy przykład z datami:

```
>>> from datetime import date
>>> d = date(2012, 12, 21)
>>> print(d)
2012-12-21
>>> str(d)
'2012-12-21'
>>>
```

Ta „ładna” reprezentacja obiektu może nie wystarczyć do debugowania. Na przykład nie ma oczywistego sposobu, aby dowiedzieć się, czy zmienna `d` jest instancją daty, czy prostym ciągiem znaków zawierającym tekst `'2012-12-21'`. Aby uzyskać więcej informacji, użyj funkcji `repr(x)`, która tworzy ciąg z reprezentacją obiektu. Na przykład:

```
>>> d = date(2012, 12, 21)
>>> repr(d)
'datetime.date(2012, 12, 21)'
>>> print(repr(d))
datetime.date(2012, 12, 21)
>>> print(f'Data to: {d!r}')
The date is: datetime.date(2012, 12, 21)
>>>
```

W formatowaniu ciągów sufiks `!r` może być dodany do wartości, aby wytworzyć jej wartość `repr()` zamiast normalnej konwersji ciągu.

4.6. Obiekty pierwszoklasowe

Mówi się, że wszystkie obiekty w Pythonie są *pierwszoklasowe*. Oznacza to, że wszystkie obiekty, które można przypisać do nazwy, mogą być również traktowane jako dane. Tak jak dane, obiekty mogą być przechowywane jako zmienne, przekazywane jako argumenty, zwracane z funkcji, porównywane z innymi obiektami i nie tylko.

Oto prosty słownik zawierający dwie wartości:

```
items = {
    'number': 42
    'text': "Witaj, świecie"
}
```

Pierwszoklasową naturę obiektów można zobaczyć, dodając do tego słownika kilka bardziej nietypowych pozycji:

```
items['func'] = abs              # Dodaj funkcję abs()
import math
items['mod'] = math              # Dodaj moduł
items['error'] = ValueError      # Dodaj typ wyjątku
liczba = [1,2,3,4]
items['append'] = nums.append    # Dodaj metodę innego obiektu
```

W tym przykładzie słownik `items` zawiera teraz funkcję, moduł, wyjątek i metodę innego obiektu. Jeśli chcesz, zamiast stosować oryginalne nazwy, możesz skorzystać z wyszukiwania słownikowego na `items`, a kod nadal będzie działał. Na przykład:

```
>>> items['func'](-45)            # Wykonuje abs(–45)
45
>>> items['mod'].sqrt(4)          # Wykonuje math.sqrt(4)
2.0
>>> try:
... x = int('a lot')
... except items['error'] as e:  # To samo co: except ValueError as e
... print("Nie można przekonwertować")
...
Nie można przekonwertować
>>> items['append'](100)          # Wykonuje nums.append(100)
>>> nums
[1, 2, 3, 4, 100]
>>>
```

Fakt, że wszystko w Pythonie jest pierwszoklasowe, często nie jest w pełni doceniany przez niedoświadczonych programistów. Dzięki temu jednak można napisać bardzo zwarty i elastyczny kod.

Załóżmy na przykład, że masz wiersz tekstu, taki jak „`ACME,100,490,10`”, i chcesz go przekonwertować na listę wartości z odpowiednimi konwersjami typu. Oto sprytny sposób, aby to zrobić. Tworzymy listę typów (które są obiektami pierwszoklasowymi) i wykonujemy kilka standardowych operacji przetwarzania list:

```
>>> line = 'ACME,100,490.10'
>>> column_types = [str, int, float]
>>> parts = line.split(',')
```

```
>>> row = [ty(val) for ty, val in zip(column_types, parts)]
>>> row
['ACME', 100, 490.1]
>>>
```

Umieszczanie funkcji lub klas w słowniku jest powszechną techniką eliminowania złożonych instrukcji `if-elif-else`. Jeśli na przykład masz taki kod:

```
if format == 'text':
    formatter = TextFormatter()
elif format == 'csv':
    formatter = CSVFormatter()
elif format == 'html':
    formatter = HTMLFormatter()
else:
    raise RuntimeError('Nieprawidłowy format')
```

możesz przepisać go, stosując słownik:

```
_formats = {
'text': TextFormatter,
'csv': CSVFormatter,
'html': HTMLFormatter
}
if format in _formats:
formatter = _formats[format]()
else:
raise RuntimeError('Nieprawidłowy format')
```

Ta ostatnia forma jest również bardziej elastyczna, ponieważ można dodawać nowe przypadki przez wstawianie większej liczby wpisów do słownika bez konieczności modyfikowania dużego bloku instrukcji `if-elif-else`.

4.7. Używanie wartości None dla opcjonalnych lub brakujących danych

Czasami w programach trzeba określić opcjonalną lub brakującą wartość. `None` jest specjalną instancją zarezerwowaną do tego celu. Wartość `None` jest zwracana przez funkcje, które nie zwracają wartości w sposób jawny. `None` jest również często używana jako wartość domyślna argumentów opcjonalnych, dzięki czemu funkcja może wykryć, czy obiekt wywołujący rzeczywiście przekazał wartość dla tego argumentu. `None` nie ma atrybutów i jest obliczana jako `False` w wyrażeniach logicznych.

Wewnętrznie `None` jest przechowywana jako *singleton* — oznacza to, że w interpreterze jest tylko jedna wartość `None`. Dlatego powszechnym sposobem testowania wartości względem `None` jest użycie operatora `is` w następujący sposób:

```
if value is None:
    instrukcje
    ...
```

Testowanie pod kątem None przy użyciu operatora == również działa, ale nie jest zalecane i może zostać oznaczone przez narzędzia do sprawdzania kodu jako błąd stylu.

4.8. Protokoły obiektu i abstrakcja danych

Większość funkcji języka Python jest definiowana przez *protokoły*. Przyjrzyj się następującej funkcji:

```
def compute_cost(unit_price, num_units):
    return unit_price * num_units
```

Teraz zadaj sobie pytanie: jakie dane wejściowe są dozwolone? Odpowiedź jest zwodniczo prosta: wszystko jest dozwolone! Na pierwszy rzut oka wygląda na to, że powyższa funkcja operuje na liczbach:

```
>>> compute_cost(1.25, 50)
62.5
>>>
```

Rzeczywiście, funkcja działa zgodnie z oczekiwaniami. Jednak może ona o wiele więcej. Możesz używać liczb specjalnych, takich jak ułamki zwykłe lub dziesiętne:

```
>>> from fractions import Fraction
>>> compute_cost(Fraction(5, 4), 50)
Fraction(125, 2)
>>> from decimal import Decimal
>>> compute_cost(Decimal('1.25'), Decimal('50'))
Decimal('62.50')
>>>
```

Mało tego — funkcja działa z tablicami i innymi złożonymi strukturami zawartymi w pakietach takich jak numpy. Na przykład:

```
>>> import numpy as np
>>> prices = np.array([1.25, 2.10, 3.05])
>>> units = np.array([50, 20, 25])
>>> compute_cost(prices, quantities)
array([62.5, 42., 76.25])
>>>
```

Funkcja może nawet działać w nieoczekiwany sposób:

```
>>> compute_cost('a lot', 10)
'a lota lota lota lota lota lota lota lota lota lot'
>>>
```

A jednak niektóre kombinacje typów wygenerują błąd:

```
>>> compute_cost(Fraction(5, 4), Decimal('50'))
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "<stdin>", line 2, in compute_cost
TypeError: unsupported operand type(s) for *: 'Fraction' and 'decimal.Decimal'
>>>
```

W przeciwieństwie do kompilatora języka statycznego, Python nie weryfikuje z góry prawidłowego zachowania programu. Zamiast tego zachowanie obiektu jest określane przez dynamiczny proces, który obejmuje wysyłanie tak zwanych specjalnych lub magicznych metod. Nazwy tych specjalnych metod są zawsze poprzedzone podwójnym podkreśleniem (__). Metody są automatycznie wyzwalane przez interpreter podczas wykonywania programu. Na przykład operacja `x * y` jest wykonywana przez metodę `x.__mul__(y)`. Nazwy tych metod i odpowiadające im operatory są ustalone na stałe. Zachowanie dowolnego obiektu zależy całkowicie od zestawu specjalnych metod, które implementuje.

Kilka następnych rozdziałów opisuje specjalne metody związane z różnymi kategoriami podstawowych funkcji interpretera. Te kategorie są czasami nazywane protokołami. Obiekt, w tym klasa zdefiniowana przez użytkownika, może definiować dowolną kombinację tych cech, aby obiekt zachowywał się na różne sposoby.

4.9. Protokół obiektu

Metody przedstawione w tabeli 4.1 dotyczą ogólnego zarządzania obiektami. Obejmuje to tworzenie, inicjowanie, niszczenie i reprezentację obiektów.

Tabela 4.1. *Metody zarządzania obiektami*

Metoda	Opis
`__new__(cls [,*args [,**kwargs]])`	Statyczna metoda wywoływana w celu utworzenia nowej instancji.
`__init__(self [,*args [,**kwargs]])`	Wywoływana w celu zainicjowania nowej instancji po jej utworzeniu.
`__del__(self)`	Wywoływana, gdy instancja jest niszczona.
`__repr__(self)`	Tworzy reprezentację w postaci ciągu.

Metody `__new__()` i `__init__()` są używane razem do tworzenia i inicjowania instancji. Kiedy obiekt jest tworzony przez wywołanie `SomeClass(args)`, jest on tłumaczony na następujące kroki:

```
x = SomeClass.__new__(SomeClass, args)
if isinstance(x, SomeClass):
    x.__init__(args)
```

Zwykle te kroki są obsługiwane w tle i nie musisz się o nie martwić. Najpopularniejszą metodą zaimplementowaną w klasie jest `__init__()`. Użycie `__new__()` prawie zawsze wskazuje na obecność zaawansowanych kroków związanych z tworzeniem instancji (na przykład jest używana w metodach klasowych, które chcą ominąć `__init__()`, lub w pewnych twórczych wzorcach projektowych, takich jak Singleton. Implementacja `__new__()` niekoniecznie musi zwracać instancję danej klasy — jeśli tego nie robi, kolejne wywołanie `__init__()` podczas tworzenia jest pomijane.

Metoda `__del__()` jest wywoływana, gdy instancja ma zostać usunięta z pamięci. Ta metoda jest wywoływana tylko wtedy, gdy instancja nie jest już używana. Zauważ, że instrukcja `del x` zmniejsza jedynie liczbę odwołań do instancji i niekoniecznie skutkuje wywołaniem tej funkcji.

`__del__()` prawie nigdy nie jest definiowana, chyba że instancja musi wykonać dodatkowe kroki zarządzania zasobami po odśmiecaniu.

Metoda `__repr__()`, wywoływana przez wbudowaną funkcję `repr()`, tworzy ciąg reprezentujący obiekt, który może być przydatny do debugowania i wyświetlania. Jest to również metoda odpowiedzialna za tworzenie danych wyjściowych, które widzisz podczas sprawdzania zmiennych w interaktywnym interpreterze. Konwencja jest taka, że `__repr__()` zwraca ciąg wyrażenia, który może zostać wykorzystany do ponownego utworzenia obiektu za pomocą `eval()`. Na przykład:

```
a = [2, 3, 4, 5]      # Utwórz listę
s = repr(a)           # s = '[2, 3, 4, 5]'
b = eval(s)           # Zamienia s z powrotem w listę
```

Jeśli nie można utworzyć wyrażenia łańcuchowego, to zgodnie z konwencją `__repr__()` zwraca łańcuch w postaci `<...wiadomość...>`, jak pokazano poniżej:

```
f = open('foo.txt')
a = repr(f)
# a = "<_io.TextIOWrapper name='foo.txt' mode='r' encoding='UTF-8'>
```

4.10. Protokół liczbowy

Tabela 4.2 zawiera listę specjalnych metod, które muszą zostać zaimplementowane przez obiekty, aby zapewnić operacje matematyczne.

Tabela 4.2. *Metody operacji matematycznych*

Metoda	Operacja
`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__mul__(self, other)`	`self * other`
`__truediv__(self, other)`	`self / other`
`__floordiv__(self, other)`	`self // other`
`__mod__(self, other)`	`self % other`
`__matmul__(self, other)`	`self @ other`
`__divmod__(self, other)`	`divmod(self, other)`
`__pow__(self, other [, modulo])`	`self ** inne, pow(self, other, modulo)`
`__lshift__(self, other)`	`self << other`
`__rshift__(self, other)`	`self >> other`
`__and__(self, other)`	`self & other`
`__or__(self, other)`	`self | other`
`__xor__(self, other)`	`self ^ other`

Tabela 4.2. *Metody operacji matematycznych (ciąg dalszy)*

Metoda	Operacja
`__radd__(self, other)`	`other + self`
`__rsub__(self, other)`	`other - self`
`__rmul__(self, other)`	`other * self`
`__rtruediv__(self, other)`	`other / self`
`__rfloordiv__(self, other)`	`other // self`
`__rmod__(self, other)`	`other % self`
`__rmatmul__(self, other)`	`other @ self`
`__rdivmod__(self, other)`	`divmod(other, self)`
`__rpow__(self, other)`	`other ** self`
`__rlshift__(self, other)`	`other << self`
`__rrshift__(self, other)`	`other >> self`
`__rand__(self, other)`	`other & self`
`__ror__(self, other)`	`other | self`
`__rxor__(self, other)`	`other ^ self`
`__iadd__(self, other)`	`self += other`
`__isub__(self, other)`	`self -= other`
`__imul__(self, other)`	`self *= other`
`__itruediv__(self, other)`	`self /= other`
`__ifloordiv__(self, other)`	`self //= other`
`__imod__(self, other)`	`self %= other`
`__imatmul__(self, other)`	`self @= other`
`__ipow__(self, other)`	`self **= other`
`__iand__(self, other)`	`self &= other`
`__ior__(self, other)`	`self |= other`
`__ixor__(self, other)`	`self ^= other`
`__ilshift__(self, other)`	`self <<= other`
`__irshift__(self, other)`	`self >>= other`
`__neg__(self)`	`-self`
`__pos__(self)`	`+self`
`__invert__(self)`	`~self`
`__abs__(self)`	`abs(self)`

Tabela 4.2. *Metody operacji matematycznych (ciąg dalszy)*

Metoda	Operacja
`__round__(self, n)`	`round(self, n)`
`__floor__(self)`	`math.floor(self)`
`__ceil__(self)`	`math.ceil(self)`
`__trunc__(self)`	`math.trunc(self)`

Po przedstawieniu wyrażenia, takiego jak `x + y`, interpreter — w celu wykonania operacji — wywołuje kombinację metod `x.__add__(y)` lub `y.__radd__(x)`. Pierwszym wyborem jest wypróbowanie `x.__add__(y)` we wszystkich przypadkach z wyjątkiem jednego, specjalnego, w którym `y` jest podtypem `x`; metoda `y.__radd__(x)` jest wówczas wykonywana jako pierwsza. Jeśli metoda początkowa się nie powiedzie i zwróci `NotImplemented`, podejmowana jest próba wywołania operacji z odwróconymi operandami, tzn. `y.__radd__(x)`. Jeśli ta druga próba się nie uda, cała operacja się nie powiedzie. Oto przykład:

```
>>> a = 42        # int
>>> b = 3.7       # float
>>> a.__add__(b)
NotImplemented
>>> b.__radd__(a)
45.7
>>>
```

Ten przykład może się wydawać zaskakujący, ale odzwierciedla fakt, że liczby całkowite są, matematycznie, specjalnym rodzajem liczb zmiennoprzecinkowych. W ten sposób odwrócony operand daje poprawną odpowiedź.

Metody `__iadd__()`, `__isub__()` itd. są używane do obsługi lokalnych operatorów arytmetycznych, takich jak `a += b` i `a -= b` (operacje nazywane również przypisaniami rozszerzonymi). Rozróżnia się operatory i standardowe metody arytmetyczne, ponieważ implementacja lokalnych operatorów może zapewnić dostosowanie do własnych potrzeb lub optymalizacje wydajności. Jeśli na przykład obiekt nie jest współdzielony, wartość obiektu może być modyfikowana lokalnie bez przydzielania nowo utworzonego obiektu do wyniku. Jeśli operatory lokalne są niezdefiniowane, operacja taka jak `a += b` jest liczona przy użyciu `a = a + b`.

Nie ma metod, których można użyć do zdefiniowania zachowania operatorów logicznych `and`, `or` lub `not`. Operatory `and` i `or` wdrażają *liczenie short-circuit*, w którym obliczanie zatrzymuje się, jeśli można już określić ostateczny wynik. Na przykład:

```
>>> True or 1/0 # Nie oblicza wyniku 1/0
True
>>>
```

To zachowanie obejmujące nieliczone podwyrażenia nie może być wyrażone przy użyciu reguł liczenia normalnej funkcji lub metody. Tak więc nie ma protokołu ani zestawu metod, które mogłyby je przedefiniować. Zamiast tego jest obsługiwane jako specjalny przypadek w samej implementacji Pythona.

4.11. Protokół porównania

Obiekty można porównywać na różne sposoby. Najbardziej podstawowym jest sprawdzenie tożsamości za pomocą operatora `is`, na przykład `a is b`. Tożsamość nie uwzględnia wartości przechowywanych w obiekcie, nawet jeśli są one takie same. Przykładowo:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
>>> c = [1, 2, 3]
>>> a is c
False
>>>
```

Operator `is` jest wewnętrzną częścią Pythona, której nie można przedefiniować. Wszystkie inne porównania na obiektach są realizowane metodami z tabeli 4.3.

Tabela 4.3. *Metody porównywania instancji i haszowanie*

Metoda	Opis
`__bool__(self)`	Zwraca `False` lub `True` w przypadku testowania prawdziwości.
`__eq__(self, other)`	`self == other`
`__ne__(self, other)`	`self != other`
`__lt__(self, other)`	`self < other`
`__le__(self, other)`	`self <= other`
`__gt__(self, other)`	`self > other`
`__ge__(self, other)`	`self >= other`
`__hash__(self)`	Oblicza całkowity indeks haszowania.

Metoda `__bool__()`, jeśli jest obecna, jest używana do określenia wartości logicznej, gdy obiekt jest testowany jako część warunku lub wyrażenia warunkowego. Na przykład:

```
if a:    # Wykonuje a.__bool__()
...
else:
...
```

Jeśli `__bool__()` jest niezdefiniowane, to używana jest metoda `__len__()`. Jeśli zarówno `__bool__()`, jak i `__len__()` są niezdefiniowane, obiekt jest po prostu uważany za `True`.

Metoda `__eq__()` służy do określenia podstawowej równości — działa z operatorami `==` i `!=`. Domyślna implementacja `__eq__()` porównuje obiekty według tożsamości przy użyciu operatora `is`. Metoda `__ne__()`, jeśli jest obecna, może być użyta do zaimplementowania specjalnego przetwarzania dla `!=`, ale zwykle nie jest to wymagane, o ile zdefiniowano `__eq__()`.

Porządkowanie jest określane przez operatory relacyjne (`<`, `>`, `<=` i `>=`) przy użyciu metod takich jak `__lt__()` i `__gt__()`. Podobnie jak w przypadku innych operacji matematycznych, zasady liczenia są precyzyjne. Aby obliczyć `a < b`, interpreter najpierw spróbuje wykonać `a.__lt__(b)`, chyba że `b` jest podtypem `a`. W tym jednym konkretnym przypadku zamiast tego wykonuje się `b.__gt__(a)`. Jeśli ta początkowa metoda nie jest zdefiniowana lub zwraca

`NotImplemented`, interpreter próbuje odwróconego porównania, wywołując `b.__gt__(a)`. Podobne zasady dotyczą operatorów takich jak `<=` i `>=`. Na przykład wyrażenie `<=` najpierw próbuje obliczyć `a.__le__(b)`, a jeśli nie zostało zaimplementowane, liczone jest `b.__ge__(a)`.

Każda z metod porównywania przyjmuje dwa argumenty i może zwrócić dowolny rodzaj wartości, w tym wartość logiczną, listę lub dowolny inny typ Pythona. Na przykład pakiet liczbowy może użyć takiej metody do porównania dwóch macierzy — element po elemencie, zwracając macierz z wynikami. Jeśli porównanie nie jest możliwe, metody powinny zwrócić wbudowany obiekt `NotImplemented`. To nie to samo co wyjątek `NotImplementedError`. Na przykład:

```
>>> a = 42       # int
>>> b = 52.3     # float
>>> a.__lt__(b)
NotImplemented
>>> b.__gt__(a)
True
>>>
```

Obiekt nie musi implementować wszystkich operacji porównania z tabeli 4.3. Jeśli chcesz mieć możliwość sortowania obiektów lub używania funkcji, takich jak `min()` lub `max()`, musi być przynajmniej zdefiniowana metoda `__lt__()`. Jeśli dodajesz operatory porównania do klasy zdefiniowanej przez użytkownika, może się przydać dekorator klas `@total_ordering` z modułu `functools`. Może on generować wszystkie metody, o ile tylko zaimplementujesz `__eq__()` i jedno z innych porównań.

Metoda `__hash__()` jest zdefiniowana na instancjach, które mają być umieszczone w zbiorze lub używane jako klucze w odwzorowaniu (słowniku). Zwracana wartość jest liczbą całkowitą, która powinna być taka sama dla dwóch wystąpień porównywanych jako równe. Co więcej, `__eq__()` należy zawsze definiować razem z `__hash__()`, ponieważ te dwie metody działają razem. Wartość zwrócona przez `__hash__()` jest zwykle używana jako wewnętrzny szczegół implementacji różnych struktur danych. Możliwe jest jednak, aby dwa różne obiekty miały tę samą wartość skrótu (ang. *hash*). Dlatego `__eq__()` jest niezbędne do rozwiązania potencjalnych kolizji.

4.12. Protokoły konwersji

Czasami musisz przekonwertować obiekt na typ wbudowany, taki jak ciąg lub liczba. W tym celu można zdefiniować metody z tabeli 4.4.

Metoda `__str__()` jest wywoływana przez wbudowaną funkcję `str()` oraz przez funkcje związane z wyświetlaniem. Metoda `__format__()` jest wywoływana przez funkcję `format()` lub metodę `format()` ciągów. Argument `format_spec` to ciąg znaków zawierający specyfikację formatu. Ten ciąg jest taki sam jak argument `format_spec` funkcji `format()`. Na przykład:

```
f'{x:spec}'                 # Wywołuje x.__format__('spec')
format(x, 'spec')           # Wywołuje x.__format__('spec')
'x to {0:spec}'.format(x)   # Wywołuje x.__format__('spec')
```

Tabela 4.4. *Metody konwersji*

Metoda	Opis
`__str__(self)`	Konwersja na ciąg znaków
`__bytes__(self)`	Konwersja na bajty
`__format__(self, format_spec)`	Tworzy sformatowaną reprezentację
`__bool__(self)`	`bool(self)`
`__int__(self)`	`int(self)`
`__float__(self)`	`float(self)`
`__complex__(self)`	`complex(self)`
`__index__(self)`	Konwersja na indeks całkowity `[self]`

Składnia specyfikacji formatu jest dowolna i może być dostosowywana do obiektu. Istnieje jednak standardowy zestaw konwencji używanych dla typów wbudowanych. Więcej informacji na temat formatowania łańcuchów, w tym ogólny format specyfikatora, można znaleźć w rozdziale 9.

Metoda `__bytes__()` służy do tworzenia reprezentacji bajtowej, jeśli instancja jest przekazywana do `bytes()`. Nie wszystkie typy obsługują konwersję bajtów.

Oczekuje się, że konwersje numeryczne `__bool__()`, `__int__()`, `__float__()` i `__complex__()` dadzą wartość odpowiedniego typu wbudowanego.

Python nigdy nie wykonuje niejawnych konwersji typów przy użyciu tych metod. Tak więc nawet jeśli obiekt `x` implementuje metodę `__int__()`, wyrażenie `3 + x` nadal wygeneruje `TypeError`. Jedynym sposobem na wykonanie `__int__()` jest jawne użycie funkcji `int()`.

Metoda `__index__()` przeprowadza konwersję obiektu na liczbę całkowitą, gdy jest używana w operacji wymagającej podania wartości całkowitej. Obejmuje to indeksowanie w operacjach sekwencyjnych. Jeśli na przykład `items` jest listą, wykonanie operacji takiej jak `items[x]` będzie próbowało wykonać `items[x.__index__()]` (jeśli `x` nie jest liczbą całkowitą). `__index__()` jest również używana w różnych podstawowych konwersjach, takich jak `oct(x)` i `hex(x)`.

4.13. Protokół kontenera

Metody przedstawione w tabeli 4.5 są używane przez obiekty, które chcą zaimplementować kontenery różnego rodzaju: listy, słowniki, zbiory i tak dalej.

Tabela 4.5. *Metody dla kontenerów*

Metoda	Opis
`__len__(self)`	Zwraca długość `self`
`__getitem__(self, key)`	Zwraca `self[key]`
`__setitem__(self, key, value)`	Ustawia `self[key] = value`
`__delitem__(self, key)`	Usuwa `self[key]`
`__contains__(self, key)`	`obj in self`

Oto przykład:

```
a = [1, 2, 3, 4, 5, 6]
len(a)                  # a.__len__()
```

```
x = a[2]          # x = a.__getitem__(2)
a[1] = 7          # a.__setitem__(1,7)
del a[2]          # a.__delitem__(2)
5 in a            # a.__contains__(5)
```

Metoda `__len__()` jest wywoływana przez wbudowaną funkcję `len()` w celu zwrócenia nieujemnej długości. Ta funkcja określa również wartości logiczne, chyba że zdefiniowano także metodę `__bool__()`.

Aby uzyskać dostęp do poszczególnych elementów, metoda `__getitem__()` może zwrócić element według wartości klucza. Kluczem może być dowolny obiekt Pythona, ale oczekuje się, że będzie to liczba całkowita dla uporządkowanych sekwencji, takich jak listy i tablice. Metoda `__setitem__()` przypisuje wartość do elementu. Metoda `__delitem__()` jest wywoływana za każdym razem, gdy operacja `del` jest stosowana do pojedynczego elementu. Metoda `__contains__()` służy do implementacji operatora `in`.

Operacje wycinania, takie jak `x = s[i:j]`, są również implementowane za pomocą `__getitem__()`, `__setitem__()` i `__delitem__()`. W przypadku wycinków kluczem jest specjalna instancja `slice`. Ta instancja ma atrybuty opisujące zakres żądanego wycinka. Na przykład:

```
a = [1,2,3,4,5,6]
x = a[1:5]          # x = a.__getitem__(slice(1, 5, None))
a[1:3] = [10,11,12] # a.__setitem__(slice(1, 3, None), [10, 11, 12])
del a[1:4]          # a.__delitem__(slice(1, 4, None))
```

Wielu programistów nawet nie zdaje sobie sprawy, jak potężnym narzędziem są funkcje wycinania w Pythonie. Przykładowe zaawansowane techniki wycinania, przydatne do pracy z wielowymiarowymi strukturami danych, takimi jak macierze i tablice, obsługiwane przez Pythona:

```
a = m[0:100:10]           # Wycinanie krokowe (krok = 10)
b = m[1:10, 3:20]         # Wycinek wielowymiarowy
c = m[0:100:10, 50:75:5]  # Wiele wymiarów z krokami
m[0:5, 5:10] = n          # Rozszerzone przypisanie wycinka
del m[:10, 15:]           # Rozszerzone usuwanie wycinka
```

Ogólny format dla każdego wymiaru rozszerzonego wycinka to `i:j[:`*krok*`]`, gdzie *krok* jest opcjonalny. Podobnie jak w przypadku zwykłego wycinania, możesz pominąć początkowe lub końcowe wartości dla każdej części wycinka.

Ponadto wielokropek (`Ellipsis`; zapisany jako `...`) jest dostępny do oznaczenia dowolnej liczby końcowego lub wiodącego wymiaru w rozszerzonym wycinku:

```
a = m[..., 10:20] # Rozszerzony dostęp do wycinków za pomocą wielokropka
m[10:20, ...] = n
```

Podczas korzystania z rozszerzonych wycinków metody `__getitem__()`, `__setitem__()` i `__delitem__()` implementują odpowiednio dostęp, modyfikację i usuwanie. Jednak zamiast liczby całkowitej wartość przekazywana do tych metod jest krotką zawierającą kombinację obiektów `slice` lub `Ellipsis`. Na przykład:

```
a = m[0:10, 0:100:5, ...]
```

wywołuje `__getitem__()` w następujący sposób:

```
a = m.__getitem__((slice(0,10,None), slice(0,100,5), Ellipsis))
```

Ciągi, krotki i listy Pythona zapewniają obecnie wsparcie dla rozszerzonych wycinków. Żadna część Pythona ani jego standardowej biblioteki nie wykorzystuje wielowymiarowego wycinania ani `Ellipsis`. Te funkcje są zarezerwowane wyłącznie dla zewnętrznych bibliotek i frameworków. Miejscem, w którym można je najczęściej zobaczyć, jest zapewne biblioteka taka jak `numpy`.

4.14. Protokół iteracji

Jeśli instancja `obj` obsługuje iterację, udostępnia metodę `obj.__iter__()`, która zwraca iterator. Z kolei iterator `iter` implementuje pojedynczą metodę, `iter.__next__()`, która zwraca następny obiekt lub wywołuje `StopIteration`, aby zasygnalizować koniec iteracji. Te metody są używane przez implementację instrukcji `for` oraz inne operacje, które niejawnie wykonują iterację. Na przykład wyrażenie `for x in s` jest wykonywane w następujących krokach:

```
_iter = s.__iter__()
while True:
    try:
        x = _iter.__next__()
    except StopIteration:
       break
    # Wykonaj instrukcje w treści pętli for
    ...
```

Obiekt może opcjonalnie dostarczać odwrócony iterator, jeśli implementuje specjalną metodę `__reversed__()`. Ta metoda powinna zwrócić obiekt iteratora z tym samym interfejsem co normalny iterator (to znaczy z metodą `__next__()`, która wywołuje `StopIteration` na końcu iteracji). Ta metoda jest używana przez wbudowaną funkcję `reversed()`. Na przykład:

```
>>> for x in reversed([1,2,3]):
... print(x)
3
2
1
>>>
```

Powszechną techniką implementacji iteracji jest użycie funkcji generatora wykorzystującej instrukcję `yield`. Przykładowo:

```
class FRange:
    def __init__(self, start, stop, step):
        self.start = start
        self.stop = stop
        self.step = step

    def __iter__(self):
        x = self.start
        while x < self.stop:
            yield x
            x += self.step
```

```
# Przykładowe użycie:
nums = FRange(0.0, 1.0, 0.1)
for x in nums:
    print(x) # 0.0, 0.1, 0.2, 0.3, ...
```

Program działa, ponieważ funkcje generatora są zgodne z samym protokołem iteracji. Trochę łatwiej jest zaimplementować iterator w ten sposób, ponieważ musisz się zatroszczyć tylko o metodę `__iter__()`. Reszta maszynerii iteracyjnej jest już dostarczana przez generator.

4.15. Protokół atrybutów

Metody z tabeli 4.6 odczytują, zapisują i usuwają atrybuty obiektu za pomocą operatora kropki (`.`) i operatora `del`.

Tabela 4.6. *Metody dostępu do atrybutów*

Metoda	Opis
`__getattribute__(self, name)`	Zwraca atrybut `self.name`.
`__getattr__(self, name)`	Zwraca atrybut `self.name`, jeśli nie został znaleziony przez `__getattribute__()`.
`__setattr__(self, name, value)`	Ustawia atrybut `self.name = value`.
`__delattr__(self, name)`	Usuwa atrybut `del self.name`.

Przy każdym dostępie do atrybutu wywoływana jest metoda `__getattribute__()`. Jeśli atrybut zostanie zlokalizowany, zwracana jest jego wartość. W przeciwnym razie wywoływana jest metoda `__getattr__()`. Domyślnym zachowaniem `__getattr__()` jest zgłoszenie wyjątku `AttributeError`. Metoda `__setattr__()` jest zawsze wywoływana podczas ustawiania atrybutu, a w trakcie jego usuwania wywoływana jest metoda `__delattr__()`.

Metody te są dość proste, ponieważ pozwalają całkowicie przedefiniować dostęp do atrybutów dla wszystkich atrybutów. Klasy zdefiniowane przez użytkownika mogą definiować właściwości i deskryptory, które pozwalają na bardziej szczegółową kontrolę dostępu do atrybutów. Zostało to omówione w rozdziale 7.

4.16. Protokół funkcji

Obiekt może emulować funkcję, udostępniając metodę `__call__()`. Jeśli obiekt `x` udostępnia tę metodę, może ją wywołać jak funkcję. Oznacza to, że `x(arg1, arg2, ...)` wywołuje `x.__call__(arg1, arg2, ...)`.

Istnieje wiele wbudowanych typów, które obsługują wywołania funkcji. Na przykład typy implementują `__call__()` do tworzenia nowych instancji. Metody powiązane implementują `__call__()`, aby przekazać argument `self` do metod instancji. Funkcje biblioteczne, takie jak `functools.partial()`, również tworzą obiekty emulujące funkcje.

4.17. Protokół menedżera kontekstu

Instrukcja `with` umożliwia wykonanie sekwencji instrukcji pod kontrolą instancji znanej jako menedżer kontekstu. Ogólna składnia jest następująca:

```
with context [as var]:
    instrukcje
```

Oczekuje się, że pokazany tutaj obiekt kontekstowy zaimplementuje metody wymienione w tabeli 4.7.

Tabela 4.7. *Metody dla menedżerów kontekstu*

Metoda	Opis
`__enter__(self)`	Wywoływana podczas wchodzenia w nowy kontekst. Wartość zwracana jest umieszczana w zmiennej wymienionej ze specyfikatorem `as` w instrukcji `with`.
`__exit__(self, type, value, tb)`	Wywoływana podczas opuszczania kontekstu. Jeśli wystąpił wyjątek, `type`, `value` oraz `tb` zawierają typ wyjątku, wartość i informacje o śledzeniu.

Podczas wykonywania instrukcji `with` wywoływana jest metoda `__enter__()`. Wartość zwracana przez tę metodę jest umieszczana w zmiennej określonej opcjonalnym specyfikatorem `var`. Metoda `__exit__()` jest wywoływana, gdy tylko program opuści blok instrukcji skojarzony z instrukcją `with`. Jeśli został zgłoszony wyjątek, metoda `__exit__()` otrzymuje w argumentach bieżący typ wyjątku, wartość i informacje o śledzeniu. Jeśli nie są obsługiwane żadne błędy, wszystkie trzy wartości są ustawione na `None`. Metoda `__exit__()` powinna zwrócić `True` lub `False`, aby wskazać, czy zgłoszony wyjątek został obsłużony. Jeśli zwracana jest wartość `True`, każdy oczekujący wyjątek jest usuwany i wykonywanie programu jest kontynuowane normalnie z pierwszą instrukcją po bloku `with`.

Podstawowym zastosowaniem interfejsu zarządzania kontekstem jest umożliwienie uproszczonej kontroli zasobów obiektów obejmujących stan systemu, takich jak otwarte pliki, połączenia sieciowe i blokady. Implementując ten interfejs, obiekt może bezpiecznie wyczyścić zasoby, gdy wykonanie programu pozostawia kontekst, w którym obiekt jest używany. Dalsze szczegóły znajdują się w rozdziale 3.

4.18. Podsumowanie: pythoniczność

Często opisywanym celem projektowym jest napisanie kodu, który jest „pythoniczny”. Może to oznaczać wiele rzeczy, ale zasadniczo zachęca do podążania za ustalonymi idiomami używanymi w języku Python. Oznacza to znajomość protokołów Pythona dotyczących kontenerów, elementów iteracyjnych, zarządzania zasobami i tak dalej. Wiele najpopularniejszych frameworków Pythona korzysta z tych protokołów, aby zapewnić dobre wrażenia użytkownika. Ty też powinieneś do tego dążyć.

Spośród różnych protokołów trzy zasługują na szczególną uwagę ze względu na ich szerokie zastosowanie. Jednym z nich jest stworzenie odpowiedniej reprezentacji obiektu za pomocą metody `__repr__()`. Programy Pythona są często debugowane w środowisku interaktywnym REPL.

Powszechna jest również prezentacja obiektów za pomocą metody `print()` lub biblioteki logowania. Jeśli ułatwisz obserwację stanu swoich obiektów, wszystko stanie się łatwiejsze.

Poza tym iteracja po danych jest jednym z najczęstszych zadań programistycznych. Jeśli zamierzasz to zrobić, powinieneś sprawić, by Twój kod działał z instrukcją `for` Pythona. Wiele podstawowych części Pythona i biblioteki standardowej zostało zaprojektowanych do pracy z obiektami iterowalnymi. Wspierając iterację w zwykły sposób, automatycznie zyskasz dodatkowe funkcjonalności, a Twój kod będzie intuicyjny dla innych programistów.

Na koniec użyj menedżerów kontekstu i instrukcji `with` dla wspólnego wzorca programowania, w którym instrukcje są umieszczane pomiędzy początkowymi i końcowymi elementami — na przykład otwarciem i zamknięciem zasobów, założeniem i zwolnieniem blokady, subskrypcją i anulowaniem subskrypcji itd.

5

Funkcje

Funkcje są podstawowymi elementami konstrukcyjnymi większości programów Pythona. W tym rozdziale opisano ich definicje i zastosowanie, reguły określania zakresu, zamknięcia, dekoratory i inne funkcje programowania funkcjonalnego. Szczególną uwagę zwraca się na różne idiomy programowania, modele liczenia i związane z nimi wzorce.

5.1. Definicje funkcji

Funkcje definiuje się za pomocą instrukcji `def`:

```
def add(x, y):
    return x + y
```

Pierwsza część definicji funkcji określa nazwę funkcji i nazwy parametrów, które reprezentują wartości wejściowe. Treść funkcji to sekwencja instrukcji wykonywanych, gdy funkcja jest wywoływana lub stosowana. Aby przypisać funkcję do argumentów, należy podać jej nazwę z argumentami w nawiasie: `a = add(3, 4)`. Przed wykonaniem treści funkcji argumenty są w pełni liczone od lewej do prawej. Na przykład w przypadku `add(1+1, 2+2)` wyrażenie — przed wywołaniem funkcji — jest redukowane do postaci `add(2, 4)`. Nosi to nazwę *aplikacyjnego porządku liczenia*. Kolejność i liczba argumentów muszą być zgodne z parametrami podanymi w definicji funkcji. Jeśli istnieje niezgodność, zgłaszany jest wyjątek `TypeError`. Struktura wywołania funkcji (taka jak liczba wymaganych argumentów) jest określana sygnaturą wywołania funkcji.

5.2. Argumenty domyślne

Do parametrów funkcji można dołączyć wartości domyślne, przypisując wartości w definicji funkcji. Na przykład:

```
def split(line, delimiter=','):
    instrukcje
```

Gdy funkcja definiuje parametr z wartością domyślną, ten parametr i wszystkie następujące po nim parametry są opcjonalne. Nie można określić parametru bez wartości domyślnej po jakimkolwiek parametrze z wartością domyślną.

Domyślne wartości parametrów są liczone raz, gdy funkcja jest definiowana po raz pierwszy, a nie za każdym razem, gdy funkcja jest wywoływana. Często prowadzi to do zaskakującego zachowania, jeśli domyślnie używane są obiekty mutowalne:

```
def func(x, items=[]):
    items.append(x)
    return items

func(1) # Zwraca [1]
func(2) # Zwraca [1, 2]
func(3) # Zwraca [1, 2, 3]
```

Zwróć uwagę, jak domyślny argument zachowuje modyfikacje dokonane z poprzednich wywołań. Aby temu zapobiec, lepiej użyć wartości `None` i dodać sprawdzanie w następujący sposób:

```
def func(x, items=None):
    if items is None:
        items = []
    items.append(x)
    return items
```

Zgodnie z ogólną praktyką, aby uniknąć takich niespodzianek, używaj tylko niezmiennych obiektów dla domyślnych wartości argumentów: liczb, ciągów, wartości logicznych, `None` i tak dalej.

5.3. Argumenty wariadyczne (zmienna liczba argumentów)

Funkcja może akceptować zmienną liczbę argumentów, jeśli w nazwie ostatniego parametru występuje przedrostek w postaci gwiazdki (*). Na przykład:

```
def product(first, *args):
    result = first
    for x in args:
        result = result * x
    return result

    product(10, 20)       # -> 200
    product(2, 3, 4, 5)   # -> 120
```

W takim przypadku wszystkie dodatkowe argumenty są umieszczane w zmiennej `args` jako krotka. Następnie możesz pracować z argumentami przy użyciu standardowych operacji sekwencyjnych: iteracji, wycinania, rozpakowywania i tak dalej.

5.4. Argumenty słów kluczowych

Argumenty funkcji można podać jawnie, nazywając każdy parametr i określając wartość. Są to tak zwane argumenty słów kluczowych. Oto przykład:

```
def func(w, x, y, z):
    instrukcje

# Wywołanie argumentu słowa kluczowego
func(x=3, y=22, w='hello', z=[1, 2])
```

W przypadku argumentów słów kluczowych kolejność argumentów nie ma znaczenia, o ile każdy wymagany parametr ma jedną wartość. Jeśli pominiesz którykolwiek z wymaganych argumentów lub jeśli nazwa słowa kluczowego nie pasuje do żadnej z nazw parametrów w definicji funkcji, zostanie zgłoszony wyjątek `TypeError`. Argumenty słów kluczowych są liczone w tej samej kolejności, w jakiej zostały określone w wywołaniu funkcji.

Argumenty pozycyjne i argumenty słów kluczowych mogą się pojawiać w tym samym wywołaniu funkcji, pod warunkiem że wszystkie argumenty pozycyjne wystąpią jako pierwsze, podane są wartości dla wszystkich argumentów nieopcjonalnych, a żaden argument nie otrzyma więcej niż jednej wartości. Oto przykład:

```
func('hello', 3, z=[1, 2], y=22)
func(3, 22, w='hello', z=[1, 2] # TypeError. Wiele wartości dla w
```

W razie potrzeby można wymusić użycie argumentów słów kluczowych. Odbywa się to poprzez wypisanie parametrów po argumencie * lub po prostu przez uwzględnienie jednego * w definicji. Na przykład:

```
def read_data(filename, *, debug=False):
    ...

def product(first, *values, scale=1):
    result = first * scale
    for val in values:
        result = result * val
    return result
```

W tym przykładzie argument debugowania funkcji `read_data()` może być określony tylko za pomocą słowa kluczowego.

To ograniczenie często poprawia czytelność kodu:

```
data = read_data('Data.csv', True) # Błąd. TypeError
data = read_data('Data.csv', debug=True) # Nie ma błędu
```

Funkcja `product()` przyjmuje dowolną liczbę argumentów pozycyjnych i opcjonalny argument zawierający słowo kluczowe. Na przykład:

```
result = product(2,3,4)             # Wynik = 24
result = product(2,3,4, scale=10)   # Wynik = 240
```

5.5. Wariadyczne argumenty słów kluczowych

Jeśli ostatni argument definicji funkcji jest poprzedzony znakiem **, wszystkie dodatkowe argumenty słów kluczowych (te, które nie pasują do żadnych innych nazw parametrów) są umieszczane w słowniku i przekazywane do funkcji. Kolejność elementów w tym słowniku gwarantuje zgodność z kolejnością, w jakiej zostały podane argumenty słów kluczowych.

Arbitralne argumenty słów kluczowych mogą być przydatne do definiowania funkcji akceptujących dużą liczbę opcji konfiguracyjnych, które byłyby zbyt nieporęczne, aby wyświetlić je jako parametry. Oto przykład:

```
def make_table(data, **parms):
    # Pobierz parametry konfiguracyjne z parms (słownik)
    fgcolor = parms.pop('fgcolor', 'black')
    bgcolor = parms.pop('bgcolor', 'white')
    width = parms.pop('width', None)
    ...
    # Nie ma więcej opcji
    if parms:
        raise TypeError(f'Nieobsługiwane opcje {list(parms)}')

make_table(items, fgcolor='black', bgcolor='white', border=1,
                  borderstyle='grooved', cellpadding=10,
                  width=400)
```

Metoda `pop()` usuwa element ze słownika, zwracając możliwą wartość domyślną, jeśli nie jest zdefiniowana. Wyrażenie `parms.pop('fgcolor', 'black')` użyte w tym kodzie naśladuje zachowanie argumentu słowa kluczowego z określoną wartością domyślną.

5.6. Funkcje akceptujące wszystkie dane wejściowe

Używając * oraz **, możesz napisać funkcję, która akceptuje dowolną kombinację argumentów. Argumenty pozycyjne są przekazywane jako krotka, a argumenty słów kluczowych jako słownik. Na przykład:

```
# Zaakceptuj zmienną liczbę argumentów pozycyjnych lub słów kluczowych
def func(*args, **kwargs):
    # args jest krotką argumentów pozycyjnych
    # kwargs to słownik słów kluczowych args
...
```

`*args` i `**kwargs` często stosuje się łącznie do pisania nakładek, dekoratorów, proxy i podobnych funkcji. Załóżmy, że masz funkcję do analizowania wierszy tekstu pobranego z elementu iteracyjnego:

```
def parse_lines(lines, separator=',', types=(), debug=False):
    for line in lines:
        ...
        instrukcje
        ...
```

Załóżmy teraz, że zamiast tego chcesz utworzyć funkcję, która będzie przetwarzać dane z pliku określonego przez jego nazwę. Aby to zrobić, możesz napisać:

```
def parse_file(filename, *args, **kwargs):
    with open(filename, 'rt') as file:
        return parse_lines(file, *args, **kwargs)
```

Zaletą tego podejścia jest to, że funkcja `parse_file()` nie musi wiedzieć nic o argumentach funkcji `parse_lines()`. Przyjmuje wszelkie dodatkowe argumenty podane przez kod wywołujący funkcję i przekazuje je dalej. Upraszcza to również obsługę funkcji `parse_file()`. Jeśli na przykład do `parse_lines()` zostaną dodane nowe argumenty, będą one również działać z funkcją `parse_file()`.

5.7. Argumenty tylko pozycyjne

Wiele wbudowanych funkcji Pythona akceptuje tylko argumenty pozycyjne. Zobaczysz, że jest to wskazywane przez obecność ukośnika (/) w sygnaturze wywołującej funkcji pokazywanej przez różne narzędzia pomocy i IDE. Możesz na przykład zobaczyć coś takiego jak `func(x, y, /)`. Oznacza to, że wszystkie argumenty pojawiające się przed ukośnikiem mogą być określone tylko przez pozycję. Możesz zatem wywołać funkcję jako `func(2, 3)`, ale nie jako `func(x=2, y=3)`. Ta składnia może być również używana podczas definiowania funkcji. Możesz na przykład napisać:

```
def func(x, y, /):
    pass

func(1, 2) # OK
func(1, y=2) # Błąd
```

Ta forma definicji jest rzadko spotykana w kodzie, ponieważ po raz pierwszy była obsługiwana tylko w Pythonie 3.8. Może to być jednak przydatny sposób na uniknięcie potencjalnych konfliktów nazw między nazwami argumentów. Rozważmy na przykład następujący kod:

```
import time

def after(seconds, func, /, *args, **kwargs):
    time.sleep(seconds)
    return func(*args, **kwargs)

def duration(*, seconds, minutes, hours):
    return seconds + 60 * minutes + 3600 * hours

after(5, duration, seconds=20, minutes=3, hours=2)
```

W tym kodzie argument `seconds` jest przekazywany jako argument słowa kluczowego, ale jest przeznaczony do użycia z funkcją `duration`, która jest przekazywana do `after()`. Użycie argumentów tylko pozycyjnych w `after()` zapobiega kolizji nazw z argumentem `seconds`, który pojawia się jako pierwszy.

5.8. Nazwy, wpisy dokumentacyjne i wskazówki dotyczące typów

Standardowa konwencja nazewnictwa funkcji polega na używaniu małych liter z podkreśleniem (_) stosowanym jako separator wyrazów, na przykład `read_data()`, a nie `readData()`. Jeśli funkcja nie będzie używana bezpośrednio, ponieważ jest pomocnicza lub jest jakimś rodzajem wewnętrznego szczegółu implementacji, do jej nazwy zwykle dodawany jest pojedynczy znak podkreślenia, na przykład `_helper()`. To jednak tylko konwencja. Możesz dowolnie nazwać funkcję, o ile nazwa jest prawidłowym identyfikatorem.

Nazwę funkcji można uzyskać poprzez atrybut `__name__`. Czasami jest to przydatne do debugowania.

```
>>> def square(x):
...     return x * x
...
>>> square.__name__
'square'
>>>
```

Często pierwsza instrukcja funkcji jest wpisem dokumentacyjnym opisującym jej użycie. Na przykład:

```
def factorial(n):
    '''
    Oblicza silnię. Na przykład:

    >>> factorial(6)
    120
    >>>
    '''
    if n <= 1:
        return 1
    else:
        return n*factorial(n-1)
```

Wpisy dokumentacyjne są przechowywane w atrybucie `__doc__` funkcji. Jest on często używany przez IDE, aby zapewnić interaktywną pomoc.

Funkcje można również opisywać za pomocą wskazówek dla typu. Na przykład:

```
def factorial(n: int) -> int:
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)
```

Wskazówki dla typu nie zmieniają niczego w sposobie liczenia funkcji. Oznacza to, że obecność podpowiedzi nie zapewnia żadnych korzyści w zakresie wydajności ani dodatkowego sprawdzania błędów w czasie wykonywania. Wskazówki są jedynie przechowywane w atrybucie `__annotations__` funkcji, który jest słownikiem mapującym nazwy argumentów na podane wskazówki. Narzędzia innych firm, takie jak środowiska IDE i programy do sprawdzania kodu, mogą wykorzystywać wskazówki do różnych celów.

Czasami zobaczysz wskazówki dla typu dołączone do zmiennych lokalnych w funkcji. Na przykład:

```
def factorial(n:int) -> int:
    result: int = 1 # Zmienna lokalna ze wskazówką dla typu
    while n > 1:
        result *= n
        n -= 1
    return result
```

Takie wskazówki są całkowicie ignorowane przez interpreter. Nie są sprawdzane, przechowywane ani nawet liczone. Ponownie celem podpowiedzi jest pomoc narzędziom zewnętrznym do sprawdzania kodu. Nie zaleca się dodawania wskazówek dla typu do funkcji, chyba że aktywnie używasz narzędzi do sprawdzania kodu, które z nich korzystają. Łatwo określić niepoprawne wskazówki dotyczące typów i — jeśli nie używasz narzędzia, które je sprawdza — błędy pozostaną niewykryte, dopóki ktoś inny nie zdecyduje się na uruchomienie takiego narzędzia.

5.9. Zastosowanie funkcji i przekazywanie parametrów

Gdy funkcja jest wywoływana, jej parametry są nazwami lokalnymi, które są powiązane z przekazanymi obiektami wejściowymi. Python przekazuje dostarczone obiekty do funkcji bez zmian — bez dodatkowego kopiowania. Należy zachować ostrożność, jeśli przekazywane są obiekty mutowalne, takie jak listy lub słowniki. Jeśli zostaną wprowadzone zmiany, będą one odzwierciedlone w oryginalnym obiekcie. Oto przykład:

```
def square(items):
    for i, x in enumerate(items):
        items[i] = x * x            # Modyfikuje elementy w miejscu

a = [1, 2, 3, 4, 5]
square(a)                           # Zmienia a na [1, 4, 9, 16, 25]
```

Mówi się, że funkcje, które mutują swoje wartości wejściowe lub zmieniają stan innych części programu w tle, mają „efekty uboczne". Zasadniczo najlepiej unikać skutków ubocznych. Mogą się stać źródłem subtelnych błędów programistycznych w miarę wzrostu rozmiaru i złożoności programów — czasami może to nie być oczywiste, czy funkcja ma skutki uboczne, czy nie. Takie funkcje również słabo współdziałają z programami obejmującymi wątki i współbieżność, ponieważ efekty uboczne zwykle muszą być chronione przez blokady.

Ważne jest, aby odróżnić modyfikację obiektu od ponownego przypisania nazwy zmiennej. Przyjrzyj się tej funkcji:

```
def sum_squares(items):
    items = [x*x for x in items] # Ponowne przypisanie do "items"
    return sum(items)

a = [1, 2, 3, 4, 5]
result = sum_squares(a)
print(a)                            # [1, 2, 3, 4, 5] (Bez zmian)
```

W tym przykładzie wygląda na to, że funkcja `sum_squares()` może nadpisywać zmienną przekazaną `items`. Tak, lokalna etykieta `items` jest ponownie przypisywana do nowej wartości. Ale pierwotna wartość wejściowa (`a`) nie jest zmieniana przez tę operację. Zamiast tego lokalna nazwa zmiennej `items` jest powiązana z zupełnie innym obiektem — wynikiem wewnętrznej listy. Jest różnica między przypisaniem nazwy zmiennej a modyfikacją obiektu. Kiedy przypisujesz wartość do nazwy, nie zastępujesz obiektu, który już tam był — po prostu przypisujesz nazwę do innego obiektu.

Stylistycznie funkcje z efektami ubocznymi często zwracają w wyniku `None`. Jako przykład zastanówmy się nad metodą `sort()` listy:

```
>>> items = [10, 3, 2, 9, 5]
>>> items.sort()       # Obserwuj: brak zwracanej wartości
>>> items
[2, 3, 5, 9, 10]
>>>
```

Metoda `sort()` wykonuje sortowanie elementów listy w miejscu. Nie zwraca żadnego wyniku. Brak wyniku jest silnym wskaźnikiem efektu ubocznego — w tym przypadku elementy listy uległy przestawieniu.

Czasami masz już w sekwencji lub mapowaniu dane, które chcesz przekazać do funkcji. Aby to zrobić, możesz użyć * i ** w wywołaniach funkcji. Na przykład:

```
def func(x, y, z):
    ...
s = (1, 2, 3)
# Przekaż sekwencję jako argumenty
result = func(*s)

# Przekaż mapowanie jako argumenty słów kluczowych
d = {'x': 1, 'y': 2, 'z': 3}
result = func(**d)
```

Możesz pobierać dane z wielu źródeł lub nawet podawać niektóre argumenty jawnie i wszystko będzie działać, o ile funkcja otrzyma wszystkie wymagane argumenty, nie ma zduplikowanych elementów, a wszystko w sygnaturze wywołania jest odpowiednio dopasowane. Możesz nawet użyć * i ** więcej niż raz w tym samym wywołaniu funkcji. Jeśli brakuje argumentu lub określisz zduplikowane wartości dla argumentu, otrzymasz błąd. Python nigdy nie pozwoli Ci wywołać funkcji z argumentami, które nie spełniają jej sygnatury.

5.10. Zwracane wartości

Instrukcja `return` zwraca wartość z funkcji. Jeśli nie określono żadnej wartości lub pominięto instrukcję `return`, zwracana jest wartość `None`. Aby zwrócić wiele wartości, umieść je w krotce:

```
def parse_value(text):
    '''
    Podziel tekst formularza name=val na wartości (name, val)
    '''
    parts = text.split('=', 1)
    return (parts[0].strip(), parts[1].strip())
```

Wartości zwracane w krotce można rozpakować do poszczególnych zmiennych:

```
name, value = parse_value('url=http://www.python.org')
```

Czasami alternatywnie są używane nazwane krotki:

```
from typing import NamedTuple

class ParseResult(NamedTuple):
    name: str
    value: str

def parse_value(text):
    '''
    Podziel tekst formularza name=val na wartości (name, val)
    '''
    parts = text.split('=', 1)
    return ParseResult(parts[0].strip(), parts[1].strip())
```

Nazwana krotka działa tak samo jak zwykła krotka (możesz wykonać te same operacje i przeprowadzić rozpakowanie), ale możesz również odwoływać się do zwracanych wartości za pomocą nazwanych atrybutów:

```
r = parse_value('url=http://www.python.org')
print(r.name, r.value)
```

5.11. Obsługa błędów

Jednym z problemów z funkcją `parse_value()`, opisaną w poprzedniej sekcji, jest obsługa błędów. Jakie działania należy podjąć, jeśli tekst wejściowy jest błędny i nie można zwrócić prawidłowego wyniku?

Jednym z podejść jest traktowanie wyniku jako opcjonalnego — to znaczy, że funkcja działa i albo zwraca odpowiedź, albo zwraca `None`, co jest powszechnie używane do wskazania brakującej wartości. Na przykład funkcję można zmodyfikować w następujący sposób:

```
def parse_value(text):
    parts = text.split('=', 1)
    if len(parts) == 2:
        return ParseResult(parts[0].strip(), parts[1].strip())
    else:
        return None
```

W tym projekcie ciężar sprawdzenia opcjonalnego wyniku spoczywa na kodzie wywołującym funkcję:

```
result = parse_value(text)
if result:
    name, value = result
```

W Pythonie 3.8+ występuje bardziej zwięzłe rozwiązanie:

```
if result := parse_value(text):
    name, value = result
```

Zamiast zwracać None, możesz potraktować nieprawidłowy tekst jako błąd, wywołując wyjątek. Na przykład:

```
def parse_value(text):
    parts = text.split('=', 1)
    if len(parts) == 2:
        return ParseResult(parts[0].strip(), parts[1].strip())
    else:
        raise ValueError('Nieprawidłowa wartość')
```

W takim przypadku kod wywołujący ma możliwość obsługi złych wartości za pomocą try-except. Na przykład:

```
try:
    name, value = parse_value(text)
    ...
except ValueError:
    ...
```

Decyzja, czy skorzystać z wyjątku, czy nie, nie zawsze jest jednoznaczna. Z reguły wyjątki są częstszym sposobem radzenia sobie z nieprawidłowym wynikiem. Jednak obsługa wyjątków sporo kosztuje, jeśli często występują. Jeśli piszesz kod, w którym wydajność ma znaczenie, lepszym rozwiązaniem może być zwrócenie wartości None, False, -1 lub innej specjalnej wartości wskazującej na niepowodzenie.

5.12. Zasady określania zakresu

Za każdym razem, gdy funkcja jest wykonywana, tworzona jest lokalna przestrzeń nazw (ang. *namespace*). Ta przestrzeń nazw to środowisko, które zawiera nazwy i wartości parametrów funkcji, a także wszystkie zmienne, które są przypisane w treści funkcji. Przypisywanie nazw jest znane z góry, gdy funkcja jest zdefiniowana, a wszystkie nazwy przypisane w treści funkcji są powiązane ze środowiskiem lokalnym. Wszystkie inne nazwy, które są używane, ale nie są przypisane w treści funkcji (wolne zmienne), są dynamicznie znajdowane w globalnej przestrzeni nazw, zawsze będącej głównym modułem, w którym zdefiniowano funkcję.

Istnieją dwa rodzaje błędów związanych z nazwami, które mogą wystąpić podczas wykonywania funkcji. Wyszukiwanie niezdefiniowanej nazwy wolnej zmiennej w środowisku globalnym skutkuje wystąpieniem wyjątku NameError. Wyszukiwanie zmiennej lokalnej, której nie przypisano jeszcze wartości, skutkuje wystąpieniem wyjątku UnboundLocalError. Ten ostatni błąd jest często wynikiem błędów kontroli przepływu. Na przykład:

```
def func(x):
    if x > 0:
        y = 42
    return x + y     # y nie zostaje przypisane, jeśli warunek jest fałszywy

func(10)     # Zwraca 52
func(-10)    # UnboundLocalError: y przywołane przed przypisaniem
```

UnboundLocalError jest również czasami spowodowany nieostrożnym użyciem operatorów przypisania w miejscu. Instrukcja taka jak n += 1 jest obsługiwana jako n = n + 1. Jeśli zostanie użyta przed przypisaniem n wartości początkowej, nie powiedzie się.

```
def func():
    n += 1 # Błąd: UnboundLocalError
```

Należy podkreślić, że nazwy zmiennych nigdy nie zmieniają swojego zakresu — są to albo zmienne globalne, albo zmienne lokalne, co jest określane w czasie definiowania funkcji.

Oto przykład, który to ilustruje:

```
x = 42
def func():
    print(x) # Błąd: UnboundLocalError
    x = 13
func()
```

Wydaje się, że w tym przykładzie funkcja `print()` wyświetla wartość zmiennej globalnej `x`. Jednak przypisanie `x`, które pojawia się później, oznacza `x` jako zmienną lokalną. Błąd jest wynikiem dostępu do zmiennej lokalnej, której nie przypisano jeszcze wartości.

Jeśli usuniesz funkcję `print()`, otrzymasz kod, który wygląda tak, jakby mógł ponownie przypisywać wartość zmiennej globalnej. Rozważmy na przykład to:

```
x = 42
def func():
    x = 13
func()
# x to nadal 42
```

Kiedy ten kod jest wykonywany, `x` zachowuje swoją wartość `42`, pomimo pozorów, że może modyfikować zmienną globalną `x` z wnętrza funkcji `func`. Gdy zmienne są przypisane wewnątrz funkcji, są zawsze powiązane jako zmienne lokalne; w rezultacie zmienna `x` w treści funkcji odnosi się do całkowicie nowego obiektu zawierającego wartość `13`, a nie do zmiennej zewnętrznej. Aby zmienić to zachowanie, użyj instrukcji `global`. Słowo `global` deklaruje nazwy jako należące do globalnej przestrzeni nazw i jest to konieczne, gdy zmienna globalna wymaga modyfikacji. Oto przykład:

```
x = 42
y = 37
def func():
    global x      # 'x' znajduje się w globalnej przestrzeni nazw
    x = 13
    y = 0
func()
# x to teraz 13; y to nadal 37
```

Należy zauważyć, że użycie wyrażenia `global` jest zwykle uważane za kiepskie rozwiązanie w Pythonie. Jeśli piszesz kod, w którym funkcja musi zakulisowo zmieniać stan, rozważ użycie definicji klasy i zmodyfikuj stan, zmieniając instancję lub zmienną klasy. Na przykład:

```
class Config:
    x = 42

def func():
    Config.x = 13
```

Python umożliwia korzystanie z zagnieżdżonych definicji funkcji. Oto przykład:

```
def countdown(start):
    n = start
    def display():     # Definicja funkcji zagnieżdżonej
        print('T-minus', n)
    while n > 0:
        display()
        n -= 1
```

Zmienne w funkcjach zagnieżdżonych są określane za pomocą zakresu leksykalnego. Oznacza to, że nazwy są rozwiązywane najpierw w zakresie lokalnym, a następnie w kolejnych wyższych zakresach, od zakresu wewnętrznego do zakresu zewnętrznego. Ponownie, nie jest to proces dynamiczny — wiązanie nazw jest określane raz w czasie definiowania funkcji na podstawie składni. Podobnie jak w przypadku zmiennych globalnych, funkcje wewnętrzne nie mogą ponownie przypisać wartości zmiennej lokalnej zdefiniowanej w funkcji zewnętrznej. Na przykład ten kod nie działa:

```
def countdown(start):
    n = start
    def display():
        print('T-minus', n)
    def decrement():
        n -= 1 # Błąd: UnboundLocalError
    while n > 0:
        display()
        decrement()
```

Aby naprawić błąd, możesz zadeklarować `n` jako zmienną `nonlocal` w następujący sposób:

```
def countdown(start):
    n = start
    def display():
        print('T-minus', n)
    def decrement():
        nonlocal n
        n -= 1 # Modyfikuje zewnętrzne n
while n > 0:
    display()
decrement()
```

`nonlocal` nie może być używane do odwoływania się do zmiennej globalnej — musi odwoływać się do zmiennej lokalnej w zewnętrznym zakresie. Jeśli więc funkcja przypisuje funkcję globalną, powinieneś nadal używać deklaracji globalnej, jak opisano wcześniej.

Używanie funkcji zagnieżdżonych i deklaracji `nonlocal` nie jest powszechnie stosowane w programowaniu. Na przykład funkcje wewnętrzne nie mają widoczności na zewnątrz, co może skomplikować testowanie i debugowanie. Niemniej funkcje zagnieżdżone są czasami przydatne do dzielenia złożonych obliczeń na mniejsze części i ukrywania wewnętrznych szczegółów implementacji.

5.13. Rekurencja

Python obsługuje funkcje rekurencyjne. Na przykład:

```
def sumn(n):
    if n == 0:
        return 0
    else:
        return n + sumn(n-1)
```

Istnieje jednak ograniczenie głębokości wywołań dla funkcji rekurencyjnych. Funkcja `sys.getrecursionlimit()` zwraca bieżącą maksymalną głębokość rekurencji, a do zmiany wartości można użyć funkcji `sys.setrecursionlimit()`. Wartość domyślna to 1000. Chociaż można zwiększyć tę wartość, programy nadal są ograniczone przez rozmiar stosu wymuszony przez system operacyjny hosta. Po przekroczeniu limitu głębokości rekursji zgłaszany jest wyjątek `RuntimeError`. Jeśli limit zostanie zbytnio zwiększony, Python może ulec awarii z błędem segmentacji (ang. *segmentation fault*) lub innym błędem systemu operacyjnego.

W praktyce problemy z limitem rekurencji pojawiają się tylko podczas pracy z głęboko zagnieżdżonymi rekurencyjnymi strukturami danych, takimi jak drzewa i grafy. Wiele algorytmów wykorzystujących drzewa w naturalny sposób nadaje się do rozwiązań rekurencyjnych — a jeśli struktura danych jest zbyt duża, możesz przekroczyć limit stosu. Istnieje jednak kilka sprytnych obejść; zobacz rozdział 6. na temat generatorów.

5.14. Wyrażenie lambda

Anonimową — nienazwaną — funkcję można zdefiniować za pomocą wyrażenia `lambda`:

```
lambda args: expression
```

`args` to rozdzielona przecinkami lista argumentów, a `expression` to wyrażenie zawierające te argumenty. Oto przykład:

```
a = lambda x, y: x + y
r = a(2, 3)             # r otrzymuje wartość 5
```

Kod zdefiniowany za pomocą `lambda` musi być prawidłowym wyrażeniem. W wyrażeniu `lambda` nie można skorzystać ze złożonych instrukcji lub instrukcji takich jak `try` i `while`. Wyrażenia `lambda` podlegają tym samym regułom określania zakresu co funkcje.

Jednym z głównych zastosowań lambdy jest definiowanie małych funkcji zwrotnych. Możesz choćby zobaczyć, że jest używana z wbudowanymi operacjami, takimi jak `sorted()`. Na przykład:

```
# Sortuj listę słów według liczby unikalnych liter
result = sorted(words, key=lambda word: len(set(word)))
```

Należy zachować ostrożność, gdy wyrażenie `lambda` zawiera wolne zmienne (nieokreślone jako parametry). Rozważ ten przykład:

```
x = 2
f = lambda y: x * y
```

```
x = 3
g = lambda y: x * y
print(f(10))          # --> Wyświetla 30
print(g(10))          # --> Wyświetla 30
```

W tym przykładzie możesz oczekiwać, że wywołanie `f(10)` wypisze `20`, odzwierciedlając fakt, że `x` było równe `2` w momencie definiowania. Tak jednak nie jest. Będąc zmienną wolną ewaluacja `f(10)` wykorzystuje jakąkolwiek wartość, którą ma `x` akurat w momencie liczenia. Może się ona różnić od wartości, którą miało podczas definiowania funkcji `lambda`. Czasami to zachowanie jest określane jako *późne wiązanie*.

Jeśli ważne jest, aby uchwycić wartość zmiennej w momencie definiowania, użyj domyślnego argumentu:

```
x = 2
f = lambda y, x=x: x * y
x = 3
g = lambda y, x=x: x * y
print(f(10))              # --> Wyświetla 20
print(g(10))              # --> Wyświetla 30
```

To działa, ponieważ domyślne wartości argumentów są liczone tylko podczas definiowania funkcji, a zatem przechwytują bieżącą wartość `x`.

5.15. Funkcje wyższego rzędu

Python obsługuje koncepcję *funkcji wyższego rzędu*. Oznacza to, że funkcje mogą być przekazywane jako argumenty do innych funkcji, umieszczane w strukturach danych i zwracane przez funkcję w wyniku. Mówi się, że funkcje są obiektami pierwszej klasy, co oznacza, że nie ma różnicy między sposobem obsługi funkcji a jakimkolwiek innym rodzajem danych. Oto przykład funkcji, która akceptuje inną funkcję jako dane wejściowe i wywołuje ją z opóźnieniem, na przykład w celu emulacji wydajności mikrousługi w chmurze:

```
import time

def after(seconds, func):
    time.sleep(seconds)
    func()

# Przykład użycia
def greeting():
    print('Witaj, Świecie')

after(10, greeting)  # Wyświetla 'Witaj, świecie" po 10 sekundach
```

Tutaj argument `func` funkcji `after()` jest przykładem tak zwanej *funkcji zwrotnej* (ang. *callback function*). Odnosi się to do faktu, że funkcja `after()` „odwołuje się" do funkcji podanej jako argument.

Gdy funkcja jest przekazywana jako dane, niejawnie przenosi informacje związane ze środowiskiem, w którym funkcja została zdefiniowana. Załóżmy na przykład, że funkcja `greeting()` wykorzystuje zmienną taką jak ta:

```
def main():
    name = 'Guido'
    def greeting():
        print('Witaj, ', name)
    after(10, greeting) # Wyświetla 'Witaj, Guido'

main()
```

W tym przykładzie nazwa zmiennej jest używana przez `greeting()`, ale jest to zmienna lokalna zewnętrznej funkcji `main()`. Gdy `greeting` jest przekazywane do `after()`, funkcja zapamiętuje swoje środowisko i używa wartości zmiennej `name`. Opiera się to na funkcji znanej jako *domknięcie* (ang. *closure*). Domknięcie to funkcja, która wraz ze środowiskiem zawiera wszystkie zmienne potrzebne do wykonania treści funkcji.

Domknięcia i funkcje zagnieżdżone są przydatne podczas pisania kodu opartego na koncepcji opóźnionego przetwarzania. Przedstawiona powyżej funkcja `after()` jest ilustracją tej koncepcji. Otrzymuje funkcję, która nie jest przetwarzana od razu — pojawia się dopiero w późniejszym czasie. Jest to powszechny wzorzec programowania, który pojawia się w innych kontekstach. Na przykład program może mieć funkcje, które są wykonywane tylko w odpowiedzi na zdarzenia: naciśnięcia klawiszy, ruch myszy, nadejście pakietów sieciowych i tak dalej. We wszystkich tych przypadkach przetwarzanie funkcji jest odraczane do momentu, gdy wydarzy się coś interesującego. Gdy funkcja zostanie ostatecznie wykonana, domknięcie zapewnia, że funkcja otrzyma wszystko, czego potrzebuje.

Możesz także pisać funkcje, które tworzą i zwracają inne funkcje. Na przykład:

```
def make_greeting(name):
    def greeting():
        print('Witaj, ', name)
    return greeting

f = make_greeting('Guido')
g = make_greeting('Ada')

f()     # Wyświetla 'Witaj, Guido'
g()     # Wyświetla 'Witaj, Ada'
```

W tym przykładzie funkcja `make_greeting()` nie wykonuje żadnych interesujących obliczeń. Zamiast tego tworzy i zwraca funkcję `greeting()`, która wykonuje rzeczywistą pracę. Dzieje się tak tylko wtedy, gdy ta funkcja zostanie później przetworzona.

W tym przykładzie dwie zmienne `f` i `g` zawierają dwie różne wersje funkcji `greeting()`. Mimo że funkcja `make_greeting()`, która utworzyła te funkcje, nie jest już wykonywana, funkcje `greeting()` nadal pamiętają zdefiniowaną zmienną `name` — jest to część domknięcia każdej funkcji.

Ważną uwagą dotyczącą domknięć jest to, że wiązanie z nazwami zmiennych nie jest „migawką”, ale procesem dynamicznym — co oznacza, że domknięcie wskazuje na zmienną nazwy i wartość, która została jej ostatnio przypisana. Oto przykład ilustrujący, gdzie mogą się pojawić problemy:

```
def make_greetings(names):
    funcs = []
```

```
    for name in names:
        funcs.append(lambda: print('Witaj, ', name))
    return funcs

# Wypróbuj to
a, b, c = make_greetings(['Guido', 'Ada', 'Margaret'])
a()    # Wyświetla 'Witaj, Margaret'
b()    # Wyświetla 'Witaj, Margaret'
c()    # Wyświetla 'Witaj, Margaret'
```

W tym przykładzie tworzona jest lista różnych funkcji (przy użyciu `lambda`). Może się wydawać, że wszystkie używają unikalnej wartości nazwy, ponieważ zmienia się ona przy każdej iteracji pętli `for`. Tak jednak nie jest. Wszystkie funkcje używają tej samej wartości `name` — wartości, jaką ma, gdy zwracana jest zewnętrzna funkcja `make_greetings()`.

Najprawdopodobniej nie takiego działania oczekujesz. Jeśli chcesz przechwycić kopię zmiennej, przechwyć ją jako domyślny argument, jak opisano wcześniej:

```
def make_greetings(names):
    funcs = []
    for name in names:
        funcs.append(lambda name=name: print('Witaj, ', name))
    return funcs

# Wypróbuj to
a, b, c = make_greetings(['Guido', 'Ada', 'Margaret'])
a()       # Wyświetla 'Witaj, Guido'
b()       # Wyświetla 'Witaj, Ada'
c()       # Wyświetla 'Witaj, Margaret'
```

W ostatnich dwóch przykładach funkcje zostały zdefiniowane za pomocą `lambda`. To rozwiązanie jest często stosowane jako skrót do tworzenia małych funkcji zwrotnych. Nie jest to jednak ścisły wymóg.

Powyższy przykład mogłeś napisać w ten sposób:

```
def make_greetings(names):
    funcs = []
    for name in names:
        def greeting(name=name):
            print('Witaj, ', name)
        funcs.append(greeting)
return funcs
```

Wybór, kiedy i gdzie użyć `lambda`, zależy od osobistych preferencji i jest kwestią przejrzystości kodu. Jeśli sprawia, że kod jest trudniejszy do odczytania, być może należy tego unikać.

5.16. Przekazywanie argumentów w funkcjach zwrotnych

Jednym z większych problemów związanych z funkcjami zwrotnymi jest przekazywanie argumentów do dostarczonej funkcji. Przyjrzyj się napisanej wcześniej funkcji `after()`:

```
import time

def after(seconds, func):
    time.sleep(seconds)
    func()
```

W tym kodzie funkcja `func()` jest na stałe wywoływana bez argumentów. Jeśli chcesz przekazać dodatkowe argumenty, możesz spróbować tego:

```
def add(x, y):
    print(f'{x} + {y} -> {x+y}')
    return x + y

after(10, add(2, 3)) # Błąd: jest natychmiast wywoływana funkcja add()
```

W tym przykładzie funkcja `add(2, 3)` działa natychmiast, zwracając wartość `5`. Funkcja `after()` ulega awarii 10 sekund później, gdy próbuje wykonać `5()`. To zdecydowanie nie to, czego oczekiwałeś. Jednak wydaje się, że nie ma oczywistego sposobu, aby takie rozwiązanie zadziałało, jeśli `add()` zostanie wywołana z żądanymi argumentami.

Problem ten wskazuje na większą trudność projektową dotyczącą wykorzystania funkcji i ogólnie programowania funkcyjnego — kompozycji funkcji. Gdy funkcje są ze sobą mieszane na różne sposoby, należy pomyśleć o tym, jak wejścia i wyjścia funkcji łączą się ze sobą. Nie zawsze jest to proste.

W tym przypadku jednym z rozwiązań jest umieszczenie obliczeń w funkcji zeroargumentowej za pomocą `lambda`. Na przykład:

```
after(10, lambda: add(2, 3))
```

Taka mała funkcja z zerowym argumentem jest czasami nazywana *thunk*. Zasadniczo jest to wyrażenie, które zostanie przetworzone później, gdy w końcu zostanie wywołane jako funkcja z zerowym argumentem. Może to być ogólny sposób na opóźnienie przetwarzania dowolnego wyrażenia: umieść wyrażenie w `lambda` i wywołaj funkcję, gdy faktycznie potrzebujesz wartości.

Alternatywnym rozwiązaniem dla `lambda` jest skorzystanie z `functools.partial()` — aby utworzyć częściowo przetworzoną funkcję, na przykład:

```
from functools import partial
after(10, partial(add, 2, 3))
```

`partial()` tworzy funkcje wywoływalne, dla których został już określony i zapamiętany co najmniej jeden argument. Może to być użyteczny sposób, aby niezgodne funkcje były zgodne z oczekiwanymi sygnaturami wywołań w wywołaniach zwrotnych i innych aplikacjach. Oto kilka innych przykładów użycia funkcji `partial()`:

```
def func(a, b, c, d):
    print(a, b, c, d)

f = partial(func, 1, 2)         # Ustawia a=1, b=2
f(3, 4)                         # func(1, 2, 3, 4)
f(10, 20)                       # func(1, 2, 10, 20)
```

```
g = partial(func, 1, 2, d=4)    # Ustawia a=1, b=2, d=4
g(3)                            # func(1, 2, 3, 4)
g(10)                           # func(1, 2, 10, 4)
```

`partial()` i `lambda` mogą być używane do podobnych celów, ale istnieje ważne rozróżnienie semantyczne między tymi dwiema technikami. Dzięki `partial()` argumenty są przetwarzane i wiązane w momencie, gdy funkcja `partial` jest po raz pierwszy zdefiniowana. W przypadku `lambda` z zerowym argumentem argumenty są przetwarzane i wiązane, gdy funkcja `lambda` faktycznie wykonuje się później (przetwarzanie wszystkiego jest opóźnione). Na przykład:

```
>>> def func(x, y):
...     return x + y
...
>>> a = 2
>>> b = 3
>>> f = lambda: func(a, b)
>>> g = partial(func, a, b)
>>> a = 10
>>> b = 20
>>> f()       # Używa bieżących wartości a, b
30
>>> g()       # Używa początkowych wartości a, b
5
>>>
```

Ponieważ funkcja `partial()` jest w pełni przetwarzana, obiekty wywoływalne tworzone przez `partial()` są obiektami, które mogą być serializowane do bajtów, zapisywane w plikach, a nawet przesyłane przez połączenia sieciowe (na przykład przy użyciu modułu standardowej biblioteki `pickle`). Nie jest to możliwe w przypadku funkcji `lambda`. Tak więc w aplikacjach, w których funkcje są przekazywane — prawdopodobnie do interpreterów Pythona działających w różnych procesach lub na różnych komputerach — okaże się, że funkcja `partial()` jest nieco bardziej elastyczna. Na marginesie: funkcja `partial()` jest ściśle związana z koncepcją znaną jako *rozwijanie funkcji* (ang. *currying*).

Jest to technika programowania funkcjonalnego, w której funkcja wieloargumentowa jest wyrażona jako łańcuch zagnieżdżonych funkcji jednoargumentowych. Oto przykład:

```
# Funkcja trójargumentowa
def f(x, y, z):
    return x + y + z

# Wersja z rozwijaniem funkcji
def fc(x):
    return lambda y: (lambda z: x + y + z)

# Przykładowe zastosowanie
a = f(2, 3, 4)      # Funkcja trójargumentowa
b = fc(2)(3)(4)     # Wersja z rozwijaniem funkcji
```

Nie jest to typowy styl programowania wykorzystywany w Pythonie, ale istnieje kilka praktycznych powodów, aby go stosować. Czasami z pewnością usłyszysz słowo „currying" w rozmowach z programistami, którzy spędzili zbyt dużo czasu na wypaczaniu swoich mózgów takimi rzeczami jak `lambda`. Ta technika radzenia sobie z wieloma argumentami

została nazwana na cześć słynnego logika Haskella Curry'ego. Wiedza na ten temat może być przydatna — zwłaszcza jeśli natkniesz się na grupę programistów funkcjonalnych, którzy podczas imprezy towarzyskiej toczą gorącą dyskusję.

Wróćmy jednak do pierwotnego problemu z przekazywaniem argumentów. Inną opcją przekazywania argumentów do funkcji zwrotnej jest dostarczanie ich osobno jako argumentów do zewnętrznej funkcji wywołującej. Przyjrzyj się tej wersji funkcji `after()`:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    func(*args)

after(10, add, 2, 3) # Wywołanie add(2, 3) po 10 sekundach
```

Zauważysz, że przekazywanie argumentów słów kluczowych do funkcji `func()` nie jest obsługiwane. Jest to zgodne z projektem. Jednym z problemów związanych z argumentami słów kluczowych jest to, że nazwy argumentów danej funkcji mogą kolidować z nazwami już używanymi (czyli `seconds` i `func`). Argumenty słów kluczowych mogą być również zarezerwowane do określania opcji samej funkcji `after()`. Na przykład:

```
def after(seconds, func, *args, debug=False):
    time.sleep(seconds)
    if debug:
        print('Wywołanie', func, args)
    func(*args)
```

Jednak nie wszystko stracone. Jeśli musisz określić argumenty słów kluczowych do funkcji `func()`, nadal możesz to zrobić za pomocą `partial()`. Na przykład:

```
after(10, partial(add, y=3), 2)
```

Jeśli chcesz, aby funkcja `after()` akceptowała argumenty słów kluczowych, bezpiecznym sposobem na to może być użycie argumentów tylko pozycyjnych. Na przykład:

```
def after(seconds, func, debug=False, /, *args, **kwargs):
    time.sleep(seconds)
    if debug:
        print('Wywołanie', func, args, kwargs)
    func(*args, **kwargs)

after(10, add, 2, y=3)
```

Innym potencjalnie niepokojącym spostrzeżeniem jest to, że funkcja `after()` w rzeczywistości reprezentuje dwa różne połączone wywołania funkcji. Być może problem przekazywania argumentów można rozłożyć na dwie funkcje w ten sposób:

```
def after(seconds, func, debug=False):
    def call(*args, **kwargs):
        time.sleep(seconds)
        if debug:
            print('Wywołanie', func, args, kwargs)
        func(*args, **kwargs)
    return call

after(10, add)(2, y=3)
```

Teraz nie ma żadnych konfliktów między argumentami funkcji `after()` i argumentami funkcji `func`. Istnieje jednak możliwość, że spowoduje to konflikt między Tobą a Twoimi współpracownikami.

5.17. Zwracanie wyników z wywołań zwrotnych

Innym problemem, który nie został poruszony w poprzedniej sekcji, jest zwracanie wyników obliczeń. Zastanów się nad tą zmodyfikowaną funkcją `after()`:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    return func(*args)
```

To działa, ale w niektórych przypadkach mogą wystąpić problemy, wynikające z faktu, że zaangażowane są dwie oddzielne funkcje: sama funkcja `after()` i dostarczona funkcja wywołania zwrotnego.

Jedna kwestia dotyczy obsługi wyjątków. Na przykład wypróbuj te dwa przykłady:

```
after("1", add, 2, 3) # Błąd: TypeError (oczekiwana liczba całkowita)
after(1, add, "2", 3) # Błąd: TypeError (nie można połączyć int z str)
```

W obu przypadkach wywoływany jest błąd `TypeError`, ale z bardzo różnych powodów i w różnych funkcjach. Pierwszy błąd jest spowodowany problemem w samej funkcji `after()`: do `time.sleep()` jest podawany zły argument. Drugi błąd wynika z problemu z wykonaniem funkcji zwrotnej `func(*args)`.

Jeśli ważne jest rozróżnienie tych dwóch przypadków, dostępnych jest kilka rozwiązań. Jednym z nich jest poleganie na połączonych wyjątkach. Chodzi o to, aby spakować błędy z wywołania zwrotnego w inny sposób, który pozwala na ich obsługę oddzielnie od innych rodzajów błędów. Na przykład:

```
class CallbackError(Exception):
    pass

def after(seconds, func, *args):
    time.sleep(seconds)
    try:
        return func(*args)
    except Exception as err:
        raise CallbackError('Nieprawidłowe działanie funkcji zwrotnej') from err
```

Ten zmodyfikowany kod izoluje błędy z podanego wywołania zwrotnego do własnej kategorii wyjątków. Użyj go w ten sposób:

```
try:
    r = after(delay, add, x, y)
except CallbackError as err:
    print("Błąd. Powód", err.__cause__)
```

Gdyby wystąpił problem z wykonaniem samego `after()`, ten wyjątek rozprzestrzeniłby się nieprzechwycony. Z drugiej strony problemy związane z wykonaniem dostarczonej funkcji zwrotnej zostałyby wyłapane i zgłoszone jako `CallbackError`. Takie podejście sprawia, że przypisanie

winy jest bardziej precyzyjne, a zachowanie `after()` łatwiejsze do udokumentowania. Jeśli wystąpi problem z wywołaniem zwrotnym, jest to zawsze zgłaszane jako `CallbackError`.

Innym rozwiązaniem jest spakowanie wyniku funkcji zwrotnej do pewnego rodzaju instancji wyniku, która zawiera zarówno wartość, jak i błąd. Na przykład zdefiniuj klasę w ten sposób:

```
class Result:
    def __init__(self, value=None, exc=None):
        self._value = value
        self._exc = exc
    def result(self):
        if self._exc:
            raise self._exc
        else:
            return self._value
```

Następnie użyj tej klasy, aby zwrócić wyniki z funkcji `after()`:

```
def after(seconds, func, *args):
    time.sleep(seconds)
    try:
        return Result(value=func(*args))
    except Exception as err:
        return Result(exc=err)

# Przykład użycia:

r = after(1, add, 2, 3)
print(r.result())        # Wyświetla 5

s = after("1", add, 2, 3) # Natychmiastowo wywołuje błąd TypeError. Zły argument sleep()

t = after(1, add, "2", 3) # Zwraca "Result"
print(t.result())         # Zgłasza TypeError
```

To drugie podejście polega na odroczeniu raportowania wyników funkcji zwrotnej do oddzielnego kroku. Jeśli wystąpi problem z funkcją `after()`, zostanie to natychmiast zgłoszone. Jeśli wystąpi problem z wywołaniem zwrotnym `func()`, zostanie on zgłoszony, gdy użytkownik spróbuje uzyskać wynik poprzez wywołanie metody `result()`.

Ten styl pakowania wyniku do specjalnej instancji, która ma zostać później rozpakowana, jest coraz bardziej powszechnym wzorcem występującym we współczesnych językach programowania. Jednym z powodów jego użycia jest to, że ułatwia sprawdzanie typu. Jeśli na przykład umieścisz wskazówkę dotyczącą typu w funkcji `after()`, jej zachowanie jest w pełni zdefiniowane — zawsze zwraca `Result` i nic więcej:

```
def after(seconds, func, *args) -> Result:
    ...
```

Chociaż nie jest to tak powszechne rozwiązanie, aby zobaczyć ten rodzaj wzorca w kodzie Pythona, pojawia się on z pewną regularnością podczas pracy z prymitywami współbieżności, takimi jak wątki i procesy. Na przykład instancje `Future` zachowują się tak podczas pracy z pulami wątków. Przykładowo:

```
from concurrent.futures import ThreadPoolExecutor
pool = ThreadPoolExecutor(16)
r = pool.submit(add, 2, 3)        # Zwraca Future
print(r.result())                 # Rozpakuj wynik Future
```

5.18. Dekoratory

Dekorator to funkcja, która opakowuje inną funkcję. Głównym celem tego opakowania jest zmiana lub poprawa zachowania opakowanego obiektu. Syntaktycznie dekoratory są oznaczane za pomocą specjalnego symbolu `@` w następujący sposób:

```
@decorate
def func(x):
    ...
```

Powyższy kod jest skrótem dla następujących elementów:

```
def func(x):
    ...
func = decorate(func)
```

W tym przykładzie zdefiniowana jest funkcja `func()`. Jednak zaraz po zdefiniowaniu sam obiekt funkcji jest przekazywany do funkcji `decorate()` zwracającej obiekt, który zastępuje oryginalną `func`.

Jako przykład konkretnej implementacji stworzymy dekorator `@trace`, który dodaje komunikaty debugowania do funkcji:

```
def trace(func):
    def call(*args, **kwargs):
        print('Wywołanie', func.__name__)
        return func(*args, **kwargs)
    return call

# Przykład użycia
@trace
def square(x):
    return x * x
```

W tym kodzie funkcja `trace()` tworzy funkcję opakowującą, która zapisuje dane wyjściowe debugowania, a następnie wywołuje oryginalny obiekt funkcji. Jeśli więc wywołasz `square()`, zobaczysz wynik funkcji `print()` z funkcji opakowującej.

Gdyby to było takie proste! W praktyce funkcje zawierają również metadane, takie jak nazwa funkcji, wpis dokumentacyjny i wskazówki dotyczące typów. Jeśli umieścisz nakładkę wokół funkcji, wszystkie te informacje zostaną ukryte. Uważa się, że najlepszą praktyką podczas pisania dekoratora jest użycie dekoratora `@wraps()`, jak pokazano w tym przykładzie:

```
from functools import wraps

def trace(func):
    @wraps(func)
    def call(*args, **kwargs):
```

```
        print('Wywołanie', func.__name__)
        return func(*args, **kwargs)
    return call
```

Dekorator `@wraps()` kopiuje różne metadane funkcji do funkcji zastępującej. W takim przypadku metadane z funkcji `func()` są kopiowane do zwracanej funkcji opakowującej `call()`.

Kiedy stosowane są dekoratory, muszą się one pojawić w osobnej linii bezpośrednio przed funkcją. Można zastosować więcej niż jeden dekorator. Oto przykład:

```
@decorator1
@decorator2
def func(x):
    pass
```

W tym przypadku dekoratory stosuje się w następujący sposób:

```
def func(x):
    pass

func = decorator1(decorator2(func))
```

Kolejność, w jakiej pojawiają się dekoratory, może mieć znaczenie. Na przykład w definicjach klas dekoratory, takie jak `@classmethod` i `@staticmethod`, często muszą być umieszczone na najbardziej zewnętrznym poziomie. Na przykład:

```
class SomeClass(object):
    @classmethod      # Dobrze
    @trace
    def a(cls):
        pass

    @trace            # Źle. Błąd
    @classmethod
    def b(cls):
        pass
```

Powód tego ograniczenia miejsca docelowego jest związany z wartościami zwracanymi przez `@classmethod`. Czasami dekorator zwraca obiekt inny niż normalna funkcja. Jeśli najbardziej zewnętrzny dekorator nie spodziewa się tego, coś może nie zadziałać. W tym przypadku `@classmethod` tworzy obiekt deskryptora `classmethod` (patrz rozdział 7.). Jeśli tutaj dekorator `@trace` nie został napisany i dekoratory zostaną wymienione w złej kolejności, wystąpi błąd.

Dekorator może również akceptować argumenty. Załóżmy, że chcesz zmienić dekorator `@trace`, aby umożliwić wyświetlanie niestandardowej wiadomości, takiej jak ta:

```
@trace("Wywołałeś {func.__name__}")
def func():
    pass
```

Jeśli zostaną podane argumenty, semantyka procesu wykonywanego przez dekorator jest następująca:

```
def func():
    pass
```

```
# Tworzenie funkcji dekoratora
temp = trace("Wywołałeś {func.__name__}")

# Zastosowanie powyższego do func
func = temp(func)
```

W tym przypadku najbardziej zewnętrzna funkcja, która akceptuje argumenty, jest odpowiedzialna za utworzenie funkcji dekoratora. Ta funkcja jest następnie wywoływana z funkcją, która ma zostać opakowana, aby uzyskać końcowy wynik. Oto jak może wyglądać implementacja dekoratora:

```
from functools import wraps

def trace(message):
    def decorate(func):
        @wraps(func)
        def wrapper(*args, **kwargs):
            print(message.format(func=func))
            return func(*args, **kwargs)
        return wrapper
    return decorate
```

Interesującą cechą tej implementacji jest to, że zewnętrzna funkcja jest w rzeczywistości rodzajem „fabryki dekoratorów". Załóżmy, że napisałeś taki kod:

```
@trace('Wywołałeś {func.__name__}')
def func1():
    pass

@trace('Wywołałeś {func.__name__}')
def func2():
    pass
```

To szybko stałoby się nużące. Możesz to uprościć, wywołując raz funkcję dekoratora zewnętrznego i ponownie wykorzystując wynik w ten sposób:

```
logged = trace('Wywołałeś {func.__name__}')

@logged
def func1():
    pass

@logged
def func2():
    pass
```

Dekoratory niekoniecznie muszą zastępować pierwotną funkcję. Czasami dekorator wykonuje jedynie czynność taką jak rejestracja. Jeśli na przykład budujesz rejestr programów obsługi zdarzeń, możesz zdefiniować dekorator, który działa w ten sposób:

```
@eventhandler('BUTTON')
def handle_button(msg):
    ...
@eventhandler('RESET')
def handle_reset(msg):
    ...
```

Oto dekorator, który tym zarządza:

```
# Dekorator obsługi zdarzeń
_event_handlers = { }
def eventhandler(event):
    def register_function(func):
        _event_handlers[event] = func
        return func
    return register_function
```

5.19. Funkcje map, filter i reduce

Programiści zaznajomieni z językami funkcjonalnymi często pytają o typowe operacje na listach, takie jak `map`, `filter` i `reduce`. Wiele z tych funkcji jest zapewnianych przez listy składane i wyrażenia generatora. Na przykład:

```
def square(x):
    return x * x

nums = [1, 2, 3, 4, 5]
squares = [square(x) for x in nums]     # [1, 4, 9, 16, 25]
```

Technicznie rzecz biorąc, nie potrzebujesz nawet krótkiej funkcji jednowierszowej. Możesz napisać:

```
squares = [x * x for x in nums]
```

Filtrowanie można również przeprowadzić za pomocą listy składanej:

```
a = [x for x in nums if x > 2]          # [3, 4, 5]
```

Jeśli użyjesz wyrażenia generatora, otrzymasz generator, który generuje wyniki stopniowo poprzez iterację. Na przykład:

```
squares = (x*x for x in nums) # Tworzy generator
for n in squares:
    print(n)
```

Python udostępnia wbudowaną funkcję `map()`, która jest taka sama jak mapowanie za pomocą wyrażenia generatora. Na przykład powyższy przykład można zapisać w ten sposób:

```
squares = map(lambda x: x*x, nums)
for n in squares:
    print(n)
```

Wbudowana funkcja `filter()` tworzy generator, który filtruje wartości:

```
for n in filter(lambda x: x > 2, nums):
    print(n)
```

Jeśli chcesz akumulować lub redukować wartości, możesz użyć funkcji `functools.reduce()`. Na przykład:

```
from functools import reduce
total = reduce(lambda x, y: x + y, nums)
```

W swojej ogólnej formie `reduce()` akceptuje funkcję dwuargumentową, iterację i wartość początkową. Oto kilka przykładów:

```
nums = [1, 2, 3, 4, 5]
total = reduce(lambda x, y: x + y, nums)        # 15
product = reduce(lambda x, y: x * y, nums, 1)   # 120
pairs = reduce(lambda x, y: (x, y), nums, None)
# (((((None, 1), 2), 3), 4), 5)
```

Funkcja `reduce()` gromadzi wartości od lewej do prawej w dostarczonym obiekcie iteracyjnym. Nazywa się to operacją składania w lewo. Oto pseudokod dla `reduce(func, items, initial)`:

```
def reduce(func, items, initial):
    result = initial
    for item in items:
        result = func(result, item)
    return result
```

Korzystanie z `reduce()` w praktyce może być mylące. Co więcej, typowe analogiczne operacje, takie jak `sum()`, `min()` i `max()`, są już wbudowane. Twój kod będzie łatwiejszy do interpretacji (i prawdopodobnie będzie działał szybciej), jeśli użyjesz jednej z nich, zamiast próbować implementować typowe operacje za pomocą funkcji `reduce()`.

5.20. Przegląd funkcji, atrybutów i sygnatur

Jak widzisz, funkcje są obiektami — co oznacza, że można je przypisywać do zmiennych, umieszczać w strukturach danych i wykorzystywać w taki sam sposób jak każdy inny rodzaj danych w programie. Można je również sprawdzać na różne sposoby. Tabela 5.1 przedstawia niektóre typowe atrybuty funkcji. Wiele z tych atrybutów jest przydatnych podczas debugowania, rejestrowania i innych operacji związanych z funkcjami.

Tabela 5.1. *Atrybuty funkcji*

Atrybut	Opis
`f.__name__`	Nazwa funkcji
`f.__qualname__`	W pełni kwalifikowana nazwa (jeśli zagnieżdżona)
`f.__module__`	Nazwa modułu, w którym zdefiniowano funkcję
`f.__doc__`	Wpis dokumentacyjny
`f.__annotations__`	Wskazówki dla typu
`f.__globals__`	Słownik będący globalną przestrzenią nazw
`f.__closure__`	Zmienne domknięcia (jeśli występują)
`f.__code__`	Bazowy obiekt kodu

Atrybut `f.__name__` zawiera nazwę, która została użyta podczas definiowania funkcji. `f.__qualname__` to dłuższa nazwa zawierająca dodatkowe informacje o otaczającym środowisku.

Atrybut `f.__module__` to ciąg zawierający nazwę modułu, w którym zdefiniowano funkcję. Atrybut `f.__globals__` to słownik, który służy jako globalna przestrzeń nazw dla funkcji. Zwykle jest to ten sam słownik, który jest dołączony do skojarzonego obiektu modułu.

`f.__doc__` przechowuje wpis dokumentacyjny funkcji. Atrybut `f.__annotations__` jest słownikiem, który zawiera wskazówki dotyczące typu, jeśli takie istnieją.

`f.__closure__` przechowuje odwołania do wartości zmiennych domykających dla funkcji zagnieżdżonych.

Powyższe wartości są nieco schowane, ale poniższy przykład pokazuje, jak je wyświetlić:

```
def add(x, y):
    def do_add():
        return x + y
    return do_add

>>> a = add(2, 3)
>>> a.__closure__
(<cell at 0x10edf1e20: int object at 0x10ecc1950>,
<cell at 0x10edf1d90: int object at 0x10ecc1970>)
>>> a.__closure__[0].cell_contents
2
>>>
```

Obiekt `f.__code__` reprezentuje skompilowany kod bajtowy interpretera dla treści funkcji.

Funkcje mogą mieć przypisane dowolne atrybuty. Oto przykład:

```
def func():
    instrukcje

func.secure = 1
func.private = 1
```

Atrybuty nie są widoczne w treści funkcji — nie są to zmienne lokalne i nie pojawiają się jako nazwy w środowisku wykonawczym. Głównym zastosowaniem atrybutów funkcji jest przechowywanie dodatkowych metadanych. Czasami frameworki lub różne techniki metaprogramowania wykorzystują tagowanie funkcji — czyli dołączanie atrybutów do funkcji. Jednym z przykładów jest dekorator `@abstractmethod`, który jest używany w metodach w ramach abstrakcyjnych klas bazowych. Jedyne, co robi dekorator, to dołączanie atrybutu:

```
def abstractmethod(func):
    func.__isabstractmethod__ = True
    return func
```

Jakiś inny fragment kodu (w tym przypadku metaklasa) szuka tego atrybutu i używa go do dodania dodatkowych sprawdzeń podczas tworzenia instancji.

Jeśli chcesz dowiedzieć się więcej o parametrach funkcji, możesz uzyskać jej sygnaturę za pomocą funkcji `inspect.signature()`:

```
import inspect

def func(x: int, y:float, debug=False) -> float:
    pass

sig = inspect.signature(func)
```

Obiekty sygnatury zapewniają wiele wygodnych funkcji do uzyskiwania szczegółowych informacji o parametrach. Na przykład:

```
# Wyświetl sygnaturę w ładnej formie
print(sig)  # Tworzy (x: int, y: float, debug=False) -> float

# Pobierz listę nazw argumentów
print(list(sig.parameters))  # Zwraca ['x', 'y', 'debug']

# Iteruj po parametrach i wyświetl różne metadane
for p in sig.parameters.values():
    print('nazwa', p.name)
    print('wskazówka', p.annotation)
    print('rodzaj', p.kind)
    print('domyślnie', p.default)
```

Sygnatura to metadane, które opisują naturę funkcji: jak można ją wywołać, wskazówki dla typu i tak dalej. Jest wiele rzeczy, do których może się ona przydać. Jedną z nich jest operacja porównania. Oto przykład, jak sprawdzić, czy dwie funkcje mają tę samą sygnaturę:

```
def func1(x, y):
    pass

def func2(x, y):
    pass

assert inspect.signature(func1) == inspect.signature(func2)
```

Tego rodzaju porównanie może być przydatne we frameworkach. Na przykład framework może wykorzystać porównanie sygnatur, aby sprawdzić, czy piszesz funkcje lub metody, które są zgodne z oczekiwanym prototypem.

Jeśli jest on przechowywany w atrybucie `__signature__` funkcji, sygnatura będzie pokazywana w komunikatach pomocy i zwracana przy użyciu `inspect.signature()`. Na przykład:

```
def func(x, y, z=None):
    ...

func.__signature__ = inspect.signature(lambda x,y: None)
```

W tym przykładzie opcjonalny argument z byłby ukryty podczas dalszej kontroli funkcji `func`. Dołączona sygnatura zostanie zwrócona przez wywołanie `inspect.signature()`.

5.21. Inspekcja środowiska

Funkcje mogą sprawdzać swoje środowisko wykonawcze za pomocą wbudowanych funkcji `globals()` i `locals()`. `globals()` zwraca słownik, który służy jako globalna przestrzeń nazw. Jest to to samo co atrybut `func.__globals__`. Jest to zwykle ten sam słownik, w którym znajduje się zawartość otaczającego modułu. `locals()` zwraca słownik zawierający wartości wszystkich zmiennych lokalnych i domykających. Słownik ten nie jest rzeczywistą strukturą danych używaną do przechowywania tych zmiennych. Zmienne lokalne mogą pochodzić z funkcji

zewnętrznych (poprzez domknięcie) lub być zdefiniowane wewnętrznie. `locals()` zbiera wszystkie te zmienne i umieszcza je w słowniku. Zmiana pozycji w słowniku `locals()` nie ma wpływu na zmienną bazową. Na przykład:

```
def func():
    y = 20
    locs = locals()
    locs['y'] = 30     # Próba zmiany y
    print(locs['y'])   # Wyświetla 30
    print(y)           # Wyświetla 20
```

Jeśli chcesz, aby zmiana zaczęła obowiązywać, musisz skopiować wartość z powrotem do zmiennej lokalnej przy użyciu normalnego przypisania.

```
def func():
    y = 20
    locs = locals()
    locs['y'] = 30
    y = locs['y']
```

Funkcja może uzyskać własną ramkę stosu za pomocą `inspect.currentframe()`. Może ona również uzyskać ramkę stosu swojego obiektu wywołującego, śledząc ślad stosu poprzez atrybuty `f.f_back` w ramce. Oto przykład:

```
import inspect

def spam(x, y):
    z = x + y
    grok(z)

def grok(a):
    b = a * 10

    # Wyjście: {'a':5, 'b':50}
    print(inspect.currentframe().f_locals)

    # Wyjście: {'x':2, 'y':3, 'z':5}
    print(inspect.currentframe().f_back.f_locals)

spam(2, 3)
```

Czasami zamiast tego zobaczysz ramki stosu uzyskane za pomocą funkcji `sys._getframe()`. Na przykład:

```
import sys
def grok(a):
    b = a * 10
    print(sys._getframe(0).f_locals) # Ja
    print(sys._getframe(1).f_locals) # Obiekt wywołujący
```

Atrybuty w tabeli 5.2 mogą być przydatne do kontrolowania ramek.

Tabela 5.2. *Atrybuty ramki*

Atrybut	Opis
`f.f_back`	Poprzednia ramka stosu (w kierunku obiektu wywołującego)
`f.f_code`	Wykonywany obiekt kodu
`f.f_locals`	Słownik zmiennych lokalnych (`locals()`)
`f.f_globals`	Słownik używany dla zmiennych globalnych (`globals()`)
`f.f_builtins`	Słownik używany dla nazw wbudowanych
`f.f_lineno`	Numer linii
`f.f_lasti`	Bieżąca instrukcja. Jest to indeks do ciągu kodu bajtowego `f_code`
`f.f_trace`	Funkcja wywoływana na początku każdej linii kodu źródłowego

Przeglądanie ramek stosu jest przydatne do debugowania i kontroli kodu. Oto przykład interesującej funkcji debugowania, która pozwala zobaczyć wartości wybranych zmiennych wywołującego obiektu:

```
import inspect
from collections import ChainMap

def debug(*varnames):
    f = inspect.currentframe().f_back
    vars = ChainMap(f.f_locals, f.f_globals)
    print(f'{f.f_code.co_filename}:{f.f_lineno}')
    for name in varnames:
        print(f' {name} = {vars[name]!r}')

# Przykład użycia
def func(x, y):
    z = x + y
    debug('x','y') # Pokazuje x i y wraz z plikiem/linią
    return z
```

5.22. Dynamiczne wykonywanie i tworzenie kodu

Funkcja `exec(str [, globals [, locals]])` wykonuje ciąg znaków zawierający dowolny kod Pythona. Kod dostarczony do `exec()` jest wykonywany tak, jakby kod faktycznie pojawił się w miejscu operacji `exec`. Oto przykład:

```
a = [3, 5, 10, 13]
exec('for i in a: print(i)')
```

Kod przekazany do `exec()` jest wykonywany w lokalnej i globalnej przestrzeni nazw obiektu wywołującego. Należy jednak pamiętać, że zmiany zmiennych lokalnych w `exec` nie mają wpływu na wartość tej zmiennej poza nią. Na przykład:

```
def func():
    x = 10
    exec("x = 20")
    print(x)        # Wyświetla 10
```

Powody tego są związane z tym, że `locals` jest słownikiem zebranych zmiennych lokalnych, a nie rzeczywistymi zmiennymi lokalnymi (więcej szczegółów znajdziesz w poprzedniej sekcji).

Opcjonalnie `exec()` może akceptować jeden lub dwa obiekty słownikowe, które służą odpowiednio jako globalna i lokalna przestrzeń nazw dla wykonywanego kodu. Oto przykład:

```
globs = {'x': 7,
           'y': 10,
           'birds': ['Parrot', 'Swallow', 'Albatross']
         }

locs = { }

# Wykonaj, używając powyższych słowników jako globalnej i lokalnej przestrzeni nazw
exec('z = 3 * x + 4 * y', globs, locs)
exec('for b in birds: print(b)', globs, locs)
```

Jeśli pominiesz jedną lub obie przestrzenie nazw, używane są bieżące wartości globalnej i lokalnej przestrzeni nazw. Jeśli udostępniasz słownik tylko dla globalnych, będzie on używany zarówno dla globalnych, jak i lokalnych przestrzeni.

Typowym zastosowaniem dynamicznego wykonywania kodu jest tworzenie funkcji i metod. Oto przykład funkcji, która tworzy metodę `__init__()` dla klasy o podanej liście nazw:

```
def make_init(*names):
    parms = ','.join(names)
    code = f'def __init__(self, {parms}):\n'
    for name in names:
        code += f' self.{name} = {name}\n'
    d = { }
    exec(code, d)
    return d['__init__']

# Przykład użycia
class Vector:
    __init__ = make_init('x','y','z')
```

Ta technika jest używana w różnych częściach standardowej biblioteki. Na przykład `namedtuple()`, `@dataclass` i podobne funkcje opierają się na dynamicznym tworzeniu kodu za pomocą `exec()`.

5.23. Funkcje asynchroniczne i await

Python zapewnia szereg funkcji językowych związanych z asynchronicznym wykonywaniem kodu. Należą do nich tak zwane funkcje *asynchroniczne* (ang. *async functions*) (lub współprogramy) i elementy *awaitable*. Najczęściej używane są przez programy wykorzystujące współbieżność i moduł `asyncio`, jednak inne biblioteki również mogą na nich bazować.

Funkcja asynchroniczna lub funkcja współbieżna jest definiowana przez poprzedzenie definicji funkcji normalnej dodatkowym słowem kluczowym `async`. Na przykład:

```
async def greeting(name):
print(f'Witaj, {name}')
```

Jeśli wywołasz taką funkcję, przekonasz się, że nie wykonuje się ona w zwykły sposób — w rzeczywistości nie wykonuje się wcale. Zamiast tego otrzymujesz w zamian instancję współprogramowego obiektu. Na przykład:

```
>>> greeting('Guido')
<coroutine object greeting at 0x104176dc8>
>>>
```

Aby funkcja działała, musi być uruchomiona pod nadzorem innego kodu. Powszechnym rozwiązaniem jest użycie modułu `asyncio`. Na przykład:

```
>>> import asyncio
>>> asyncio.run(greeting('Guido'))
Witaj, Guido
>>>
```

Ten przykład przedstawia najważniejszą cechę funkcji asynchronicznych — nigdy nie wykonują się samodzielnie. Do ich wykonania zawsze wymagany jest jakiś kod menedżera lub biblioteki. Niekoniecznie jest to `asyncio`, jak pokazano, ale coś zawsze jest zaangażowane w uruchamianie funkcji asynchronicznych.

Oprócz zarządzania funkcja asynchroniczna jest przetwarzana w taki sam sposób jak każda inna funkcja Pythona. Instrukcje są uruchamiane w odpowiedniej kolejności i działają wszystkie zwykłe funkcje kontroli przepływu. Jeśli chcesz zwrócić wynik, użyj zwykłej instrukcji `return`. Na przykład:

```
async def make_greeting(name):
    return f'Witaj, {name}'
```

Wartość podana do zwrócenia jest zwracana przez zewnętrzną funkcję `run()` używaną do wykonania funkcji asynchronicznej. Na przykład:

```
>>> import asyncio
>>> a = asyncio.run(make_greeting('Paula'))
>>> a
'Witaj, Paula'
>>>
```

Funkcje asynchroniczne mogą wywoływać inne funkcje asynchroniczne za pomocą wyrażenia `await`, na przykład:

```
async def make_greeting(name):
    return f'Witaj, {name}'

async def main():
    for name in ['Paula', 'Thomas', 'Lewis']:
        a = await make_greeting(name)
        print(a)

# Uruchom. Zobaczysz pozdrowienia dla Pauli, Thomasa i Lewisa
asyncio.run(main())
```

Użycie `await` jest prawidłowe tylko w ramach obejmującej definicji funkcji `async`. Jest to również wymagana część wykonywania funkcji asynchronicznych. Jeśli opuścisz wywołanie `await`, przekonasz się, że kod się zepsuje.

Istnieje wymóg używania `await` przy wykorzystywaniu funkcji asynchronicznych, ponieważ ich inny model przetwarzania uniemożliwia wykorzystywanie ich w połączeniu z innymi częściami Pythona. Nie jest możliwe napisanie kodu, który wywołuje funkcję asynchroniczną z funkcji nieasynchronicznej:

```
async def twice(x):
    return 2 * x

def main():
    print(twice(2))        # Błąd. Nie wykonuje funkcji
    print(await twice(2)) # Błąd. Nie można tutaj użyć await
```

Łączenie funkcji asynchronicznych i nieasynchronicznych w tej samej aplikacji to złożony temat, zwłaszcza jeśli weźmie się pod uwagę niektóre techniki programowania obejmujące funkcje wyższego rzędu, wywołania zwrotne i dekoratory. W większości przypadków obsługa funkcji asynchronicznych musi być specjalnie zaimplementowana.

Przyjrzyjmy się protokołom iteratora i menedżera kontekstu w Pytonie. Na przykład asynchroniczny menedżer kontekstu można zdefiniować za pomocą metod `__aenter__()` i `__aexit__()` w klasie takiej jak ta:

```
class AsyncManager(object):
    def __init__(self, x):
        self.x = x

    async def yow(self):
        pass

    async def __aenter__(self):
        return self

    async def __aexit__(self, ty, val, tb):
       pass
```

Zauważ, że te metody są funkcjami asynchronicznymi i dlatego mogą wykonywać inne funkcje asynchroniczne za pomocą `await`. Aby użyć takiego menedżera, musisz skorzystać ze specjalnej składni `async with`, która jest dozwolona tylko w ramach funkcji asynchronicznej:

```
# Przykład
async def main():
    async with AsyncManager(42) as m:
        await m.yow()

asyncio.run(main())
```

Klasa może podobnie zdefiniować iterator asynchroniczny poprzez zdefiniowanie metod `__aiter__()` i `__anext__()`. Są one używane przez instrukcję `async for`, która również może się pojawiać tylko wewnątrz funkcji asynchronicznej.

Z praktycznego punktu widzenia funkcja asynchroniczna zachowuje się dokładnie tak samo jak normalna funkcja — po prostu musi działać w środowisku zarządzanym, takim jak `asyncio`.

Jeśli nie podjąłeś świadomej decyzji o pracy w takim środowisku, powinieneś iść dalej i ignorować funkcje asynchroniczne. Będziesz dużo szczęśliwszy.

5.24. Podsumowanie: przemyślenia na temat funkcji i kompozycji

Każdy system jest zbudowany jako kompozycja komponentów. W Pythonie te komponenty zawierają różnego rodzaju biblioteki i obiekty. U podstaw wszystkiego leżą jednak funkcje. To one są spoiwem pomiędzy systemem i podstawowym mechanizmem przenoszenia danych.

Wiele dyskusji w tym rozdziale koncentrowało się na naturze funkcji i ich interfejsach. W jaki sposób funkcja otrzymuje dane wejściowe? Jak obsługiwane są wyjścia? Jak zgłaszane są błędy? Jak wszystkie te rzeczy mogą być ściślej kontrolowane i lepiej rozumiane?

Warto przemyśleć sposób interakcji funkcji z innymi komponentami programu podczas pracy nad większymi projektami. Często może to oznaczać różnicę między intuicyjnym, łatwym w użyciu interfejsem API a bałaganem.

6

Generatory

Funkcje generatora to jedna z najciekawszych i najbardziej zaawansowanych możliwości Pythona. Generatory są często przedstawiane jako wygodny sposób definiowania nowych rodzajów wzorców iteracji. Jednak jest w nich znacznie więcej: generatory mogą również fundamentalnie zmienić cały model wykonywania funkcji. W tym rozdziale omówiono generatory, ich delegowanie, współprogramy na nich oparte i typowe zastosowania generatorów.

6.1. Generatory i yield

Jeśli funkcja używa słowa kluczowego `yield`, definiuje obiekt nazywany generatorem. Podstawowym jego zastosowaniem jest generowanie wartości do użycia w iteracji. Oto przykład:

```
def countdown(n):
    print('Odliczanie w dół od', n)
    while n > 0:
        yield n
        n -= 1

# Przykład użycia
for x in countdown(10):
    print('T-minus', x)
```

Jeśli wywołasz tę funkcję, przekonasz się, że żaden jej kod nie zacznie się wykonywać. Na przykład:

```
>>> c = countdown(10)
>>> c
<generator object countdown at 0x105f73740>
>>>
```

Zamiast tego tworzony jest obiekt generatora. Z kolei obiekt generatora wykonuje funkcję tylko wtedy, gdy zaczynasz na nim iterować. Jednym ze sposobów, aby to zrobić, jest wywołanie na nim metody `next()`:

```
>>> next(c)
Odliczanie w dół od 10
10
>>> next(c)
9
```

Po wywołaniu metody `next()` funkcja generatora wykonuje instrukcje, dopóki nie osiągnie instrukcji `yield`. Instrukcja `yield` zwraca wynik i wykonywanie funkcji jest zawieszane do momentu ponownego wywołania metody `next()`. W trakcie zawieszenia funkcja zachowuje wszystkie swoje lokalne zmienne i środowisko wykonawcze. Po wznowieniu wykonanie jest kontynuowane z instrukcją, która występuje po `yield`.

`next()` to skrót do wywoływania metody `__next__()` w generatorze. Na przykład możesz również zrobić to:

```
>>> c.__next__()
8
>>> c.__next__()
7
>>>
```

Zwykle nie wywołujesz metody `next()` bezpośrednio w generatorze, ale używasz instrukcji `for` lub innej operacji, która przechodzi przez elementy. Na przykład:

```
for n in countdown(10):
    instrukcje

a = sum(countdown(10))
```

Funkcja generatora tworzy elementy, dopóki nie zwróci wyniku — przez dotarcie do końca funkcji lub użycie instrukcji `return`. Powoduje to wywołanie wyjątku `StopIteration`, który kończy pętlę `for`. Jeśli funkcja generatora zwraca wartość inną niż `None`, jest ona dołączona do wyjątku `StopIteration`. Na przykład ta funkcja generatora używa zarówno instrukcji `yield`, jak i `return`:

```
def func():
    yield 37
    return 42
```

Oto jak wykonałby się kod:

```
>>> f = func()
>>> f
<generator object func at 0x10b7cd480>
>>> next(f)
37
>>> next(f)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration: 42
>>>
```

Zauważ, że zwracana wartość jest dołączona do `StopIteration`. Aby zebrać tę wartość, musisz jawnie przechwycić wyjątek `StopIteration` i wyodrębnić wartość:

```
try:
    next(f)
except StopIteration as e:
    value = e.value
```

Zwykle funkcje generatora nie zwracają wartości. Generatory są prawie zawsze wykorzystywane przez pętlę `for`, w której nie ma możliwości uzyskania wartości wyjątku. Oznacza to, że jedynym praktycznym sposobem uzyskania wartości jest ręczne sterowanie generatorem za pomocą jawnych wywołań funkcji `next()`. Większość kodu wykorzystującego generatory po prostu tego nie robi.

Subtelny problem z generatorami polega na tym, że funkcja generatora jest wykorzystywana tylko częściowo. Rozważmy poniższy kod, który wcześnie przerywa pętlę:

```
for n in countdown(10):
    if n == 2:
        break
    instrukcje
```

W tym przykładzie pętla `for` zostaje przerwana przez wywołanie `break`, a skojarzony generator nigdy się w pełni nie zakończy. Jeśli ważne jest, aby funkcja generatora wykonała jakąś akcję czyszczenia, upewnij się, że używasz instrukcji `try-finally` lub menedżera kontekstu. Na przykład:

```
def countdown(n):
    print('Odliczanie w dół od', n)
    try:
        while n > 0:
            yield n
                n = n - 1
    finally:
        print('Udało się tylko do', n)
```

Jest zagwarantowane, że generatory wykonają blok `finally`, nawet jeśli generator nie zostanie w pełni przetworzony — zostanie wykonany, gdy porzucony generator zostanie zebrany przez system odśmiecania pamięci. Podobnie, każdy kod czyszczący obejmujący menedżer kontekstu wykona się również po zakończeniu działania generatora:

```
def func(filename):
    with open(filename) as file:
        ...
        yield data
        ...
    # Plik zamknięty tutaj, nawet jeśli generator zostanie wcześniej porzucony
```

Właściwe czyszczenie zasobów to trudny problem. Dopóki używasz konstrukcji takich jak `try-finally` lub menedżerów kontekstu, generatory mają gwarancję, że zrobią to właściwie, nawet jeśli zostaną przedwcześnie zakończone.

6.2. Generatory z możliwością ponownego uruchomienia

Zwykle funkcja generatora jest wykonywana tylko raz. Na przykład:

```
>>> c = countdown(3)
>>> for n in c:
... print('T-minus', n)
...
T-minus 3
T-minus 2
T-minus 1
>>> for n in c:
... print('T-minus', n)
...
>>>
```

Jeśli potrzebujesz obiektu, który umożliwia wielokrotne iteracje, zdefiniuj go jako klasę i ustaw metodę `__iter__()` jako generator:

```
class countdown:
    def __init__(self, start):
        self.start = start

    def __iter__(self):
        n = self.start
        while n > 0:
            yield n
            n -= 1
```

Takie rozwiązanie działa, ponieważ za każdym razem, gdy wykonujesz iterację, `__iter__()` tworzy nowy generator.

6.3. Delegowanie generatora

Istotną cechą generatorów jest to, że funkcja związana z `yield` nigdy nie jest wykonywana sama — zawsze musi być sterowana przez inny kod przy użyciu pętli `for` lub jawnych wywołań `next()`. To sprawia, że pisanie funkcji bibliotecznych obejmujących `yield` jest nieco trudne, ponieważ wywołanie funkcji generatora nie wystarczy, aby została wykonana. Aby rozwiązać ten problem, można użyć instrukcji `yield from`. Na przykład:

```
def countup(stop):
    n = 1
    while n <= stop:
        yield n
        n += 1

def countdown(start):
    n = start
    while n > 0:
        yield n
        n -= 1
```

```
def up_and_down(n):
    yield from countup(n)
    yield from countdown(n)
```

`yield from` skutecznie deleguje proces iteracji do iteracji zewnętrznej. Aby na przykład sterować iteracją, możesz napisać taki kod:

```
>>> for x in up_and_down(5):
...     print(x, end=' ')
1 2 3 4 5 5 4 3 2 1
>>>
```

Instrukcja `yield` przede wszystkim pozwala uniknąć konieczności samodzielnego prowadzenia iteracji. Bez tej funkcji musiałbyś napisać `up_and_down(n)` w następujący sposób:

```
def up_and_down(n):
    for x in countup(n):
        yield x
    for x in countdown(n):
        yield x
```

Instrukcja `yield from` jest szczególnie przydatna podczas pisania kodu, który musi rekursywnie iterować przez zagnieżdżone obiekty. Na przykład ten kod spłaszcza zagnieżdżone listy:

```
def flatten(items):
    for i in items:
        if isinstance(i, list):
            yield from flatten(i)
        else:
            yield i
```

Oto przykład, jak to działa:

```
>>> a = [1, 2, [3, [4, 5], 6, 7], 8]
>>> for x in flatten(a):
...     print(x, end=' ')
...
1 2 3 4 5 6 7 8
>>>
```

Jednym z ograniczeń tej implementacji jest to, że nadal polega ona na limitach wyznaczonych przez rekurencję Pythona, więc nie będzie w stanie obsłużyć głęboko zagnieżdżonych struktur. Zostanie to omówione w następnej sekcji.

6.4. Używanie generatorów w praktyce

Na pierwszy rzut oka może nie być oczywiste, jak używać generatorów do rozwiązywania praktycznych problemów poza definiowaniem prostych iteratorów. Jednak generatory są szczególnie skuteczne przy różnych problemach z obsługą danych związanych z potokami i przepływami pracy.

Jednym z użytecznych zastosowań generatorów jest narzędzie do restrukturyzacji kodu, który składa się z głęboko zagnieżdżonych pętli `for` i instrukcji warunkowych. Przyjrzyjmy się skryptowi, który przeszukuje katalog plików Pythona w poszukiwaniu wszystkich komentarzy zawierających słowo „spam”:

```
import pathlib
import re

for path in pathlib.Path('.').rglob('*.py'):
    if path.exists():
        with path.open('rt', encoding='latin-1') as file:
            for line in file:
                m = re.match('.*(#.*)$', line)
                if m:
                    comment = m.group(1)
                    if 'spam' in comment:
                        print(comment)
```

Zwróć uwagę na liczbę poziomów zagnieżdżonego przepływu sterowania. Wygląda to skomplikowanie. Teraz zerknij na wersję z wykorzystaniem generatorów:

```
import pathlib
import re

def get_paths(topdir, pattern):
    for path in pathlib.Path(topdir).rglob(pattern)
        if path.exists():
            yield path

def get_files(paths):
    for path in paths:
        with path.open('rt', encoding='latin-1') as file:
            yield file

def get_lines(files):
    for file in files:
        yield from file

def get_comments(lines):
    for line in lines:
        m = re.match('.*(#.*)$', line)
        if m:
            yield m.group(1)

def print_matching(lines, substring):
    for line in lines:
        if substring in lines:
             print(substring)

paths = get_paths('.', '*.py')
files = get_files(paths)
lines = get_lines(files)
comments = get_comments(lines)
print_matching(comments, 'spam')
```

W tej sekcji problem został podzielony na mniejsze, samodzielne komponenty. Każdy komponent zajmuje się określonym zadaniem. Na przykład generator `get_paths()` zajmuje się tylko nazwami ścieżek, generator `get_files()` jedynie otwieraniem plików i tak dalej. Dopiero na końcu te generatory są połączone razem, aby rozwiązać problem.

Dobrą techniką abstrakcji jest to, aby każdy komponent był mały i odizolowany. Przyjrzyjmy się na przykład generatorowi `get_comments()`. Jako dane wejściowe przyjmuje on wszystkie iterowalne linie tekstu. Ten tekst może pochodzić niemal z dowolnego miejsca: z pliku, listy, generatora i tak dalej. W rezultacie ta funkcja jest znacznie potężniejsza i bardziej elastyczna niż wtedy, gdy była osadzona w głęboko zagnieżdżonej pętli `for`. W ten sposób generatory zachęcają do ponownego wykorzystania, gdyż dzielą problemy na małe, dobrze zdefiniowane zadania obliczeniowe. Mniejsze zadania są również łatwiejsze do uzasadnienia, debugowania i testowania.

Generatory są także przydatne do zmiany normalnych zasad wyliczania funkcji. Zwykle po zastosowaniu funkcji wykonuje się ona natychmiast, dając wynik. Generatory tego nie robią. Kiedy zastosowana jest funkcja generatora, jej wykonanie jest opóźniane, dopóki jakiś inny fragment kodu nie wywoła na niej funkcji `next()` (albo jawnie, albo przez pętlę `for`).

Jako przykład rozważmy ponownie funkcję generatora do spłaszczania zagnieżdżonych list:

```
def flatten(items):
    for i in items:
        if isinstance(i, list):
            yield from flatten(i)
        else:
            yield i
```

Jednym z problemów związanych z tą implementacją jest to, że ze względu na limit rekurencji Pythona nie będzie ona działać z głęboko zagnieżdżonymi strukturami. Można to naprawić, prowadząc iterację w inny sposób — za pomocą stosu. Zerknij na tę wersję:

```
def flatten(items):
    stack = [iter(items)]
    while stack:
        try:
            item = next(stack[-1])
            if isinstance(item, list):
                stack.append(iter(item))
            else:
                yield item
        except StopIteration:
            stack.pop()
```

Ta implementacja buduje wewnętrzny stos iteratorów. Nie podlega ograniczeniom rekurencji Pythona, ponieważ umieszcza dane na wewnętrznej liście, w przeciwieństwie do budowania ramek na wewnętrznym stosie interpretera. Jeśli więc musisz spłaszczyć kilka milionów warstw jakiejś niezwykle głębokiej struktury danych, przekonasz się, że ta wersja działa dobrze.

Czy te przykłady oznaczają, że należy przepisać cały kod przy użyciu takich wzorców tworzenia generatora? Nie. Najważniejsze jest to, że opóźnione przetwarzanie przez generatory pozwala na zmianę w wyliczaniu funkcji. Istnieją różne rzeczywiste scenariusze, w których te techniki mogą być przydatne i stosowane w nieoczekiwany sposób.

6.5. Ulepszone generatory i wyrażenia yield

Wewnątrz funkcji generatora instrukcja `yield` może być również używana jako wyrażenie, które występuje po prawej stronie operatora przypisania. Na przykład:

```
def receiver():
    print('Gotowy do odbioru')
    while True:
        n = yield
        print('Otrzymano', n)
```

Funkcja, która wykorzystuje instrukcję `yield` w ten sposób, jest czasami nazywana „ulepszonym generatorem" lub „współprogramem opartym na generatorze". Niestety ta terminologia jest nieco nieprecyzyjna i jeszcze bardziej zagmatwana, ponieważ „współprogramy" są ostatnio kojarzone z funkcjami asynchronicznymi. Aby uniknąć tego zamieszania, użyjemy terminu „ulepszony generator", aby wyjaśnić, że wciąż mówimy o standardowych funkcjach, które wykorzystują instrukcję `yield`.

Funkcja, która używa `yield` jako wyrażenia, jest nadal generatorem, ale jej użycie jest inne. Zamiast wytwarzać wartości, wykonuje się w odpowiedzi na wysłane do niej wartości. Na przykład:

```
>>> r = receiver()
>>> r.send(None) # Przejście do pierwszego wystąpienia yield
Gotowy do odbioru
>>> r.send(1)
Otrzymano 1
>>> r.send(2)
Otrzymano 2
>>> r.send('Witaj')
Otrzymano Witaj
>>>
```

W tym przykładzie początkowe wywołanie `r.send(None)` jest konieczne, aby generator wykonał instrukcje prowadzące do pierwszego wyrażenia `yield`. W tym momencie generator zawiesza się, czekając na przesłanie wartości za pomocą metody `send()` powiązanego obiektu generatora `r`. Wartość przekazana do `send()` jest zwracana przez wyrażenie `yield` w generatorze. Po otrzymaniu wartości generator wykonuje instrukcje do momentu napotkania następnego wyrażenia `yield`.

Jak napisano, funkcja działa w nieskończoność. Do wyłączenia generatora można wykorzystać metodę `close()`:

```
>>> r.close()
>>> r.send(4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
>>>
```

Operacja `close()` wywołuje wyjątek `GeneratorExit` wewnątrz generatora przy bieżącej instrukcji `yield`. Zwykle powoduje to ciche zakończenie działania generatora; możesz jednak przechwycić ten wyjątek, aby wykonać czynności porządkowe. Po zamknięciu generatora zostanie zgłoszony wyjątek `StopIteration`, jeśli do generatora zostaną wysłane dalsze wartości.

Wyjątki można zgłaszać wewnątrz generatora za pomocą metody `throw(ty [,val [,tb]])`, gdzie `ty` jest typem wyjątku, `val` jest argumentem wyjątku (lub krotką argumentów), a `tb` jest opcjonalnym elementem śledzenia. Na przykład:

```
>>> r = receiver()
Ready to receive
>>> r.throw(RuntimeError, "Dead")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "receiver.py", line 14, in receiver
n = yield
RuntimeError: Dead
>>>
```

Wyjątki zgłoszone w dowolny sposób będą propagowane z aktualnie wykonywanej instrukcji `yield` w generatorze. Generator może przechwycić wyjątek i odpowiednio go obsłużyć. Jeśli generator nie obsługuje wyjątku, propaguje się on poza generator z możliwością obsługi na wyższym poziomie.

6.6. Zastosowania ulepszonych generatorów

Ulepszone generatory to dziwna konstrukcja programistyczna. W przeciwieństwie do prostego generatora, który w naturalny sposób zasila pętlę `for`, nie ma podstawowej funkcji języka, która napędza ulepszony generator. Dlaczego więc miałbyś kiedykolwiek chcieć funkcji, która wymaga wysłania do niej wartości? Czy to czysto akademickie podejście?

Historycznie ulepszone generatory były używane w kontekście bibliotek współbieżności, zwłaszcza tych opartych na asynchronicznych operacjach wejścia-wyjścia. W tym kontekście są one zwykle określane jako „współprogramy” lub „współprogramy oparte na generatorze”. Jednak w większości korzystają one z właściwości `async` oraz `await`. Nie ma powodu, aby korzystać z instrukcji `yield` w tym konkretnym przypadku. Mimo to nadal istnieje kilka praktycznych zastosowań takich generatorów.

Podobnie jak inne generatory, ulepszony generator może być używany do implementacji różnego rodzaju wyliczeń i przepływu kontroli. Jednym z przykładów jest dekorator `@contextmanager` znajdujący się w module `contextlib`. Na przykład:

```
from contextlib import contextmanager
@contextmanager
def manager():
    print("Wejście")
    try:
        yield 'jakaś_wartość'
    except Exception as e:
        print("Wystąpił błąd", e)
    finally:
        print("Wyjście")
```

Tutaj do sklejenia dwóch połówek menedżera kontekstu używany jest generator. Przypomnij sobie, że menedżery kontekstu są definiowane przez obiekty implementujące następujący protokół:

```
class Manager:
    def __enter__(self):
        return somevalue

    def __exit__(self, ty, val, tb):
        if ty:
            # Wystąpił wyjątek
            ...
            # Zwróć True, jeśli zostanie obsłużone, w przeciwnym razie False
```

W generatorze `@contextmanager`, w momencie gdy menedżer rozpoczyna pracę (za pomocą metody `__enter__()`), jest wykonywane wszystko przed instrukcją `yield`. Wszystko po instrukcji `yield` jest wykonywane, gdy menedżer kończy pracę (za pomocą metody `__exit__()`). Jeśli wystąpił błąd, jest on zgłaszany jako wyjątek w deklaracji `yield`. Oto przykład:

```
>>> with manager() as val:
... print(val)
...
Wejście
jakaś_wartość
Wyjście
>>> with manager() as val:
... print(int(val))
...
Wejście
Wystąpił błąd invalid literal for int() with base 10: 'jakaś_wartość'
Wyjście
>>>
```

Aby to zaimplementować, używana jest klasa opakowująca. Jest to uproszczona implementacja, która ilustruje podstawową ideę:

```
class Manager:
    def __init__(self, gen):
        self.gen = gen

    def __enter__(self):
        # Idź do yield
        return self.gen.send(None)

    def __exit__(self, ty, val, tb):
        # Propaguj wyjątek (jeśli wystąpi)
        try:
            if ty:
                try:
                    self.gen.throw(ty, val, tb)
                except ty:
                    return False
            else:
                self.gen.send(None)
        except StopIteration:
            return True
```

Innym zastosowaniem rozszerzonych generatorów jest wykorzystanie funkcji do hermetyzacji zadania „workera". Jedną z głównych cech wywołania funkcji jest to, że tworzone jest środowisko zmiennych lokalnych. Dostęp do tych zmiennych jest wysoce zoptymalizowany — jest znacznie szybszy niż dostęp do atrybutów klas i instancji. Ponieważ generatory pozostają w użyciu, dopóki nie zostaną wyraźnie zamknięte lub zniszczone, można użyć generatora do skonfigurowania długotrwałego zadania. Oto przykład generatora, który odbiera fragmenty bajtów i składa je w linie:

```
def line_receiver():
    data = bytearray()
    line = None
    linecount = 0
    while True:
        part = yield line
        linecount += part.count(b'\n')
        data.extend(part)
        if linecount > 0:
            index = data.index(b'\n')
            line = bytes(data[:index+1])
            data = data[index+1:]
            linecount -= 1
        else:
            line = None
```

W tym przykładzie generator odbiera fragmenty bajtów, które są zbierane w tablicy bajtów. Jeśli tablica zawiera znak nowej linii, wiersz jest wyodrębniany i zwracany. W przeciwnym razie zwracana jest wartość `None`. Oto przykład ilustrujący, jak to działa:

```
>>> r = line_receiver()
>>> r.send(None) # Uruchom generator
>>> r.send(b'hello')
>>> r.send(b'world\nit')
b'hello world\n'
>>> r.send(b'works!')
>>> r.send(b'\n')
b'it works!\n''
>>>
```

Podobny kod można napisać jako klasę, taką jak ta:

```
class LineReceiver:
    def __init__(self):
        self.data = bytearray()
        self.linecount = 0
    def send(self, part):
        self.linecount += part.count(b'\n')
        self.data.extend(part)
        if self.linecount > 0:
             index = self.data.index(b'\n')
             line = bytes(self.data[:index+1])
             self.data = self.data[index+1:]
             self.linecount -= 1
             return line
        else:
            return None
```

Chociaż pisanie klasy może być bardziej intuicyjne, kod jest bardziej złożony i działa wolniej. Przetestowane na maszynie autora: podawanie dużej kolekcji porcji danych do odbiornika jest o około 40 – 50% szybsze przy użyciu generatora niż przy wykorzystaniu kodu tej klasy. Większość tych oszczędności wynika z wyeliminowania wyszukiwania atrybutów instancji — zmienne lokalne są szybsze.

Chociaż istnieje wiele innych potencjalnych aplikacji, ważne jest, aby pamiętać, że jeśli widzisz, że w kontekście jest używana instrukcja `yield`, to aplikacja prawdopodobnie korzysta z ulepszonych funkcji, takich jak `send()` lub `throw()`.

6.7. Generatory i obsługa await

Klasycznym zastosowaniem funkcji generatora są biblioteki związane z asynchroniczną obsługą urządzeń wejścia-wyjścia, takie jak w standardowym module `asyncio`. Jednak od czasu Pythona 3.5 znaczna część tej funkcjonalności została przeniesiona do innej funkcji językowej związanej z funkcjami `async` i instrukcją `await` (patrz ostatnia część rozdziału 5.).

Instrukcja `await` implementuje ukrytą interakcję z generatorem. Oto przykład ilustrujący podstawowy protokół używany przez `await`:

```
class Awaitable:
    def __await__(self):
        print('Czas na await')
        yield # Musi być generatorem
        print('Wznawianie')

# Funkcja kompatybilna z "await". Zwraca typ "awaitable".
def function():
    return Awaitable()

async def main():
    await function()
```

Oto jak możesz wypróbować kod wykorzystujący `asyncio`:

```
>>> import asyncio
>>> asyncio.run(main())
Czas na await
Wznawianie
>>>
```

Czy koniecznie trzeba wiedzieć, jak to działa? Prawdopodobnie nie. Cała ta maszyneria jest zwykle niewidoczna. Jeśli jednak kiedykolwiek będziesz używał funkcji asynchronicznych, po prostu wiedz, że gdzieś w środku jest ukryta funkcja generatora. W końcu ją znajdziesz, jeśli będziesz wystarczająco głęboko wnikał w szczegóły techniczne.

6.8. Podsumowanie: krótka historia generatorów i patrzenie w przyszłość

Generatory przyczyniają się do sukcesu języka Python. Są także częścią większej opowieści dotyczącej iteracji. Iteracja jest jednym z najczęstszych zadań programistycznych. We wczesnych wersjach Pythona iteracja była implementowana za pomocą indeksowania sekwencji i metody `__getitem__()`. Później przekształciła się w obecny protokół iteracji oparty na metodach `__iter__()` i `__next__()`. Niedługo potem generatory pojawiły się jako wygodniejszy sposób implementacji iteratora. We współczesnym Pythonie prawie nie ma powodu, aby kiedykolwiek implementować iterator przy użyciu czegokolwiek innego niż generator. Nawet na obiektach iterowalnych, które możesz zdefiniować samodzielnie, sama metoda `__iter__()` jest również zaimplementowana w ten sposób.

W późniejszych wersjach Pythona, gdy rozwinęły się ulepszone funkcje związane ze współprogramami — metody `send()` i `throw()` — generatory przejęły nową rolę. Nie ograniczały się one już do iteracji, ale otwierały możliwości wykorzystania generatorów w innych kontekstach. Przede wszystkim stanowiło to podstawę wielu tak zwanych frameworków „async", używanych do programowania sieci i współbieżności. Jednak wraz z ewolucją programowania asynchronicznego większość z nich przekształciła się w późniejsze funkcje, które używają składni `async/await`. Dlatego nie jest to tak powszechne, że funkcje generatora są używane poza kontekstem iteracji — ich pierwotnym celem. W rzeczywistości, jeśli odkryjesz, że definiujesz funkcję generatora i nie wykonujesz iteracji, prawdopodobnie powinieneś ponownie rozważyć swoje podejście. Może istnieć lepszy lub bardziej nowoczesny sposób na osiągnięcie tego, co robisz.

7

Klasy i programowanie obiektowe

Klasy służą do tworzenia nowych rodzajów obiektów. Ten rozdział zawiera informacje na temat klas, ale nie ma na celu szczegółowego odniesienia do programowania i projektowania obiektowego. Omówiono niektóre wzorce programowania powszechne w Pythonie, a także sposoby dostosowywania klas, aby zachowywały się w interesujący sposób. W pierwszej kolejności opisano koncepcje i techniki wysokiego poziomu dotyczące używania klas. W dalszej części rozdziału materiał staje się bardziej techniczny i koncentruje się na wewnętrznej implementacji.

7.1. Obiekty

Prawie cały kod w Pythonie obejmuje tworzenie i wykonywanie akcji na obiektach. Na przykład możesz utworzyć obiekt typu `string` i manipulować nim w następujący sposób:

```
>>> s = "Hello World"
>>> s.upper()
'HELLO WORLD'
>>> s.replace('Hello', 'Hello Cruel')
'Hello Cruel World'
>>> s.split()
['Hello', 'World']
>>>
```

Możesz także utworzyć obiekt listy:

```
>>> names = ['Paula', 'Thomas']
>>> names.append('Lewis')
>>> names
['Paula', 'Thomas', 'Lewis']
>>> names[1] = 'Tom'
>>>
```

Istotną cechą każdego obiektu jest to, że zwykle ma on jakiś stan: znaki ciągu, elementy listy i tak dalej, a także metody operujące na tym stanie. Metody są wywoływane przez sam obiekt, tak jakby były funkcjami dołączonymi do obiektu za pomocą operatora kropki (`.`).

Obiekty zawsze mają skojarzony typ. Możesz go wyświetlić za pomocą funkcji `type()`:

```
>>> type(names)
<class 'list'>
>>>
```

Mówi się, że obiekt jest *instancją* tego typu. Na przykład `names` jest instancją listy.

7.2. Wyrażenie class

Nowe obiekty są definiowane za pomocą instrukcji `class`. Klasa zazwyczaj składa się ze zbioru funkcji, które tworzą metody. Oto przykład:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def inquiry(self):
        return self.balance
```

Należy zauważyć, że sama instrukcja `class` nie tworzy żadnych wystąpień klasy. Na przykład w poprzednim przykładzie nie są tworzone żadne konta. Klasa przechowuje jedynie metody, które będą dostępne w instancjach utworzonych później. Możesz myśleć o tym jako o planie.

Funkcje zdefiniowane wewnątrz klasy są nazywane metodami. Metoda instancji to funkcja, która działa na instancji klasy, która jest przekazywana jako pierwszy argument. Zgodnie z konwencją ten argument nazywa się `self`. W poprzednim przykładzie `deposit()`, `extract()` i `query()` są przykładami metod instancji.

Metody `__init__()` i `__repr__()` klasy są przykładami tak zwanych metod *specjalnych* lub *magicznych*. Te metody mają specjalne znaczenie dla środowiska wykonawczego interpretera. Metoda `__init__()` służy do inicjowania stanu podczas tworzenia nowej instancji. Metoda `__repr__()` zwraca ciąg znaków do przeglądania obiektu. Zdefiniowanie tej metody jest opcjonalne, ale upraszcza debugowanie i ułatwia wyświetlanie obiektów z interaktywnego monitu.

Definicja klasy może opcjonalnie zawierać wpis dokumentacyjny i wskazówki dotyczące typu. Na przykład:

```
class Account:
    '''
    Proste konto bankowe
```

```
    '''
    owner: str
    balance: float

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def inquiry(self):
        return self.balance
```

Wskazówki dotyczące typów nie zmieniają żadnego aspektu działania klasy — to znaczy nie wprowadzają żadnego dodatkowego sprawdzania ani walidacji. Są to wyłącznie metadane, które mogą być przydatne w narzędziach lub środowiskach IDE zewnętrznych firm lub wykorzystywane w niektórych zaawansowanych technikach programowania. Nie są one używane w większości poniższych przykładów.

7.3. Instancje

Instancje klasy są tworzone przez wywołanie obiektu klasy jako funkcji. Tworzy to nową instancję, która jest następnie przekazywana do metody `__init__()`. Argumenty funkcji `__init__()` składają się z nowo utworzonej instancji `self` wraz z argumentami dostarczonymi podczas wywoływania obiektu klasy.

Na przykład:

```
# Utwórz kilka kont

a = Account('Guido', 1000.0)
# Wywołuje Account.__init__(a, 'Guido', 1000.0)

b = Account('Eva', 10.0)
# Wywołuje Account.__init__(b, 'Eva', 10.0)
```

Wewnątrz `__init__()` atrybuty są zapisywane w instancji poprzez przypisanie do `self`. Na przykład `self.owner = owner` zapisuje atrybut w instancji. Gdy nowo utworzona instancja zostanie zwrócona, dostęp do tych atrybutów oraz metod klasy uzyskuje się za pomocą operatora kropki (`.`):

```
a.deposit(100.0) # Wywołuje Account.deposit(a, 100.0)
b.withdraw(50.00 # Wywołuje Account.withdraw(b, 50.0)
owner = a.owner # Pobiera konto właściciela
```

Należy podkreślić, że każda instancja ma swój własny stan. Możesz wyświetlić zmienne instancji za pomocą funkcji `vars()`. Na przykład:

```
>>> a = Account('Guido', 1000.0)
>>> b = Account('Eva', 10.0)
>>> vars(a)
{'owner': 'Guido', 'balance': 1000.0}
>>> vars(b)
{'owner': 'Eva', 'balance': 10.0}
>>>
```

Zauważ, że nie pojawiają się tutaj metody. Zamiast tego znajdują się one w klasie. Każda instancja utrzymuje link do swojej klasy poprzez skojarzony typ. Na przykład:

```
>>> type(a)
<class 'Account'>
>>> type(b)
<class 'Account'>
>>> type(a).deposit
<function Account.deposit at 0x10a032158>
>>> type(a).inquiry
<function Account.inquiry at 0x10a032268>
>>>
```

W dalszej części omówiono szczegóły implementacji powiązania atrybutów i relacji między instancjami oraz klasami.

7.4. Dostęp do atrybutów

Istnieją tylko trzy podstawowe operacje, które można wykonać na instancji: pobieranie, ustawianie i usuwanie atrybutu. Na przykład:

```
>>> a = Account('Guido', 1000.0)
>>> a.owner     # Pobieranie
'Guido'
>>> a.balance = 750.0 # Ustawianie
>>> del a.balance # Usuwanie
>>> a.balance
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Account' object has no attribute 'balance'
>>>
```

W Pythonie wszystko dzieje się dynamicznie — z bardzo nielicznymi ograniczeniami. Jeśli chcesz dodać nowy atrybut do obiektu po jego utworzeniu, możesz to zrobić. Na przykład:

```
>>> a = Account('Guido', 1000.0)
>>> a.creation_date = '2019-02-14'
>>> a.nickname = 'Former BDFL'
>>> a.creation_date
'2019-02-14'
>>>
```

Zamiast używać kropki (`.`) do wykonywania tych operacji, możesz podać nazwę atrybutu do funkcji `getattr()`, `setattr()` i `delattr()`. Funkcja `hasattr()` sprawdza istnienie atrybutu. Na przykład:

```
>>> a = Account('Guido', 1000.0)
>>> getattr(a, 'owner')
'Guido'
>>> setattr(a, 'balance', 750.0)
>>> delattr(a, 'balance')
>>> hasattr(a, 'balance')
False
>>> getattr(a, 'withdraw')(100) # Wywołanie metody
>>> a
Account('Guido', 650.0)
>>>
```

Metody `a.attr` i `getattr(a, 'attr')` są wymienne, więc `getattr(a,'withdraw')(100)` jest tym samym co `a.withdraw(100)`. Nie ma znaczenia, że `withdraw()` jest metodą.

Funkcja `getattr()` jest godna uwagi ze względu na przyjmowanie opcjonalnej wartości domyślnej. Jeśli chcesz wyszukać atrybut, który może istnieć lub nie, możesz to zrobić:

```
>>> a = Account('Guido', 1000.0)
>>> getattr(s, 'balance', 'unknown')
1000.0
>>> getattr(s, 'creation_date', 'unknown')
'unknown'
>>>
```

Gdy uzyskujesz dostęp do metody jako atrybutu, otrzymujesz obiekt określany jako *metoda powiązana* (ang. *bound method*).

Na przykład:

```
>>> a = Account('Guido', 1000.0)
>>> w = a.withdraw
>>> w
<bound method Account.withdraw of Account('Guido', 1000.0)>
>>> w(100)
>>> a
Account('Guido', 900.0)
>>>
```

Metoda powiązana to obiekt, który zawiera zarówno instancję (`self`), jak i funkcję implementującą metodę. Po wywołaniu metody powiązanej przez dodanie nawiasów i argumentów wykonywana jest metoda i jako pierwszy argument przekazywana jest dołączona instancja. Na przykład wywołanie `w(100)` powyżej zamienia się w wywołanie `Account.withdraw(a, 100)`.

7.5. Zasady ustalania zakresu

Chociaż klasy definiują wyizolowaną przestrzeń nazw dla metod, ta przestrzeń nazw nie służy jako zakres rozpoznawania nazw używanych wewnątrz metod. Dlatego podczas implementacji klasy odwołania do atrybutów i metod muszą być w pełni kwalifikowane. Na przykład w metodach

zawsze odwołujesz się do atrybutów instancji poprzez `self`. Dlatego używasz wywołania `self.balance`, a nie `balance`. Dotyczy to również sytuacji, gdy chcesz wywołać metodę z innej metody. Załóżmy na przykład, że chcesz zaimplementować metodę `withdraw()`, w której podajesz ujemną kwotę:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.deposit(-amount)      # Musisz użyć self.deposit()

    def inquiry(self):
        return self.balance
```

Brak zasięgu na poziomie klasy to jeden z obszarów, w którym Python różni się od C++ lub Javy. Jeśli używałeś tych języków, parametr `self` w Pythonie jest taki sam jak tak zwany wskaźnik `this`, z wyjątkiem tego, że w Pythonie zawsze musisz go używać jawnie.

7.6. Przeciążanie operatora i protokoły

W rozdziale 4. omówiliśmy model danych Pythona. Szczególną uwagę zwrócono na tzw. metody specjalne, które implementują operatory i protokoły Pythona. Na przykład funkcja `len(obj)` wywołuje `obj.__len__()`, a `obj[n]` wywołuje `obj.__getitem__(n)`.

Podczas tworzenia nowych klas często definiuje się niektóre z tych metod. Metoda `__repr__()` w klasie `Account` była jedną z takich metod. Zdefiniowano ją w celu debugowania. Możesz zdefiniować ich więcej, jeśli tworzysz coś bardziej skomplikowanego, na przykład niestandardowy kontener. Załóżmy na przykład, że chcesz stworzyć portfel kont:

```
class AccountPortfolio:
    def __init__(self):
        self.accounts = []

    def add_account(self, account):
        self.accounts.append(account)

    def total_funds(self):
        return sum(account.inquiry() for account in self)

    def __len__(self):
        return len(self.accounts)

    def __getitem__(self, index):
        return self.accounts[index]
```

```
    def __iter__(self):
        return iter(self.accounts)

# Przykład
port = AccountPortfolio()
port.add_account(Account('Guido', 1000.0))
port.add_account(Account('Eva', 50.0))

print(port.total_funds()) # -> 1050.0
len(port) # -> 2

# Wyświetl rachunki
for account in port:
    print(account)

# Uzyskaj dostęp do indywidualnego konta przez indeks
port[1].inquiry()     # -> 50,0
```

Specjalne metody, które pojawiają się na końcu, takie jak `__len__()`, `__getitem__()` i `__iter__()`, sprawiają, że `AccountPortfolio` współpracuje z operatorami Pythona, takimi jak indeksowanie i iteracja.

Niekiedy z pewnością zetkniesz się ze słowem „Pythonic", na przykład: „Ten kod jest Pythonic". Termin ten jest używany potocznie i wskazuje, czy obiekt dobrze współgra z resztą środowiska Pythona. Oznacza to obsługę — w zakresie, w jakim ma to sens — podstawowych funkcji Pythona, takich jak iteracja, indeksowanie i innych operacji. Prawie zawsze robisz to, implementując w swojej klasie predefiniowane metody specjalne, jak opisano w rozdziale 4.

7.7. Dziedziczenie

Dziedziczenie to mechanizm tworzenia nowej klasy, która specjalizuje lub modyfikuje zachowanie istniejącej. Oryginalna klasa nazywana jest klasą bazową, nadklasą lub klasą nadrzędną. Nowa klasa jest nazywana klasą pochodną, klasą podrzędną, podklasą lub podtypem. Kiedy klasa jest tworzona poprzez dziedziczenie, dziedziczy ona atrybuty zdefiniowane przez jej klasy bazowe. Jednak klasa pochodna może przedefiniować dowolny z tych atrybutów i dodać własne nowe atrybuty.

Dziedziczenie jest określone za pomocą rozdzielonej przecinkami listy nazw klas bazowych w instrukcji `class`. Jeśli nie ma określonej klasy bazowej, klasa niejawnie dziedziczy po `object`. `object` to klasa, która jest korzeniem wszystkich obiektów Pythona; zapewnia domyślną implementację niektórych popularnych metod, takich jak `__str__()` i `__repr__()`.

Jednym z zastosowań dziedziczenia jest rozszerzenie istniejącej klasy o nowe metody. Załóżmy na przykład, że chcesz dodać do konta metodę `panic()`, która wypłaciłaby wszystkie środki.

```
class MyAcount(Account):
    def panic(self):
        self.withdraw(self.balance)
```

```
# Przykład
a = MyAcount('Guido', 1000.0)
a.withdraw(23.0)                # a.balance = 977.0
a.panic()                       # a.balance = 0
```

Dziedziczenie można również wykorzystać do przedefiniowania już istniejących metod. Na przykład oto specjalistyczna wersja `Account`, która na nowo definiuje metodę `query()`, okresowo zawyżającą saldo — z nadzieją, że ktoś nie zwróci na to szczególnej uwagi i w wyniku przekroczenia stanu swojego konta poniesie dużą karę podczas spłaty kredytu hipotecznego:

```
import random

class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
            return self.balance * 1.10
        else:
            return self.balance

a = EvilAccount('Guido', 1000.0)
a.deposit(10.0)                 # Wywołuje Account.deposit(a, 10.0)
available = a.inquiry()         # Wywołuje EvilAccount.inquiry(a)
```

W tym przykładzie instancje `EvilAccount` są identyczne z instancjami `Account`, z wyjątkiem przedefiniowanej metody `inquiry()`.

Czasami klasa pochodna może ponownie zaimplementować metodę, ale także musi wywołać oryginalną implementację. Metoda może jawnie wywołać oryginalną metodę za pomocą instrukcji `super()`:

```
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
            return 1.10 * super().inquiry()
        else:
            return super().inquiry()
```

W tym przykładzie metoda `super()` umożliwia dostęp do metody, tak jak została ona wcześniej zdefiniowana. Wywołanie `super().inquiry()` używa oryginalnej definicji `inquiry()`, zanim została przedefiniowana przez `EvilAccount`.

Jest to mniej popularne rozwiązanie, ale dziedziczenie może być również używane do dodawania dodatkowych atrybutów do instancji. Poniżej pokazano przykład, w którym współczynnik `1.10` jest atrybutem na poziomie instancji — w związku z tym można go dostosować:

```
class EvilAccount(Account):
    def __init__(self, owner, balance, factor):
        super().__init__(owner, balance)
        self.factor = factor

    def inquiry(self):
        if random.randint(0,4) == 1:
            return self.factor * super().inquiry()
        else:
            return super().inquiry()
```

Dodaliśmy atrybut do istniejącej metody `__init__()`. W tym przykładzie definiujemy nową wersję `__init__()`, która zawiera naszą dodatkową zmienną `factor`. Jednak gdy `__init__()` zostanie przedefiniowana, obowiązkiem klasy potomnej (dziecka) jest zainicjowanie swojego rodzica za pomocą `super().__init__()`, jak pokazano. Jeśli zapomnisz tego zrobić, skończysz z w połowie zainicjowanym obiektem i wszystko się zepsuje. Ponieważ inicjalizacja rodzica wymaga dodatkowych argumentów, nadal muszą one zostać przekazane do metody potomnej `__init__()`.

Dziedziczenie może w subtelny sposób zepsuć kod. Rozważmy metodę `__repr__()` klasy `Account`:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self.balance!r})'
```

Celem tej metody jest pomoc w debugowaniu poprzez tworzenie przejrzystych komunikatów. Jednak metoda jest stworzona w taki sposób, aby na stałe używać nazwy `Account`. Jeśli zaczniesz stosować dziedziczenie, przekonasz się, że dane wyjściowe są nieprawidłowe:

```
>>> class EvilAccount(Account):
... pass
...
>>> a = EvilAccount('Eva', 10.0)
>>> a
Account('Eva', 10.0) # Zwróć uwagę na wprowadzające w błąd dane wyjściowe
>>> type(a)
<class 'EvilAccount'>
>>>
```

Aby to naprawić, musisz zmodyfikować metodę `__repr__()` tak, aby używała właściwej nazwy typu.

Na przykład:

```
class Account:
    ...
    def __repr__(self):
        return f'{type(self).__name__}({self.owner!r}, {self.balance!r})'
```

Teraz zobaczysz dokładniejsze dane wyjściowe. Dziedziczenie nie jest stosowane z każdą klasą, ale jeśli jest to przewidywany przypadek użycia klasy, którą piszesz, musisz zwrócić uwagę na takie szczegóły. Zasadniczo unikaj kodowania nazw klas.

Dziedziczenie ustanawia relację w systemie typów, w której każda klasa podrzędna będzie sprawdzać typ jako klasa nadrzędna. Na przykład:

```
>>> a = EvilAccount('Eva', 10)
>>> type(a)
<class 'EvilAccount'>
>>> isinstance(a, Account)
True
>>>
```

Jest to tak zwana relacja „jest”: `EvilAccount` jest tym samym co `Account`. Czasami relacja dziedziczenia bywa używana do definiowania ontologii lub taksonomii typu obiektów. Na przykład:

```
class Food:
    pass

class Sandwich(Food):
    pass
class RoastBeef(Sandwich):
    pass

class GrilledCheese(Sandwich):
    pass

class Taco(Food):
    pass
```

W praktyce organizowanie obiektów w ten sposób może być dość trudne i ryzykowne. Załóżmy, że chcesz dodać klasę `HotDog` do powyższej hierarchii. Dokąd to zmierza? Biorąc pod uwagę, że hot dog ma bułkę, możesz być skłonny zrobić z niego podklasę `Sandwich`. Jednak biorąc pod uwagę ogólny zakrzywiony kształt bułki ze smacznym nadzieniem w środku, być może hot dog jest bardziej podobny do `Taco`. A może zdecydujesz się na podklasę obu:

```
class HotDog(Sandwich, Taco):
    pass
```

W tym momencie wśród Twoich współpracowników może dojść do awantury. Równie dobrze może to być dobry moment, aby wspomnieć, że Python obsługuje dziedziczenie wielokrotne. Aby to zrobić, wymień więcej niż jedną klasę jako rodzica. Powstała klasa potomna odziedziczy wszystkie połączone cechy rodziców. Więcej informacji o dziedziczeniu wielokrotnym można znaleźć w sekcji 7.19.

7.8. Unikanie dziedziczenia poprzez kompozycję

Jednym z problemów związanych z dziedziczeniem jest tak zwane dziedziczenie implementacji. Aby to zilustrować, załóżmy, że chcesz utworzyć strukturę danych stosu za pomocą operacji `push` i `pop`. Szybkim sposobem na to byłoby dziedziczenie z `list` i dodanie do niej nowej metody:

```
class Stack(list):
    def push(self, item):
        self.append(item)

# Przykład
s = Stack()
s.push(1)
s.push(2)
s.push(3)
s.pop()       # -> 3
s.pop()       # -> 2
```

Rzeczywiście, ta struktura danych działa jak stos, ale ma też wszystkie inne funkcje list: wstawianie, sortowanie, wycinanie i tak dalej. To jest dziedziczenie implementacji — użyłeś dziedziczenia, aby ponownie zastosować kod, na którym zbudowałeś coś innego. Masz też wiele funkcji, które nie są związane z faktycznie rozwiązywanym problemem. Użytkownicy prawdopodobnie uznają ten obiekt za dziwny. Dlaczego stos ma metody sortowania?

Lepszym podejściem jest kompozycja. Zamiast budować stos przez dziedziczenie z listy, powinieneś zbudować stos jako niezależną klasę, która zawiera w sobie listę. Fakt, że w środku znajduje się lista, jest szczegółem implementacji. Na przykład:

```
class Stack:
    def __init__(self):
        self._items = list()

    def push(self, item):
        self._items.append(item)

    def pop(self):
        return self._items.pop()

    def __len__(self):
        return len(self._items)

# Przykład
s = Stack()
s.push(1)
s.push(2)
s.push(3)
s.pop() # -> 3
s.pop() # -> 2
```

Ten obiekt działa dokładnie tak samo jak poprzednio, ale skupia się wyłącznie na „byciu stosem”. Nie ma żadnych dodatkowych metod tworzenia list ani funkcji niestosowanych w stosie. Występuje większa przejrzystość działania.

Niewielkie rozszerzenie tej implementacji może przyjąć jako opcjonalny argument klasę `list`:

```
class Stack:
    def __init__(self, *, container=None):
        if container is None:
            container = list()
        self._items = container

    def push(self, item):
        self._items.append(item)

    def pop(self):
        return self._items.pop()

    def __len__(self):
        return len(self._items)
```

Jedną z zalet tego podejścia jest to, że promuje luźne łączenie komponentów. Na przykład możesz chcieć utworzyć stos, który przechowuje swoje elementy w tablicy zamiast na liście. Oto jak możesz to zrobić:

```
import array

s = Stack(container=array.array('i'))
s.push(42)
s.push(23)
s.push('a lot') # TypeError
```

Jest to również przykład tak zwanego *wstrzykiwania zależności* (ang. *dependency injection*). Zamiast kodowania klasy `Stack`, tak aby była zależna od `list`, możesz uzależnić ją od dowolnego kontenera, który użytkownik zdecyduje się przekazać, pod warunkiem że implementuje wymagany interfejs.

Mówiąc szerzej, uczynienie listy wewnętrznej ukrytym szczegółem implementacji wiąże się z problemem abstrakcji danych. Być może później zdecydujesz, że nie chcesz nawet używać listy. Powyższy projekt sprawia, że łatwo to zmienić. Jeśli na przykład zmienisz implementację tak, aby używała połączonych krotek w następujący sposób, użytkownicy `Stack` nawet tego nie zauważą:

```
class Stack:
    def __init__(self):
        self._items = None
        self._size = 0

    def push(self, item):
        self._items = (item, self._items)
        self._size += 1

    def pop(self):
        (item, self._items) = self._items
        self._size -= 1
        return item

    def __len__(self):
        return self._size
```

Aby zdecydować, czy użyć dziedziczenia, czy nie, powinieneś się cofnąć i zadać sobie pytanie, czy obiekt, który budujesz, jest wyspecjalizowaną wersją klasy nadrzędnej, czy używasz go tylko jako komponentu do budowania czegoś innego. Jeśli to drugie, nie stosuj dziedziczenia.

7.9. Unikanie dziedziczenia poprzez funkcje

Czasami może się zdarzyć, że tworzysz klasy składające się tylko z jednej metody, którą trzeba dostosować. Na przykład mogłeś napisać klasę parsującą dane:

```
class DataParser:
    def parse(self, lines):
        records = []
        for line in lines:
            row = line.split(',')
            record = self.make_record(row)
            records.append(row)
        return records

    def make_record(self, row):
```

```
        raise NotImplementedError()

class PortfolioDataParser(DataParser):
    def make_record(self, row):
        return {
            'name': row[0],
            'shares': int(row[1]),
            'price': float(row[2])
        }

parser = PortfolioDataParser()
data = parser.parse(open('portfolio.csv'))
```

Za dużo się tu dzieje. Jeśli piszesz wiele klas z jedną metodą, zamiast tego rozważ użycie funkcji. Na przykład:

```
def parse_data(lines, make_record):
    records = []
    for line in lines:
        row = line.split(',')
        record = make_record(row)
        records.append(row)
    return records

def make_dict(row):
    return {
        'name': row[0],
        'shares': int(row[1]),
        'price': float(row[2])
    }

data = parse_data(open('portfolio.csv'), make_dict)
```

Ten kod jest znacznie prostszy i równie elastyczny, a proste funkcje są łatwiejsze do przetestowania. Jeśli istnieje potrzeba rozszerzenia go na klasy, zawsze możesz to zrobić później. Przedwczesna abstrakcja często nie jest dobrym rozwiązaniem.

7.10. Wiązanie dynamiczne i technika kaczego typowania

Wiązanie dynamiczne to mechanizm wykonawczy, którego Python używa do znajdowania atrybutów obiektów. To pozwala Pythonowi pracować z instancjami bez względu na ich typ. W Pythonie nazwy zmiennych nie mają skojarzonego typu. W ten sposób proces wiązania atrybutów jest niezależny od rodzaju obiektu `obj`. Jeśli wykonasz wyszukiwanie, takie jak `obj.name`, będzie ono działać na dowolnym `obj`, który ma atrybut `name`. Taka technika jest czasami określana kaczym typowaniem (ang. *duck typing*), gdyż nawiązuje do powiedzenia: „Jeśli coś wygląda jak kaczka, kwacze jak kaczka i chodzi jak kaczka, to jest to kaczka”.

Programiści Pythona często piszą programy, które opierają się na tym zachowaniu. Jeśli na przykład chcesz stworzyć dostosowaną wersję istniejącego obiektu, możesz albo dziedziczyć po nim, albo stworzyć zupełnie nowy obiekt, który wygląda i zachowuje się jak on, ale w żaden

inny sposób nie jest z nim powiązany. To drugie podejście jest często stosowane w celu utrzymania luźnego sprzężenia składników programu. Na przykład kod może być napisany do pracy z dowolnym obiektem, o ile ma określony zestaw metod. Jednym z najczęstszych przykładów są różne iterowalne obiekty zdefiniowane w standardowej bibliotece. Istnieje wiele rodzajów obiektów, które współpracują z pętlą `for`, generując wartości: listy, pliki, generatory, ciągi znaków i tak dalej. Jednak żaden z nich nie dziedziczy z żadnego rodzaju specjalnej klasy bazowej `Iterable`. Po prostu wdrażają metody wymagane do wykonania iteracji — i to wszystko działa.

7.11. Niebezpieczeństwo dziedziczenia po typach wbudowanych

Python umożliwia dziedziczenie z typów wbudowanych. Takie postępowanie grozi jednak niebezpieczeństwem. Jeśli na przykład zdecydujesz się skorzystać z podklasy `dict`, aby wymusić na wszystkich kluczach pisanie wielkimi literami, możesz przedefiniować metodę `__setitem__()` w następujący sposób:

```
class udict(dict):
def __setitem__(self, key, value):
super().__setitem__(key.upper(), value)
```

Rzeczywiście, początkowo wydaje się, że takie rozwiązanie działa:

```
>>> u = udict()
>>> u['name'] = 'Guido'
>>> u['number'] = 37
>>> u
{'NAME': 'Guido', 'NUMBER': 37}
>>>
```

Dalsze użytkowanie pokazuje jednak, że tylko wydaje się działać. W rzeczywistości raczej w ogóle nie działa:

```
>>> u = udict(name='Guido', number=37)
>>> u
{'name': 'Guido', 'number': 37}
>>> u.update(color='blue')
>>> u
{'name': 'Guido', 'number': 37, 'color': 'blue'}
>>>
```

Kwestią sporną jest tutaj fakt, że wbudowane typy Pythona nie są implementowane jak normalne klasy Pythona — są implementowane w C. Większość metod działa w świecie C. Na przykład metoda `dict.update()` bezpośrednio manipuluje danymi słownika bez przechodzenia przez przedefiniowaną powyżej metodę `__setitem__()` w Twojej niestandardowej klasie `udict`.

Moduł `collections` zawiera specjalne klasy `UserDict`, `UserList` i `UserString`, których można używać do tworzenia bezpiecznych podklas typów `dict`, `list` i `str`. Na przykład przekonasz się, że to rozwiązanie działa o wiele lepiej:

```
from collections import UserDict

class udict(UserDict):
```

```
    def __setitem__(self, key, value):
        super().__setitem__(key.upper(), value)
```

Oto przykład działania nowej wersji:

```
>>> u = udict(name='Guido', num=37)
>>> u.update(color='Blue')
>>> u
{'NAME': 'Guido', 'NUM': 37, 'COLOR': 'Blue'}
>>> v = udict(u)
>>> v['title'] = 'BDFL'
>>> v
{'NAME': 'Guido', 'NUM': 37, 'COLOR': 'Blue', 'TITLE': 'BDFL'}
>>>
```

W większości przypadków można uniknąć tworzenia podklas typu wbudowanego. Na przykład w trakcie budowania nowych kontenerów prawdopodobnie lepiej stworzyć nową klasę, jak pokazano to dla klasy `Stack` w rozdziale 7.8. Jeśli naprawdę potrzebujesz podklasy typu wbudowanego, może to wymagać dużo więcej pracy, niż myślisz.

7.12. Zmienne i metody klasy

W definicji klasy zakłada się, że wszystkie funkcje działają na instancji, która jest zawsze przekazywana jako pierwszy parametr `self`. Jednak sama klasa jest również obiektem, który może przenosić stan i także nim manipulować. Możesz przykładowo śledzić, ile instancji zostało utworzonych, za pomocą zmiennej klasy `num_accounts`:

```
class Account:
    num_accounts = 0

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance
        Account.num_accounts += 1

    def __repr__(self):
        return f'{type(self).__name__}({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.deposit(-amount) # Musi użyć self.deposit()

    def inquiry(self):
        return self.balance
```

Zmienne klas są zdefiniowane poza normalną metodą `__init__()`. Aby je zmodyfikować, używaj klasy, a nie `self`. Na przykład:

```
>>> a = Account('Guido', 1000.0)
>>> b = Account('Eva', 10.0)
```

```
>>> Account.num_accounts
2
>>>
```

To trochę nietypowe, ale do zmiennych klas można również uzyskać dostęp za pośrednictwem instancji. Na przykład:

```
>>> a.num_accounts
2
>>> c = Account('Ben', 50.0)
>>> Account.num_accounts
3
>>> a.num_accounts
3
>>>
```

Takie rozwiązanie działa, ponieważ wyszukiwanie atrybutów w instancjach sprawdza skojarzoną klasę, jeśli nie ma pasującego atrybutu w samej instancji. Jest to ten sam mechanizm, za pomocą którego Python zwykle znajduje metody.

Możliwe jest również zdefiniowanie tak zwanej *metody klasy*. Metoda klasy to metoda stosowana do samej klasy, a nie do instancji. Powszechnym zastosowaniem metod klas jest definiowanie alternatywnych konstruktorów instancji. Załóżmy na przykład, że istnieje wymóg utworzenia instancji `Account` za pomocą innego formatu danych wejściowych:

```
data = '''
<account>
  <owner>Guido</owner>
  <amount>1000.0</amount>
</account>
'''
```

Aby to zrobić, możesz napisać `@classmethod` w ten sposób:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    @classmethod
    def from_xml(cls, data):
        from xml.etree.ElementTree import XML
        doc = XML(data)
        return cls(doc.findtext('owner'), float(doc.findtext('amount')))

# Przykład
data = '''
<account>
  <owner>Guido</owner>
  <amount>1000.0</amount>
</account>
'''

a = Account.from_xml(data)
```

Pierwszym argumentem metody klasy jest zawsze sama klasa. Zgodnie z konwencją ten argument nazywa się zwykle `cls`. W tym przykładzie `cls` jest ustawiony na `Account`. Jeśli celem metody klasy jest utworzenie nowej instancji, należy podjąć wyraźne kroki. W ostatnim wierszu przykładu wywołanie `cls(..., ...)` jest takie samo jak wywołanie `Account(..., ...)` na dwóch argumentach.

Fakt, że klasa jest przekazywana jako argument, rozwiązuje ważny problem związany z dziedziczeniem. Załóżmy, że definiujesz podklasę `Account`, a teraz chcesz utworzyć instancję tej klasy. Przekonasz się, że nadal działa:

```
class EvilAccount(Account):
    pass

e = EvilAccount.from_xml(data) # Tworzy 'EvilAccount'
```

Powodem, dla którego ten kod działa, jest to, że `EvilAccount` jest teraz przekazywane jako `cls`. Tak więc ostatnia instrukcja metody klasy `from_xml()` tworzy instancję `EvilAccount`.

Zmienne klas i metody klas są czasami używane razem do konfigurowania i kontrolowania działania instancji. Przyjrzyjmy się przykładowo następującej klasie `Date`:

```
import time

class Date:
    datefmt = '{year}-{month:02d}-{day:02d}'
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    def __str__(self):
        return self.datefmt.format(year=self.year,
                                   month=self.month,
                                   day=self.day)

    @classmethod
    def from_timestamp(cls, ts):
        tm = time.localtime(ts)
        return cls(tm.tm_year, tm.tm_mon, tm.tm_mday)

    @classmethod
    def today(cls):
         return cls.from_timestamp(time.time())
```

Ta klasa zawiera zmienną klasy `datefmt`, która dostosowuje dane wyjściowe z metody `__str__()`. Można tu zastosować dziedziczenie:

```
class MDYDate(Date):
    datefmt = '{month}/{day}/{year}'

class DMYDate(Date):
    datefmt = '{day}/{month}/{year}'

# Przykład
a = Date(1967, 4, 9)
print(a)      # 1967-04-09
```

```
b = MDYDate(1967, 4, 9)
print(b)      # 4/9/1967

c = DMYDate(1967, 4, 9)
print(c)      # 9/4/1967
```

Konfiguracja przy użyciu zmiennych klas i takie dziedziczenie są powszechnie wykorzystywane do dostosowywania zachowania instancji. Użycie metod klas ma kluczowe znaczenie dla działania, ponieważ zapewniają one utworzenie odpowiedniego rodzaju obiektu. Na przykład:

```
a = MDYDate.today()
b = DMYDate.today()
print(a)      # 2/13/2019
print(b)      # 13/2/2019
```

Alternatywna konstrukcja instancji jest zdecydowanie najczęstszym zastosowaniem metod klas. Powszechną konwencją nazewnictwa dla takich metod jest dołączenie słowa `from_` jako przedrostka, na przykład `from_timestamp()`. Taka konwencja nazw jest używana w metodach klasowych w bibliotece standardowej oraz w pakietach innych firm. Na przykład słowniki mają metodę klasy do tworzenia wstępnie zainicjowanego słownika z zestawu kluczy:

```
>>> dict.from_keys(['a','b','c'], 0)
{'a': 0, 'b': 0, 'c': 0}
>>>
```

Python nie zarządza metodami klas w przestrzeni nazw oddzielnej od metod instancji. W rezultacie nadal można je wywoływać na instancji.

Na przykład:

```
d = Date(1967,4,9)
b = d.today()      # Wywołuje Date.now(Date)
```

Jest to potencjalnie dość mylące, ponieważ wywołanie `d.today()` tak naprawdę nie ma nic wspólnego z instancją `d`. W Twoim IDE i w dokumentacji metoda `today()` może być prawidłowa dla instancji `Date`.

7.13. Metody statyczne

Czasami klasa jest jedynie używana jako przestrzeń nazw dla funkcji zadeklarowanych jako metody statyczne — przy użyciu `@staticmethod`. W przeciwieństwie do zwykłej metody lub metody klasy metoda statyczna nie wymaga dodatkowego argumentu `self` lub `cls`. Metoda statyczna to po prostu zwykła funkcja, która jest zdefiniowana w klasie. Na przykład:

```
class Ops:
    @staticmethod
    def add(x, y):
        return x + y

    @staticmethod
    def sub(x, y):
        return x - y
```

Zwykle nie tworzy się instancji takiej klasy. Zamiast tego wywołuje się funkcje bezpośrednio przez klasę:

```
a = Ops.add(2, 3) # a = 5
b = Ops.sub(4, 5) # a = -1
```

Czasami inne klasy będą używać kolekcji metod statycznych, takich jak ta, aby zaimplementować bardziej elastyczne zachowanie lub jako coś, co naśladuje zachowanie modułu importu. Rozważ użycie dziedziczenia we wcześniejszym przykładzie `Account`:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __repr__(self):
        return f'{type(self).__name__}({self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

    def inquiry(self):
        return self.balance

# Specjalne konto "Evil"
class EvilAccount(Account):
    def deposit(self, amount):
        self.balance += 0.95 * amount

    def inquiry(self):
        if random.randint(0,4) == 1:
            return 1.10 * self.balance
        else:
            return self.balance
```

W tym przypadku użycie dziedziczenia jest trochę dziwne. Wprowadza dwa różne rodzaje obiektów: `Account` i `EvilAccount`. Nie ma również oczywistego sposobu na zmianę istniejącej instancji `Account` na `EvilAccount` lub z powrotem, ponieważ wiąże się to ze zmianą typu instancji. Oto alternatywna definicja `Account` — za pomocą metod statycznych:

```
class StandardPolicy:
    @staticmethod
    def deposit(account, amount):
        account.balance += amount

    @staticmethod
    def withdraw(account, amount):
        account.balance -= amount

    @staticmethod
    def inquiry(account):
        return account.balance
```

```
class EvilPolicy(StandardPolicy):
    @staticmethod
    def deposit(account, amount):
        account.balance += 0.95*amount

    @staticmethod
    def inquiry(account):
        if random.randint(0,4) == 1:
            return 1.10 * account.balance
        else:
            return account.balance

class Account:
    def __init__(self, owner, balance, *, policy=StandardPolicy):
        self.owner = owner
        self.balance = balance
        self.policy = policy

    def __repr__(self):
        return f'Account({self.policy}, {self.owner!r}, {self.balance!r})'

    def deposit(self, amount):
        self.policy.deposit(self, amount)

    def withdraw(self, amount):
        self.policy.withdraw(self, amount)

    def inquiry(self):
        return self.policy.inquiry(self)
```

W powyższym przykładzie tworzony jest tylko jeden rodzaj instancji, `Account`. Ma jednak specjalny atrybut `policy`, który zapewnia implementację różnych metod. W razie potrzeby wartość `policy` może być dynamicznie zmieniana na istniejącej instancji `Account`:

```
>>> a = Account('Guido', 1000.0)
>>> a.policy
<class 'StandardPolicy'>
>>> a.deposit(500)
>>> a.inquiry()
1500.0
>>> a.policy = EvilPolicy
>>> a.deposit(500)
>>> a.inquiry() # Może być losowo 1,10× więcej
1975.0
>>>
```

Jednym z powodów, dla których `@staticmethod` ma tutaj sens, jest to, że nie ma potrzeby tworzenia instancji `StandardPolicy` lub `EvilPolicy`. Głównym celem tych klas jest zorganizowanie pakietu metod, a nie przechowywanie dodatkowych danych instancji związanych z kontem. Mimo to luźno powiązana natura Pythona może z pewnością pozwolić na aktualizację `policy` tak, aby zawierała własne dane. Zmień metody statyczne na standardowe metody instancji, takie jak:

```
class EvilPolicy(StandardPolicy):
    def __init__(self, deposit_factor, inquiry_factor):
        self.deposit_factor = deposit_factor
        self.inquiry_factor = inquiry_factor

    def deposit(self, account, amount):
        account.balance += self.deposit_factor * amount

    def inquiry(self, account):
        if random.randint(0,4) == 1:
            return self.inquiry_factor * account.balance
        else:
            return account.balance

# Przykład
a = Account('Guido', 1000.0, policy=EvilPolicy(0.95, 1.10))
```

Takie podejście, polegające na delegowaniu metod do klas wspierających, jest powszechną strategią implementacji dla maszyn stanowych i podobnych obiektów. Każdy stan operacyjny może być zawarty we własnej klasie metod (często statycznych). Zmienna instancji, taka jak atrybut `policy` w tym przykładzie, może być następnie użyta do przechowywania szczegółów specyficznych dla implementacji związanych z bieżącym stanem operacyjnym.

7.14. Słowo na temat wzorców projektowych

Pisząc programy obiektowe, programiści czasami fiksują się na implementacji znanych wzorców projektowych, takich jak Strategia, Pyłek, Singleton i tak dalej. Wiele z nich pochodzi ze słynnej książki *Wzorce projektowe* autorstwa Ericha Gammy, Richarda Helma, Ralpha Johnsona i Johna Vlissidesa.

Jeśli znasz takie wzorce, ogólne zasady projektowania stosowane w innych językach z pewnością możesz zastosować w Pythonie. Jednak wiele z tych udokumentowanych wzorców ma na celu obejście konkretnych problemów, które wynikają ze ścisłej natury systemu statycznych typów C++ lub Javy. Dynamiczna natura Pythona sprawia, że wiele z tych wzorców jest przestarzałych, przesadnych lub po prostu niepotrzebnych.

To powiedziawszy, trzeba wspomnieć, że istnieje kilka nadrzędnych zasad pisania dobrego oprogramowania, takich jak dążenie do pisania kodu, który można debugować, testować i rozszerzać. Podstawowe strategie, takie jak pisanie klas z przydatnymi metodami `__repr__()`, przedkładanie kompozycji nad dziedziczenie i zezwalanie na wstrzykiwanie zależności, mogą się znacznie przyczynić do osiągnięcia tych celów. Programiści Pythona również lubią pracować z kodem, o którym można powiedzieć, że jest „Pythonic". Zwykle oznacza to, że obiekty podlegają różnym wbudowanym protokołom, takim jak iteracja, kontenery lub zarządzanie kontekstem. Na przykład zamiast próbować zaimplementować jakiś egzotyczny wzorzec operujący na danych z książki programowania Javy, programista Pythona prawdopodobnie skorzystałby z funkcji generatora z pętlą `for` lub po prostu zamieniłby cały wzorzec na kilka wyszukiwań słownikowych.

7.15. Enkapsulacja danych i atrybuty prywatne

W Pythonie wszystkie atrybuty i metody klasy są publiczne, czyli dostępne bez żadnych ograniczeń. Jest to często niepożądane w aplikacjach zorientowanych obiektowo, które mają powody do ukrywania lub hermetyzacji wewnętrznych szczegółów implementacji.

Aby rozwiązać ten problem, Python opiera się na konwencjach nazewnictwa jako środkach sygnalizowania zamierzonego użycia. Jedną z takich konwencji jest to, że nazwy zaczynające się od jednego wiodącego podkreślenia (_) wskazują implementację wewnętrzną. Oto przykładowo wersja klasy `Account`, w której saldo (ang. *balance*) ma teraz atrybut „prywatny":

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance

    def __repr__(self):
        return f'Account({self.owner!r}, {self._balance!r})'

    def deposit(self, amount):
        self._balance += amount

    def withdraw(self, amount):
        self._balance -= amount

    def inquiry(self):
        return self._balance
```

W tym kodzie atrybut `_balance` ma być wewnętrznym szczegółem. Nic nie stoi na przeszkodzie, aby użytkownik mógł uzyskać do niego bezpośredni dostęp, ale wiodący znak podkreślenia jest silnym wskaźnikiem, że użytkownik powinien szukać bardziej publicznego interfejsu — takiego jak metoda `Account.inquiry()`.

To, czy atrybuty wewnętrzne są dostępne do użycia w podklasie, nie jest jasno określone. Czy poprzedni przykład dziedziczenia może mieć bezpośredni dostęp do atrybutu `_balance` swojego rodzica?

```
class EvilAccount(Account):
    def inquiry(self):
        if random.randint(0,4) == 1:
            return 1.10 * self._balance
        else:
            return self._balance
```

Z reguły jest to akceptowalne w Pythonie. IDE i inne narzędzia mogą ujawniać takie atrybuty. Jeśli przechodzisz z C++, Javy lub innego podobnego języka obiektowego, rozważ stosowanie `_balance` podobnie do atrybutu `protected`.

Jeśli chcesz mieć jeszcze bardziej prywatny atrybut, poprzedź nazwę dwoma wiodącymi podkreśleniami (__). Wszystkie nazwy, takie jak `__name`, są automatycznie zmieniane na nową o postaci `_Classname__name`. Gwarantuje to, że prywatne nazwy używane w nadklasie nie zostaną zastąpione identycznymi nazwami w klasie podrzędnej. Oto przykład ilustrujący to zachowanie:

```
class A:
    def __init__(self):
        self.__x = 3            # Przekształcone na self._A__x

    def __spam(self):           # Przekształcone na _A__spam()
        print('A.__spam', self.__x)

    def bar(self):
        self.__spam()            # Wywołuje tylko A.__spam()

class B(A):
    def __init__(self):
         A.__init__(self)
         self.__x = 37          # Przekształcone na self._B__x

    def __spam(self):           # Przekształcone na _B__spam()
         print('B.__spam', self.__x)

    def grok(self):
         self.__spam()          # Wywołuje B.__spam()
```

W tym przykładzie istnieją dwa różne przypisania do atrybutu `__x`. Ponadto wygląda na to, że klasa `B` próbuje przesłonić metodę `__spam()` poprzez dziedziczenie. Tak jednak nie jest. Przekształcenie nazw powoduje, że dla każdej definicji są używane unikalne nazwy. Wypróbuj następujący przykład:

```
>>> b = B()
>>> b.bar()
A.__spam 3
>>> b.grok()
B.__spam 37
>>>
```

Możesz zobaczyć zniekształcone nazwy bardziej bezpośrednio, jeśli spojrzysz na podstawowe zmienne instancji:

```
>>> vars(b)
{'_A__x': 3, '_B__x': 37}
>>> b._A__spam()
A.__spam 3
>>> b._B__spam
B.__spam 37
>>>
```

Chociaż ten schemat zapewnia iluzję ukrywania danych, nie ma mechanizmu, który faktycznie uniemożliwiałby dostęp do prywatnych atrybutów klasy. Jeśli znane są nazwy klasy i odpowiadający im atrybut prywatny, nadal można uzyskać dostęp do atrybutów przy użyciu przekształconej nazwy. Jeśli taki dostęp do prywatnych atrybutów nadal stanowi problem, możesz rozważyć bardziej bolesny proces przeglądu kodu.

Na pierwszy rzut oka przekształcanie nazw może się wydawać dodatkowym etapem przetwarzania. Jednak ten proces w rzeczywistości występuje tylko raz, gdy klasa jest zdefiniowana. Nie występuje podczas wykonywania metod ani nie daje dodatkowego narzutu na wykonanie

programu. Należy pamiętać, że przekształcanie nazw nie występuje w funkcjach, takich jak `getattr()`, `hasattr()`, `setattr()` lub `delattr()`, gdzie nazwa atrybutu jest określona jako ciąg znaków.

W przypadku tych funkcji, aby uzyskać dostęp do atrybutu, musisz jawnie użyć przekształconej nazwy, takiej jak `'_Classname__name'`. W praktyce prawdopodobnie najlepiej nie zastanawiać się nad prywatnością atrybutów. Nazwy z pojedynczym podkreśleniem są dość powszechne; nazwy podwójnie podkreślone także. Chociaż możesz podjąć dalsze kroki, aby naprawdę ukryć atrybuty, dodatkowe wysiłek i złożoność nie są warte uzyskanych korzyści. Być może najbardziej przydatną rzeczą jest zapamiętanie, że jeśli widzisz wiodące podkreślenia przy nazwie, prawie na pewno jest to jakiś wewnętrzny szczegół, który najlepiej pozostawić w spokoju.

7.16. Wskazówka typu

Atrybuty klas zdefiniowanych przez użytkownika nie mają ograniczeń co do ich typu ani wartości. W rzeczywistości możesz ustawić atrybut na wszystko, co chcesz. Na przykład:

```
>>> a = Account('Guido', 1000.0)
>>> a.owner
'Guido'
>>> a.owner = 37
>>> a.owner
37
>>> b = Account('Eva', 'a lot')
>>> b.deposit(' more')
>>> b.inquiry()
'a lot more'
>>>
```

Jeśli jest to problem praktyczny, istnieje kilka możliwych rozwiązań. Jedno z nich jest łatwe — nie rób tego! Innym jest poleganie na zewnętrznych narzędziach, takich jak *linter*, i sprawdzanie typu. W tym celu klasy pozwalają na określenie opcjonalnych wskazówek typu (ang. *type hinting*) dla wybranych atrybutów. Na przykład:

```
class Account:
    owner: str          # Wskazówka typu
    _balance: float     # Wskazówka typu

    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance
    ...
```

Włączenie wskazówek typu nie zmienia niczego w rzeczywistym zachowaniu klasy w czasie wykonywania — to znaczy nie ma dodatkowego sprawdzania i nic nie uniemożliwia użytkownikowi ustawienia złych wartości w kodzie. Jednak wskazówki mogą dać użytkownikom więcej przydatnych informacji w ich edytorze, co pozwala uniknąć błędów, zanim się pojawią.

W praktyce używanie dokładnych wskazówek typu może sprawiać problemy. Na przykład czy klasa `Account` pozwala na użycie `int` zamiast `float`? A co z typem `Decimal`? Przekonasz się, że wszystkie te operacje działają, nawet jeśli wskazówka może sugerować coś innego.

```
from decimal import Decimal
a = Account('Guido', Decimal('1000.0'))
a.withdraw(Decimal('50.0'))
print(a.inquiry())        # -> 950.0
```

Wiedza o tym, jak właściwie organizować typy w takich sytuacjach, wykracza poza zakres tej książki. Jeśli masz wątpliwości, prawdopodobnie lepiej nie zgadywać, chyba że aktywnie korzystasz z narzędzi, które sprawdzają typ kodu.

7.17. Właściwości

Jak wspomniano w poprzedniej sekcji, Python nie nakłada żadnych ograniczeń na wartości lub typy atrybutów. Jest to jednak możliwe, jeśli atrybut będzie zarządzany poprzez *właściwość* (ang. *property*). Właściwość to specjalny rodzaj atrybutu, który przechwytuje dostęp do niego i obsługuje go za pomocą metod zdefiniowanych przez użytkownika. Te metody mają pełną swobodę w zarządzaniu atrybutem. Oto przykład:

```
import string

class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self._balance = balance

    @property
    def owner(self):
        return self._owner

    @owner.setter
    def owner(self, value):
        if not isinstance(value, str):
            raise TypeError('Spodziewany ciąg znaków')
        if not all(c in string.ascii_uppercase for c in value):
            raise ValueError('Musi być dużą literą ASCII')
        if len(value) > 10:
            raise ValueError('Musi być 10 znaków lub mniej')
        self._owner = value
```

W tym przypadku atrybut `owner` jest ograniczony do 10-znakowego ciągu ASCII i wielkich liter. Oto jak to działa, gdy próbujesz użyć klasy:

```
>>> a = Account('GUIDO', 1000.0)
>>> a.owner = 'EVA'
>>> a.owner = 42
Traceback (most recent call last):
...
TypeError: Spodziewany ciąg znaków
>>> a.owner = 'Carol'
Traceback (most recent call last):
...
ValueError: Musi być dużą literą ASCII
>>> a.owner = 'RENÉE'
```

```
Traceback (most recent call last):
...
ValueError: Musi być dużą literą ASCII
>>> a.owner = 'RAMAKRISHNAN'
Traceback (most recent call last):
...
ValueError: Musi być 10 znaków lub mniej
>>>
```

Dekorator `@property` służy do ustalenia atrybutu jako właściwości. W tym przykładzie jest stosowany do atrybutu `owner`. Ten dekorator jest zawsze najpierw stosowany do metody, która pobiera wartość atrybutu. W tym przypadku metoda zwraca rzeczywistą wartość, która jest przechowywana w prywatnym atrybucie `_owner`. Poniższy dekorator `@owner.setter` służy do opcjonalnego zaimplementowania metody ustawiania wartości atrybutu. Ta metoda wykonuje różne sprawdzenia typu i wartości przed zapisaniem wartości w prywatnym atrybucie `_owner`.

Krytyczną cechą właściwości jest to, że powiązana nazwa, taka jak `owner` w powyższym przykładzie, staje się „magiczna". Oznacza to, że każde użycie tego atrybutu automatycznie wywołuje zaimplementowane metody pobierające/ustawiające. Nie musisz zmieniać żadnego istniejącego kodu, aby to zadziałało. Na przykład nie trzeba wprowadzać żadnych zmian w metodzie `Account.__init__()`. Może to Cię zaskoczyć, ponieważ `__init__()` nie korzysta z prywatnego atrybutu `self._owner`, tylko stosuje przypisanie `self.owner = owner`. Jest to zgodne z projektem — cały sens własności `owner` polega na walidacji wartości atrybutów. Na pewno chcesz to zrobić, gdy zostaną utworzone instancje. Przekonasz się, że walidacja działa zgodnie z przeznaczeniem:

```
>>> a = Account('Guido', 1000.0)
Traceback (most recent call last):
  File "account.py", line 5, in __init__
    self.owner = owner
  File "account.py", line 15, in owner
    raise ValueError('Musi być dużą literą ASCII')
ValueError: Musi być dużą literą ASCII
>>>
```

Ponieważ każdy dostęp do atrybutu właściwości automatycznie wywołuje metodę, rzeczywista wartość musi być przechowywana pod inną nazwą. Właśnie dlatego `_owner` jest używany w metodach pobierających i ustawiających. Nie możesz użyć `owner` jako lokalizacji przechowywania, ponieważ spowodowałoby to nieskończoną rekurencję.

Ogólnie rzecz biorąc, właściwości pozwalają na przechwycenie dowolnej nazwy konkretnego atrybutu. Możesz zaimplementować metody pobierania, ustawiania lub usuwania wartości atrybutu. Na przykład:

```
class SomeClass:
    @property
    def attr(self):
        print('Pobieranie')

@attr.setter
def attr(self, value):
    print('Ustawianie', value)

@attr.deleter
```

```
def attr(self):
    print('Usuwanie)

# Przykład
s = SomeClass()
s.attr       # Pobieranie
s.attr = 13  # Ustawianie
del s.attr   # Usuwanie
```

Nie ma konieczności implementacji wszystkich części właściwości. W rzeczywistości często używa się właściwości do implementowania atrybutów danych przeznaczonych tylko do odczytu. Na przykład:

```
class Box(object):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    @property
    def area(self):
        return self.width * self.height

    @property
    def perimeter(self):
        return 2*self.width + 2*self.height

# Przykład
b = Box(4, 5)
print(b.area)      # -> 20
print(b.perimeter) # -> 18
b.area = 5         # Błąd: nie można ustawić atrybutu
```

Jedną z rzeczy, o których należy pomyśleć podczas definiowania klasy, jest ujednolicenie interfejsu programistycznego. Bez właściwości niektóre wartości byłyby dostępne jako proste atrybuty, takie jak `b.width` lub `b.height`, podczas gdy inne wartości byłyby dostępne jako metody, takie jak `b.area()` i `b.perimeter()`. Śledzenie atrybutów, dla których można dodać dodatkowe `()`, powoduje niepotrzebne zamieszanie. Właściwość może pomóc to naprawić.

Programiści Pythona często nie zdają sobie sprawy, że same metody są traktowane niejawnie jako rodzaj własności. Przyjrzyj się tej klasie:

```
class SomeClass:
    def yow(self):
        print('Yow!')
```

Kiedy tworzysz instancję, taką jak `s = SomeClass()`, a następnie uzyskujesz dostęp do `s.yow`, oryginalny obiekt funkcji `yow` nie jest zwracany. Zamiast tego otrzymujesz powiązaną metodę w ten sposób:

```
>>> s = SomeClass()
>>> s.yow
<bound method SomeClass.yow of <__main__.SomeClass object at 0x10e2572b0>>
>>>
```

Jak to się stało? Okazuje się, że funkcje zachowują się bardzo podobnie do właściwości, gdy są umieszczane w klasie. A dokładnie funkcje magicznie przechwytują dostęp do atrybutów i tworzą pod spodem metodę powiązaną. Kiedy definiujesz metody statyczne i metody klasowe za pomocą `@staticmethod` oraz `@classmethod`, w rzeczywistości zmieniasz ten proces. `@staticmethod` zwraca funkcję metody bez specjalnego opakowywania lub przetwarzania. Więcej informacji na temat tego procesu znajduje się w sekcji 7.28.

7.18. Typy, interfejsy i klasy abstrakcyjne

Kiedy tworzysz instancję klasy, typem tej instancji jest sama klasa. Aby przetestować członkostwo w klasie, użyj wbudowanej funkcji `isinstance(obj, cls)`. Ta funkcja zwraca `True`, jeśli obiekt `obj`, należy do klasy `cls` lub dowolnej klasy pochodnej `cls`.

Oto przykład:

```
class A:
    pass

class B(A):
    pass

class C:
    pass

a = A() # Instancja 'A'
b = B() # Instancja 'B'
c = C() # Instancja 'C'

type(a)          # Zwraca obiekt klasy A
isinstance(a, A) # Zwraca True
isinstance(b, A) # Zwraca True, B dziedziczy po A
isinstance(b, C) # Zwraca False, B nie dziedziczy po C
```

Podobnie, wbudowana funkcja `issubclass(A, B)` zwraca `True`, jeśli klasa `A` jest podklasą klasy `B`. Oto przykład:

```
issubclass(B, A) # Zwraca True
issubclass(C, A) # Zwraca False
```

Powszechnym zastosowaniem relacji typowania klas jest specyfikacja interfejsów programistycznych. Jako przykład można zaimplementować klasę bazową najwyższego poziomu w celu określenia wymagań interfejsu programistycznego. Ta klasa bazowa może być następnie użyta dla wskazówki typu lub do zabezpieczenia przed wymuszeniem typu za pomocą `isinstance()`:

```
class Stream:
    def receive(self):
        raise NotImplementedError()

    def send(self, msg):
        raise NotImplementedError()
```

```
    def close(self):
        raise NotImplementedError()

# Przykład
def send_request(stream, request):
    if not isinstance(stream, Stream):
        raise TypeError('Spodziewany obiekt Stream')
    stream.send(request)
    return stream.receive()
```

W powyższym przykładzie obiekt `Stream` nie będzie używany bezpośrednio. Zamiast tego różne klasy dziedziczą po `Stream` i implementują wymaganą funkcjonalność. Użytkownik mógłby stworzyć instancję jednej z tych klas. Na przykład:

```
class SocketStream(Stream):
    def receive(self):
        ...

    def send(self, msg):
        ...

    def close(self):
        ...

class PipeStream(Stream):
    def receive(self):
        ...

    def send(self, msg):
        ...

    def close(self):
        ...

# Przykład
s = SocketStream()
send_request(s, request)
```

W tym przykładzie warto omówić wymuszenie środowiska wykonawczego interfejsu w `send_request()`. Czy zamiast tego należy użyć wskazówki typu?

```
# Określanie interfejsu jako wskazówki typu
def send_request(stream:Stream, request):
    stream.send(request)
    return stream.receive()
```

Biorąc pod uwagę, że wskazówki typu nie są wymuszane, decyzja o tym, jak sprawdzić poprawność argumentu względem interfejsu, naprawdę zależy od tego, kiedy chcesz, aby miało to miejsce — w czasie wykonywania, w kroku sprawdzania kodu lub wcale.

Takie użycie klas interfejsów jest bardziej powszechne w organizacji dużych frameworków i aplikacji. Jednak przy takim podejściu musisz się upewnić, że podklasy faktycznie implementują wymagany interfejs. Jeśli na przykład dla podklasy nie zdecydowano się zaimplementować jednej z wymaganych metod lub wystąpił w niej prosty błąd pisowni, efekty mogą początkowo

pozostać niezauważone, ponieważ kod może nadal działać w typowym przypadku. Jednak jeśli później zostanie wywołana niezaimplementowana metoda, program ulegnie awarii. Oczywiście miałoby to miejsce tylko o 3:30 w środowisku produkcyjnym.

Aby zapobiec temu problemowi, interfejsy często są definiowane jako *abstrakcyjne klasy bazowe* przy użyciu modułu `abc`. Ten moduł definiuje klasę bazową (`ABC`) i dekorator (`@abstractmethod`), które są używane razem do opisywania interfejsu. Oto przykład:

```
from abc import ABC, abstractmethod

class Stream(ABC):
    @abstractmethod

    def receive(self):
        pass

    @abstractmethod
    def send(self, msg):
        pass

    @abstractmethod
    def close(self):
        pass
```

Klasa abstrakcyjna nie może być tworzona bezpośrednio. W rzeczywistości, jeśli spróbujesz utworzyć instancję `Stream`, otrzymasz błąd:

```
>>> s = Stream()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Stream with abstract methods close, receive,
send
>>>
```

Komunikat o błędzie informuje dokładnie, jakie metody muszą zostać zaimplementowane przez `Stream`. Służy jako przewodnik do pisania podklas. Załóżmy, że piszesz podklasę, ale popełniasz błąd:

```
class SocketStream(Stream):
    def read(self): # Błędna nazwa
        ...

    def send(self, msg):
        ...

    def close(self):
        ...
```

Abstrakcyjna klasa bazowa wychwyci błąd podczas tworzenia instancji. Jest to przydatne, ponieważ błędy są wcześnie wykrywane.

```
>>> s = SocketStream()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class SocketStream with abstract methods receive
>>>
```

Chociaż nie można utworzyć instancji klasy abstrakcyjnej, może ona definiować metody i właściwości do użytku w podklasach. Co więcej, metoda abstrakcyjna nadal może być wywoływana z podklasy. Na przykład wywoływanie `super().receive()` z podklasy jest dozwolone.

7.19. Wielokrotne dziedziczenie, interfejsy i domieszki

Python obsługuje dziedziczenie wielokrotne. Jeśli klasa podrzędna zawiera więcej niż jednego rodzica, dziecko dziedziczy wszystkie cechy rodziców. Na przykład:

```
class Duck:
    def walk(self):
        print('Chodzenie')

class Trombonist:
    def noise(self):
        print('Głos!')

class DuckBonist(Duck, Trombonist):
    pass

d = DuckBonist()
d.walk()      # -> Chodzenie
d.noise()     # -> Głos !
```

Koncepcyjnie jest to fajny pomysł, ale może to inaczej działać w rzeczywistości. Co się na przykład stanie, jeśli `Duck` i `Trombonist` zdefiniują metodę `__init__()`? Lub jeśli obie klasy definiują metodę `noise()`? Nagle zaczynasz sobie zdawać sprawę, że wielokrotne dziedziczenie jest obarczone niebezpieczeństwem.

Aby lepiej zrozumieć faktyczne wykorzystanie dziedziczenia wielokrotnego i spójrz na nie jak na wysoce wyspecjalizowane narzędzie do organizacji i ponownego wykorzystania kodu — w przeciwieństwie do techniki programowania ogólnego przeznaczenia. W szczególności pobieranie kolekcji arbitralnie niepowiązanych klas i łączenie ich razem z wielokrotnym dziedziczeniem w celu stworzenia dziwnych, zmutowanych klas nie jest standardową praktyką. Nigdy tego nie rób.

Bardziej powszechnym zastosowaniem dziedziczenia wielokrotnego jest organizowanie relacji typu i interfejsu. Na przykład ostatnia sekcja wprowadziła pojęcie abstrakcyjnej klasy bazowej. Celem abstrakcyjnej bazy jest określenie interfejsu programistycznego. Na przykład możesz mieć różne klasy abstrakcyjne, takie jak ta:

```
from abc import ABC, abstractmethod

class Stream(ABC):
    @abstractmethod
    def receive(self):
        pass

    @abstractmethod
    def send(self, msg):
        pass
```

```
    @abstractmethod
    def close(self):
        pass

class Iterable(ABC):
    @abstractmethod
    def __iter__(self):
        pass
```

W przypadku tych klas dziedziczenie wielokrotne może być użyte do określenia, które interfejsy zostały zaimplementowane przez klasę potomną:

```
class MessageStream(Stream, Iterable):
    def receive(self):
        ...
    def send(self):
        ...
    def close(self):
        ...
    def __iter__(self):
        ...
```

Ponownie, powyższe użycie wielokrotnego dziedziczenia nie dotyczy implementacji, ale relacji typów. Na przykład żadna z dziedziczonych metod nic nie robi w tym przykładzie. Nie ma możliwości ponownego wykorzystania kodu. Relacja dziedziczenia umożliwia przeprowadzanie kontroli typu głównie w następujący sposób:

```
m = MessageStream()
isinstance(m, Stream)       # -> True
isinstance(m, Iterable)     # -> True
```

Innym zastosowaniem dziedziczenia wielokrotnego jest zdefiniowanie *klas domieszek*. Klasa domieszki to klasa, która modyfikuje lub rozszerza funkcjonalność innych klas. Przyjrzyj się następującym definicjom klas:

```
class Duck:
    def noise(self):
        return 'Kwak'

    def waddle(self):
        return 'Chodzenie'

class Trombonist:
    def noise(self):
        return 'Głos!'

    def march(self):
        return 'Stąpanie'

class Cyclist:
    def noise(self):
        return 'Po lewej stronie!'

    def pedal(self):
        return 'Pedałowanie'
```

Klasy te są ze sobą zupełnie niezwiązane. Nie ma pomiędzy nimi relacji dziedziczenia i stosują różne metody. Istnieje jednak cecha wspólna — każda z nich definiuje metodę `noise()`. Kierując się tym, możesz zdefiniować następujące klasy:

```
class LoudMixin:
    def noise(self):
        return super().noise().upper()

class AnnoyingMixin:
    def noise(self):
        return 3*super().noise()
```

Na pierwszy rzut oka te klasy wyglądają źle. Jest tylko jedna izolowana metoda i używa metody `super()` do delegowania do nieistniejącej klasy nadrzędnej. Klasy nawet nie działają:

```
>>> a = AnnoyingMixin()
>>> a.noise()
Traceback (most recent call last):
...
AttributeError: 'super' object has no attribute 'noise'
>>>
```

To są klasy domieszek. Jedynym sposobem, w jaki działają, jest połączenie z innymi klasami, które implementują brakującą funkcjonalność. Na przykład:

```
class LoudDuck(LoudMixin, Duck):
    pass

class AnnoyingTrombonist(AnnoyingMixin, Trombonist):
    pass

class AnnoyingLoudCyclist(AnnoyingMixin, LoudMixin, Cyclist):
    pass

d = LoudDuck()
d.noise() # -> KWAK'

t = AnnoyingTrombonist()
t.noise() # -> 'Głos! Głos! Głos!

c = AnnoyingLoudCyclist()
c.noise() # -> 'PO LEWEJ STRONIE!PO LEWEJ STRONIE!PO LEWEJ STRONIE!'
```

Ponieważ klasy domieszek są definiowane w taki sam sposób jak zwykłe klasy, najlepiej dołączyć słowo `Mixin` jako część nazwy klasy. Ta konwencja nazewnictwa zapewnia większą jasność celu.

Aby w pełni zrozumieć domieszki, musisz się nieco więcej dowiedzieć o tym, jak działają dziedziczenie i funkcja `super()`.

Za każdym razem, gdy używasz dziedziczenia, Python buduje liniowy łańcuch klas (ang. *Method Resolution Order*; MRO). Jest on dostępny w klasie jako atrybut `__mro__`. Oto kilka przykładów pojedynczego dziedziczenia:

```
class Base:
    pass
class A(Base):
```

```
    pass
class B(A):
    pass
Base.__mro__  # -> (<class 'Base'>, <class 'object'>)
A.__mro__  # -> (<class 'A'>, <class 'Base'>, <class 'object'>)
B.__mro__  # -> (<class 'B'>, <class 'A'>, <class 'Base'>, <class 'object'>)
```

MRO określa kolejność dla wyszukiwania atrybutów. Za każdym razem, gdy szukasz atrybutu w instancji lub klasie, każda klasa w MRO jest sprawdzana w podanej kolejności. Wyszukiwanie zatrzymuje się po pierwszym dopasowaniu. Klasa obiektu jest wymieniona w MRO, ponieważ wszystkie klasy dziedziczą po obiekcie, niezależnie od tego, czy jest on wymieniony jako rodzic.

Aby obsługiwać wielokrotne dziedziczenie, Python implementuje tak zwane współdzielone dziedziczenie wielokrotne. Przy dziedziczeniu współdzielonym wszystkie klasy są umieszczane na liście MRO zgodnie z dwiema podstawowymi zasadami porządkowania. Pierwsza zasada mówi, że klasa potomna musi być zawsze sprawdzana przed którymkolwiek z jej rodziców. Druga zasada mówi, że jeśli klasa ma wielu rodziców, rodzice ci muszą być sprawdzani w tej samej kolejności, w jakiej są zapisani na liście dziedziczenia dziecka. W większości te zasady tworzą sensowny MRO. Jednak precyzyjny algorytm porządkujący klasy jest w rzeczywistości dość złożony i nie opiera się na żadnym prostym podejściu, takim jak wyszukiwanie w głąb lub wszerz. Zamiast tego kolejność określana jest zgodnie z algorytmem linearyzacji C3, który jest opisany w artykule *A Monotonic Superclass Linearization for Dylan* (K. Barrett i in., przedstawionym na OOPSLA'96). Subtelnym aspektem tego algorytmu jest to, że niektóre hierarchie klas zostaną odrzucone przez Pythona z błędem `TypeError`. Oto przykład:

```
class X: pass
class Y(X): pass
class Z(X,Y): pass # TypeError
                   # Nie można stworzyć spójnego MRO
```

W tym przypadku algorytm rozpoznawania metod odrzuca klasę `Z`, ponieważ nie może określić kolejności klas bazowych, które mają sens. Tutaj klasa `X` pojawia się przed klasą `Y` na liście dziedziczenia, więc należy ją najpierw sprawdzić. Jednak klasa `Y` dziedziczy po `X`, więc jeśli `X` jest sprawdzane jako pierwsze, narusza to regułę, że najpierw sprawdzane są dzieci. W praktyce problemy te pojawiają się rzadko — a jeśli już, to zwykle wskazuje to na poważniejszy problem projektowy.

Oto praktyczny przykład MRO dla klasy `AnnoyingLoudCyclist`, pokazanej wcześniej:

```
class AnnoyingLoudCyclist(AnnoyingMixin, LoudMixin, Cyclist):
    pass

AnnoyingLoudCyclist.__mro__
# (<class 'AnnoyingLoudCyclist'>, <class 'AnnoyingMixin'>,
# <class 'LoudMixin'>, <class 'Cyclist'>, <class 'object'>)
```

W tym MRO zobaczysz, jak spełnione są obie zasady. Każda klasa podrzędna jest zawsze wymieniona przed jej rodzicami. Klasa obiektu jest wymieniona jako ostatnia, ponieważ jest rodzicem wszystkich innych klas. Rodzice wymienione są w kolejności, w jakiej pojawiają się w kodzie.

Zachowanie funkcji `super()` jest powiązane z podstawowym MRO. Jego rolą jest delegowanie atrybutów do następnej klasy w MRO. Jest to oparte na klasie, w której używana jest `super()`. Na przykład gdy klasa `AnnoyingMixin` używa `super()`, sprawdza MRO instancji, aby znaleźć własną pozycję. Stamtąd deleguje wyszukiwanie atrybutów do następnej klasy. W tym przykładzie użycie `super().noise()` w klasie `AnnoyingMixin` wywołuje `LoudMixin.noise()`. Dzieje się tak, ponieważ `LoudMixin` to kolejna klasa wymieniona na MRO dla `AnnoyingLoudCyclist`. Operacja `super().noise()` w klasie `LoudMixin` jest następnie delegowana do klasy `Cyclist`. Przy każdym użyciu `super()` wybór następnej klasy różni się w zależności od typu instancji. Jeśli na przykład utworzysz instancję `AnnoyingTrombonist`, to `super().noise()` wywoła `Trombonist.noise()`.

Projektowanie pod kątem współdzielonego dziedziczenia wielokrotnego i domieszek jest wyzwaniem. Oto kilka wskazówek projektowych. Po pierwsze, klasy podrzędne są zawsze sprawdzane przed jakąkolwiek klasą bazową w MRO. Tak więc często zdarza się, że domieszki mają wspólnego rodzica i ten rodzic zapewnia pustą implementację metod. Jeśli w tym samym czasie używanych jest wiele klas domieszek, ustawią się one jedna za drugą. Wspólny rodzic pojawi się jako ostatni, gdzie może zapewnić domyślną implementację lub sprawdzenie błędów. Na przykład:

```
class NoiseMixin:
def noise(self):
    raise NotImplementedError('noise() nie zaimplementowana')

class LoudMixin(NoiseMixin):
    def noise(self):
        return super().noise().upper()

class AnnoyingMixin(NoiseMixin):
    def noise(self):
        return 3 * super().noise()
```

Po drugie, wszystkie implementacje metody domieszki powinny mieć identyczną sygnaturę funkcji. Jednym z problemów związanych z domieszkami jest to, że są one opcjonalne i często mieszają się ze sobą w nieprzewidywalnej kolejności. Aby implementacja zadziałała, musisz zagwarantować, że operacje obejmujące `super()` powiodą się, niezależnie od tego, jaka klasa będzie następna. Aby to zrobić, wszystkie metody w łańcuchu wywołań muszą mieć zgodną sygnaturę wywołania.

Na koniec musisz się upewnić, że wszędzie używasz metody `super()`. Czasami napotkasz klasę, która odwołuje się bezpośrednio do rodzica:

```
class Base:
    def yow(self):
        print('Base.yow')

class A(Base):
    def yow(self):
        print('A.yow')
        Base.yow(self) # Bezpośrednie odwołanie do rodzica
class B(Base):
    def yow(self):
        print('B.yow')
        super().yow(self)
```

```
class C(A, B):
    pass

c = C()
c.yow()
# Dane wyjściowe:
# A.yow
# Base.yow
```

Takie klasy z wielokrotnym dziedziczeniem nie są bezpieczne. Takie postępowanie przerywa prawidłowy łańcuch wywołań metod i powoduje zamieszanie. Na przykład w powyższym przykładzie żadne dane wyjściowe nigdy nie pojawiają się z `B.yow()`, mimo że jest częścią hierarchii dziedziczenia. Jeśli robisz cokolwiek z wielokrotnym dziedziczeniem, powinieneś używać metody `super()`, zamiast wykonywać bezpośrednie wywołania metod w superklasach.

7.20. Dyspozycja oparta na typie

Czasami trzeba napisać kod, który wykonuje dyspozycję (ang. *dispatch*) na podstawie określonego typu. Na przykład:

```
if isinstance(obj, Duck):
    handle_duck(obj)
elif isinstance(obj, Trombonist):
    handle_trombonist(obj)
elif isinstance(obj, Cyclist):
    handle_cyclist(obj)
else:
    raise RuntimeError('Nieznany obiekt')
```

Tak duży blok `if-elif-else` jest nieelegancki i delikatny. Często stosowanym rozwiązaniem jest wykonywanie dyspozycji poprzez słownik:

```
handlers = {
    Duck: handle_duck,
    Trombonist: handle_trombonist,
    Cyclist: handle_cyclist
}

# Dyspozycja
def dispatch(obj):
    func = handlers.get(type(obj))
    if func:
        return func(obj)
    else:
        raise RuntimeError(f'Brak obsługi dla {obj}')
```

To rozwiązanie zakłada dokładne dopasowanie typu. Jeśli w takiej dyspozycji ma być również obsługiwane dziedziczenie, tworzony będzie MRO:

```
def dispatch(obj):
    for ty in type(obj).__mro__:
        func = handlers.get(ty)
        if func:
            return func(obj)
    raise RuntimeError(f'Brak obsługi dla {obj}')
```

Czasami dyspozycja jest realizowana przez interfejs oparty na klasach za pomocą metody `getattr()`:

```
class Dispatcher:
    def handle(self, obj):
        for ty in type(obj).__mro__:
            meth = getattr(self, f'handle_{ty.__name__}', None)
            if meth:
                return meth(obj)
        raise RuntimeError(f'Brak obsługi dla {obj}')

def handle_Duck(self, obj):
    ...

def handle_Trombonist(self, obj):
    ...

def handle_Cyclist(self, obj):
    ...

# Przykład
dispatcher = Dispatcher()
dispatcher.handle(Duck()) # -> handle_Duck()
dispatcher.handle(Cyclist()) # -> handle_Cyclist()
```

Ten ostatni przykład użycia `getattr()` do wykonania dyspozycji metod klasy jest dość powszechnym wzorcem programowania.

7.21. Dekoratory klas

Czasami po zdefiniowaniu klasy chcesz wykonać dodatkowe kroki przetwarzania — takie jak dodanie klasy do rejestru lub wygenerowanie dodatkowego kodu pomocy. Jednym z podejść jest użycie dekoratora klas. Dekorator klas to funkcja, która pobiera klasę jako dane wejściowe i zwraca klasę jako dane wyjściowe. Oto przykład, jak możesz prowadzić rejestr:

```
_registry = { }
def register_decoder(cls):
    for mt in cls.mimetypes:
        _registry[mt.mimetype] = cls
    return cls

# Funkcja fabryczna, która korzysta z rejestru
def create_decoder(mimetype):
    return _registry[mimetype]()
```

W tym przykładzie funkcja `register_decoder()` szuka w klasie atrybutu `mimetypes`. Jeśli zostanie znaleziony, jest używany do dodania klasy do słownika mapującego typy MIME do obiektów klas. Aby użyć tej funkcji, zastosuj ją jako dekorator tuż przed definicją klasy:

```
@register_decoder
class TextDecoder:
    mimetypes = ['text/plain']
```

```
    def decode(self, data):
        ...

@register_decoder
class HTMLDecoder:
    mimetypes = ['text/html']
    def decode(self, data):
        ...

@register_decoder
class ImageDecoder:
    mimetypes = ['image/png', 'image/jpg', 'image/gif']
    def decode(self, data):
        ...

# Przykład
decoder = create_decoder('image/jpg')
```

Dekorator klas może dowolnie modyfikować zawartość podanej przez siebie klasy. Na przykład może nawet nadpisać istniejące metody. Jest to powszechna alternatywa dla klas domieszek lub wielokrotnego dziedziczenia. Rozważmy na przykład te dekoratory:

```
def loud(cls):
    orig_noise = cls.noise
    def noise(self):
        return orig_noise(self).upper()
    cls.noise = noise
    return cls

def annoying(cls):
    orig_noise = cls.noise
    def noise(self):
        return 3 * orig_noise(self)
    cls.noise = noise
    return cls

@annoying
@loud
class Cyclist(object):
    def noise(self):
        return 'Po lewej stronie!'

    def pedal(self):
        return 'Pedałowanie'
```

Ten przykład daje taki sam wynik jak przykład domieszki z poprzedniej sekcji. Nie ma tutaj jednak wielokrotnego dziedziczenia ani metody `super()`. W każdym dekoratorze wyszukiwanie `cls.noise` wykonuje tę samą akcję co `super()`. Ale ponieważ dzieje się to tylko raz, gdy zastosowany jest dekorator (w czasie definicji), wywołania `noise()` będą działać nieco szybciej.

Dekoratory klas mogą być również używane do tworzenia zupełnie nowego kodu. Na przykład typowym zadaniem podczas tworzenia klasy jest napisanie użytecznej metody `__repr__()` w celu usprawnienia debugowania:

```
class Point:
def __init__(self, x, y):
self.x = x
```

```
self.y = y
def __repr__(self):
return f'{type(self).__name__}({self.x!r}, {self.y!r})'
```

Pisanie takich metod bywa irytujące. Może zamiast tego dekorator klas mógłby stworzyć metodę?

```
import inspect
def with_repr(cls):
    args = list(inspect.signature(cls).parameters)
    argvals = ', '.join('{self.%s!r}' % arg for arg in args)
    code = 'def __repr__(self):\n'
    code += f' return f"{cls.__name__}({argvals})"\n'
    locs = { }
    exec(code, locs)
    cls.__repr__ = locs['__repr__']
    return cls

# Przykład
@with_repr
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

W tym przykładzie metoda `__repr__()` jest generowana na podstawie sygnatury wywołującej metody `__init__()`. Metoda jest tworzona jako ciąg tekstowy i przekazywana do `exec()` w celu utworzenia funkcji. Ta funkcja jest dołączona do klasy.

Podobne techniki generowania kodu są używane w częściach standardowej biblioteki. Na przykład wygodnym sposobem definiowania struktur danych jest użycie `dataclass`:

```
from dataclasses import dataclass
@dataclass
class Point:
    x: int
    y: int
```

`Dataclass` (klasa danych) automatycznie tworzy metody, takie jak `__init__()` i `__repr__()`, na podstawie wskazówek typu klasy. Metody są tworzone przy użyciu `exec()`, podobnie jak w poprzednim przykładzie.

Oto jak działa klasa `Point`:

```
>>> p = Point(2, 3)
>>> p
Point(x=2, y=3)
>>>
```

Jedną z wad takiego podejścia jest słaba wydajność uruchamiania. Dynamiczne tworzenie kodu za pomocą `exec()` omija optymalizacje kompilacji, które Python zwykle stosuje do modułów. Zdefiniowanie dużej liczby klas w ten sposób może zatem znacznie spowolnić importowanie kodu.

Przykłady pokazane w tej sekcji ilustrują typowe zastosowania dekoratorów klas: rejestrację, przepisywanie kodu, generowanie kodu, walidację i tak dalej. Jednym z problemów z dekoratorami klas jest to, że muszą być one jawnie stosowane do każdej klasy, w której są używane. Nie zawsze jest to pożądane. W następnej sekcji opisano funkcję, która pozwala na niejawną manipulację klasami.

7.22. Nadzorowane dziedziczenie

Jak widziałeś w poprzedniej sekcji, czasami chcesz zdefiniować klasę i wykonać dodatkowe akcje. Dekorator klas jest jednym z mechanizmów stosowanych do tego celu. Jednak klasa nadrzędna może również wykonywać dodatkowe akcje w imieniu swoich podklas. Jest to realizowane przez zaimplementowanie metody klasy `__init_subclass__(cls)`. Na przykład:

```
class Base:
    @classmethod
    def __init_subclass__(cls):
        print('Inicjalizacja', cls)

# Przykład (powinieneś zobaczyć komunikat „Inicjalizacja" dla każdej klasy)
class A(Base):
    pass

class B(A):
    pass
```

Jeśli istnieje metoda `__init_subclass__()`, jest ona wyzwalana automatycznie po zdefiniowaniu dowolnej klasy podrzędnej. Dzieje się tak nawet wtedy, gdy dziecko jest głęboko schowane w hierarchii dziedziczenia.

Wiele zadań często wykonywanych za pomocą dekoratorów klas można zamiast tego wykonać przy użyciu `__init_subclass__()`. Na przykład rejestracja klas może przebiegać następująco:

```
class DecoderBase:
    _registry = { }
    @classmethod
    def __init_subclass__(cls):
        for mt in cls.mimetypes:
            DecoderBase._registry[mt.mimetype] = cls

# Funkcja fabryczna, korzystająca z rejestru
def create_decoder(mimetype):
    return DecoderBase._registry[mimetype]()

class TextDecoder(DecoderBase):
    mimetypes = ['text/plain']
    def decode(self, data):
        ...

class HTMLDecoder(DecoderBase):
    mimetypes = ['text/html']
    def decode(self, data):
        ...

class ImageDecoder(DecoderBase):
    mimetypes = ['image/png', 'image/jpg', 'image/gif']
    def decode(self, data):
        ...

# Przykład
decoder = create_decoder('image/jpg')
```

Oto przykład klasy, która automatycznie tworzy metodę `__repr__()` na podstawie sygnatury metody `__init__()` klasy:

```
import inspect

class Base:
    @classmethod
    def __init_subclass__(cls):
        # Tworzenie metody __repr__
        args = list(inspect.signature(cls).parameters)
        argvals = ', '.join('{self.%s!r}' % arg for arg in args)
        code = 'def __repr__(self):\n'
        code += f' return f"{cls.__name__}({argvals})"\n'
        locs = { }
        exec(code, locs)
        cls.__repr__ = locs['__repr__']

class Point(Base):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Jeśli używane jest dziedziczenie wielokrotne, należy zastosować metodę `super()`, aby upewnić się, że wszystkie klasy implementujące `__init_subclass__()` zostaną wywołane. Na przykład:

```
class A:
    @classmethod
    def __init_subclass__(cls):
        print('A.init_subclass')
        super().__init_subclass__()

class B:
    @classmethod
    def __init_subclass__(cls):
        print('B.init_subclass')
        super().__init_subclass__()

# Powinieneś zobaczyć wynik działania obu klas
class C(A, B):
    pass
```

Nadzorowanie dziedziczenia za pomocą `__init_subclass__()` to jedna z najpotężniejszych funkcji Pythona. Duża część jego mocy pochodzi z ukrytej natury. Klasa bazowa najwyższego poziomu może to wykorzystać do cichego nadzorowania całej hierarchii klas podrzędnych. Taki nadzór może rejestrować klasy, przepisywać metody, przeprowadzać walidację i nie tylko.

7.23. Cykl życia obiektu i zarządzanie pamięcią

Kiedy klasa jest zdefiniowana, wynikowa klasa jest fabryką do tworzenia nowych instancji. Na przykład:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

# Tworzenie kilku instancji Account
a = Account('Guido', 1000.0)
b = Account('Eva', 25.0)
```

Tworzenie instancji odbywa się w dwóch krokach przy użyciu specjalnej metody `__new__()`, która tworzy nową instancję, oraz `__init__()`, która ją inicjuje. Na przykład operacja `a = Account('Guido', 1000.0)` wykonuje następujące kroki:

```
a = Account.__new__(Account, 'Guido', 1000.0)
if isinstance(a, Account):
    Account.__init__('Guido', 1000.0)
```

Z wyjątkiem pierwszego argumentu, który jest klasą, a nie instancją, `__new__()` zwykle otrzymuje te same argumenty co `__init__()`. Jednak domyślna implementacja `__new__()` po prostu je ignoruje. Czasami zobaczysz `__new__()` wywołane tylko jednym argumentem. Na przykład ten kod również działa:

```
a = Account.__new__(Account)
Account.__init__('Guido', 1000.0)
```

Bezpośrednie użycie metody `__new__()` jest rzadkością, ale czasami jest ona używana do tworzenia instancji z pominięciem wywołania metody `__init__()`. Jednym z takich zastosowań są metody klasowe. Na przykład:

```
import time

class Date:
    def __init__(self, year, month, day):
        self.year = year
        self.month = month
        self.day = day

    @classmethod
    def today(cls):
        t = time.localtime()
        self = cls.__new__(cls) # Stwórz instancję
        self.year = t.tm_year
        self.month = t.tm_month
        self.day = t.tm_day
        return self
```

Moduły wykonujące serializację obiektów, takie jak `pickle`, również wykorzystują `__new__()` do odtworzenia instancji, gdy obiekty są deserializowane. Odbywa się to bez wywoływania `__init__()`.

Czasami klasa definiuje `__new__()`, jeśli chce zmienić jakiś aspekt tworzenia instancji. Typowe aplikacje obejmują buforowanie instancji, singletony i niezmienność. Na przykład możesz chcieć, aby klasa `Date` wykonywała kopiowanie daty, czyli buforowanie i ponowne używanie wystąpień `Date`, które mają identyczny rok, miesiąc i dzień. Oto jeden ze sposobów, który można wdrożyć:

```
class Date:
    _cache = { }

    @staticmethod
    def __new__(cls, year, month, day):
        self = Date._cache.get((year,month,day))
        if not self:
            self = super().__new__(cls)
            self.year = year
            self.month = month
            self.day = day
            Date._cache[year,month,day] = self
        return self

    def __init__(self, year, month, day):
        pass

# Przykład
d = Date(2012, 12, 21)
e = Date(2012, 12, 21)
assert d is e # Ten sam obiekt
```

W tym przykładzie klasa utrzymuje wewnętrzny słownik wcześniej utworzonych instancji `Date`. Podczas tworzenia nowego obiektu `Date` najpierw zaglądamy do pamięci podręcznej. Jeśli zostanie znalezione dopasowanie, instancja jest zwracana. W przeciwnym razie tworzona jest nowa instancja i jest inicjowana.

Subtelnym szczegółem tego rozwiązania jest pusta metoda `__init__()`. Nawet jeśli instancje są buforowane, każde odwołanie do `Date()` wciąż wywołuje `__init__()`. Aby uniknąć powielonego obciążenia, metoda nic nie robi — tworzenie instancji odbywa się w `__new__()`, gdy instancja jest tworzona za pierwszym razem.

Istnieją sposoby uniknięcia dodatkowego odwołania do `__init__()`, ale wymaga to podstępnych sztuczek. Jednym ze sposobów jest taki, aby `__new__()` zwracała zupełnie inny typ instancji, na przykład należący do innej klasy. Kolejne rozwiązanie, opisane później, wymaga użycia metaklasy.

Po utworzeniu instancje są zarządzane za pomocą zliczania referencji. Jeśli liczba referencji osiąga zero, instancja zostanie natychmiast zniszczona. Gdy instancja ma zostać zniszczona, interpreter najpierw wyszukuje metodę `__del__()` powiązaną z obiektem i wywołuje ją. Na przykład:

```
class Account(object):
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __del__(self):
        print('Usuwanie obiektu Account')
```

```
>>> a = Account('Guido', 1000.0)
>>> del a
Usuwanie obiektu Account
>>>
```

Od czasu do czasu program użyje instrukcji `del`, aby usunąć odniesienie do obiektu, jak pokazano. Jeśli powoduje to, że liczba referencji do obiektu osiąga zero, wywoływana jest metoda `__del__()`. Jednak powszechnie instrukcja `del` nie wywołuje bezpośrednio `__del__()`,

ponieważ mogą występować inne odniesienia do obiektów — w innym miejscu. Istnieje wiele innych sytuacji, kiedy obiekt może zostać usunięty — na przykład przy ponownym przypisaniu nazwy zmiennej lub przy wyjściu zmiennej poza zakres w funkcji:

```
>>> a = Account('Guido', 1000.0)
>>> a = 42
Usuwanie obiektu Account
>>> def func():
...     a = Account('Guido', 1000.0)
...
>>> func()
Usuwanie obiektu Account
>>>
```

W praktyce rzadko jest konieczne definiowanie metody `__del__()` dla klasy. Występują wyjątki, gdy zniszczenie obiektu wymaga dodatkowej akcji czyszczenia, na przykład w przypadku zamknięcia pliku, zamknięcia połączenia sieciowego lub uwolnienia innych zasobów systemowych. Nawet w tych przypadkach niebezpiecznie jest polegać na `__del__()`, ponieważ nie ma gwarancji, że ta metoda zostanie wywołana, kiedy jest potrzebna. W przypadku prawidłowego zwalniania zasobu należy na obiekcie wywołać metodę `close()`. Powinieneś się również upewnić, że Twoja klasa ma wsparcie dla protokołu menedżera kontekstu, dzięki czemu można skorzystać z instrukcji `with`. Oto przykład, który obejmuje wszystkie przypadki:

```
class SomeClass:
    def __init__(self):
        self.resource = open_resource()

    def __del__(self):
        self.close()

    def close(self):
        self.resource.close()

    def __enter__(self):
        return self

    def __exit__(self, ty, val, tb):
        self.close()

# Zamknięcie za pomocą __del__()
s = SomeClass()
del s

# Wywołanie close
s = SomeClass()
s.close()

# Zamknięcie na końcu kontekstu bloku
with SomeClass() as s:
    ......
```

Ponownie należy podkreślić, że tworzenie metody `__del__()` w klasie prawie nigdy nie jest konieczne. Python ma już mechanizm odśmiecania pamięci i po prostu nie ma potrzeby tego robić, chyba że istnieje dodatkowe działanie, które musi się odbywać po zniszczeniu obiektów.

Nawet wtedy nadal możesz nie potrzebować metody `__del__()`, ponieważ możliwe jest, że obiekt jest już zaprogramowany do prawidłowego oczyszczenia się, nawet jeśli nic nie zrobisz.

I choć istnieje dużo niebezpieczeństw związanych z liczeniem referencji i zniszczeniem obiektu, dostępne są pewne rodzaje wzorców programowania — zwłaszcza tych, które obejmują relacje rodzic – dziecko, wykresy lub buforowanie — gdzie obiekty mogą tworzyć tak zwany *cykl referencji* (ang. *reference cycle*). Oto przykład:

```
class SomeClass:
    def __del__(self):
        print('Usuwanie')

parent = SomeClass()
child = SomeClass()

# Tworzenie cyklu referencji dziecko – rodzic
parent.child = child
child.parent = parent

# Próba usuwania (nie pojawia się wynik działania __del__ )
del parent
del child
```

W tym przykładzie nazwy zmiennych zostaną zniszczone, ale nigdy nie widzisz wykonania metody `__del__()`. Każdy z dwóch obiektów trzyma wzajemnie odwołania wewnętrzne, nie ma więc możliwości, aby licznik referencji kiedykolwiek spadł do 0. Aby sobie z tym poradzić, działa specjalny mechanizm odśmiecania pamięci, kontrolujący cykl. Ostatecznie obiekty zostaną zwolnione, ale trudno przewidzieć, kiedy może się to wydarzyć. Jeśli chcesz wymusić odśmiecanie, możesz wywołać `gc.collect()`. Moduł `gc` ma wiele innych funkcji związanych z cyklicznym odśmiecaniem pamięci i jej monitorowaniem.

Ze względu na nieprzewidywalny termin odśmiecania metoda `__del__()` ma kilka ograniczeń. Po pierwsze, każdy wyjątek, który propaguje metodę `__del__()`, jest wysyłany do `sys.stderr` — w przeciwnym wypadku jest ignorowany. Po drugie, metoda `__del__()` powinna unikać operacji, takich jak tworzenie blokad lub innych zasobów. Może to spowodować zakleszczenie, gdy `__del__()` jest niespodziewanie wywoływany w środku wykonywania niepowiązanej funkcji. Jeśli musisz zdefiniować metodę `__del__()`, niech będzie prosta.

7.24. Słabe referencje

Czasami obiekty działają, mimo że chciałbyś, aby było inaczej. We wcześniejszym przykładzie omówiono klasę `Date` z wewnętrznym buforowaniem instancji. Jednym z problemów tego przykładu jest to, że nie ma sposobu, aby instancja została usunięta z pamięci podręcznej. Jako taka pamięć podręczna będzie stale rosła.

Jednym ze sposobów naprawienia tego problemu jest stworzenie słabej referencji za pomocą modułu `weakref`. Słabe referencje to sposób na tworzenie odniesienia do obiektu bez zwiększenia liczby referencji. Aby wspierać w programie słabe referencje, musisz dopisać dodatkowy kod, aby sprawdzić, czy obiekt, o którym mowa, nadal istnieje. Przykład:

```
>>> a = Account('Guido', 1000.0)
>>> import weakref
>>> a_ref = weakref.ref(a)
>>> a_ref
<weakref at 0x104617188; to 'Account' at 0x1046105c0>
>>>
```

W przeciwieństwie do normalnego odniesienia słaba referencja umożliwia usunięcie oryginalnego obiektu. Na przykład:

```
>>> del a
>>> a_ref
<weakref at 0x104617188; dead>
>>>
```

Słaba referencja zawiera opcjonalne odniesienie do obiektu. Aby uzyskać rzeczywisty obiekt, musisz wywołać słabą referencję jako funkcję bez argumentów. Spowoduje to zwrócenie obiektu, który jest wskazany, lub `None`. Na przykład:

```
acct = a_ref()
if acct is not None:
    acct.withdraw(10)

# Alternatywnie
if acct := a_ref():
    acct.withdraw(10)
```

Słabe referencje są powszechnie stosowane w połączeniu z buforowaniem i innym zaawansowanym zarządzaniem pamięcią. Oto zmodyfikowana wersja klasy `Date`, która automatycznie usuwa obiekty z pamięci podręcznej, gdy nie ma więcej odniesień:

```
import weakref

class Date:
    _cache = { }

@staticmethod
    def __new__(cls, year, month, day):
        selfref = Date._cache.get((year,month,day))
        if not selfref:
            self = super().__new__(cls)
            self.year = year
            self.month = month
            self.day = day
            Date._cache[year,month,day] = weakref.ref(self)
        else:
            self = selfref()
        return self

    def __init__(self, year, month, day):
        pass

    def __del__(self):
        del Date._cache[self.year,self.month,self.day]
```

Zrozumienie kodu może wymagać odrobiny czasu, ale poniższa interaktywna sesja pokazuje, jak to działa. Zauważ, że kiedy wpis zostanie usunięty z pamięci podręcznej, nie ma już do niego więcej odniesień:

```
>>> Date._cache
{}
>>> a = Date(2012, 12, 21)
>>> Date._cache
{(2012, 12, 21): <weakref at 0x10c7ee2c8; to 'Date' at 0x10c805518>}
>>> b = Date(2012, 12, 21)
>>> a is b
True
>>> del a
>>> Date._cache
{(2012, 12, 21): <weakref at 0x10c7ee2c8; to 'Date' at 0x10c805518>}
>>> del b
>>> Date._cache
{}
>>>
```

Jak wspomniano wcześniej, metoda `__del__()` jest wywoływana tylko wtedy, gdy liczba referencji obiektu osiąga wartość zero. W tym przykładzie pierwsza instrukcja `del` zmniejsza liczbę referencji. Ponieważ nadal jest jeszcze inne odniesienie do tego samego obiektu, obiekt pozostaje w `Date._cache`. Gdy drugi obiekt zostanie usunięty, jest wywołana metoda `__del__()`, a pamięć podręczna zostaje wyczyszczona.

Wsparcie dla słabych referencji wymaga, aby instancje posiadały modyfikowalny atrybut `__weakref__`. Instancje klas zdefiniowanych przez użytkownika zwykle domyślnie mają taki atrybut. Jednak wbudowane typy i pewne rodzaje specjalnych struktur danych, takie jak tuple czy klasy ze slotami — nie. Jeśli chcesz skonstruować słabe referencje do tych typów, możesz to zrobić, definiując warianty z dodanym atrybutem `__weakref__`:

```
class wdict(dict):
    __slots__ = ('__weakref__',)

w = wdict()
w_ref = weakref.ref(w) # Teraz działa
```

Korzystanie ze slotów ma na celu uniknięcie marnowania pamięci, jak wkrótce wyjaśnimy.

7.25. Wewnętrzna reprezentacja obiektów i wiązanie atrybutu

Stan powiązany z instancją jest przechowywany w słowniku, który jest dostępny jako atrybut instancji `__dict__`. Ten słownik zawiera dane, które są unikalne dla każdej instancji. Oto przykład:

```
>>> a = Account('Guido', 1100.0)
>>> a.__dict__
{'owner': 'Guido', 'balance': 1100.0}
```

Nowe atrybuty można dodać do instancji w dowolnym momencie:

```
a.number = 123456 # Dodanie atrybutu 'number' do a.__dict__
a.__dict__['number'] = 654321
```

Modyfikacje instancji są zawsze odzwierciedlane w lokalnym atrybucie `__dict__`, chyba że atrybut jest zarządzany przez właściwość. Podobnie, jeśli dokonujesz modyfikacji `__dict__` bezpośrednio, te modyfikacje są odzwierciedlone w atrybutach.

Instancje linkują z powrotem do swojej klasy za pomocą specjalnego atrybutu `__class__`. Sama klasa tworzy również tylko cienką warstwę nad słownikiem, który można znaleźć w atrybucie `__dict__`. Słownik klasy jest miejscem, w którym znajdują się metody. Na przykład:

```
>>> a.__class__
<class '__main__.Account'>
>>> Account.__dict__.keys()
dict_keys(['__module__', '__init__', '__repr__', 'deposit', 'withdraw',
'inquiry', '__dict__', '__weakref__', '__doc__'])
>>> Account.__dict__['withdraw']
<function Account.withdraw at 0x108204158>
>>>
```

Klasy są połączone ze swoimi klasami bazowymi przez specjalny atrybut `__bases__`, który jest krotką klas bazowych. Atrybut `__bases__` ma jedynie charakter informacyjny. Rzeczywista implementacja dziedziczenia używa atrybutu `__mro__`, który jest krotką wszystkich klas nadrzędnych wymienionych w kolejności wyszukiwania. Ta struktura jest podstawą wszystkich operacji, które pobierają, ustawiają lub usuwają atrybuty instancji.

Za każdym razem, gdy atrybut jest ustawiany za pomocą `obj.name = value`, wywoływana jest specjalna metoda `obj.__setattr__('name', value)`. Jeśli atrybut zostanie usunięty za pomocą `del obj.name`, wywoływana jest specjalna metoda `obj.__delattr__('name')`. Domyślnym zachowaniem tych metod jest modyfikowanie lub usuwanie wartości z lokalnego atrybutu `__dict__` obiektu `obj`, chyba że żądany atrybut odpowiada właściwości lub deskryptorowi. W takim przypadku operacje ustawiania i usuwania będą wykonywane przez funkcje ustawiania i usuwania skojarzone z właściwością.

W przypadku wyszukiwania atrybutów, takich jak `obj.name`, wywoływana jest specjalna metoda `obj.__getattribute__('name')`. Ta metoda przeprowadza wyszukiwanie atrybutu, które zwykle obejmuje sprawdzenie właściwości, przeszukanie lokalnego `__dict__`, sprawdzenie słownika klas i przeszukanie MRO. Jeśli to wyszukiwanie się nie powiedzie, podejmowana jest ostateczna próba znalezienia atrybutu przez wywołanie metody `obj.__getattr__('name')` klasy (jeśli jest zdefiniowana). Jeśli to się nie uda, zostanie zgłoszony wyjątek `AttributeError`.

W razie potrzeby klasy zdefiniowane przez użytkownika mogą implementować własne wersje funkcji dostępu do atrybutów. Oto przykład klasy sprawdzającej nazwy atrybutów, które można ustawić:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def __setattr__(self, name, value):
        if name not in {'owner', 'balance'}:
```

```
            raise AttributeError(f'Brak atrybutu {name}')
        super().__setattr__(name, value)

# Przykład
a = Account('Guido', 1000.0)
a.balance = 940.25 # OK
a.amount = 540.2 # AttributeError. Brak atrybutu amount
```

Klasa, która reimplementuje te metody, powinna polegać na domyślnej implementacji dostarczonej przez `super()` w celu wykonania rzeczywistej pracy związanej z manipulowaniem atrybutem. Dzieje się tak, ponieważ domyślna implementacja dba o bardziej zaawansowane funkcje klas, takie jak deskryptory i właściwości. Jeśli nie używasz `super()`, będziesz musiał sam zadbać o te szczegóły.

7.26. Proxy, wrapper i delegacje

Czasami klasy implementują warstwę opakowującą (ang. *wrapper layer*) wokół innego obiektu, aby stworzyć coś w rodzaju obiektu proxy. Proxy to obiekt, który udostępnia ten sam interfejs co inny obiekt, ale z jakiegoś powodu nie jest powiązany z oryginalnym obiektem poprzez dziedziczenie. Różni się to od kompozycji, w której z innych obiektów tworzony jest całkowicie nowy obiekt, ale z własnym, unikalnym zestawem metod i atrybutów.

Istnieje wiele rzeczywistych scenariuszy, w których występuje takie rozwiązanie. Na przykład w obliczeniach rozproszonych rzeczywista implementacja obiektu może się znajdować na zdalnym serwerze w chmurze. Klienci, którzy wchodzą w interakcję z tym serwerem, mogą używać serwera proxy, który wygląda jak obiekt na serwerze, ale w tle deleguje wszystkie wywołania metod za pośrednictwem komunikatów sieciowych.

Powszechną techniką implementacji serwerów proxy jest metoda `__getattr__()`.

Oto prosty przykład:

```
class A:
    def spam(self):
        print('A.spam')

    def grok(self):
        print('A.grok')

    def yow(self):
        print('A.yow')

class LoggedA:
    def __init__(self):
        self._a = A()

    def __getattr__(self, name):
        print("Uzyskiwanie dostępu do", name)
        # Delegacja do wewnętrznej instancji A
        return getattr(self._a, name)

# Przykład
a = LoggedA()
```

```
a.spam()  # Wyświetla "Uzyskiwanie dostępu do spam" i "A.spam"
a.yow()   # Wyświetla "Uzyskiwanie dostępu do  yow" i "A.yow"
```

Delegowanie jest czasami używane jako alternatywa dla dziedziczenia. Oto przykład:

```
class A:
    def spam(self):
        print('A.spam')

    def grok(self):
        print('A.grok')

    def yow(self):
        print('A.yow')

class B:
    def __init__(self):
        self._a = A()

    def grok(self):
        print('B.grok')

    def __getattr__(self, name):
        return getattr(self._a, name)

# Przykład
b = B()
b.spam()    # -> A.spam
b.grok()    # -> B.grok (redefiniowana metoda)
b.yow()     # -> A.yow
```

W tym przykładzie wydaje się, że klasa `B` może dziedziczyć po klasie `A` i redefiniować pojedynczą metodę. Tak wynika z obserwacji — jednak dziedziczenie nie jest używane. Zamiast tego `B` zawiera wewnętrzne odniesienie do `A`. Niektóre metody `A` można przedefiniować. Jednak wszystkie pozostałe metody są delegowane za pomocą metody `__getattr__()`.

Technika przekazywania wyszukiwania atrybutów za pośrednictwem `__getattr__()` jest powszechną praktyką. Należy jednak pamiętać, że nie dotyczy to operacji mapowanych na metody specjalne. Rozważmy na przykład tę klasę:

```
class ListLike:
    def __init__(self):
        self._items = list()

    def __getattr__(self, name):
        return getattr(self._items, name)

# Przykład
a = ListLike()
a.append(1)      # Działa
a.insert(0, 2)   # Działa
a.sort()         # Działa
len(a)           # Nie powiodło się. Brak metody __len__()
a[0]             # Nie powiodło się. Brak metody __getitem__()
```

W tym przypadku klasa pomyślnie przekazuje wszystkie standardowe metody list (`list.sort()`, `list.append()` itd.) do listy wewnętrznej. Jednak żaden ze standardowych operatorów Pythona nie działa. Aby to zmienić, musiałbyś jawnie zaimplementować wymagane metody specjalne. Na przykład:

```
class ListLike:
    def __init__(self):
        self._items = list()

    def __getattr__(self, name):
        return getattr(self._items, name)

    def __len__(self):
        return len(self._items)

    def __getitem__(self, index):
        return self._items[index]

    def __setitem__(self, index, value):
        self._items[index] = value
```

7.27. Zmniejszenie wykorzystania pamięci za pomocą __slots__

Jak widzieliśmy, instancja przechowuje swoje dane w słowniku. Jeśli tworzysz dużą liczbę instancji, może to spowodować duże obciążenie pamięci. Jeśli wiesz, że nazwy atrybutów są stałe, możesz określić je w specjalnej zmiennej klasy o nazwie `__slots__`.

Oto przykład:

```
class Account(object):
    __slots__ = ('owner', 'balance')
    ...
```

Sloty są wskazówkami dotyczącymi definicji, która umożliwia Pythonowi optymalizację wydajności zarówno pod kątem wykorzystania pamięci, jak i szybkości wykonywania. Instancje klasy wykorzystujące `__slots__` nie używają już słownika do przechowywania danych instancji. Zamiast tego wykorzystywana jest znacznie bardziej zwarta struktura danych oparta na tablicy. W programach, które tworzą dużą liczbę obiektów, zastosowanie `__slots__` może skutkować znacznym zmniejszeniem użycia pamięci i niewielkim skróceniem czasu wykonywania.

Jedyne wpisy zawarte w `__slots__` to atrybuty instancji. Nie wymieniasz metod, właściwości, zmiennych klas ani żadnych innych atrybutów na poziomie klasy. Zasadniczo są to te same nazwy, które zwykle pojawiają się jako klucze słownika w `__dict__` instancji.

Pamiętaj, że `__slots__` mają problematyczną interakcję z dziedziczeniem. Jeśli klasa dziedziczy z klasy bazowej, która używa `__slots__`, musi również zdefiniować `__slots__` do przechowywania własnych atrybutów (nawet jeśli nie dodaje żadnych), aby skorzystać z możliwości, jakie zapewniają. Jeśli o tym zapomnisz, klasa pochodna będzie działać wolniej i zużyje jeszcze więcej pamięci, niż gdyby `__slots__` nie były używane w żadnej z klas!

`__slots__` są niezgodne z dziedziczeniem wielokrotnym. Jeśli określono wiele klas bazowych, każdą z niepustymi slotami, otrzymasz wyjątek `TypeError`.

Użycie `__slots__` może również zepsuć kod, który oczekuje, że instancje będą miały bazowy atrybut `__dict__`. Chociaż często nie dotyczy to kodu użytkownika, biblioteki narzędziowe i inne narzędzia do obsługi obiektów mogą być zaprogramowane do przeglądania `__dict__` w celu debugowania, serializacji obiektów i innych operacji.

Obecność `__slots__` nie ma wpływu na wywoływanie metod, takich jak `__getattribute__()`, `__getattr__()` i `__setattr__()`, jeśli zostaną przedefiniowane w klasie. Jeśli jednak wdrażasz takie metody, pamiętaj, że nie ma już żadnego atrybutu instancji `__dict__`. Twoja implementacja będzie musiała to uwzględnić.

7.28. Deskryptory

Standardowo dostęp do atrybutów odpowiada operacjom słownikowym. Jeśli potrzebna jest większa kontrola, dostęp do atrybutów może być kierowany przez zdefiniowane przez użytkownika funkcje `get`, `set` i `delete`. Wykorzystanie właściwości zostało już opisane. Jednak właściwość jest faktycznie implementowana przy użyciu konstrukcji niższego poziomu znanej jako deskryptor. Deskryptor to obiekt na poziomie klasy, który zarządza dostępem do atrybutu. Implementując jedną lub więcej specjalnych metod `__get__()`, `__set__()` i `__delete__()`, możesz podłączyć się bezpośrednio do mechanizmu dostępu do atrybutów i dostosować te operacje. Oto przykład:

```
class Typed:
    expected_type = object

    def __set_name__(self, cls, name):
        self.key = name

    def __get__(self, instance, cls):
        if instance:
            return instance.__dict__[self.key]
        else:
            return self

    def __set__(self, instance, value):
        if not isinstance(value, self.expected_type):
            raise TypeError(f'Oczekiwano {self.expected_type}')
        instance.__dict__[self.key] = value

    def __delete__(self,instance):
        raise AttributeError("Nie można usunąć atrybutu")

class Integer(Typed):
    expected_type = int

class Float(Typed):
    expected_type = float
```

```
class String(Typed):
    expected_type = str

# Przykład:
class Account:
    owner = String()
    balance = Float()

    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance
```

W tym przykładzie klasa `Typed` definiuje deskryptor, w którym sprawdzanie typu jest wykonywane po przypisaniu atrybutu, a w przypadku próby usunięcia atrybutu generowany jest błąd. Podklasy `Integer`, `Float` i `String` wykorzystują `Type` w celu dopasowania do określonego typu. Użycie tych klas w innej klasie (takiej jak `Account`) powoduje, że te atrybuty automatycznie wywołują odpowiednie metody `__get__()`, `__set__()` lub `__delete__()`. Na przykład:

```
a = Account('Guido', 1000.0)
b = a.owner          # Wywołuje Account.owner.__get__(a, Account)
a.owner = 'Eva'      # Wywołuje Account.owner.__set__(a, 'Eva')
del f.owner          # Wywołuje Account.owner.__delete__(a)
```

Deskryptory można tworzyć tylko na poziomie klasy. Niedozwolone jest tworzenie deskryptorów na podstawie instancji poprzez tworzenie obiektów deskryptorów wewnątrz `__init__()` i innych metod. Metoda `__set_name__()` deskryptora jest wywoływana po zdefiniowaniu klasy, ale przed utworzeniem jakichkolwiek instancji, aby poinformować deskryptor o nazwie użytej w klasie. Na przykład definicja `balance = Float()` wywołuje `Float.__set_name__(Account, 'balance')`, aby poinformować deskryptor klasy o używanej nazwie.

Deskryptory z metodą `__set__()` zawsze mają pierwszeństwo przed elementami w słowniku instancji. Jeśli na przykład deskryptor ma taką samą nazwę jak klucz w słowniku instancji, to deskryptor ma pierwszeństwo. W powyższym przykładzie `Account` zobaczysz deskryptor stosujący sprawdzanie typu, mimo że słownik instancji zawiera pasujący wpis:

```
>>> a = Account('Guido', 1000.0)
>>> a.__dict__
{'owner': 'Guido', 'balance': 1000.0}
>>> a.balance = 'a lot'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "descrip.py", line 63, in __set__
    raise TypeError(f'Oczekiwano {self.expected_type}')
TypeError: Expected <class 'float'>
>>>
```

Metoda `__get__(instance, cls)` deskryptora pobiera argumenty zarówno dla instancji, jak i dla klasy. Możliwe jest, że `__get__()` jest wywoływana na poziomie klasy, w którym to przypadku argumentem instancji jest `None`. W większości przypadków `__get__()` zwraca deskryptor, jeśli nie podano żadnej instancji. Na przykład:

```
>>> Account.balance
<__main__.Float object at 0x110606710>
>>>
```

Deskryptor, który implementuje tylko `__get__()`, jest nazywany deskryptorem metody. Ma słabsze powiązanie niż deskryptor z obiema możliwościami pobierania/ustawiania. Metoda `__get__()` deskryptora metody jest wywoływana tylko wtedy, gdy w słowniku instancji nie ma pasującego wpisu. Powodem, dla którego nazywa się go deskryptorem metody, jest to, że ten rodzaj deskryptora jest używany głównie do implementacji różnych typów metod Pythona — w tym metod instancji, metod klas i metod statycznych.

Oto przykładowy szkielet implementacji, który pokazuje, jak `@classmethod` i `@staticmethod` mogą być zaimplementowane od zera (rzeczywista implementacja jest bardziej wydajna):

```
import types
class classmethod:
    def __init__(self, func):
        self.__func__ = func

    # Zwróć metodę powiązaną z cls jako pierwszy argument
    def __get__(self, instance, cls):
        return types.MethodType(self.__func__, cls)

class staticmethod:
    def __init__(self, func):
        self.__func__ = func

    # Zwróć samą funkcję
    def __get__(self, instance, cls):
        return self.__func__
```

Ponieważ deskryptory metod działają tylko wtedy, gdy w słowniku instancji nie ma pasującego wpisu, można ich również użyć do zaimplementowania różnych form leniwego wartościowania (ang. *lazy evaluation*) atrybutów. Na przykład:

```
class Lazy:
    def __init__(self, func):
        self.func = func

    def __set_name__(self, cls, name):
        self.key = name

    def __get__(self, instance, cls):
        if instance:
            value = self.func(instance)
            instance.__dict__[self.key] = value
            return value
        else:
            return self

class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    area = Lazy(lambda self: self.width * self.height)
    perimeter = Lazy(lambda self: 2*self.width + 2*self.height)
```

W tym przykładzie `area` i `perimeter` są atrybutami obliczanymi na żądanie i przechowywanymi w słowniku instancji. Po obliczeniu wartości są po prostu zwracane bezpośrednio ze słownika instancji.

```
>>> r = Rectangle(3, 4)
>>> r.__dict__
{'width': 3, 'height': 4}
>>> r.area
12
>>> r.perimeter
14
>>> r.__dict__
{'width': 3, 'height': 4, 'area': 12, 'perimeter': 14}
>>>
```

7.29. Proces definicji klasy

Definicja klasy jest procesem dynamicznym. Kiedy definiujesz klasę za pomocą instrukcji `class`, tworzony jest nowy słownik, który służy jako lokalna przestrzeń nazw klas. Następnie ciało klasy jest wykonywane jako skrypt w tej przestrzeni nazw. Ostatecznie przestrzeń nazw staje się atrybutem `__dict__` wynikowego obiektu klasy.

W treści klasy dozwolone są wszelkie dopuszczalne instrukcje Pythona. Zwykle po prostu definiujesz funkcje i zmienne, ale dozwolone są także kontrola przepływu, importy, klasy zagnieżdżone i wszystko inne. Oto przykład klasy, która warunkowo definiuje metody:

```
debug = True

class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    if debug:
        import logging
        log = logging.getLogger(f'{__module__}.{__qualname__}')
        def deposit(self, amount):
            Account.log.debug('Wpłata %f', amount)
            self.balance += amount

        def withdraw(self, amount):
            Account.log.debug('Wypłata %f', amount)
            self.balance -= amount
    else:
        def deposit(self, amount):
            self.balance += amount

        def withdraw(self, amount):
            self.balance -= amount
```

W tym przykładzie debugowanie zmiennych globalnych jest używane do warunkowego definiowania metod. Zmienne `__qualname__` i `__module__` są predefiniowanymi ciągami, które przechowują informacje o nazwie klasy i module obejmującym. Mogą być używane przez instrukcje w treści klasy. W tym przykładzie zostały one wykorzystane do konfigurowania systemu logowania. Prawdopodobnie istnieją czystsze sposoby organizowania powyższego kodu, ale kluczową kwestią jest to, że możesz umieścić w klasie wszystko, co chcesz.

Jednym z krytycznych punktów definicji klasy jest to, że przestrzeń nazw używana do przechowywania zawartości treści klasy nie jest zakresem zmiennych. Każda nazwa stosowana w ramach metody (na przykład `Account.log` w powyższym przykładzie) musi być w pełni kwalifikowana.

Jeśli funkcja taka jak `locals()` jest używana w treści klasy (ale nie wewnątrz metody), zwraca ona słownik stosowany dla przestrzeni nazw klasy.

7.30. Dynamiczne tworzenie klas

Zwykle klasy są tworzone przy użyciu instrukcji `class`, ale nie jest to wymagane. Jak wspomniano w poprzedniej sekcji, klasy są definiowane przez wykonanie ciała klasy w celu wypełnienia przestrzeni nazw. Jeśli jesteś w stanie wypełnić słownik własnymi definicjami, możesz utworzyć klasę bez użycia instrukcji `class`. Aby to zrobić, skorzystaj z `types.new_class()`:

```
import types

# Kilka metod (nie w klasie)
def __init__(self, owner, balance):
    self.owner = owner
    self.balance = balance

def deposit(self, amount):
    self.balance -= amount

def withdraw(self, amount):
    self.balance += amount

methods = {
    '__init__': __init__,
    'deposit': deposit,
    'withdraw': withdraw,
}

Account = types.new_class('Account', (),
                          exec_body=lambda ns: ns.update(methods))

# Teraz masz klasę
a = Account('Guido', 1000.0)
a.deposit(50)
a.withdraw(25)
```

Funkcja `new_class()` wymaga nazwy klasy, krotki klas bazowych oraz funkcji wywołania zwrotnego odpowiedzialnej za wypełnienie przestrzeni nazw klas. To wywołanie zwrotne odbiera słownik przestrzeni nazw klas jako argument. Słownik powinien zostać zaktualizowany. Powrotna wartość wywołania zwrotnego jest ignorowana.

Dynamiczne tworzenie klas może być przydatne, jeśli chcesz tworzyć klasy ze struktur danych. Na przykład w części dotyczącej deskryptorów zdefiniowano następujące klasy:

```
class Integer(Typed):
    expected_type = int
```

```
class Float(Typed):
    expected_type = float

class String(Typed):
    expected_type = str
```

Ten kod jest bardzo powtarzalny. Być może lepsze byłoby podejście oparte na danych:

```
typed_classes = [
    ('Integer', int),
    ('Float', float),
    ('String', str),
    ('Bool', bool),
    ('Tuple', tuple),
]

globals().update(
    (name, types.new_class(name, (Typed,),
           exec_body=lambda ns: ns.update(expected_type=ty)))
    for name, ty in typed_classes)
```

W tym przykładzie globalna przestrzeń nazw modułu jest aktualizowana dynamicznie tworzonymi klasami za pomocą `types.new_class()`. Jeśli chcesz utworzyć więcej klas, umieść odpowiedni wpis na liście `typed_classes`.

Czasami zobaczysz, że `type()` jest używane do dynamicznego tworzenia klasy. Na przykład:

```
Account = type('Account', (), methods)
```

Ten przykład działa, ale nie bierze pod uwagę niektórych bardziej zaawansowanych technik, takich jak metaklasy (które zostaną omówione wkrótce). We współczesnym kodzie spróbuj zamiast tego skorzystać z `types.new_class()`.

7.31. Metaklasy

Kiedy definiujesz klasę w Pythonie, sama definicja klasy staje się obiektem. Oto przykład:

```
class Account:
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

isinstance(Account, object) # -> True
```

Jeśli się nad tym zastanowisz, zdasz sobie sprawę, że skoro `Account` jest obiektem, to coś musiało go stworzyć. Tworzenie obiektu klasy jest kontrolowane przez specjalny rodzaj klasy zwany *metaklasą*. Mówiąc najprościej, metaklasa to klasa, która tworzy instancje klas. W poprzednim

przykładzie metaklasa, która utworzyła `Account`, jest wbudowaną klasą o nazwie `type`. W rzeczywistości, jeśli sprawdzisz typ `Account`, zobaczysz, że jest to instancja typu:

```
>>> Account.__class__
<type 'type'>
>>>
```

To trochę dziwne, ale podobnie jest w przypadku typu liczb całkowitych. Jeśli spojrzysz na `x.__class__` i na przykład napiszesz `x = 42`, to otrzymasz `int`, czyli klasę, która tworzy liczby całkowite. Podobnie `type` tworzy instancje typów lub klas.

Kiedy nowa klasa jest definiowana za pomocą instrukcji `class`, dzieje się wiele rzeczy. Najpierw tworzona jest nowa przestrzeń nazw dla klasy. Następnie ciało klasy jest wykonywane w tej przestrzeni nazw. Wreszcie, nazwa klasy, klasy bazowe i wypełniona przestrzeń nazw są używane do tworzenia instancji klasy. Poniższy kod ilustruje kroki wykonywane na niższym poziomie:

```
# Krok 1. Utwórz przestrzeń nazw klas
namespace = type.__prepare__('Account', ())

# Krok 2. Wykonaj treść klasy
exec('''
def __init__(self, owner, balance):
    self.owner = owner
    self.balance = balance

def deposit(self, amount):
    self.balance += amount

def withdraw(self, amount):
    self.balance -= amount
''', globals(), namespace)

# Krok 3. Utwórz ostateczny obiekt klasy
Account = type('Account', (), namespace)
```

W procesie definicji zachodzi interakcja z klasą `type` w celu utworzenia przestrzeni nazw klasy i końcowego obiektu klasy. Korzystanie z `type` można dostosować — klasa może wybrać przetwarzanie przez inną klasę typu, określając inną metaklasę. Odbywa się to za pomocą argumentu słowa kluczowego `metaclass` w dziedziczeniu:

```
class Account(metaclass=type):
    ...
```

Jeśli nie podano argumentu `metaclass`, instrukcja `class` sprawdza typ pierwszego wpisu w krotce klas bazowych (jeśli istnieje) i używa go jako metaklasy. Dlatego jeśli napiszesz klasę `Account(object)`, wynikowa klasa `Account` będzie miała ten sam typ co `object` (czyli `type`). Zauważ, że klasy, które w ogóle nie określają żadnego rodzica, zawsze dziedziczą po obiekcie.

Aby utworzyć nową metaklasę, zdefiniuj klasę, która dziedziczy po `type`. W ramach tej klasy możesz przedefiniować jedną lub więcej metod używanych podczas procesu tworzenia klasy. Zazwyczaj obejmuje to metodę `__prepare__()` używaną do tworzenia przestrzeni nazw klasy, metodę `__new__()` stosowaną do tworzenia instancji klasy, metodę `__init__()` wywoływaną po utworzeniu klasy oraz metodę `__call__()` wykorzystywaną do tworzenia nowych instancji.

Poniższy przykład implementuje metaklasę, która wyświetla argumenty wejściowe do każdej metody, dzięki czemu przyda się do eksperymentów:

```
class mytype(type):

# Tworzy przestrzeń nazw klas
@classmethod
def __prepare__(meta, clsname, bases):
    print("Przygotowanie:", clsname, bases)
    return super().__prepare__(clsname, bases)

# Tworzy instancję klasy po wykonaniu treści
@staticmethod
def __new__(meta, clsname, bases, namespace):
    print("Tworzenie:", clsname, bases, namespace)
    return super().__new__(meta, clsname, bases, namespace)

# Inicjuje instancję klasy
def __init__(cls, clsname, bases, namespace):
    print("Inicjalizacja:", clsname, bases, namespace)
    super().__init__(clsname, bases, namespace)

# Tworzy nowe instancje klasy
def __call__(cls, *args, **kwargs):
    print("Tworzenie instancji:", args, kwargs)
    return super().__call__(*args, **kwargs)

# Przykład
class Base(metaclass=mytype):
    pass

# Definicja bazy daje następujące dane wyjściowe
# Przygotowanie: Base ()
# Tworzenie: Base () {'__module__': '__main__', '__qualname__': 'Base'}
# Initcjalizacja: Base () {'__module__': '__main__', '__qualname__': 'Base'}

b = Base()
# Tworzenie instancji: () {}
```

Jednym z bardziej kłopotliwych aspektów pracy z metaklasami jest nazywanie zmiennych i śledzenie różnych zaangażowanych jednostek. W powyższym kodzie nazwa `meta` odnosi się do samej metaklasy. Nazwa `cls` odwołuje się do wystąpienia klasy utworzonego przez metaklasę. Chociaż nie jest tutaj używana, nazwa `self` odnosi się do normalnej instancji utworzonej przez klasę.

Metaklasy są propagowane poprzez dziedziczenie. Tak więc jeśli zdefiniowałeś klasę bazową, aby korzystała z innej metaklasy, wszystkie klasy podrzędne będą również używać tej metaklasy. Wypróbuj ten przykład, aby zobaczyć swoją niestandardową metaklasę w pracy:

```
class Account(Base):
    def __init__(self, owner, balance):
        self.owner = owner
        self.balance = balance
```

```
    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount

print(type(Account)) # -> <class 'mytype'>
```

Podstawowym zastosowaniem metaklas jest sytuacja, w której chcesz zachować ekstremalną kontrolę na niskim poziomie nad środowiskiem definicji klasy i procesem tworzenia. Zanim jednak przejdziesz dalej, pamiętaj, że Python już udostępnia wiele funkcji do monitorowania i modyfikowania definicji klas (takich jak metoda `__init_subclass__()`, dekoratory klas, deskryptory, domieszki itd.). Przez większość czasu prawdopodobnie nie potrzebujesz metaklasy. Kilka następnych przykładów pokazuje sytuacje, w których metaklasa zapewnia jedyne rozsądne rozwiązanie.

Jednym z zastosowań metaklasy jest przepisanie zawartości przestrzeni nazw klasy przed utworzeniem obiektu klasy. Pewne cechy klas są ustalane w momencie definiowania i nie mogą być później modyfikowane. Jedną z takich funkcji są `__slots__`. Jak wspomniano wcześniej, `__slots__` optymalizują wydajność związaną z układem pamięci instancji. Oto metaklasa, która automatycznie ustawia atrybut `__slots__` na podstawie sygnatury wywołującej metody `__init__()`.

```
import inspect

class SlotMeta(type):
    @staticmethod
    def __new__(meta, clsname, bases, methods):
        if '__init__' in methods:
            sig = inspect.signature(methods['__init__'])
            __slots__ = tuple(sig.parameters)[1:]
        else:
            __slots__ = ()
        methods['__slots__'] = __slots__
        return super().__new__(meta, clsname, bases, methods)

class Base(metaclass=SlotMeta):
    pass

# Przykład
class Point(Base):
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

W tym przykładzie klasa `Point` jest automatycznie tworzona za pomocą `__slots__ of ('x', 'y')`. Powstałe instancje `Point` oszczędzają teraz pamięć, ponieważ nie wiedzą, że używane są sloty. Nie trzeba tego określać bezpośrednio. Tego rodzaju sztuczka nie jest możliwa przy użyciu dekoratorów klas ani za pomocą `__init_subclass__()`, ponieważ te funkcje działają tylko na klasie po jej utworzeniu. Potem jest już za późno.

Innym zastosowaniem metaklas jest zmiana środowiska definicji klasy. Na przykład zduplikowane definicje nazwy podczas definicji klasy zwykle skutkują cichym błędem — druga definicja zastępuje pierwszą. Załóżmy, że chcesz to wyłapać. Oto metaklasa, która robi to, definiując inny rodzaj słownika dla przestrzeni nazw klasy:

```
class NoDupeDict(dict):
    def __setitem__(self, key, value):
        if key in self:
            raise AttributeError(f'{key} został już zdefiniowany')
        super().__setitem__(key, value)

class NoDupeMeta(type):
    @classmethod
    def __prepare__(meta, clsname, bases):
        return NoDupeDict()

class Base(metaclass=NoDupeMeta):
    pass

# Przykład
class SomeClass(Base):
    def yow(self):
        print('Yow!')

    def yow(self, x): # Błąd. Już zdefiniowane
        print('Inna Yow!')
```

To tylko niewielka próbka tego, co jest możliwe. Dla twórców frameworków metaklasy oferują możliwość ścisłej kontroli tego, co dzieje się podczas definiowania klasy — dzięki temu klasy mogą służyć jako rodzaj języka dziedzinowego (ang. *domain-specific language*).

Historycznie metaklasy były używane do wykonywania różnych zadań, które są teraz możliwe do zrealizowania za pomocą innych środków. Metoda `__init_subclass__()` może być wykorzystywana do rozwiązywania wielu różnych problemów, w których kiedyś stosowano metaklasy. Obejmuje to rejestrację klas w centralnym rejestrze, automatyczne dekorowanie metod i generowanie kodu.

7.32. Obiekty wbudowane dla instancji i klas

Ta sekcja zawiera szczegółowe informacje na temat obiektów niskiego poziomu używanych do reprezentowania typów i instancji. Te informacje mogą być przydatne w niskopoziomowym metaprogramowaniu i kodzie, który musi bezpośrednio manipulować typami.

Tabela 7.1 przedstawia często używane atrybuty obiektu typu `cls`.

Atrybut `cls.__name__` zawiera krótką nazwę klasy. Atrybut `cls.__qualname__` zawiera w pełni kwalifikowaną nazwę z dodatkowymi informacjami o otaczającym kontekście (może to być przydatne, jeśli klasa jest zdefiniowana wewnątrz funkcji lub jeśli tworzysz definicję klasy zagnieżdżonej). Słownik `cls.__annotations__` zawiera wskazówki dotyczące typu na poziomie klasy (jeśli istnieją).

Tabela 7.1. *Atrybuty typów*

Atrybut	Opis
`cls.__name__`	Nazwa klasy
`cls.__module__`	Nazwa modułu, w którym zdefiniowana jest klasa
`cls.__qualname__`	W pełni kwalifikowana nazwa klasy
`cls.__bases__`	Krotka klas bazowych
`cls.__mro__`	Krotka reguł wyszukiwania metod (ang. *Method Resolution Order*; MRO)
`cls.__dict__`	Słownik zawierający metody i zmienne klas
`cls.__doc__`	Wpis dokumentacyjny
`cls.__annotations__`	Słownik podpowiedzi typu klasy
`cls.__abstractmethods__`	Zestaw nazw metod abstrakcyjnych (może być niezdefiniowany, jeśli ich nie ma)

Tabela 7.2 przedstawia specjalne atrybuty instancji `i`.

Tabela 7.2. *Atrybuty instancji*

Atrybut	Opis
`i.__class__`	Klasa, do której należy instancja
`i.__dict__`	Słownik przechowujący dane instancji (jeśli zdefiniowano)

Atrybut `__dict__` jest zwykle miejscem przechowywania wszystkich danych związanych z instancją. Jeśli jednak klasa zdefiniowana przez użytkownika korzysta ze `__slots__`, używana jest bardziej wydajna reprezentacja wewnętrzna, a instancje nie będą miały atrybutu `__dict__`.

7.33. Podsumowanie: zachowaj prostotę

W tym rozdziale przedstawiono wiele informacji o klasach oraz sposobach ich dostosowywania i kontrolowania. Podczas pisania klas często jednak najlepszą strategią jest prostota. Tak, możesz używać klas abstrakcyjnych, metaklas, deskryptorów, dekoratorów klas, właściwości, wielokrotnego dziedziczenia, domieszek, wzorców i wskazówek dotyczących typów. Ale możesz też po prostu napisać zwykłą klasę. Istnieje spora szansa, że ta klasa będzie wystarczająco dobra i wszyscy zrozumieją, co ona robi.

Ogólnie rzecz biorąc, warto się cofnąć i rozważyć kilka ogólnie pożądanych cech kodu. Przede wszystkim bardzo liczy się czytelność, która może ucierpieć, jeśli nałożysz zbyt wiele warstw abstrakcji. Poza tym powinieneś dążyć do stworzenia kodu, który będzie łatwy do analizy oraz debugowania, i nie zapominaj o użyciu REPL. Wreszcie, umożliwienie testowania kodu jest często czynnikiem mającym dobry wpływ na projekt. Jeśli nie można przetestować kodu lub testowanie nie jest zbyt wygodne, może istnieć lepszy sposób na przedstawienie rozwiązania.

8

Moduły i pakiety

Programy Pythona są zorganizowane w moduły i pakiety, które są ładowane za pomocą instrukcji `import`. W tym rozdziale bardziej szczegółowo opisano moduły i system pakietów. Główny nacisk kładziony jest na programowanie za pomocą modułów i pakietów, a nie na proces łączenia kodu w celu wdrożenia u innych. Zapoznaj się z najnowszą dokumentacją pod adresem *https://packaging.python.org/tutorials/packaging-projects/*.

8.1. Moduły i wyrażenie import

Dowolny plik źródłowy Pythona można zaimportować jako moduł. Na przykład:

```
# module.py

a = 37

def func():
    print(f'Funkcja func mówi, że a ma wartość {a}')

class SomeClass:
    def method(self):
        print('Metoda method mówi cześć')

print('loaded module')
```

Ten plik zawiera typowe elementy programowania: zmienną globalną, funkcję, definicję klasy i instrukcję izolowaną. Ten przykład ilustruje niektóre ważne (i czasami subtelne) cechy ładowania modułu.

Aby załadować moduł, użyj instrukcji `import module`. Na przykład:

```
>>> import module
loaded module
>>> module.a
37
```

```
>>> module.func()
Funkcja func mówi, że a ma wartość 37
>>> s = module.SomeClass()
>>> s.method()
Metoda method mówi cześć
>>>
```

Podczas importu wykonywane są następujące kroki:

1. Lokalizowany jest kod źródłowy modułu. Jeśli nie można go znaleźć, zgłaszany jest wyjątek `ImportError`.
2. Tworzony jest nowy obiekt modułu. Ten obiekt służy jako kontener dla wszystkich definicji globalnych zawartych w module. Czasami nazywa się go *przestrzenią nazw* (ang. *namespace*).
3. Kod źródłowy modułu jest wykonywany w nowo utworzonej przestrzeni nazw modułu.
4. Jeśli nie wystąpią żadne błędy, wewnątrz obiektu wywołującego tworzona jest nazwa odwołująca się do nowego obiektu modułu. To nazwa modułu, ale bez żadnego rozszerzenia. Jeśli na przykład kod znajduje się w pliku *module.py*, nazwa modułu to `module`.

Spośród tych kroków pierwszy (lokalizacja modułów) jest najbardziej skomplikowany. Częstym źródłem niepowodzeń nowicjuszy jest użycie złej nazwy pliku lub umieszczenie kodu w nieznanej lokalizacji. Nazwa pliku modułu musi używać tych samych reguł co nazwy zmiennych (litery, cyfry oraz podkreślenie) i mieć przyrostek *.py*, na przykład *module.py*. Używając instrukcji `import`, określasz nazwę bez sufiksu: `import module`, a nie `import module.py` (ten ostatni generuje raczej mylący komunikat o błędzie). Plik należy umieścić w jednym z katalogów znajdujących się w ścieżce `sys.path`.

Pozostałe kroki dotyczą modułu definiującego izolowane środowisko dla kodu. Wszystkie definicje, które pojawiają się w module, pozostają odizolowane od tego modułu. Dzięki temu nie ma ryzyka kolizji nazw zmiennych, funkcji i klas z identycznymi nazwami w innych modułach. Uzyskując dostęp do definicji w module, użyj w pełni kwalifikowanej nazwy, takiej jak `module.func()`.

`import` wykonuje wszystkie instrukcje w załadowanym pliku źródłowym. Jeśli moduł oprócz definiowania obiektów wykonuje obliczenia lub generuje dane wyjściowe, zobaczysz wynik — taki jak komunikat `loaded module` wyświetlony w powyższym przykładzie. Często problemy podczas pracy z modułami dotyczą dostępu do klas. Moduł zawsze definiuje przestrzeń nazw, więc jeśli plik *module.py* definiuje klasę `SomeClass`, użyj nazwy `module.SomeClass`, aby odwołać się do tej klasy.

Aby zaimportować wiele modułów za pomocą jednej instrukcji `import`, użyj listy nazw oddzielonych przecinkami:

```
import socket, os, re
```

Czasami nazwa lokalna używana do odwoływania się do modułu jest zmieniana za pomocą kwalifikatora `as`:

```
import module as mo
mo.func()
```

Jest to standardowa praktyka w świecie analizy danych. Na przykład często widzisz to:

```
import numpy as np
import pandas as pd
import matplotlib as plt
...
```

W przypadku zmiany nazwy modułu nowa nazwa dotyczy tylko kontekstu, w którym pojawiła się instrukcja `import`. Inne niepowiązane moduły programu mogą nadal ładować moduł przy użyciu jego oryginalnej nazwy.

Nadanie innej nazwy importowanemu modułowi może być przydatnym rozwiązaniem w przypadku zarządzania różnymi implementacjami wspólnej funkcjonalności lub do pisania rozszerzalnych programów. Jeśli na przykład masz dwa moduły — *unixmodule.py* i *winmodule.py* — które definiują funkcję `func()`, ale zawierają szczegóły implementacji zależne od platformy, możesz napisać kod, który selektywnie importuje moduł:

```
if platform == 'unix':
    import unixmodule as module
elif platform == 'windows':
    import winmodule as module

...
r = module.func()
```

Moduły w Pythonie są obiektami pierwszej klasy. Oznacza to, że można je przypisać do zmiennych, umieścić w strukturach danych i przekazać w programie jako dane. Na przykład nazwa `module` w powyższym przykładzie jest zmienną, która odnosi się do odpowiedniego obiektu modułu.

8.2. Buforowanie modułów

Kod źródłowy modułu jest ładowany i wykonywany tylko raz, niezależnie od tego, jak często używasz instrukcji `import`. Kolejne instrukcje importu wiążą nazwę modułu z obiektem modułu już utworzonym przez poprzedni import.

Początkujący często napotykają problemy, gdy moduł jest importowany do sesji interaktywnej, a następnie jego kod źródłowy jest modyfikowany (na przykład w celu naprawienia błędu) — nowe polecenie `import` nie może załadować zmodyfikowanego kodu. Winę za to ponosi pamięć podręczna modułów. Python nigdy nie przeładuje wcześniej zaimportowanego modułu, nawet jeśli bazowy kod źródłowy został zaktualizowany.

Pamięć podręczną wszystkich aktualnie załadowanych modułów można znaleźć w `sys.modules`, który jest słownikiem mapującym nazwy modułów na obiekty modułów. Zawartość tego słownika jest używana do określenia, czy `import` ładuje nową kopię modułu, czy nie. Usunięcie modułu z pamięci podręcznej zmusi go do ponownego załadowania przy następnej instrukcji `import`. Rzadko jednak jest to bezpieczne z powodów wyjaśnionych w sekcji 8.5 dotyczącej ponownego ładowania modułów.

Czasami zobaczysz instrukcję `import` używaną w funkcji takiej jak ta:

```
def f(x):
    import math
    return math.sin(x) + math.cos(x)
```

Na pierwszy rzut oka wydaje się, że taka implementacja byłaby potwornie powolna — ładowanie modułu przy każdym wywołaniu. W rzeczywistości koszt importu jest minimalny — to tylko jedno wyszukiwanie w słowniku, ponieważ Python natychmiast znajduje moduł w pamięci podręcznej. Główne przeciwwskazanie dla importu wewnątrz funkcji związane jest ze stylem — najczęściej wszystkie importy modułów są wymienione na górze pliku, gdzie są łatwo widoczne. Z drugiej strony, jeśli masz wyspecjalizowaną funkcję, która jest rzadko wywoływana, umieszczenie importu wewnątrz treści funkcji może przyspieszyć ładowanie programu. W takim przypadku załadujesz tylko wymagane moduły, które są rzeczywiście potrzebne.

8.3. Importowanie wybranych nazw z modułu

Za pomocą instrukcji `from module import name` możesz załadować określone definicje z modułu do bieżącej przestrzeni nazw. Jej działanie jest identyczne jak funkcji `import`, z tym wyjątkiem, że zamiast tworzyć nazwę odwołującą się do nowo powstałej przestrzeni nazw modułu, umieszcza odniesienia do jednego lub większej liczby obiektów zdefiniowanych w module w bieżącej przestrzeni nazw:

```
from module import func # Importuje moduł i umieszcza func w bieżącej przestrzeni nazw
func()                  # Wywołuje func() zdefiniowaną w module
module.func()           # Błąd. NameError: module
```

Jeśli chcesz mieć wiele definicji, instrukcja `from` akceptuje nazwy oddzielone przecinkami. Na przykład:

```
from module import func, SomeClass
```

Semantycznie instrukcja `from module import name` wykonuje kopię nazwy z pamięci podręcznej modułu do lokalnej przestrzeni nazw. Oznacza to, że Python najpierw, w tle, wykonuje instrukcję `import module`. Następnie wykonuje przypisanie z pamięci podręcznej do nazwy lokalnej, takie jak `name = sys.modules['module'].name`.

Powszechnym błędnym przekonaniem jest to, że instrukcja `from module import name` jest bardziej wydajna, gdyż ładuje tylko część modułu. To nieprawda. Tak czy inaczej ładowany jest cały moduł, który jest przechowywany w pamięci podręcznej.

Importowanie funkcji przy użyciu składni `from` nie zmienia ich reguł określania zakresu. Gdy funkcje szukają zmiennych, zaglądają tylko do pliku, w którym funkcja została zdefiniowana, a nie do przestrzeni nazw, do której funkcja jest importowana i w której jest wywoływana. Na przykład:

```
>>> from module import func
>>> a = 42
>>> func()
Funkcja func mówi, że a ma wartość 37
>>> func.__module__
'module'
>>> func.__globals__['a']
37
>>>
```

Zwróć uwagę na zachowanie zmiennych globalnych. Przyjrzyj się fragmentowi kodu importującemu zarówno zmienną `func`, jak i zmienną globalną `a`, której używa:

```
from module import a, func
a = 42        # Zmodyfikuj zmienną
func()        # Wyświetla "Funkcja func mówi, że a ma wartość 37"
print(a)      # Wyświetla "42"
```

Przypisanie zmiennych w Pythonie nie jest operacją przechowywania. Oznacza to, że nazwa `a` w tym przykładzie nie reprezentuje jakiegoś rodzaju pamięci, w którym jest przechowywana wartość. Początkowa instrukcja `import` kojarzy nazwę lokalną `a` z oryginalnym obiektem `module.a`. Jednak późniejsze przypisanie `a = 42` przenosi nazwę lokalną `a` do zupełnie innego obiektu. W tym momencie `a` nie jest już powiązane z wartością w importowanym module. Z tego powodu nie jest możliwe użycie instrukcji `from` w taki sposób, aby zmienne zachowywały się jak zmienne globalne, tak jak w języku takim jak C. Jeśli chcesz mieć zmienne parametry globalne w swoim programie, umieść je w module i jawnie użyj nazwy modułu za pomocą instrukcji `import`, na przykład `module.a`.

Symbol wieloznaczny gwiazdki (`*`) jest czasami używany do załadowania wszystkich definicji w module z wyjątkiem tych, które zaczynają się od podkreślenia. Oto przykład:

```
# Załaduj wszystkie definicje do bieżącej przestrzeni nazw
from module import *
```

Instrukcja `from module import *` może być używana tylko w zakresie najwyższego poziomu modułu. Nielegalne jest zwłaszcza używanie tej formy importu wewnątrz treści funkcji.

Moduły mogą precyzyjnie kontrolować zestaw nazw importowanych przez instrukcję `from module import *` poprzez zdefiniowanie listy `__all__`. Oto przykład:

```
# Moduł: module.py
__all__ = ['func', 'SomeClass']

a = 37            # Nie jest eksportowane

def func():       # Wyeksportowane
    ...
class SomeClass: # Wyeksportowane
    ...
```

Gdy pojawi się interaktywny znak zachęty Pythona, użycie instrukcji `from module import *` może być wygodnym sposobem pracy z modułem. Jednak używanie tego stylu importu w programie nie jest pożądane. Nadużywanie importu może zanieczyszczać lokalną przestrzeń nazw i prowadzić do zamieszania. Na przykład:

```
from math import *
from random import *
from statistics import *

a = gauss(1.0, 0.25) # Z którego modułu?
```

Zwykle lepiej jest jasno określić nazwy:

```
from math import sin, cos, sqrt
from random import gauss
from statistics import mean

a = gauss(1.0, 0.25)
```

8.4. Importy cykliczne

Specyficzny problem pojawia się, gdy dwa moduły wzajemnie się importują. Załóżmy na przykład, że masz dwa pliki:

```
# ----------------------------
# moda.py

import modb

def func_a():
    modb.func_b()

class Base:
    pass

# ----------------------------
# modb.py

import moda

def func_b():
    print('B')

class Child(moda.Base):
    pass
```

W tym kodzie występuje dziwna zależność kolejności importu. Użycie `import modb` najpierw działa dobrze, ale jeśli najpierw wywołasz `import moda`, zostanie wyświetlony błąd dotyczący niezdefiniowania `moda.Base`.

Aby zrozumieć, co się dzieje, musisz podążać za tokiem programu. `import moda` rozpoczyna wykonywanie pliku *moda.py*. Pierwsza napotkana instrukcja to `import modb`. W ten sposób sterowanie przełącza się na *modb.py*. Pierwsza instrukcja w tym pliku to `import moda`. Zamiast wchodzić w cykl rekurencyjny, ten import jest realizowany przez pamięć podręczną modułu, a kontrola jest kontynuowana od następnej instrukcji w *modb.py*. To dobrze — importy cykliczne nie powodują zakleszczenia Pythona ani wejścia w nowy wymiar czasoprzestrzeni. Jednak w tym momencie moduł `moda` został tylko częściowo przetworzony. Kiedy kontrola programu dotrze do instrukcji `class Child(moda.Base)`, wszystko kończy się błędem. Wymagana klasa `Base` nie została jeszcze zdefiniowana.

Jednym ze sposobów rozwiązania tego problemu jest przeniesienie instrukcji `import modb` w inne miejsce. Na przykład możesz przenieść import do funkcji `func_a()`, gdzie definicja jest rzeczywiście potrzebna:

```
# moda.py

def func_a():
    import modb
    modb.func_b()

class Base:
    pass
```

Możesz również przenieść import do późniejszej pozycji w pliku:

```
# moda.py

def func_a():
    modb.func_b()

class Base:
    pass

import modb  # Musi być po zdefiniowaniu Base
```

Oba te rozwiązania mogą wywołać zdziwienie podczas przeglądu kodu. W większości przypadków importy modułów nie pojawiają się na końcu pliku. Obecność importów cyklicznych prawie zawsze sugeruje problem w organizacji kodu. Lepszym sposobem poradzenia sobie z tym może być przeniesienie definicji `Base` do oddzielnego pliku *base.py* i przepisanie *modb.py* w następujący sposób:

```
# modb.py

import base

def func_b():
    print('B')

class Child(base.Base):
    pass
```

8.5. Ponowne ładowanie i zwolnienie modułu

Nie ma niezawodnej obsługi ponownego ładowania lub zwolnienia wcześniej zaimportowanych modułów. Chociaż można usunąć moduł z `sys.modules`, nie powoduje to zwolnienia modułu z pamięci. Dzieje się tak, ponieważ odwołania do obiektu modułu buforowanego nadal istnieją w innych modułach, które zaimportowały ten moduł. Co więcej, jeśli istnieją instancje klas zdefiniowane w module, te instancje zawierają referencje z powrotem do swoich obiektów klas, przechowujących referencje do modułu, w którym zostały zdefiniowane.

Fakt, że odwołania do modułów istnieją w wielu miejscach, sprawia, że przeładowanie modułu po wprowadzeniu zmian w jego implementacji jest z reguły niepraktyczne. Jeśli na przykład usuniesz moduł z `sys.modules` i użyjesz instrukcji `import`, aby go przeładować, nie spowoduje to wstecznej zmiany wszystkich poprzednich odwołań do modułu używanego w programie. Zamiast tego będziesz mieć jedno odniesienie do nowego modułu utworzonego przez najnowszą

instrukcję importu oraz zestaw odniesień do starego modułu utworzonego przez `import` w innych częściach kodu. Rzadko tego chcesz. Ponowne ładowanie modułów nigdy nie jest bezpieczne w żadnym rozsądnie napisanym kodzie produkcyjnym, chyba że jesteś w stanie dokładnie kontrolować całe środowisko wykonawcze.

Istnieje funkcja `reload()` służąca do ponownego ładowania modułu, którą można znaleźć w bibliotece `importlib`. Jako argument przekazujesz jej już załadowany moduł. Na przykład:

```
>>> import module
>>> import importlib
>>> importlib.reload(module)
loaded module
<module 'module' from 'module.py'>
>>>
```

`reload()` ładuje nową wersji kodu źródłowego modułu, a następnie wykonuje go na szczycie już istniejącej przestrzeni nazw modułu. Odbywa się to bez czyszczenia poprzedniej przestrzeni nazw. To dosłownie to samo co wpisanie nowego kodu źródłowego w miejsce starego kodu bez ponownego uruchamiania interpretera.

Jeśli inne moduły wcześniej zaimportowały ponownie załadowany moduł za pomocą standardowej instrukcji `import`, takiej jak `import module`, kolejne wczytanie sprawi, że zobaczą zaktualizowany kod — jak za dotknięciem czarodziejskiej różdżki. Wciąż jednak istnieje wiele niebezpieczeństw. Po pierwsze, ponowne ładowanie nie powoduje powtórnego załadowania żadnego z modułów, które mogą zostać zaimportowane przez ponownie załadowany plik. Ta operacja nie jest rekurencyjna — dotyczy tylko pojedynczego modułu przekazanego funkcji `reload()`. Po drugie, jeśli jakikolwiek moduł użył formy importu `from module import`, te importy nie „zauważą" efektu ponownego załadowania. Wreszcie, jeśli utworzono instancje klas, ponowne ładowanie nie aktualizuje ich podstawowej definicji klasy. W rzeczywistości będziesz mieć teraz dwie różne definicje tej samej klasy w tym samym programie — starą, która pozostaje używana we wszystkich istniejących instancjach w momencie przeładowania, oraz nową, która jest wykorzystywana dla nowych instancji. To prawie zawsze jest mylące.

Na koniec należy zauważyć, że rozszerzenia C/C++ do Pythona nie mogą być w żaden sposób bezpiecznie zwolnione lub ponownie załadowane. Nie zapewnia się żadnej obsługi tego rozwiązania, a bazowy system operacyjny może i tak tego zabronić. Najlepszym rozwiązaniem dla tego scenariusza jest ponowne uruchomienie procesu interpretera Pythona.

8.6. Kompilacja modułów

Kiedy moduły są importowane po raz pierwszy, są kompilowane do kodu bajtowego interpretera. Ten kod jest zapisywany w pliku *.pyc* w specjalnym katalogu `__pycache__`. Ten katalog znajduje się zwykle w tym samym katalogu co oryginalny plik *.py*. Gdy ten sam import wystąpi ponownie w innym przebiegu programu, zamiast tego ładowany jest skompilowany kod bajtowy. To znacznie przyspiesza proces importu.

Buforowanie kodu bajtowego to automatyczny proces, o który prawie nigdy nie musisz się martwić. Pliki są automatycznie odtwarzane, jeśli oryginalny kod źródłowy ulegnie zmianie. To po prostu działa.

Jednak nadal istnieją powody, aby wiedzieć więcej o procesie buforowania i kompilacji. Po pierwsze, czasami pliki Pythona są instalowane (często przypadkowo) w środowisku, w którym użytkownicy nie mają uprawnień do tworzenia wymaganego katalogu `__pycache__`. Python nadal będzie działał, ale każdy import ładuje teraz oryginalny kod źródłowy i kompiluje go do kodu bajtowego. Ładowanie programu będzie dużo wolniejsze, niż powinno. Podobnie przy wdrażaniu lub pakowaniu aplikacji w Pythonie korzystne może być dołączenie skompilowanego kodu bajtowego, ponieważ może to znacznie przyspieszyć uruchamianie programu.

Innym dobrym powodem, aby wiedzieć więcej na temat buforowania modułów, jest to, że przeszkadzają mu niektóre techniki programowania. Zaawansowane techniki metaprogramowania obejmujące dynamiczne generowanie kodu i funkcję `exec()` niwelują zalety buforowania kodu bajtowego. Godnym uwagi przykładem jest użycie `dataclass`:

```
from dataclasses import dataclass

@dataclass
class Point:
    x: float
    y: float
```

Dekoratory `dataclass` generują funkcje metod jako fragmenty tekstu i wykonują je za pomocą `exec()`. Żaden z wygenerowanych kodów nie jest buforowany przez system importu. W przypadku definicji jednej klasy nie zauważysz różnicy. Jeśli jednak masz moduł składający się ze 100 dekoratorów `dataclass`, może się okazać, że importuje on prawie 20 razy wolniej niż porównywalny moduł, w którym właśnie zapisałeś klasy w normalny, choć mniej zwięzły sposób.

8.7. Ścieżka wyszukiwania modułów

Podczas importowania modułów interpreter przeszukuje listę katalogów w `sys.path`. Pierwszy wpis w `sys.path` jest często pustym ciągiem `''`, który odnosi się do bieżącego katalogu roboczego. Alternatywnie, jeśli uruchomisz skrypt, pierwszy wpis w `sys.path` to katalog, w którym znajduje się skrypt. Inne wpisy w `sys.path` zwykle składają się z kombinacji nazw katalogów i plików archiwum *.zip*. Kolejność, w jakiej wpisy są wyświetlane w `sys.path`, określa kolejność wyszukiwania stosowaną podczas importowania modułów. Aby utworzyć nowe wpisy do ścieżki wyszukiwania, dodaj je do tej listy. Można to zrobić bezpośrednio lub poprzez ustawienie zmiennej środowiskowej `PYTHONPATH`. Na przykład w systemie UNIX:

```
bash $ env PYTHONPATH=/jakaś/ścieżka python3 script.py
```

Pliki archiwum ZIP to wygodny sposób na połączenie kolekcji modułów w jeden plik. Załóżmy na przykład, że utworzyłeś dwa moduły, *foo.py* i *bar.py*, i umieściłeś je w pliku *mymodules.zip*. Plik można dodać do ścieżki wyszukiwania Pythona w następujący sposób:

```
import sys
sys.path.append('mymodules.zip')
import foo, bar
```

Ścieżkami mogą być także określone lokalizacje w strukturze katalogów pliku *.zip*. Ponadto pliki *.zip* można mieszać ze zwykłymi ścieżkami innych komponentów. Oto przykład:

```
sys.path.append('/tmp/modules.zip/lib/python')
```

Plik ZIP nie musi mieć rozszerzenia „.zip". Historycznie rzecz biorąc, w ścieżce często spotykano również pliki *.egg*. Pochodzą one z wczesnego narzędzia do zarządzania pakietami Pythona o nazwie *setuptools*. Jednak plik *.egg* to nic innego jak zwykły plik *.zip* lub katalog z dodanymi do niego metadanymi (takimi jak numer wersji, zależności itd.).

8.8. Wykonanie jako program główny

Chociaż ta sekcja dotyczy instrukcji `import`, pliki Pythona są często wykonywane jako główny skrypt. Na przykład:

```
% python3 module.py
```

Każdy moduł zawiera zmienną `__name__`, która przechowuje nazwę modułu. Kod może zbadać tę zmienną, aby określić moduł, w którym jest wykonywany. Moduł najwyższego poziomu interpretera nosi nazwę `__main__`. Programy określone w wierszu poleceń lub wprowadzone interaktywnie działają wewnątrz modułu `__main__`. Czasami program może zmienić swoje zachowanie, w zależności od tego, czy został zaimportowany jako moduł, czy działa w `__main__`. Na przykład moduł może zawierać kod, który jest wykonywany, jeśli moduł jest używany jako program główny, oraz taki, który nie jest wykonywany, jeśli moduł jest po prostu importowany przez inny moduł.

```
# Sprawdź, czy działa jako program
if __name__ == '__main__':
    # Tak. Działa jako główny skrypt
    instrukcje
else:
    # Nie, musiałem zostać zaimportowany jako moduł
    instrukcje
```

Pliki źródłowe przeznaczone do użytku jako biblioteki mogą wykorzystywać tę technikę w celu dołączenia opcjonalnego testowania lub przykładowego kodu. Podczas tworzenia modułu możesz umieścić kod debugowania do testowania funkcji Twojej biblioteki wewnątrz instrukcji `if`, jak pokazano, i uruchomić Pythona w swoim module jako główny program. Ten kod nie zostanie uruchomiony dla użytkowników, którzy zaimportują Twoją bibliotekę.

Jeśli stworzyłeś katalog z kodem Pythona, możesz uruchomić w nim aplikację, pod warunkiem że zawiera specjalny plik *__main__.py*. Jeśli na przykład stworzysz taki katalog:

```
myapp/
    foo.py
    bar.py
    __main__.py
```

możesz w nim uruchomić Pythona, wpisując komendę *python3 myapp*. Wykonywanie rozpocznie się w pliku *__main__.py*. Działa to również, jeśli zmienisz katalog *myapp* w archiwum ZIP. Wpisanie *python3 myapp.zip* spowoduje wyszukanie pliku *__main__.py* najwyższego poziomu i wykonanie go, jeśli zostanie znaleziony.

8.9. Pakiety

Poza najprostszymi aplikacjami kod Pythona jest zazwyczaj zorganizowany w postaci *pakietów*. Pakiet to zbiór modułów zgrupowanych pod wspólną nazwą najwyższego poziomu. Grupowanie pomaga rozwiązywać konflikty między nazwami modułów używanymi w różnych aplikacjach i utrzymuje Twój kod oddzielnie od kodu innych osób. Pakiet jest definiowany przez utworzenie katalogu o charakterystycznej nazwie i umieszczenie w nim początkowo pustego pliku *__init__.py*. Następnie w razie potrzeby umieszcza się w tym katalogu dodatkowe pliki i podpakiety Pythona. Na przykład pakiet może być zorganizowany w następujący sposób:

```
graphics/
    __init__.py
    primitive/
        __init__.py
        lines.py
        fill.py
        text.py
        ...
    graph2d/
        __init__.py
        plot2d.py
        ...
    graph3d/
        __init__.py
        plot3d.py
        ...
    formats/
        __init__.py
        gif.py
        png.py
        tiff.py
        jpeg.py
```

Instrukcja `import` służy do ładowania modułów z pakietu w taki sam sposób jak w przypadku prostych modułów, z wyjątkiem tego, że masz teraz dłuższe nazwy. Na przykład:

```
# Pełna ścieżka
import graphics.primitive.fill
...
graphics.primitive.fill.floodfill(img, x, y, color)

# Załaduj określony podmoduł
from graphics.primitive import fill
...
fill.floodfill(img, x, y, color)

# Załaduj określoną funkcję z podmodułu
from graphics.primitive.fill import floodfill
...
floodfill(img, x, y, color)
```

Za każdym razem, gdy jakakolwiek część pakietu jest importowana po raz pierwszy, kod w pliku *__init__.py* jest wykonywany jako pierwszy (jeśli istnieje). Jak wspomniano, ten plik może być pusty, ale może również zawierać kod do wykonywania inicjalizacji specyficznych dla pakietu.

Jeśli importujesz głęboko zagnieżdżony podmoduł, wykonywane są wszystkie pliki *__init__.py* napotkane podczas przechodzenia przez strukturę katalogów. W ten sposób instrukcja `import graphics.primitive.fill` najpierw wykona plik *__init__.py* w katalogu *graphics/*, a następnie plik *__init__.py* w katalogu *primitive/*.

Bystrzy użytkownicy Pythona mogą zauważyć, że pakiet nadal wydaje się działać, jeśli pominięto pliki *__init__.py*. To prawda — możesz użyć katalogu kodu Pythona jako pakietu, nawet jeśli nie zawiera on *__init__.py*. Jednak nie jest oczywiste, że katalog z brakującym plikiem *__init__.py* w rzeczywistości definiuje inny rodzaj pakietu, nazywany *przestrzenią nazw pakietu*. Jest to zaawansowana funkcja, która jest czasami wykorzystywana przez bardzo duże biblioteki i frameworki do implementowania uszkodzonych systemów wtyczek. Prawdopodobnie nie będziesz tworzyć takich funkcji, więc zawsze powinieneś dodawać odpowiednie pliki *__init__.py* podczas konstrukcji pakietu.

8.10. Import wewnątrz pakietu

Kluczową cechą instrukcji `import` jest to, że wszystkie importy modułów wymagają bezwzględnej lub w pełni kwalifikowanej ścieżki pakietu, w tym instrukcji importu używanych w samym pakiecie.

Załóżmy na przykład, że moduł `graphics.primitive.fill` chce zaimportować moduł `graphics.primitive.lines`. Prosta instrukcja, taka jak `import lines`, nie zadziała — otrzymasz wyjątek `ImportError`. Zamiast tego musisz skorzystać z w pełni kwalifikowanego importu w następujący sposób:

```
# graphics/primitives/fill.py

# W pełni kwalifikowany import podmodułów
from graphics.primitives import lines
```

Niestety, napisanie takiej pełnej nazwy pakietu jest często denerwujące. Czasami będziesz chciał zmienić nazwę pakietu aby skorzystać z innej wersji. Jeśli nazwa pakietu jest wpisana w kod, nie możesz tego zrobić. Lepszym wyborem jest użycie importu względnego, takiego jak ten:

```
# graphics/primitives/fill.py

# Import względny pakietu
from . import lines
```

Użyty w powyższym wyrażeniu znak kropki `from . import lines` odnosi się do tego samego katalogu co importowany moduł. Tak więc instrukcja ta szuka wierszy modułu w tym samym katalogu co plik *fill.py*.

Importy względne mogą również określać podmoduły zawarte w różnych katalogach tego samego pakietu. Jeśli na przykład moduł `graphics.graph2d.plot2d` ma zaimportować `graphics.primitive.lines`, może użyć instrukcji takiej jak ta:

```
# graphics/graph2d/plot2d.py

from ..primitive import lines
```

Tutaj `..` przesuwa się w górę o jeden poziom katalogu, a `primitive` spada do innego katalogu podpakietów.

Importy względne można określić tylko za pomocą odpowiedniej formy instrukcji `import`: `from module import symbol`. Dlatego instrukcje takie jak `import ..primitive.lines` lub `import .lines` są błędem składniowym. Ponadto `symbol` musi być prostym identyfikatorem, więc wyrażenie takie jak `from .. import primitive.lines` również jest niedozwolone. Wreszcie, import względny może być używany tylko z wnętrza pakietu; niedozwolone jest stosowanie względnego importu w odniesieniu do modułów, które są po prostu zlokalizowane w innym katalogu w systemie plików.

8.11. Uruchamianie podmodułu pakietu jako skryptu

Kod zorganizowany w pakiet ma inne środowisko wykonawcze niż prosty skrypt. Zawiera on nazwę pakietu, moduły podrzędne i powiązania do względnych importów (które działają tylko wewnątrz pakietu). Jedną z funkcji, która już nie działa, jest możliwość uruchamiania Pythona bezpośrednio na pliku źródłowym pakietu. Załóżmy na przykład, że pracujesz nad plikiem *graphics/graph2d/plot2d.py* i dodajesz kod testowy na dole:

```
# graphics/graph2d/plot2d.py
from ..primitive import lines, text

class Plot2D:
    ...

if __name__ == '__main__':
    print('Testowanie klasy Plot2D')
    p = Plot2D()
    ...
```

Jeśli spróbujesz uruchomić go bezpośrednio, otrzymasz błąd związany ze względnymi instrukcjami importu:

```
bash $ python3 graphics/graph2d/plot2d.py
Traceback (most recent call last):
  File "graphics/graph2d/plot2d.py", line 1, in <module>
    from ..primitive import line, text
ValueError: attempted relative import beyond top-level package
bash $
```

Nie możesz przenieść się do katalogu pakietów i tam go uruchomić:

```
bash $ cd graphics/graph2d/
bash $ python3 plot2d.py
Traceback (most recent call last):
  File "plot2d.py", line 1, in <module>
    from ..primitive import line, text
ValueError: attempted relative import beyond top-level package
bash $
```

Aby uruchomić podmoduł jako główny skrypt, musisz użyć opcji `–m` w interpreterze. Na przykład:

```
bash $ python3 -m graphics.graph2d.plot2d
Testowanie klasy Plot2D
bash $
```

Opcja `-m` określa moduł lub pakiet jako program główny. Python uruchomi moduł w odpowiednim środowisku, aby upewnić się, że importy działają. Wiele wbudowanych pakietów Pythona ma „tajne" funkcje, których można użyć za pomocą opcji `-m`. Jedną z najbardziej znanych jest komenda `python3 -m http.server`, która służy do uruchomienia serwera WWW w bieżącym katalogu.

Podobną funkcjonalność możesz zapewnić dla własnych pakietów. Jeśli nazwa dostarczona do `python -m name` odpowiada katalogowi pakietu, Python szuka w tym katalogu *__main__.py* i uruchamia go jako skrypt.

8.12. Kontrolowanie przestrzeni nazw pakietu

Podstawowym celem pakietu jest służenie jako kontener najwyższego poziomu dla kodu. Czasami użytkownicy importują nazwę najwyższego poziomu i nic więcej. Na przykład:

```
import graphics
```

Ten import nie określa żadnego konkretnego podmodułu. Nie udostępnia też żadnej innej części pakietu. Na przykład przekonasz się, że taki kod się nie wykona:

```
import graphics
graphics.primitive.fill.floodfill(img,x,y,color) # Błąd!
```

Gdy podany jest tylko import pakietu najwyższego poziomu, jedynym importowanym plikiem jest powiązany plik *__init__.py*. W tym przykładzie jest to *graphics/__init__.py*.

Podstawowym zastosowaniem pliku *__init__.py* jest budowanie zawartości przestrzeni nazw pakietu najwyższego poziomu i (lub) zarządzanie nią. Często wiąże się to z importowaniem wybranych funkcji, klas i innych obiektów z podmodułów niższego poziomu. Jeśli na przykład pakiet `graphics` w tym przykładzie składa się z setek funkcji niskiego poziomu, ale większość tych szczegółów jest zawarta w kilku klasach wysokiego poziomu, plik *__init__.py* może udostępniać tylko te klasy:

```
# graphics/__init__.py

from .graph2d.plot2d import Plot2D
from .graph3d.plot3d import Plot3D
```

W przypadku pliku *__init__.py* nazwy `Plot2D` i `Plot3D` pojawią się na najwyższym poziomie pakietu. Użytkownik mógłby wtedy używać tych nazw tak, jakby `graphics` był prostym modułem:

```
from graphics import Plot2D

plt = Plot2D(100, 100)
plt.clear()
...
```

Często jest to znacznie wygodniejsze dla użytkownika, ponieważ nie musi on wiedzieć, jak właściwie zorganizowałeś swój kod. W pewnym sensie umieszczasz wyższą warstwę abstrakcji na szczycie swojej struktury kodu. Wiele modułów w standardowej bibliotece Pythona jest w ten sposób skonstruowanych. Na przykład popularny moduł `collections` to tak naprawdę pakiet. Plik *collections/__init__.py* konsoliduje definicje z kilku różnych miejsc i przedstawia je użytkownikowi jako pojedynczą, skonsolidowaną przestrzeń nazw.

8.13. Kontrolowanie eksportu pakietów

Zagadnienie to dotyczy interakcji między plikiem *__init__.py* a podmodułami niskiego poziomu. Na przykład użytkownik pakietu może chcieć zajmować się tylko obiektami i funkcjami, które znajdują się w przestrzeni nazw pakietu najwyższego poziomu. Twórca pakietu może być jednak zaniepokojony problemem organizowania kodu w podmoduły, które można zmieniać.

Aby lepiej zarządzać tą złożonością organizacyjną, podmoduły pakietów często deklarują jawną listę eksportów, definiując zmienną `__all__`. To lista nazw, które powinny być przesunięte o jeden poziom wyżej w przestrzeni nazw pakietu. Na przykład:

```
# graphics/graph2d/plot2d.py

__all__ = ['Plot2D']

class Plot2D:
    ...
```

Powiązany plik *__init__.py* następnie dołącza swoje podmoduły za pomocą importu * w następujący sposób:

```
# graphics/graph2d/__init__.py

# Ładuje tylko nazwy wyraźnie wymienione w zmiennych __all__
from .plot2d import *

# Propaguje __all__ do następnego poziomu (w razie potrzeby)
__all__ = plot2d.__all__
```

Ten proces podnoszenia jest następnie kontynuowany aż do pakietu najwyższego poziomu *__init__.py*.

Na przykład:

```
# graphics/__init__.py

from .graph2d import *
from .graph3d import *

# Konsolidacja eksportu
__all__ = [
    *graph2d.__all__,
    *graph3d.__all__
]
```

Kluczem jest to, że każdy składnik pakietu jawnie określa swoje eksporty za pomocą zmiennej `__all__`. Następnie pliki *__init__.py* propagują eksport w górę. W praktyce może się to skomplikować, ale takie podejście pozwala uniknąć problemu wpisywania na stałe określonych nazw eksportu do pliku *__init__.py*. Zamiast tego, jeśli podmoduł ma coś wyeksportować, jego nazwa jest wyświetlana tylko w jednym miejscu — w zmiennej `__all__`. Następnie rozprzestrzenia się do właściwego miejsca w przestrzeni nazw pakietu.

Warto zauważyć, że chociaż stosowanie importów * w kodzie użytkownika jest niemile widziane, jest to powszechna praktyka w plikach pakietów *__init__.py*. Powodem, dla którego takie rozwiązanie działa w pakietach, jest to, że jest ono zwykle znacznie bardziej kontrolowane i ograniczone — jest sterowane zawartością zmiennych `_all__`, w przeciwieństwie do swobodnego podejścia „po prostu zaimportujmy wszystko".

8.14. Dane pakietu

Czasami pakiet zawiera pliki danych, które należy załadować (w przeciwieństwie do kodu źródłowego). W pakiecie zmienna `__file__` poda informacje o lokalizacji określonego pliku źródłowego. Jednak pakiety są skomplikowane. Mogą być spakowane jako archiwum ZIP lub ładowane z nietypowych środowisk. Sama zmienna `__file__` może być zawodna lub nawet niezdefiniowana. W rezultacie ładowanie pliku danych, czyli przekazanie nazwy pliku do wbudowanej funkcji `open()` i odczytanie niektórych danych, często nie jest prostą sprawą.

Aby odczytać dane pakietu, użyj `pkgutil.get_data(package, resource)`. Twój pakiet może na przykład wyglądać tak:

```
mycode/
    resources/
        data.json
    __init__.py
    spam.py
    yow.py
```

Aby załadować plik *data.json* z pliku *spam.py*, wykonaj następujące czynności:

```
# mycode/spam.py

import pkgutil
import json

def func():
    rawdata = pkgutil.get_data(__package__,
        'resources/data.json')
    textdata = rawdata.decode('utf-8')
    data = json.loads(textdata)
    print(data)
```

Funkcja `get_data()` próbuje znaleźć określony zasób i zwraca jego zawartość jako surowy ciąg bajtów. Zmienna `__package__` pokazana w przykładzie jest łańcuchem zawierającym nazwę otaczającego pakietu. Dalsze dekodowanie (takie jak konwersja bajtów na tekst) i interpretacja zależą od Ciebie. W tym przykładzie dane są dekodowane i przetwarzane z formatu JSON do słownika Pythona.

Pakiet nie jest dobrym miejscem do przechowywania gigantycznych plików danych. Zarezerwuj zasoby pakietu dla danych konfiguracyjnych i innych elementów potrzebnych do działania pakietu.

8.15. Obiekty modułu

Moduły to obiekty najwyższej klasy. Tabela 8.1 zawiera listę atrybutów często występujących w modułach.

Tabela 8.1. *Atrybuty modułu*

Atrybut	Opis
`__name__`	Pełna nazwa modułu
`__doc__`	Wpis dokumentacyjny
`__dict__`	Słownik modułu
`__file__`	Nazwa pliku, gdzie jest zdefiniowany
`__package__`	Nazwa otaczającego pakietu (jeśli istnieje)
`__path__`	Lista podkatalogów do wyszukiwania podmodułów pakietu
`__annotations__`	Wskazówki dotyczące typów na poziomie modułu

Atrybut `__dict__` jest słownikiem reprezentującym przestrzeń nazw modułu. Tutaj znajduje się wszystko, co jest zdefiniowane w module.

Atrybut `__name__` jest często używany w skryptach. Sprawdzenie, takie jak `if __name__ == '__main__'`, jest często wykonywane, aby zobaczyć, czy plik działa jako samodzielny program.

Atrybut `__package__` zawiera nazwę otaczającego pakietu, jeśli taki istnieje. Jeśli jest ustawiony, atrybut `__path__` jest listą katalogów, które będą przeszukiwane w celu zlokalizowania podmodułów pakietu. Zwykle zawiera pojedynczy wpis z katalogiem, w którym znajduje się pakiet. Czasami duże frameworki będą manipulować wartością `__path__`, aby dołączyć dodatkowe katalogi w celu obsługi wtyczek i innych zaawansowanych funkcji.

Nie wszystkie atrybuty są dostępne we wszystkich modułach. Na przykład wbudowane moduły mogą nie mieć ustawionego atrybutu `__file__`. Podobnie, atrybuty związane z pakietem nie są ustawiane dla modułów najwyższego poziomu (niezawartych w pakiecie).

Atrybut `__doc__` jest wpisem dokumentacyjnym modułu (jeśli istnieje). Jest to ciąg znaków, który pojawia się jako pierwsza instrukcja w pliku. Atrybut `__annotations__` jest słownikiem wskazówek typu na poziomie modułu. Wygląda mniej więcej tak:

```
# mymodule.py
'''
Wpis dokumentacyjny
'''
# Wskazówki typu (umieszczone w __annotations__)
x: int
y: float
...
```

Podobnie jak w przypadku innych wskazówek dotyczących typów, wskazówki na poziomie modułu nie zmieniają żadnej części zachowania Pythona ani nie definiują zmiennych. Są to czyste metadane, które inne narzędzia mogą przeglądać, jeśli chcą.

8.16. Wdrażanie pakietów Pythona

Ostatnim problemem związanym z modułami i pakietami jest udostępnienie kodu innym. Jest to obszerne zagadnienie, które przez wiele lat było w centrum uwagi deweloperów języka. Przedstawianie tutaj dokumentacji procesu, która z pewnością zdezaktualizuje się do czasu, gdy to przeczytasz, nie ma sensu. Zamiast tego zapoznaj się z dokumentacją pod adresem *https://packaging.python.org/tutorials/packaging-projects*.

Na potrzeby codziennego programowania najważniejszą rzeczą jest wyizolowanie kodu jako samodzielnego projektu. Cały Twój kod powinien się znajdować w odpowiednim pakiecie. Nadaj pakietowi unikalną nazwę, aby nie kolidowała z innymi możliwymi zależnościami. Zapoznaj się z indeksem pakietów Pythona na *https://pypi.org*, aby wybrać odpowiednią nazwę. Przy konstruowaniu kodu staraj się zachować prostotę. Jak widziałeś, za pomocą systemu modułów i pakietów można zrobić wiele wysoce wyrafinowanych rzeczy. Jest na to czas i miejsce, ale nie powinieneś od tego zaczynać.

Mając na uwadze absolutną prostotę, najbardziej minimalistycznym sposobem dystrybucji czystego kodu Pythona jest użycie modułu `setuptools` lub wbudowanego modułu `distutils`. Załóżmy, że napisałeś jakiś kod i jest to projekt, który wygląda następująco:

```
spam-project/
    README.txt
    Documentation.txt
    spam/                       # Kod pakietu
        __init__.py
        foo.py
        bar.py
        runspam.py          # Skrypt do uruchomienia jako: python runspam.py
```

Aby utworzyć dystrybucję, dodaj plik *setup.py* w najwyższym katalogu (w tym przykładzie *spam-project/*). W tym pliku umieść następujący kod:

```
# setup.py
from setuptools import setup

setup(name="spam",
      version="0.0"
      packages=['spam'],
      scripts=['runspam.py'],
)
```

W wywołaniu `setup()` pakiety to lista wszystkich katalogów pakietów, a `scripts` to lista plików skryptów. Każdy z tych argumentów można pominąć, jeśli oprogramowanie ich nie posiada (na przykład jeśli nie ma skryptów). `name` to nazwa Twojego pakietu, a `version` to numer wersji w postaci ciągu. Wywołanie `setup()` obsługuje wiele innych parametrów, które dostarczają różnych metadanych dotyczących Twojego pakietu. Zobacz pełną listę na *https://docs.python.org/3/distutils/apiref.html*.

Dodanie pliku *setup.py* wystarczy, aby utworzyć dystrybucję źródłową oprogramowania. Wpisz następujące polecenie powłoki, aby uzyskać dystrybucję źródłową:

```
bash $ python setup.py sdist
…
bash $
```

Spowoduje to utworzenie pliku archiwum, takiego jak *spam-1.0.tar.gz* lub *spam-1.0.zip*, w katalogu *spam/dist*. Jest to plik, który możesz przekazać innym, aby zainstalować swoje oprogramowanie. Aby je zainstalować, użytkownik może użyć polecenia takiego jak *pip*. Na przykład:

```
shell $ python3 -m pip install spam-1.0.tar.gz
```

Spowoduje to zainstalowanie oprogramowania w lokalnej dystrybucji Pythona i udostępnienie go do ogólnego użytku. Kod jest zwykle instalowany w katalogu o nazwie *site-packages* w bibliotece Pythona. Aby znaleźć dokładną lokalizację tego katalogu, sprawdź wartość `sys.path`. Skrypty są najczęściej instalowane w tym samym katalogu co sam interpreter Pythona.

Jeśli pierwsza linia skryptu zaczyna się od `#!` i zawiera tekst `python`, instalator przepisze linię, aby wskazać lokalną instalację Pythona. Tak więc jeśli Twoje skrypty zostały zakodowane na stałe w określonej lokalizacji Pythona, takiej jak */usr/local/bin/python*, powinny nadal działać po zainstalowaniu w innych systemach, w których Python znajduje się w innej lokalizacji.

Należy podkreślić, że użycie opisanych tutaj narzędzi konfiguracyjnych jest absolutnie minimalne. Większe projekty mogą obejmować rozszerzenia C/C++, skomplikowane struktury pakietów, przykłady i nie tylko. Omówienie wszystkich narzędzi i możliwych sposobów wdrożenia takiego kodu wykracza poza zakres tej książki. Aby uzyskać najbardziej aktualne porady, zapoznaj się z różnymi zasobami na *https://python.org* i *https://pypi.org*.

8.17. Przedostatnie słowo: zacznij od pakietu

Kiedy po raz pierwszy uruchamiasz nowy program, łatwo jest zacząć od prostego pliku Pythona. Na przykład możesz napisać skrypt o nazwie *program.py* i zacząć od tego. Chociaż będzie to działać dobrze w przypadku jednorazowych programów i krótkich zadań, Twój „skrypt" może zacząć się rozwijać i mogą być do niego dodawane nowe funkcje. W końcu możesz podzielić go na wiele plików. W tym momencie często pojawiają się problemy.

W związku z tym warto już od samego początku wyrobić sobie nawyk uruchamiania wszystkich programów jako pakiet. Na przykład zamiast tworzyć plik o nazwie *program.py*, powinieneś utworzyć katalog pakietu programu o nazwie *program*:

```
program/
    __init__.py
    __main__.py
```

Umieść swój kod startowy w *__main__.py* i uruchom program za pomocą polecenia takiego jak `python -m program`. Jeśli potrzebujesz więcej kodu, dodaj nowe pliki do swojego pakietu i skorzystaj z importów względnych. Zaletą korzystania z pakietu jest to, że cały kod pozostaje odizolowany. Możesz dowolnie nazywać pliki i nie martwić się o kolizje z innymi pakietami,

standardowymi modułami bibliotek lub kodem napisanym przez współpracowników. Chociaż konfiguracja pakietu wymaga nieco więcej pracy na początku, prawdopodobnie zaoszczędzi Ci to później wielu problemów.

8.18. Podsumowanie: zachowaj prostotę

Istnieje wiele bardziej zaawansowanych kreatorów związanych z modułem i systemem pakietów niż to, co pokazano tutaj. Zapoznaj się z samouczkiem *Modules and Packages: Live and Let Die!* na *https://dabeaz.com/modulepackage/index.html*, aby zorientować się, jakie są możliwości.

Biorąc to wszystko pod uwagę, prawdopodobnie lepiej *nie* robić żadnych skomplikowanych zmian modułów. Zarządzanie modułami, pakietami i dystrybucją oprogramowania zawsze generowało problemy w społeczności Pythona. Duża część tych problemów jest konsekwencją wprowadzania przez ludzi zmian systemu modułów. Nie rób tego. Zachowaj prostotę i znajdź moc, aby po prostu powiedzieć „nie", gdy Twoi współpracownicy zaproponują zmodyfikowanie importu, aby działał z blockchainem.

9

Obsługa operacji wejścia-wyjścia

Obsługa operacji wejścia-wyjścia (I/O) jest częścią wszystkich programów. W tym rozdziale opisano podstawowe operacje wejścia-wyjścia Pythona, w tym kodowanie danych, opcje wiersza poleceń, zmienne środowiskowe, operacje wejścia-wyjścia plików i serializację danych. Szczególną uwagę zwrócono na techniki programowania i abstrakcje, które zachęcają do właściwej obsługi operacji wejścia-wyjścia. Na końcu tego rozdziału znajduje się przegląd popularnych standardowych modułów bibliotecznych związanych z obsługą takich operacji.

9.1. Reprezentacja danych

Głównym problemem wiążącym się z operacjami wejścia-wyjścia jest świat zewnętrzny. Aby się z nim komunikować, dane muszą być odpowiednio reprezentowane, co pozwala na manipulowanie nimi. Na najniższym poziomie Python pracuje z dwoma podstawowymi typami danych: *bajtami*, reprezentującymi surowe, niezinterpretowane dane dowolnego rodzaju, oraz *tekstem*, reprezentującym znaki Unicode.

Do reprezentowania bajtów używane są dwa wbudowane typy: `bytes` i `bytearray`. `bytes` to niezmienny ciąg wartości całkowitych bajtów. `bytearray` to zmienna tablica bajtów, która zachowuje się jak kombinacja ciągu bajtów i listy. Jego zmienność sprawia, że nadaje się do budowania grup bajtów w sposób bardziej przyrostowy, na przykład podczas łączenia danych z fragmentów.

Poniższy przykład ilustruje kilka funkcji typów `bytes` i `bytearray`:

```
# Określ literał bajtów (zwróć uwagę na prefiks b')
a = b'hello'

# Określ bajty na podstawie listy liczb całkowitych
b = bytes([0x68, 0x65, 0x6c, 0x6c, 0x6f])

# Utwórz i wypełnij bytearray z części
c = bytearray()
```

```
c.extend(b'world') # c = bytearray(b'world')
c.append(0x21)     # c = bytearray(b'world!')

# Dostęp do wartości bajtów
print(a[0]) # --> Wyświetla 104
for x in b: # Wyjście 104 101 108 108 111
    print(x)
```

Dostęp do poszczególnych elementów obiektów `byte` i `bytearray` daje całkowite wartości bajtów, a nie jednoznakowe ciągi bajtów. To zachowanie różni się dla ciągów tekstowych, więc może powodować częsty błąd.

Tekst jest reprezentowany przez typ danych `str` i przechowywany jako tablica znaków kodowych Unicode. Na przykład:

```
d = 'hello'     # Tekst (Unicode)
len(d)          # --> 5
print(d[0])     # Wyświetla 'h'
```

Python utrzymuje ścisłą separację między bajtami i tekstem. Nigdy nie ma automatycznej konwersji między tymi dwoma typami, porównania między nimi są wyliczane jako `False`, a każda operacja, która łączy bajty i tekst, skutkuje błędem. Na przykład:

```
a = b'hello'    # Bajty
b = 'hello'     # Tekst
c = 'world'     # Tekst

print(a == b)   # -> False
d = a + c       # TypeError: nie można połączyć str z bytes
e = b + c       # -> 'helloworld' (oba są ciągami znaków)
```

Podczas wykonywania operacji wejścia-wyjścia upewnij się, że pracujesz z odpowiednią reprezentacją danych. Jeśli manipulujesz tekstem, użyj ciągów tekstowych. Jeśli manipulujesz danymi binarnymi, użyj bajtów.

9.2. Kodowanie i dekodowanie tekstu

Jeśli pracujesz z tekstem, wszystkie dane odczytane z wejścia muszą zostać zdekodowane, a wszystkie dane zapisane na wyjściu muszą być zakodowane. Dla jawnej konwersji między tekstem a bajtami istnieją metody `encode(text [,errors])` i `decode(bytes [,errors])` odpowiednio dla obiektów tekstowych i bajtowych. Na przykład:

```
a = 'hello'           # Tekst
b = a.encode('utf-8') # Koduj do bajtów

c = b'world'          # Bajty
d = c.decode('utf-8') # Dekoduj do tekstu
```

Zarówno `encode()`, jak i `decode()` wymagają nazwy kodowania, takiej jak `'utf-8'` lub `'latin-1'`. W tabeli 9.1 przedstawiono najpopularniejsze rodzaje kodowania.

Tabela 9.1. *Popularne rodzaje kodowania*

Kodowanie	Opis
`'ascii'`	Wartości znaków z zakresu [`0x00, 0x7f`].
`'latin1'`	Wartości znaków z zakresu [`0x00, 0xff`]. Znane również jako `'iso-8859-1'`.
`'utf-8'`	Kodowanie o zmiennej długości, które umożliwia reprezentację wszystkich znaków Unicode.
`'cp1252'`	Powszechne kodowanie tekstu w systemie Windows.
`'macroman'`	Powszechne kodowanie tekstu na komputerach Macintosh.

Ponadto metody kodowania akceptują opcjonalny argument `errors`, który określa zachowanie w przypadku błędów kodowania. Jest to jedna z wartości w tabeli 9.2.

Tabela 9.2. *Opcje obsługi błędów*

Wartość	Opis
`'strict'`	Zgłasza wyjątek `UnicodeError` z powodu błędów kodowania i dekodowania (wartość domyślna).
`'ignore'`	Ignoruje nieprawidłowe znaki.
`'replace'`	Zamienia nieprawidłowe znaki na znak zastępczy (`U+FFFD` w Unicode, `b'?'` w bajtach).
`'backslashreplace'`	Zastępuje każdy nieprawidłowy znak sekwencją wyjścia Pythona. Na przykład znak `U+1234` jest zastępowany przez `'\u1234'` (tylko przy kodowaniu).
`'xmlcharrefreplace'`	Zastępuje każdy nieprawidłowy znak odniesieniem do znaku XML. Na przykład znak `U+1234` jest zastępowany przez `'&#4660; '` (tylko przy kodowaniu).
`'surrogateescape'`	Zamienia każdy nieprawidłowy bajt `'\xhh'` na `U+DChh` podczas dekodowania, zamienia `U+DChh` na bajt `'\xhh'` w trakcie kodowania.

Zasady obsługi błędów `'backslashreplace'` i `'xmlcharrefreplace'` przedstawiają niedrukowalne znaki w formie, która umożliwia ich wyświetlanie jako prosty tekst ASCII lub jako odwołania do znaków XML. Może to być przydatne do debugowania.

Zasady obsługi błędów `'surrogateescape'` pozwalają zdegenerowanym danym bajtowym — danym, które nie są zgodne z oczekiwanymi regułami kodowania — przetrwać w stanie nienaruszonym proces dekodowania/kodowania w obie strony, niezależnie od używanego kodowania tekstu. W szczególności dotyczy to, `s.decode(enc, 'surrogateescape').encode(enc, 'surrogateescape') == s`. Zachowywanie danych w obie strony jest przydatne w przypadku niektórych rodzajów interfejsów systemowych, w których oczekuje się kodowania tekstu, ale nie można tego zagwarantować z powodu problemów pozostających poza kontrolą Pythona. Zamiast niszczyć dane za pomocą złego kodowania, Python używa kodowania zastępczego.

Oto przykład tego zachowania z nieprawidłowo zakodowanym ciągiem UTF-8:

```
>>> a = b'Spicy Jalape\xf1o' # Nieprawidłowe kodowanie UTF-8
>>> a.decode('utf-8')
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xf1
in position 12: invalid continuation byte
>>> a.decode('utf-8', 'surrogateescape')
'Spicy Jalape\udcf1o'
>>> # Zakoduj wynikowy ciąg z powrotem do bajtów
>>> _.encode('utf-8', 'surrogateescape')
b'Spicy Jalape\xf1o'
>>>
```

9.3. Formatowanie tekstu i bajtów

Podczas pracy z tekstem i ciągami bajtów problematyczne są często konwersja i formatowanie ciągów — na przykład konwersja liczby zmiennoprzecinkowej na ciąg o określonej szerokości i precyzji. Aby sformatować pojedynczą wartość, użyj funkcji `format()`:

```
x = 123,456
format(x, '0.2f')     # '123.46'
format(x, '10.4f')    # ' 123.4560'
format(x, '<*10.2f') # '123.46****'
```

Drugim argumentem `format()` jest specyfikator formatu. Ogólny format specyfikatora to `[[wypełnienie[wyrównanie]][znak][0][szerokość][,][.precyzja][typ]`, gdzie każda część zawarta w [] jest opcjonalna. `szerokość` określa minimalną szerokość pola do użycia, a specyfikator wyrównania jest jednym ze znaków <, > lub ^ dla wyrównania do lewej, do prawej i do środka.

Opcjonalne `wypełnienie` jest używane do wypełnienia spacjami. Na przykład:

```
r = format(name, '<10')      # r = 'Elwood    '
r = format(name, '>10')      # r = '    Elwood'
r = format(name, '^10')      # r = '  Elwood  '
r = format(name, '*^10')     # r = '**Elwood**'
```

Specyfikator `typ` wskazuje typ danych. Tabela 9.3 zawiera listę obsługiwanych kodów formatów. Jeśli nie zostanie podany, domyślnym kodem formatu jest `s` dla łańcuchów, `d` dla liczb całkowitych i `f` dla liczb zmiennoprzecinkowych.

Część znakowa specyfikatora formatu to znak +, - lub spacja. Znak + wskazuje, że znak wiodący powinien być używany na wszystkich liczbach. Znak - jest wartością domyślną i dodaje znak tylko dla liczb ujemnych. Spacja dodaje wiodącą spację do liczb dodatnich.

Opcjonalny przecinek (`,`) może się pojawić między szerokością a precyzją. Dodaje znak separatora części tysięcznej. Na przykład:

```
x = 123456,78
format(x, '16,.2f') # ' 123,456,78'
```

Część specyfikatora `.precyzja` dostarcza liczbę cyfr dokładności, które mają być używane dla ułamków dziesiętnych. Jeśli do szerokości pola liczb dodawane jest początkowe 0, wartości liczbowe są dopełniane wiodącymi zerami, aby wypełnić przestrzeń. Oto kilka przykładów formatowania różnych rodzajów liczb:

Tabela 9.3. *Kody formatowania*

Znak	Format wyjściowy znaków
`d`	Dziesiętna liczba całkowita lub liczba całkowita typu `long`
`b`	Binarna liczba całkowita lub liczba całkowita typu `long`
`o`	Ósemkowa liczba całkowita lub liczba całkowita typu `long`
`x`	Szesnastkowa liczba całkowita lub liczba całkowita typu `long`
`X`	Szesnastkowa liczba całkowita (wielkie litery)
`f, F`	Liczba zmiennoprzecinkowa w formacie `[-]m.dddddd`
`e`	Liczba zmiennoprzecinkowa w formacie `[-]m.dddddde±xx`
`E`	Liczba zmiennoprzecinkowa w formacie `[-]m.ddddddE±xx`
`g, G`	Użyj `e` lub `E` dla wykładników mniejszych niż `[nd]4` lub większych niż precyzja; w przeciwnym razie użyj `f`
`n`	To samo co `g`, z wyjątkiem tego, że bieżące ustawienia regionalne określają znak kropki dziesiętnej
`%`	Mnoży liczbę przez 100 i wyświetla ją w formacie `f`, po którym następuje znak `%`
`s`	Ciąg lub dowolny obiekt. Kod formatujący używa `str()` do generowania ciągów
`c`	Pojedynczy znak

```
x = 42
r = format(x, '10d')      # r = ' 42'
r = format(x, '10x')      # r = ' 2a'
r = format(x, '10b')      # r = ' 101010'
r = format(x, '010b')     # r = '0000101010'

y = 3.1415926
r = format(y, '10.2f')    # r = ' 3.14'
r = format(y, '10.2e')    # r = ' 3.14e+00'
r = format(y, '+10.2f')   # r = ' +3.14'
r = format(y, '+010.2f')  # r = '+000003.14'
r = format(y, '+10.2%')   # r = ' +314.16%'
```

Aby uzyskać bardziej złożone formatowanie ciągów, możesz użyć f-stringów:

```
x = 123.456
f'Wartość to {x:0.2f}'        # 'Wartość to 123.46'
f'Wartość to {x:10.4f}'       # 'Wartość to 123.4560'
f'Wartość to {2*x:*<10.2f}' # 'Wartość to 246.91****'
```

Wewnątrz f-stringa tekst w postaci `{expr:spec}` jest zastępowany wartością `format(expr, spec)`. `expr` może być dowolnym wyrażeniem, o ile nie zawiera znaków `{`, `}` lub `\`. Części samego specyfikatora formatu mogą być opcjonalnie dostarczane przez inne wyrażenia. Na przykład:

```
y = 3.1415926
width = 8
precision=3

r = f'{y:{width}.{precision}f}' # r = ' 3.142'
```

Jeśli zakończysz `expr` znakiem `=`, to dosłowny tekst `expr` również zostanie uwzględniony w wyniku. Na przykład:

```
x = 123.456

f'{x=:0.2f}'    # 'x=123.46'
f'{2*x=:0.2f}' # '2*x=246.91'
```

Jeśli dodasz `!r` do wartości, formatowanie zostanie zastosowane do wyjścia funkcji `repr()`. Jeśli używasz `!s`, formatowanie jest stosowane do wyjścia `str()`. Na przykład:

```
f'{x!r:spec}'   # Wywołuje (repr(x).__format__('spec'))
f'{x!s:spec}'   # Wywołuje (str(x).__format__('spec'))
```

Jako alternatywę dla f-stringów możesz użyć metody dla ciągów znaków `.format()`:

```
x = 123.456

'Wartość to {:0.2f}'.format(x)             # 'Wartość to 123.46'
'Wartość to {0:10.2f}'.format(x)           # 'Wartość to 123.4560'
'Wartość to {val:<*10.2f}'.format(val=x) # 'Wartość to 123.46****'
```

W przypadku ciągu sformatowanego przez `.format()` tekst w postaci `{arg:spec}` jest zastępowany wartością `format(arg, spec)`. Tu `arg` odnosi się do jednego z argumentów podanych w metodzie `format()`. W przypadku całkowitego pominięcia `arg` argumenty są rozpatrywane w podanej kolejności. Na przykład:

```
name = 'IBM'
shares = 50
price = 490.1

r = '{:>10s} {:10d} {:10.2f}'.format(name, shares, price)
# r = '    IBM         50     490.10'
```

`arg` może się również odnosić do określonego numeru lub nazwy argumentu. Na przykład:

```
tag = 'p'
text = 'hello world'
r = '<{0}>{1}</{0}>'.format(tag, text) # r = '<p>hello world</p>'
r = '<{tag}>{text}</{tag}>'.format(tag='p', text='hello world')
```

W przeciwieństwie do f-stringów wartość `arg` specyfikatora nie może być dowolnym wyrażeniem. Jednak metoda `format()` może wykonywać ograniczone wyszukiwanie atrybutów, indeksowanie i zagnieżdżone podstawienia. Na przykład:

```
y = 3.1415926
width = 8
precision = 3

r = 'Wartość to {0:{1}.{2}f}'.format(y, width, precision)

d = {
   'name': 'IBM',
   'shares': 50,
   'price': 490.1
```

```
}
r = '{0[shares]:d} udziałów {0[name]} po {0[price]:0.2f}'.format(d)
# r = '50 udziałów IBM po 490.10'
```

Instancje bytes i bytearray można sformatować za pomocą operatora %. Semantyka tego operatora jest wzorowana na funkcji sprintf() z języka C. Oto kilka przykładów:

```
name = b'ACME'
x = 123.456

b'Value is %0.2f' % x             # b'The value is 123.46'
bytearray(b'Value is %0.2f') % x  # b'Value is 123.46'
b'%s = %0.2f' % (name, x)         # b'ACME = 123.46'
```

Przy tym formatowaniu sekwencje w postaci %spec są zastępowane wartościami z krotki dostarczonej jako drugi operand do operatora %. Podstawowe kody formatu (d, f, s itd.) są takie same jak te używane w funkcji format(). Brakuje jednak bardziej zaawansowanych funkcji albo są one nieznacznie zmienione. Aby na przykład dostosować wyrównanie, użyj znaku - w następujący sposób:

```
x = 123.456
b'%10.2f' % x     # b' 123.46'
b'%-10.2f' % x    # b'123.46 '
```

Użycie kodu formatu %r generuje dane wyjściowe funkcji ascii(), które mogą być przydatne podczas debugowania i logowania.

Podczas pracy z bajtami należy pamiętać, że ciągi tekstowe nie są obsługiwane. Muszą być wyraźnie zakodowane.

```
name = 'Dave'

b'Hello %s' % name                  # TypeError!
b'Hello %s' % name.encode('utf-8')  # OK
```

Taki sposób formatowania może być również używana z ciągami tekstowymi, ale jest uważana za przestarzały styl programowania. Nadal jednak pojawia się w niektórych bibliotekach. W ten sposób formatowane są na przykład komunikaty generowane przez moduł logging:

```
import logging
log = logging.getLogger(__name__)

log.debug('%s dostaje wartość %d', name, value) # '%s dostaje wartość %d' % (name, value)
```

Moduł logging został pokrótce opisany w dalszej części tego rozdziału, w sekcji 9.15.12.

9.4. Czytanie opcji wiersza poleceń

Po uruchomieniu Pythona opcje wiersza poleceń są umieszczane na liście sys.argv jako ciągi tekstowe. Pierwsza pozycja to nazwa programu. Kolejne pozycje to opcje dodawane w wierszu poleceń po nazwie programu. Poniższy program jest minimalnym prototypem ręcznego przetwarzania argumentów wiersza poleceń:

```
def main(argv):
    if len(argv) != 3:
        raise SystemExit(
            f'Przykład użycia: python {argv[0]} plik_wejściowy plik_wyjściowy\n')
    inputfile = argv[1]
    outputfile = argv[2]
    ...

if __name__ == '__main__':
    import sys
    main(sys.argv)
```

Ze względu na lepszą organizację kodu, testowanie i z podobnych powodów warto napisać funkcję `main()`, która akceptuje opcje wiersza poleceń (jeśli istnieją) jako listę, w przeciwieństwie do bezpośredniego czytania `sys.argv`. Dołącz mały fragment kodu na końcu programu, aby przekazać opcje wiersza poleceń do funkcji `main()`.

`sys.argv[0]` zawiera nazwę wykonywanego skryptu. Pisanie opisowego komunikatu pomocy i zgłaszanie wyjątku `SystemExit` to standardowa praktyka w przypadku tych skryptów wiersza poleceń, które chcą zgłosić błąd.

Chociaż w prostych skryptach możesz ręcznie przetwarzać opcje poleceń, przy bardziej skomplikowanej obsłudze wiersza poleceń rozważ użycie modułu `argparse`. Oto przykład:

```
import argparse

def main(argv):
    p = argparse.ArgumentParser(description="To jest jakiś program")

    # Argument pozycyjny
    p.add_argument("infile")

    # Opcja przyjmująca argument
    p.add_argument("-o","--output", action="store")

    # Opcja, która ustawia flagę logiczną
    p.add_argument("-d","--debug", action="store_true", default=False)

    # Analiza wiersz poleceń
    args = p.parse_args(args=argv)

    # Pobierz ustawienia opcji
    infile = args.infile
    output = args.output
    debugmode = args.debug

    print(infile, output, debugmode)

if __name__ == '__main__':
    import sys
    main(sys.argv[1:])
```

Ten przykład pokazuje tylko najprostsze użycie modułu `argparse`. Dokumentacja biblioteki standardowej dostarcza bardziej zaawansowanych przykładów wykorzystania. Istnieją również moduły innych firm, takie jak `click` i `docopt`, które mogą uprościć pisanie bardziej złożonych parserów wiersza poleceń.

Wreszcie, opcje wiersza poleceń mogą być dostarczane do Pythona w nieprawidłowym kodowaniu tekstu. Takie argumenty są nadal akceptowane, ale zostaną zakodowane przy użyciu metody obsługi błędów `'surrogateescape'`, jak opisano w sekcji 9.2. Musisz być tego świadomy, jeśli takie argumenty zostaną później uwzględnione w jakimkolwiek wyniku tekstowym i bardzo ważne będzie, aby uniknąć awarii. Może to jednak nie być krytyczne — nie komplikuj nadmiernie kodu w skrajnych przypadkach, które nie mają znaczenia.

9.5. Zmienne środowiskowe

Czasami dane są przekazywane do programu za pośrednictwem zmiennych środowiskowych ustawionych w powłoce systemowej. Na przykład program w Pythonie może zostać uruchomiony za pomocą polecenia powłoki, takiego jak *env*:

```
bash $ env SOMEVAR=jakaś_wartość python3 somescript.py
```

Zmienne środowiskowe są dostępne jako ciągi tekstowe w mapowaniu `os.environ`. Oto przykład:

```
import os
path = os.environ['PATH']
user = os.environ['USER']
editor = os.environ['EDITOR']
val = os.environ['SOMEVAR']
...itd. ...
```

Aby zmodyfikować zmienne środowiskowe, ustaw zmienną `os.environ`. Na przykład:

```
os.environ['NAZWA'] = 'WARTOŚĆ'
```

Modyfikacje `os.environ` wpływają zarówno na uruchomiony program, jak i na wszelkie podprocesy utworzone później, na przykład te utworzone przez moduł `subprocess`.

Podobnie jak w przypadku opcji wiersza poleceń, źle zakodowane zmienne środowiskowe mogą generować ciągi znaków, które używają zasad obsługi błędów `'surrogateescape'`.

9.6. Pliki i obiekty plików

Aby otworzyć plik, użyj wbudowanej funkcji `open()`. Zwykle `open()` otrzymuje jako dane wejściowe nazwę pliku i tryb jego otwarcia. Instrukcja ta jest również często używana w połączeniu z instrukcją `with` jako menedżer kontekstu. Oto kilka typowych wzorców użycia podczas pracy z plikami:

```
# Odczytaj plik tekstowy od razu jako ciąg
with open('filename.txt', 'rt') as file:
    data = file.read()

# Czytaj plik wiersz po wierszu
with open('filename.txt', 'rt') as file:
    for line in file:
        ...
```

```
# Zapisz do pliku tekstowego
with open('out.txt', 'wt') as file:
    file.write('Jakieś dane wyjściowe\n')
    print('Więcej danych wyjściowych', file=file)
```

W większości przypadków użycie funkcji `open()` jest proste. Podajesz jej nazwę pliku, który chcesz otworzyć, wraz z trybem pliku. Na przykład:

```
open('name.txt')        # Otwiera plik "name.txt" do czytania
open('name.txt', 'rt') # Otwiera plik "name.txt" do czytania (to samo co wyżej)
open('name.txt', 'wt') # Otwiera "name.txt" do zapisu
open('data.bin', 'rb') # Odczyt w trybie binarnym
open('data.bin', 'wb') # Zapis w trybie binarnym
```

W przypadku większości programów podczas pracy z plikami wystarczą ci informacje zawarte w tych prostych przykładach. Jest jednak kilka specjalnych przypadków i bardziej zaawansowanych cech `open()`, o których warto wiedzieć. W następnych kilku sekcjach bardziej szczegółowo omówiono metodę `open()` i operacje wejścia-wyjścia plików.

9.6.1. Nazwy plików

Aby otworzyć plik, musisz podać funkcji `open()` nazwę pliku. Nazwa może być w pełni określoną ścieżką bezwzględną, taką jak `'/Users/guido/Desktop/files/old/data.csv'`, lub względną nazwą ścieżki, taką jak `'data.csv'` bądź `'..\old\data.csv'`. W przypadku względnych nazw plików położenie pliku jest określane względem bieżącego katalogu roboczego zwracanego przez `os.getcwd()`. Bieżący katalog roboczy można zmienić za pomocą `os.chdir(newdir)`.

Sama nazwa może być zakodowana w wielu formach. Jeśli jest to ciąg tekstowy, nazwa jest interpretowana zgodnie z kodowaniem tekstu zwracanym przez `sys.getfilesystemencoding()` przed przekazaniem do systemu operacyjnego hosta. Jeśli nazwa pliku jest ciągiem bajtów, pozostaje niezakodowana i jest przekazywana bez zmian. Ta ostatnia opcja może być przydatna, jeśli piszesz programy, które muszą obsługiwać możliwość błędnie zakodowanych nazw plików — zamiast przekazywać nazwę pliku jako tekst, możesz przekazać surową binarną reprezentację nazwy. Może się to wydawać niejasnym przypadkiem, ale Python jest powszechnie używany do pisania skryptów systemowych, które manipulują systemem plików. Nadużywanie systemu plików jest powszechną techniką stosowaną przez hakerów do ukrywania swoich śladów lub łamania narzędzi systemowych.

Oprócz tekstu i bajtów jako nazwy można użyć dowolnego obiektu, który implementuje specjalną metodę `__fspath__()`. Metoda `__fspath__()` musi zwracać obiekt tekstowy lub bajtowy odpowiadający rzeczywistej nazwie. Jest to mechanizm, który zapewnia moduł biblioteczny, taki jak `pathlib`. Na przykład:

```
>>> from pathlib import Path
>>> p = Path('Data/portfolio.csv')
>>> p.__fspath__()
'Data/portfolio.csv'
>>>
```

Potencjalnie mógłbyś stworzyć własny niestandardowy obiekt `Path`, który działałby z `open()`, o ile implementowałby `__fspath__()` odwołujący się do właściwej nazwy pliku w systemie.

Wreszcie, nazwy plików mogą być podane jako deskryptory plików. Wymaga to, aby plik był już w jakiś sposób otwarty w systemie. Być może odpowiada gniazdu sieciowemu (ang. *socket*), potokowi lub innemu zasobowi systemowemu, który udostępnia deskryptor pliku.

Oto przykład otwierania pliku bezpośrednio za pomocą modułu `os`, a następnie przekształcania go w odpowiedni obiekt pliku:

```
>>> import os
>>> fd = os.open('/etc/passwd', os.O_RDONLY) # Deskryptor fd
>>> fd
3
>>> file = open(fd, 'rt') # Właściwy obiekt pliku
>>> file
<_io.TextIOWrapper name=3 mode='rt' encoding='UTF-8'>
>>> data = file.read()
>>>
```

Podobnie jak wyżej, podczas otwierania istniejącego deskryptora pliku metoda `close()` otrzymanego obiektu pliku również zamknie deskryptor bazowy. Można to wyłączyć, przekazując `closefd=False` do funkcji `open()`. Na przykład:

```
file = open(fd, 'rt', closefd=False)
```

9.6.2. Tryby plików

Podczas otwierania pliku musisz określić jego tryb. Podstawowe tryby plików to `'r'` do czytania, `'w'` do zapisu i `'a'` do dołączania. Tryb `'w'` zastępuje istniejący plik nową zawartością, zaś `'a'` otwiera plik do zapisu i umieszcza wskaźnik pliku na końcu pliku, aby można było dołączyć nowe dane.

Specjalny tryb pliku `'x'` może być użyty do zapisu do pliku, ale tylko wtedy, gdy jeszcze nie istnieje. Jest to przydatny sposób zapobiegania przypadkowemu nadpisaniu istniejących danych. W tym trybie zgłaszany jest wyjątek `FileExistsError`, jeśli plik już istnieje.

Python dokonuje ścisłego rozróżnienia między danymi tekstowymi i binarnymi. Aby określić rodzaj danych, dodajesz `'t'` lub `'b'` do trybu pliku. Na przykład tryb plikowy `'rt'` otwiera plik do odczytu w trybie tekstowym, a `'rb'` otwiera plik do odczytu w trybie binarnym. Tryb określa rodzaj danych zwracanych przez metody związane z plikami, takie jak `f.read()`. W trybie tekstowym zwracane są ciągi. W trybie binarnym zwracane są bajty.

Pliki binarne można otwierać w celu aktualizacji w miejscu, podając znak plus (+), taki jak `'rb+'` lub `'wb+'`. Gdy plik jest otwierany do aktualizacji, można wykonywać zarówno operacje wejścia, jak i wyjścia, o ile wszystkie operacje wyjścia opróżniają swoje dane przed kolejnymi operacjami wejścia. Jeśli plik jest otwierany w trybie `'wb+'`, jego długość jest najpierw obcinana do zera. Typowym zastosowaniem trybu aktualizacji jest zapewnienie losowego dostępu do odczytu/zapisu zawartości pliku w połączeniu z operacjami wyszukiwania.

9.6.3. Buforowanie operacji wejścia-wyjścia

Domyślnie pliki są otwierane z włączonym buforowaniem operacji wejścia-wyjścia. Dzięki buforowaniu operacje te są wykonywane w większych porcjach, aby uniknąć nadmiernych wywołań systemowych. Na przykład operacje zapisu rozpoczęłyby wypełnianie wewnętrznego

bufora pamięci, a dane wyjściowe byłyby dostępne dopiero wtedy, gdy bufor jest zapełniony. To zachowanie można zmienić, podając argument `buffering` dla funkcji `open()`. Na przykład:

```
# Otwórz plik w trybie binarnym bez buforowania I/O
with open('data.bin', 'wb', buffering=0) as file:
    file.write(data)
    ...
```

Wartość `0` określa niebuforowane operacje wejścia-wyjścia i jest prawidłowa tylko dla plików otwieranych w trybie binarnym. Wartość `1` określa buforowanie linii i zwykle ma znaczenie jedynie dla plików w trybie tekstowym. Każda inna wartość dodatnia wskazuje rozmiar bufora do użycia (w bajtach). Jeśli nie określono wartości buforowania, domyślne zachowanie zależy od rodzaju pliku. Jeśli jest to normalny plik na dysku, buforowanie jest zarządzane blokowo, a rozmiar bufora ma wartość `io.DEFAULT_BUFFER_SIZE`. Zazwyczaj jest to mała wielokrotność 4096 bajtów, choć może się różnić w zależności od systemu. Jeśli obiekt plikowy reprezentuje terminal interaktywny, używane jest buforowanie linii.

W przypadku normalnych programów buforowanie operacji wejścia-wyjścia nie jest zazwyczaj poważnym problemem. Jednak buforowanie może mieć wpływ na aplikacje wymagające aktywnej komunikacji między procesami. Na przykład czasami pojawia się problem dwóch komunikujących się podprocesów, które ulegają zakleszczeniu z powodu wewnętrznego problemu z buforowaniem — przykładowo jeden proces zapisuje do bufora, ale odbiorca nigdy nie widzi tych danych, ponieważ bufor nie został opróżniony. Takie problemy można rozwiązać, określając niebuforowane operacje wejścia-wyjścia lub wywołując w sposób jawny funkcję `flush()` dla powiązanego pliku. Na przykład:

```
file.write(data)
file.write(data)
...
file.flush()  # Upewnij się, że wszystkie dane są zapisywane z buforów
```

9.6.4. Kodowanie w trybie tekstowym

W przypadku plików otwieranych w trybie tekstowym można określić opcjonalne zasady kodowania i obsługi błędów za pomocą argumentów kodowania i błędów. Na przykład:

```
with open('file.txt', 'rt',
          encoding='utf-8', errors='replace') as file:
    data = file.read()
```

Wartości podane w argumentach `encoding` i `errors` mają takie samo znaczenie jak w przypadku metod `encode()` i `decode()` odpowiednio łańcuchów i bajtów.

Domyślne kodowanie tekstu jest określone przez wartość `sys.getdefaultencoding()` i może się różnić w zależności od systemu. Jeśli znasz kodowanie, często lepiej podać je jawnie, nawet jeśli pasuje do domyślnego kodowania w Twoim systemie.

9.6.5. Obsługa wiersza w trybie tekstowym

W przypadku plików tekstowych jedną z komplikacji jest kodowanie znaków nowej linii. Znaki nowej linii są kodowane jako `'\n'`, `'\r\n'` lub `'\r'` — w zależności od systemu operacyjnego hosta, na przykład `'\n'` w systemie UNIX i `'\r\n'` w systemie Windows. Domyślnie podczas czytania Python tłumaczy wszystkie te zakończenia linii na standardowy znak `'\n'`. Podczas zapisu znaki nowego wiersza są tłumaczone z powrotem na domyślne zakończenie linii używane w systemie. Zachowanie to jest czasami określane w dokumentacji Pythona jako „uniwersalna obsługa nowego wiersza".

Możesz zmienić zachowanie nowego wiersza, podając argument `newline` dla funkcji `open()`. Na przykład:

```
# Wymagaj '\r\n' i pozostaw nienaruszone
file = open('somefile.txt', 'rt', newline='\r\n')
```

Określenie `newline=None` włącza domyślne zachowanie obsługi wiersza, w którym wszystkie zakończenia linii są tłumaczone na standardowy znak `'\n'`. Nadanie `newline=''` sprawia, że Python rozpoznaje wszystkie zakończenia linii, ale wyłącza krok tłumaczenia — jeśli wiersze byłyby zakończone przez `'\r\n'`, kombinacja `'\r\n'` pozostałaby nienaruszona w danych wejściowych. Określenie wartości `'\n'`, `'\r'` lub `'\r\n'` powoduje, że oczekiwany jest koniec wiersza.

9.7. Warstwy abstrakcyjne wejścia-wyjścia

Funkcja `open()` jest rodzajem wysokopoziomowej funkcji fabryki do tworzenia instancji różnych klas wejścia-wyjścia. Te klasy zawierają różne tryby obsługi plików, kodowania i zachowania buforowania. Są również skomponowane w warstwach. W module `io` zdefiniowane są następujące klasy:

`FileIO(filename, mode='r', closefd=True, opener=None)`

Otwiera plik dla surowego, niebuforowanego binarnego wejścia-wyjścia. Nazwa pliku to dowolna prawidłowa akceptowana przez funkcję `open()` nazwa pliku. Inne argumenty mają takie samo znaczenie jak dla funkcji `open()`.

`BufferedReader(file [, buffer_size])`

`BufferedWriter(file [, buffer_size])`

`BufferedRandom(file [, buffer_size])`

Implementuje buforowaną binarną warstwę wejścia-wyjścia dla pliku. `file` jest instancją obiektu `FileIO`.

`buffer_size` określa rozmiar wewnętrznego bufora. Wybór klasy zależy od tego, czy plik odczytuje, zapisuje, czy aktualizuje dane.

Opcjonalny argument `buffer_size` określa używany rozmiar bufora wewnętrznego.

```
TextIOWrapper(buffered, [encoding, [errors [, newline [, line_buffering [,
write_through]]]]])
```

Implementuje obsługę wejścia-wyjścia w trybie tekstowym. `buffered` to buforowany plik w trybie binarnym, taki jak `BufferedReader` lub `BufferedWriter`. Argumenty kodowania, błędów i nowej linii mają takie samo znaczenie jak dla `open()`. `line_buffering` to flaga logiczna, która wymusza opróżnianie wejścia-wyjścia w przypadku napotkania znaków nowej linii (domyślnie `False`). `write_through` to flaga logiczna, która wymusza opróżnienie bufora dla wszystkich zapisów (domyślnie `False`).

Oto przykład, który pokazuje, jak tworzony jest plik w trybie tekstowym, warstwa po warstwie:

```
>>> raw = io.FileIO('filename.txt', 'r')              # Tryb raw-binary
>>> bufor = io.BufferedReader(raw)                    # Czytnik buforowany binarnie
>>> file = io.TextIOWrapper(buffer, encoding='utf-8') # Tryb tekstowy
>>>
```

Zwykle nie musisz ręcznie konstruować warstw w ten sposób — wbudowana funkcja `open()` zajmuje się całą pracą. Jeśli jednak masz już istniejący obiekt pliku i chcesz w jakiś sposób zmienić jego obsługę, możesz manipulować warstwami, jak pokazano poniżej.

Aby usunąć warstwy, użyj metody `detach()`. Oto przykład, jak przekonwertować obsługę pliku z trybu tekstowego na tryb binarny:

```
f = open('something.txt', 'rt') # Obsługa pliku w trybie tekstowym
fb = f.detach()                 # Odłącz tryb binarny
data = fb.read()                # Zwraca bajty
```

9.7.1. Metody plików

Dokładny typ obiektu zwracanego przez `open()` zależy od kombinacji trybu obsługi pliku i dostarczonych opcji buforowania. Wynikowy obiekt pliku obsługuje metody przedstawione w tabeli 9.4.

Tabela 9.4. *Metody plików*

Metoda	Opis
`f.readable()`	Zwraca `True`, jeśli plik może być odczytany.
`f.read([n])`	Czyta co najwyżej `n` bajtów.
`f.readline([n])`	Czyta pojedynczy wiersz danych wejściowych do `n` znaków. Jeśli pominięto `n`, ta metoda odczytuje całą linię.
`f.readlines([size])`	Czyta wszystkie wiersze i zwraca listę. `size` opcjonalnie określa przybliżoną liczbę znaków do odczytania z pliku.
`f.readinto(buffer)`	Czyta dane do bufora pamięci.
`f.writable()`	Zwraca `True`, jeśli można zapisać plik.
`f.write(s)`	Zapisuje ciąg `s`.
`f.writelines(lines)`	Zapisuje wszystkie łańcuchy w iterowalnych liniach.
`f.close()`	Zamyka plik.

Tabela 9.4. *Metody plików (ciąg dalszy)*

Metoda	Opis
`f.seekable()`	Zwraca `True`, jeśli plik obsługuje wyszukiwanie o dostępie swobodnym.
`f.tell()`	Zwraca wskaźnik bieżącego pliku.
`f.seek(offset [, where])`	Szuka nowej pozycji pliku.
`f.isatty()`	Zwraca `True`, jeśli `f` jest terminalem interaktywnym.
`f.flush()`	Opróżnia bufory wyjściowe.
`f.truncate([size])`	Obcina plik do co najwyżej `size` bajtów.
`f.fileno()`	Zwraca deskryptor pliku liczb całkowitych.

Metody `readable()`, `writeable()` i `seekable()` testują obsługiwane możliwości i tryby plików. Metoda `read()` zwraca cały plik jako ciąg znaków, chyba że opcjonalny parametr długości określa maksymalną liczbę znaków. Metoda `readline()` zwraca następny wiersz danych wejściowych, w tym kończący znak nowego wiersza; metoda `readlines()` zwraca cały plik wejściowy jako listę ciągów. Metoda `readline()` opcjonalnie przyjmuje maksymalną długość linii (`n`). Jeśli odczytywana jest linia dłuższa niż `n` znaków, zwracane jest pierwsze `n` znaków. Pozostałe dane wiersza nie są usuwane i zostaną zwrócone w kolejnych operacjach odczytu. Metoda `readlines()` akceptuje parametr `size` określający przybliżoną liczbę znaków do odczytania. Rzeczywista liczba odczytanych znaków może być większa niż `size`, w zależności od tego, ile danych zostało już zbuforowanych. Metoda `readinto()` służy do unikania tworzenia kopii pamięci i została omówiona później.

`read()` i `readline()` wskazują koniec pliku (EOF) przez zwrócenie pustego ciągu. W związku z tym poniższy kod pokazuje, jak można wykryć warunek EOF:

```
while True:
    line = file.readline()
    if not line: # EOF
        break
    instrukcje
    ...
```

Możesz również napisać ten kod w następujący sposób:

```
while (line:=file.readline()):
    instrukcje
    ...
```

Wygodnym sposobem odczytania wszystkich wierszy w pliku jest użycie iteracji z pętlą `for`:

```
for line in file: # Iteruj po wszystkich wierszach pliku
    # Zrób coś z linią
    ...
```

Metoda `write()` zapisuje dane do pliku, a metoda `writelines()` zapisuje do pliku ciągi iteracyjne. `write()` i `writelines()` nie dodają znaków nowej linii do danych wyjściowych, więc wszystkie dane wyjściowe, które tworzysz, powinny już zawierać niezbędne formatowanie.

Wewnętrznie każdy otwarty obiekt plikowy zawiera wskaźnik pliku przechowujący przesunięcie bajtów, przy którym nastąpi następna operacja odczytu lub zapisu. Metoda `tell()` zwraca bieżącą wartość wskaźnika pliku. Metoda `seek(offset [,whence])` służy do losowego dostępu do części pliku, w którym podano przesunięcie i regułę `whence`. Jeśli `whence` ma wartość `os.SEEK_SET` (wartość domyślna), `seek()` zakłada, że przesunięcie jest względne w stosunku do początku pliku; jeśli `whence` ma wartość `os.SEEK_CUR`, pozycja jest przesuwana względem aktualnej pozycji; a jeśli `whence` ma wartość `os.SEEK_END`, przesunięcie jest pobierane od końca pliku.

Metoda `fileno()` zwraca deskryptor pliku w postaci liczby całkowitej dla pliku i jest czasami używana w niskopoziomowych operacjach wejścia-wyjścia w niektórych modułach bibliotecznych. Na przykład moduł `fcntl` używa deskryptora pliku do wykonywania niskopoziomowych operacji kontroli plików w systemach UNIX.

Metoda `readinto()` służy do wykonywania operacji odczytu danych do buforów pamięci. Jest najczęściej używana — w połączeniu z wyspecjalizowanymi bibliotekami, takimi jak `numpy` — na przykład do odczytywania danych bezpośrednio do pamięci przydzielonej dla tablicy liczbowej.

Obiekty plikowe mają również atrybuty danych tylko do odczytu przedstawione w tabeli 9.5.

Tabela 9.5. *Atrybuty plików*

Atrybut	Opis
`f.closed`	Wartość logiczna wskazuje stan pliku: `False`, jeśli plik jest otwarty, `True`, jeśli jest zamknięty.
`f.mode`	Tryb wejścia-wyjścia dla pliku.
`f.name`	Nazwa pliku utworzonego za pomocą `open()`. W przeciwnym razie będzie to ciąg znaków wskazujący źródło pliku.
`f.newlines`	Reprezentacja nowego wiersza, znaleziona w pliku. Wartość to `None`, jeśli nie napotkano nowej linii, ciągu znaków zawierającego `'\n'`, `'\r'` lub `'\r\n'` albo krotki zawierającej wszystkie widoczne kodowania nowych linii.
`f.encoding`	Ciąg wskazujący kodowanie pliku, jeśli istnieje (na przykład `'latin-1'` lub `'utf-8'`). Wartość to `None`, jeśli nie jest używane żadne kodowanie.
`f.errors`	Zasady obsługi błędów.
`f.write_through`	Wartość logiczna wskazująca, czy zapisy w pliku tekstowym przekazują dane bezpośrednio do pliku binarnego bez buforowania.

9.8. Standardowe wejście, wyjście i błąd

Interpreter udostępnia trzy standardowe obiekty plikopodobne, znane jako standardowe wejście, standardowe wyjście i standardowe błędy, dostępne odpowiednio jako `sys.stdin`, `sys.stdout` i `sys.stderr`. `stdin` to obiekt pliku odpowiadający strumieniowi znaków wejściowych dostarczonych do interpretera, `stdout` to obiekt pliku, który otrzymuje dane wyjściowe generowane przez `print()`, a `stderr` to plik, który otrzymuje komunikaty o błędach. Najczęściej `stdin` jest mapowany na klawiaturę użytkownika, podczas gdy `stdout` i `stderr` wyświetlają tekst na ekranie.

Metody opisane w poprzedniej sekcji mogą służyć do wykonywania operacji wejścia-wyjścia użytkownika. Na przykład poniższy kod zapisuje na standardowe wyjście i odczytuje wiersz ze standardowego wejścia:

```
import sys
sys.stdout.write("Wprowadź swoje imię: ")
name = sys.stdin.readline()
```

Alternatywnie wbudowana funkcja `input(prompt)` może odczytać wiersz tekstu z `stdin` i opcjonalnie wyświetlić monit:

```
name = input("Wprowadź swoje imię: ")
```

Wiersze odczytywane przez `input()` nie zawierają końca nowej linii. Różni się to od czytania bezpośrednio z `sys.stdin`, gdzie znaki nowej linii są zawarte w tekście wejściowym.

Jeśli to konieczne, wartości `sys.stdout`, `sys.stdin` i `sys.stderr` można zastąpić innymi obiektami plików, w którym to przypadku funkcje `print()` i `input()` będą używać nowych wartości. Jeśli kiedykolwiek zajdzie potrzeba przywrócenia oryginalnej wartości `sys.stdout`, należy ją najpierw zapisać. Oryginalne wartości `sys.stdout`, `sys.stdin` i `sys.stderr` podczas uruchamiania interpretera są również dostępne odpowiednio w `sys.__stdout__`, `sys.__stdin__` i `sys.__stderr__`.

9.9. Katalogi

Aby uzyskać listę katalogów, użyj funkcji `os.listdir(pathname)`. Oto przykład, jak wydrukować listę nazw plików w katalogu:

```
import os

names = os.listdir('dirname')
for name in names:
    print(name)
```

Nazwy zwracane przez `listdir()` są zwykle dekodowane zgodnie z kodowaniem zwracanym przez `sys.getfilesystemencoding()`. Jeśli określisz początkową ścieżkę jako bajty, nazwy plików są zwracane jako nieodkodowane ciągi bajtów. Na przykład:

```
import os

# Zwróć surowe nieodkodowane nazwy
names = os.listdir(b'dirname')
```

Przydatną operacją związaną z listowaniem katalogów jest dopasowywanie nazw plików według wzorca znanego jako *globbing* (obsługa symboli wieloznacznych). W tym celu można wykorzystać moduły `pathlib`. Oto przykład dopasowania wszystkich plików **.txt* w określonym katalogu:

```
import pathlib

for filename in path.Path('dirname').glob('*.txt'):
    print(filename)
```

Jeśli użyjesz funkcji `rglob()` zamiast `glob()`, zostaną rekursywnie przeszukane wszystkie podkatalogi w poszukiwaniu nazw plików, które pasują do wzorca. Obie funkcje — `glob()` i `rglob()` — zwracają generator, który tworzy nazwy plików poprzez iterację.

9.10. Funkcja print()

Aby wydrukować serię wartości oddzielonych spacjami, podaj je wszystkie do funkcji `print()` w następujący sposób:

```
print('Wartościami są', x, y, z)
```

Aby pominąć lub zmienić zakończenie wiersza, użyj argumentu słowa kluczowego `end`:

```
# Pomiń znak nowej linii
print('Wartościami są', x, y, z, end='')
```

Aby przekierować dane wyjściowe do pliku, użyj argumentu słowa kluczowego `file`:

```
# Przekieruj do obiektu pliku f
print('Wartościami są', x, y, z, file=f)
```

Aby zmienić znak separatora między elementami, użyj argumentu słowa kluczowego `sep`:

```
# Umieść przecinki między wartościami
print('Wartościami są', x, y, z, sep=',')
```

9.11. Generowanie wyjścia

Najbardziej znana programistom jest bezpośrednia praca z plikami. Jednak funkcje generatora mogą być również używane do emitowania strumienia wejścia-wyjścia jako sekwencji fragmentów danych. Aby to zrobić, użyj instrukcji `yield` tak samo jak funkcji `write()` lub `print()`. Oto przykład:

```
def countdown(n):
    while n > 0:
        yield f'T-minus {n}\n'
        n -= 1
    yield 'Kaboom!\n'
```

Tworzenie strumienia wyjściowego w ten sposób zapewnia elastyczność, ponieważ jest on oddzielony od kodu, który faktycznie kieruje strumień do miejsca docelowego. Jeśli na przykład chcesz skierować powyższe dane wyjściowe do pliku `f`, możesz to zrobić:

```
lines = countdown(5)
f.writelines(lines)
```

Jeśli zamiast tego chcesz przekierować wyjście przez gniazdo `s`, możesz to zrobić w następujący sposób:

```
for chunk in lines:
    s.sendall(chunk.encode('utf-8'))
```

A jeśli chcesz po prostu przechwycić wszystkie dane wyjściowe w jednym ciągu, możesz to zrobić tak:

```
out = ''.join(lines)
```

Bardziej zaawansowane aplikacje mogą wykorzystać to podejście do implementacji własnego buforowania wejścia-wyjścia. Na przykład generator może emitować małe fragmenty tekstu, ale inna funkcja zbierałaby następnie fragmenty do większych buforów, aby utworzyć bardziej wydajną pojedynczą operację wejścia-wyjścia.

```
chunks = []
buffered_size = 0
for chunk in count:
    chunks.append(chunk)
    buffered_size += len(chunk)
    if buffered_size >= MAXBUFFERSIZE:
        outf.write(''.join(chunks))
        chunks.clear()
        buffered_size = 0
outf.write(''.join(chunks))
```

W przypadku programów, które kierują dane wyjściowe do plików lub połączeń sieciowych, podejście generatora może również skutkować znacznym zmniejszeniem użycia pamięci, ponieważ cały strumień wyjściowy może często być generowany i przetwarzany w małych fragmentach, w przeciwieństwie do podejścia, w którym dane są najpierw zbierane w jeden duży ciąg znaków lub listę ciągów.

9.12. Pobieranie danych wejściowych

W przypadku programów, które pobierają fragmentaryczne dane wejściowe, ulepszone generatory mogą być przydatne do dekodowania protokołów i innych rodzajów operacji wejścia-wyjścia. Oto przykład ulepszonego generatora, który odbiera fragmenty bajtów i składa je w linie:

```
def line_receiver():
    data = bytearray()
    line = None
    linecount = 0
    while True:
        part = yield line
        linecount += part.count(b'\n')
        data.extend(part)
        if linecount > 0:
            index = data.index(b'\n')
            line = bytes(data[:index+1])
            data = data[index+1:]
            linecount -= 1
        else:
            line = None
```

W tym przykładzie generator został zaprogramowany do odbierania fragmentów bajtów, które są gromadzone w tablicy bajtów. Jeśli tablica zawiera znak nowej linii, wiersz jest wyodrębniany i zwracany.

W przeciwnym razie zwracana jest wartość `None`. Oto przykład ilustrujący, jak to działa:

```
>>> r = line_receiver()
>>> r.send(None)       # Niezbędny pierwszy krok
>>> r.send(b'hello')
>>> r.send(b'world\nit ')
b'hello world\n'
>>> r.send(b'works!')
>>> r.send(b'\n')
b'it works!\n''
>>>
```

Interesującym efektem ubocznym tego podejścia jest to, że udostępnia ono na zewnątrz rzeczywiste operacje wejścia-wyjścia, które należy wykonać, aby uzyskać dane wejściowe. Implementacja `line_receiver()` nie zawiera żadnych operacji wejścia-wyjścia! Oznacza to, że może być używana w różnych kontekstach. Na przykład z gniazdami:

```
r = line_receiver()
data = None
while True:
    while not (line:=r.send(data)):
        data = sock.recv(8192)

    # Przetwarzanie linii
    ...
```

lub z plikami:

```
r = line_receiver()
data = None
while True:
    while not (line:=r.send(data)):
        data = file.read(10000)

    # Przetwarzanie linii
    ...
```

albo nawet w kodzie asynchronicznym:

```
async def reader(ch):
    r = line_receiver()
    data = None
    while True:
        while not (line:=r.send(data)):
            data = await ch.receive(8192)

        # Przetwarzanie linii
        ...
```

9.13. Serializacja obiektów

Czasami konieczna jest serializacja reprezentacji obiektu, aby mogła być przesyłana przez sieć, zapisywana do pliku lub przechowywana w bazie danych. Jednym ze sposobów jest przekonwertowanie danych na standardowe kodowanie, takie jak JSON lub XML. Istnieje również wspólny format serializacji danych, specyficzny dla Pythona, o nazwie *Pickle*.

Moduł `pickle` serializuje obiekt w strumień bajtów, który można wykorzystać do zrekonstruowania obiektu w późniejszym czasie. Interfejs `pickle` jest prosty i składa się z dwóch operacji: `dump()` i `load()`. Na przykład poniższy kod zapisuje obiekt do pliku:

```
import pickle
obj = SomeObject()
with open(filename, 'wb') as file:
    pickle.dump(obj, file)          # Zapisz obiekt do pliku
```

Aby przywrócić obiekt, użyj:

```
with open(filename, 'rb') as file:
    obj = pickle.load(file)         # Przywróć obiekt
```

Format danych używany przez `pickle` zawiera własne dzielenie rekordów. W ten sposób sekwencję obiektów można zapisać, wykonując serię operacji `dump()`, jedna po drugiej. Aby przywrócić te obiekty, po prostu użyj podobnej sekwencji operacji `load()`.

W przypadku programowania sieciowego moduł `pickle` jest zazwyczaj używany do tworzenia wiadomości zakodowanych bajtowo. Aby to zrobić, użyj funkcji `dumps()` i `load()`. Zamiast odczytywać/zapisywać dane do pliku, te funkcje działają z ciągami bajtów.

```
obj = SomeObject()

# Zamień obiekt w bajty
data = pickle.dumps(obj)
...

# Zamień bajty z powrotem w obiekt
obj = pickle.loads(data)
```

Normalnie nie jest konieczne, aby obiekty zdefiniowane przez użytkownika robiły cokolwiek dodatkowego w celu pracy z `pickle`, jednak niektórych rodzajów obiektów nie można w ten sposób serializować. Zwykle są to obiekty zawierające stan środowiska uruchomieniowego: otwarte pliki, wątki, zamknięcia, generatory i tak dalej. Aby obsłużyć te trudne przypadki, klasa może zdefiniować metody specjalne `__getstate__()` i `__setstate__()`.

Metoda `__getstate__()`, jeśli jest zdefiniowana, zostanie wywołana w celu utworzenia wartości reprezentującej stan obiektu. Wartość zwracana przez `__getstate__()` jest zazwyczaj ciągiem, krotką, listą lub słownikiem. Metoda `__setstate__()` otrzymuje tę wartość podczas deserializacji i powinna z niej odtworzyć stan obiektu.

Podczas kodowania obiektu `pickle` nie zawiera kodu źródłowego. Zamiast tego koduje odwołanie do nazwy do klasy definiującej. Podczas deserializacji ta nazwa jest używana do wyszukiwania kodu źródłowego w systemie. Aby serializacja zadziałała, odbiorca musi mieć już zainstalowany odpowiedni kod źródłowy. Należy również podkreślić, że serializacja jest

z natury niepewna — deserializacja niezaufanych danych jest znanym przypadkiem zdalnego wykonywania kodu. Dlatego moduł `pickle` powinien być używany tylko wtedy, gdy można całkowicie zabezpieczyć środowisko uruchomieniowe.

9.14. Operacje blokujące i współbieżność

Podstawowym aspektem operacji wejścia-wyjścia jest koncepcja *blokowania*. Ze swej natury interfejs wejścia-wyjścia jest połączony ze światem rzeczywistym. Często wiąże się to z oczekiwaniem na gotowość danych wejściowych lub urządzeń. Na przykład kod, który odczytuje dane w sieci, może wykonać operację odbierania na gnieździe w taki sposób:

```
data = sock.recv(8192)
```

Po wykonaniu tej instrukcji może ona natychmiast zwrócić wynik, jeśli dane są dostępne. Jeśli jednak tak nie jest — czeka na nadejście danych. To blokuje program. Gdy program jest zablokowany, nic się nie dzieje.

W przypadku skryptu do analizy danych lub prostego programu blokowanie nie jest czymś, o co się martwisz. Jeśli jednak chcesz, aby Twój program robił coś innego, gdy operacja jest zablokowana, musisz zastosować inne podejście. To jest podstawowy problem współbieżności — działanie programu wykonującego więcej niż jedną operację naraz. Jednym z powszechnych problemów jest odczytywanie programu z dwóch lub większej liczby różnych gniazd sieciowych w tym samym czasie:

```
def reader1(sock):
    while (data := sock.recv(8192)):
        print('reader1 otrzymał:', data)

def reader2(sock):
    while (data := sock.recv(8192)):
        print('reader2 otrzymał:', data)

# Problem: co zrobić, aby reader1() i reader2()
# działały w tym samym czasie?
```

W pozostałej części tej sekcji przedstawiono kilka różnych podejść do rozwiązania tego problemu. Nie jest to jednak pełny samouczek dotyczący współbieżności. W tym celu musisz się zapoznać z innymi zasobami.

9.14.1. Nieblokujące operacje wejścia-wyjścia

Jednym ze sposobów uniknięcia blokowania jest użycie tak zwanych *nieblokujących* operacji wejścia-wyjścia. Jest to specjalny tryb, który należy włączyć — na przykład na gnieździe:

```
sock.setblocking(False)
```

Po włączeniu wyjątek zostanie zgłoszony, jeśli operacja zostałaby zablokowana. Na przykład:

```
try:
    data = sock.recv(8192)
except BlockingIOError as e:
```

```
    # Brak dostępnych danych
    ...
```

W odpowiedzi na wyjątek `BlockingIOError` program może wybrać pracę nad czymś innym. Może później ponowić operację wejścia-wyjścia, aby sprawdzić, czy dotarły jakieś dane. Oto przykład, jak możesz czytać na dwóch gniazdach jednocześnie:

```
def reader1(sock):
    try:
        data = sock.recv(8192)
        print('reader1 otrzymał:', data)
    except BlockingIOError:
        pass

def reader2(sock):
    try:
        data = sock.recv(8192)
        print('reader2 otrzymał:', data)
    except BlockingIOError:
        pass

def run(sock1, sock2):
    sock1.setblocking(False)
    sock2.setblocking(False)
    while True:
        reader1(sock1)
        reader2(sock2)
```

W praktyce poleganie tylko na nieblokujących operacjach wejścia-wyjścia jest niezgrabne i nieefektywne. Funkcja `run()` będzie działać w nieefektywnej pętli, ponieważ nieustannie próbuje czytać na gniazdach. To działa, ale nie jest to dobry projekt.

9.14.2. Odpytywanie operacji wejścia-wyjścia

Zamiast polegać na wyjątkach i czytaniu w pętli, możliwe jest odpytywanie kanałów wejścia-wyjścia (ang. *I/O polling*), aby sprawdzić, czy dane są dostępne. W tym celu można użyć modułu `select` lub `selectors`. Oto nieco zmodyfikowana wersja funkcji `run()`:

```
from selectors import DefaultSelector, EVENT_READ, EVENT_WRITE

def run(sock1, sock2):
    selector = DefaultSelector()
    selector.register(sock1, EVENT_READ, data=reader1)
    selector.register(sock2, EVENT_READ, data=reader2)
    # Czekaj, aż coś się wydarzy
    while True:
        for key, evt in selector.select():
            func = key.data
            func(key.fileobj)
```

W tym kodzie pętla uruchamia funkcję `reader1()` lub `reader2()` jako wywołanie zwrotne po wykryciu danych operacji wejścia-wyjścia w odpowiednim gnieździe. Sama operacja

`selector.select()` blokuje się, czekając na wystąpienie tych danych. Tak więc w przeciwieństwie do poprzedniego przykładu nie spowoduje to gwałtownego wykorzystania procesora.

Takie podejście do operacji wejścia-wyjścia jest podstawą wielu tak zwanych frameworków „async", takich jak `asyncio`, chociaż zwykle nie widać wewnętrznego działania pętli zdarzeń.

9.14.3. Wątki

W ostatnich dwóch przykładach współbieżność wymagała użycia specjalnej funkcji `run()` do kierowania obliczeniami. Alternatywnie można skorzystać z programowania wielowątkowego i modułu `threading`. Pomyśl o wątku jako o niezależnym zadaniu, które działa w Twoim programie. Oto przykład kodu, który odczytuje dane z dwóch gniazd jednocześnie:

```
import threading

def reader1(sock):
    while (data := sock.recv(8192)):
        print('reader1 otrzymał:', data)

def reader2(sock):
    while (data := sock.recv(8192)):
        print('reader2 otrzymał:', data)

t1 = threading.Thread(target=reader1, args=[sock1])
t2 = threading.Thread(target=reader2, args=[sock2])

# Rozpocznij wątki
t1.start()
t2.start()

# Poczekaj, aż wątki się zakończą
t1.join()
t2.join()
```

W tym programie funkcje `reader1()` i `reader2()` są wykonywane jednocześnie. Jest to zarządzane przez system operacyjny hosta, więc nie musisz dużo wiedzieć o tym, jak to działa. Jeśli operacja blokowania występuje w jednym wątku, nie wpływa to na drugi wątek.

Temat programowania wątków w całości wykracza poza zakres tej książki. Jednak kilka dodatkowych przykładów znajduje się w sekcji dotyczącej modułu `threading` w dalszej części tego rozdziału.

9.14.4. Równoczesne wykonywanie z asyncio

Moduł `asyncio` zapewnia implementację współbieżności alternatywną dla wątków. Wewnętrznie opiera się na pętli zdarzeń, która wykorzystuje odpytywanie operacji wejścia-wyjścia. Jednak model programowania wysokiego poziomu wygląda bardzo podobnie do wątków dzięki zastosowaniu specjalnych funkcji asynchronicznych. Oto przykład:

```
import asyncio

async def reader1(sock):
```

```
    loop = asyncio.get_event_loop()
    while (data := await loop.sock_recv(sock, 8192)):
        print('reader1 otrzymał:', data)

async def reader2(sock):
    loop = asyncio.get_event_loop()
    while (data := await loop.sock_recv(sock, 8192)):
        print('reader2 otrzymał:', data)

async def main(sock1, sock2):
    loop = asyncio.get_event_loop()
    t1 = loop.create_task(reader1(sock1))
    t2 = loop.create_task(reader2(sock2))

    # Poczekaj na zakończenie zadań
    await t1
    await t2

...
# Uruchom
asyncio.run(main(sock1, sock2))
```

Pełne szczegóły korzystania z `asyncio` wymagałyby osobnej, poświęconej tylko im książki. Powinieneś wiedzieć, że wiele bibliotek i frameworków oferuje wsparcie dla obsługi operacji asynchronicznych. Zwykle oznacza to, że współbieżne wykonywanie jest obsługiwane przez `asyncio` lub podobny moduł. Znaczna część kodu prawdopodobnie będzie obejmować funkcje `async` i powiązane funkcje.

9.15. Standardowe moduły biblioteczne

Wiele standardowych modułów bibliotecznych jest używanych do różnych zadań związanych z operacjami wejścia-wyjścia. Ta sekcja zawiera krótki przegląd najczęściej używanych modułów wraz z kilkoma przykładami. Kompletny materiał referencyjny można znaleźć online lub w IDE i nie jest tu powtarzany. Głównym celem tej sekcji jest wskazanie Ci właściwego kierunku poprzez podanie nazw modułów, których powinieneś używać, oraz kilku przykładów bardzo typowych zadań programistycznych dotyczących każdego modułu.

Wiele przykładów pokazano jako interaktywne sesje Pythona. Są to eksperymenty — mają za zadanie zachęcić Cię do samodzielnego wypróbowania.

9.15.1. Moduł asyncio

Moduł `asyncio` zapewnia obsługę współbieżnych operacji wejścia-wyjścia przy użyciu odpytywania operacji wejścia-wyjścia i podstawowej pętli zdarzeń. Jego głównym zastosowaniem jest kod obejmujący zagadnienia sieciowe i systemy rozproszone. Oto przykład serwera echa TCP używającego gniazd niskiego poziomu:

```
import asyncio
from socket import *

async def echo_server(address):
```

```
    loop = asyncio.get_event_loop()
    sock = socket(AF_INET, SOCK_STREAM)
    sock.setsockopt(SOL_SOCKET, SO_REUSEADDR, 1)
    sock.bind(address)
    sock.listen(5)
    sock.setblocking(False)
    print('Serwer nasłuchuje na adresie', address)
    with sock:
        while True:
            client, addr = await loop.sock_accept(sock)
            print('Połączenie z', addr)
            loop.create_task(echo_client(loop, client))

async def echo_client(loop, client):
    with client:
        while True:
            data = await loop.sock_recv(client, 10000)
            if not data:
                break
            await loop.sock_sendall(client, b'Got:' + data)
    print('Połączenie zamknięte')

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.create_task(echo_server(('', 25000)))
    loop.run_forever()
```

Aby przetestować powyższy kod, użyj programu takiego jak *nc* lub *telnet* w celu połączenia się z portem 25000 na swoim komputerze. Kod powinien wywołać funkcję `echo` wpisywanego tekstu. Jeśli połączysz się więcej niż raz przy użyciu wielu okien terminali, przekonasz się, że kod może obsługiwać wszystkie połączenia jednocześnie.

Większość aplikacji używających `asyncio` prawdopodobnie będzie działać na wyższym poziomie niż gniazda. Jednak w takich aplikacjach nadal będziesz musiał korzystać ze specjalnych funkcji `async` i w jakiś sposób wchodzić w interakcję z podstawową pętlą zdarzeń.

9.15.2. Moduł binascii

Moduł `binascii` posiada funkcje do konwertowania danych binarnych na różne reprezentacje tekstowe, takie jak kod szesnastkowy i base64. Na przykład:

```
>>> binascii.b2a_hex(b'hello')
b'68656c6c6f'
>>> binascii.a2b_hex(_)
b'hello'
>>> binascii.b2a_base64(b'hello')
b'aGVsbG8=\n'
>>> binascii.a2b_base64(_)
b'hello'
>>>
```

Podobną funkcjonalność można znaleźć w module `base64` oraz w metodach `hex()` i `fromhex()` dla bajtów. Na przykład:

```
>>> a = b'hello'
>>> a.hex()
'68656c6c6f'
>>> bytes.fromhex(_)
b'hello'
>>> import base64
>>> base64.b64encode(a)
b'aGVsbG8='
>>>
```

9.15.3. Moduł cgi

Załóżmy, że chcesz po prostu umieścić podstawowy formularz na swojej stronie internetowej. Być może jest to formularz zapisu na cotygodniowy biuletyn „Koty i kategorie”. Oczywiście, możesz zainstalować najnowszą platformę internetową i spędzać cały czas na majstrowaniu przy niej. Możesz też jednak po prostu napisać podstawowy skrypt CGI — w starym stylu. Do tego służy moduł `cgi`.

Załóżmy, że na stronie internetowej znajduje się następujący fragment formularza:

```
<form method="POST" action="cgi-bin/register.py">
    <p>
    Aby się zarejestrować, proszę podać imię oraz adres e-mail.
    </p>
    <div>
        <input name="name" type="text">Twoje imię:</input>
    </div>
    <div>
        <input name="email" type="email">Twój e-mail:</input>
    </div>
    <div class="modal-footer justify-content-center">
        <input type="submit" name= "submit" value="Zarejestruj"></input>
    </div>
</form>
```

Oto skrypt CGI, który odbiera dane z formularza po drugiej stronie:

```
#!/usr/bin/env python
import cgi
try:
    form = cgi.FieldStorage()
    name = form.getvalue('name')
    email = form.getvalue('email')
    # Waliduj odpowiedzi i rób cokolwiek
    ...
    # Zwróć wynik HTML (lub przekieruj)
    print("Status: 302 Moved\r")
    print("Location: https://www.mywebsite.com/thanks.html\r")
    print("\r")
except Exception as e:
    print("Status: 501 Error\r")
    print("Content-type: text/plain\r")
    print("\r")
    print("Wystąpił jakiś błąd.\r")
```

Czy napisanie takiego skryptu CGI da Ci pracę w internetowym startupie? Prawdopodobnie nie. Czy rozwiąże Twój rzeczywisty problem? Prawdopodobnie tak.

9.15.4. Moduł configparser

Pliki INI są powszechnym formatem kodowania informacji na temat konfiguracji programu w formie czytelnej dla człowieka. Oto przykład:

```
# config.ini

; Komentarz
[section1]
name1 = value1
name2 = value2

[section2]
; Alternatywna składnia
name1: value1
name2: value2
```

Moduł `configparser` służy do odczytywania plików *.ini* i wyodrębniania wartości. Oto podstawowy przykład:

```
import configparser

# Utwórz parser konfiguracji i odczytaj plik
cfg = configparser.ConfigParser()
cfg.read('config.ini')

# Wyodrębnij wartości
a = cfg.get('section1', 'name1')
b = cfg.get('section2', 'name2')
...
```

Dostępna jest również bardziej zaawansowana funkcjonalność, w tym funkcje interpolacji ciągów, możliwość łączenia wielu plików *.ini*, dostarczania wartości domyślnych i nie tylko. Więcej przykładów znajdziesz w oficjalnej dokumentacji.

9.15.5. Moduł csv

Moduł `csv` służy do odczytu/zapisu plików o wartościach oddzielonych przecinkami (CSV), wytwarzanych przez programy takie jak Microsoft Excel lub eksportowanych z bazy danych. Aby z niego skorzystać, otwórz plik, a następnie otocz go dodatkową warstwą kodowania/dekodowania CSV. Na przykład:

```
import csv

# Wczytaj plik CSV do listy krotek
def read_csv_data(filename):
    with open(filename, newline='') as file:
        rows = csv.reader(file)
        # Pierwsza linia to często nagłówek
```

```
        headers = next(rows)
        # Teraz przeczytaj resztę danych
        for row in rows:
            # Zrób coś z wierszem
            ...

# Zapisz dane Pythona do pliku CSV
def write_csv_data(filename, headers, rows):
    with open(filename, 'w', newline='') as file:
    out = csv.writer(file)
    out.writerow(headers)
    out.writerows(rows)
```

Często używanym udogodnieniem jest zastosowanie funkcji `DictReader()`. Interpretuje ona pierwszy wiersz pliku CSV jako nagłówki i zwraca każdy wiersz jako słownik — zamiast krotki.

```
Import csv

def find_nearby(filename):
    with open(filename, newline='') as file:
        rows = csv.DictReader(file)
        for row in rows:
            lat = float(rows['latitude'])
            lon = float(rows['longitude'])
            if close_enough(lat, lon):
                print(row)
```

Moduł `csv` niewiele robi z danymi CSV poza ich odczytywaniem lub zapisywaniem. Główną oferowaną korzyścią jest to, że moduł wie, jak prawidłowo kodować/dekodować dane, i obsługuje wiele przypadków brzegowych obejmujących cytaty, znaki specjalne i inne szczegóły.

Jest to moduł, którego możesz użyć do pisania prostych skryptów do czyszczenia lub przygotowywania danych do użycia z innymi programami. Jeśli chcesz wykonywać zadania analizy danych na danych CSV, rozważ użycie pakietu innej firmy, takiego jak popularna biblioteka `pandas`.

9.15.6. Moduł errno

Za każdym razem, gdy wystąpi błąd na poziomie systemu, Python zgłasza go jako wyjątek, który jest podklasą `OSError`. Niektóre z bardziej powszechnych rodzajów błędów systemowych są reprezentowane przez oddzielne podklasy `OSError`, takie jak `PermissionError` lub `FileNotFoundError`. W praktyce mogą jednak wystąpić setki innych błędów. W tym przypadku każdy wyjątek `OSError` zawiera numeryczny atrybut `errno`, który można sprawdzić. Moduł `errno` zapewnia stałe symboliczne odpowiadające tym kodom błędów. Są często używane podczas pisania wyspecjalizowanych programów obsługi wyjątków. Oto przykładowa procedura obsługi wyjątków, która sprawdza, czy na urządzeniu jest wolne miejsce:

```
import errno

def write_data(file, data):
    try:
        file.write(data)
```

```
    except OSError as e:
        if e.errno == errno.ENOSPC:
            print("Skończyło się miejsce na dysku!")
        else:
            raise # Jakiś inny błąd
```

9.15.7. Moduł fcntl

Moduł `fcntl` jest używany do wykonywania niskopoziomowych operacji sterowania operacji wejścia-wyjścia w systemie UNIX przy użyciu wywołań systemowych `fcntl()` i `ioctl()`. Jest to również moduł, którego należy użyć, jeśli chcesz wykonać jakiekolwiek blokowanie plików — problem, który czasami pojawia się w kontekście współbieżności i systemów rozproszonych. Oto przykład otwierania pliku w połączeniu z algorytmem wzajemnego wykluczania we wszystkich procesach za pomocą `fcntl.flock()`:

```
import fcntl

with open("somefile", "r") as file:
    try:
        fcntl.flock(file.fileno(), fcntl.LOCK_EX)
        # Użyj pliku
        ...
    finally:
        fcntl.flock(file.fileno(), fcntl.LOCK_UN)
```

9.15.8. Moduł hashlib

Moduł `hashlib` udostępnia funkcje do obliczania kryptograficznych wartości skrótu, takich jak MD5 i SHA-1. Poniższy przykład ilustruje sposób korzystania z modułu:

```
>>> h = hashlib.new('sha256')
>>> h.update(b'Hello') # Dane kanału
>>> h.update(b'World')
>>> h.digest()
b'\xa5\x91\xa6\xd4\x0b\xf4 @J\x01\x173\xcf\xb7\xb1\x90\xd6,e\xbf\x0b\xcd
\xa3+W\xb2w\xd9\xad\x9f\x14n'
>>> h.hexdigest()
'a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e'
>>> h.digest_size
32
>>>
```

9.15.9. Pakiet http

Pakiet `http` zawiera dużą ilość kodu związanego z niskopoziomową implementacją protokołu internetowego http. Pakiet może służyć do implementacji zarówno serwerów, jak i klientów. Jednak większość tego pakietu jest uważana za przestarzałą i zbyt niskopoziomową do codziennej pracy. Poważni programiści pracujący z HTTP częściej korzystają z bibliotek innych firm, takich jak `requests`, `httpx`, `Django`, `flask` i inne.

Niemniej jedną z użytecznych funkcji pakietu `http` jest możliwość uruchomienia przez Python samodzielnego serwera WWW. Przejdź do katalogu z kolekcją plików i wpisz:

```
files and type the following:
bash $ python -m http.server
```

Teraz Python będzie wyświetlać pliki w Twojej przeglądarce, jeśli wskażesz odpowiedni port. Nie wykorzystasz tego do uruchomienia strony internetowej, ale może Ci się to przydać do testowania i debugowania programów związanych z siecią. Na przykład można skorzystać z tego rozwiązania do lokalnego testowania programów stworzonych w technologiach HTML, JavaScript i WebAssembly.

9.15.10. Moduł io

Moduł `io` zawiera przede wszystkim definicje klas używanych do implementacji obiektów plików zwracanych przez funkcję `open()`. Bezpośredni dostęp do tych klas nie jest zbyt powszechny. Moduł zawiera jednak również parę klas, które są przydatne do przedstawiania pliku w postaci ciągów znaków i bajtów. Może to być pomocne w przypadku testowania oraz dla innych aplikacji, w których musisz dostarczyć „plik", ale uzyskałeś dane w inny sposób.

Klasa `StringIO()` zapewnia interfejs podobny do pliku, ale operujący na ciągach znaków. Oto jak możesz zapisać dane wyjściowe do ciągu znaków:

```
# Funkcja, która oczekuje pliku
def greeting(file):
    file.write('Hello\n')
    file.write('World\n')

# Wywołaj funkcję, używając prawdziwego pliku
with open('out.txt', 'w') as file:
    greeting(file)

# Wywołaj funkcję z "fałszywym" plikiem
import io
file = io.StringIO()
greeting(file)
# Uzyskaj wynik
output = file.getvalue()
```

Podobnie możesz stworzyć obiekt `StringIO` i użyć go do czytania danych:

```
file = io.StringIO('hello\nworld\n')
while (line := file.readline()):
    print(line, end='')
```

Klasa `BytesIO()` służy do podobnych celów — do emulowania binarnych operacji wejścia/wyjścia z użyciem bajtów.

9.15.11. Moduł json

Moduł `json` może służyć do kodowania i dekodowania danych w formacie JSON, powszechnie używanych w API mikroserwisów i aplikacji internetowych. Istnieją dwie podstawowe funkcje

konwersji danych: `dumps()` i `load()`. `dumps()` pobiera słownik Pythona i koduje go jako ciąg znaków w formacie JSON Unicode:

```
>>> import json
>>> data = {'name: 'Mary A. Python', 'email': 'mary123@python.org'}
>>> s = json.dumps(data)
>>> s
'{"name": "Mary A. Python", "email": "mary123@python.org"}'
>>>
```

Funkcja `load()` idzie w innym kierunku:

```
>>> d = json.loads(s)
>>> d == data
True
>>>
```

Zarówno funkcje `dumps()`, jak i `load()` mają wiele opcji do kontrolowania aspektów konwersji, a także komunikowania się z instancjami klas Pythona. Temat ten wykracza poza zakres tej sekcji, ale wiele informacji można znaleźć w oficjalnej dokumentacji.

9.15.12. Moduł logging

Moduł `logging` jest de facto standardowym modułem używanym do raportowania diagnostyki programu i debugowania w postaci wyświetlanych komunikatów. Może służyć do kierowania danych wyjściowych do pliku dziennika i zapewnia dużą liczbę opcji konfiguracyjnych. Powszechną praktyką jest pisanie kodu, który tworzy instancję `Logger` i wysyła na niej komunikaty w następujący sposób:

```
import logging
log = logging.getLogger(__name__)

# Funkcja korzystająca z logowania
def func(args):
    log.debug("Komunikat na potrzeby debugowania")
    log.info("Informacja")
    log.warning("Ostrzeżenie")
    log.error("Błąd")
    log.critical("Krytyczna wiadomość")
# Konfiguracja logowania (występuje podczas uruchamiania programu)
if __name__ == '__main__':
    logging.basicConfig(
        level=logging.WARNING,
        filename="output.log"
    )
```

Istnieje pięć wbudowanych poziomów logowania uporządkowanych według rosnącej istotności. Podczas konfigurowania systemu logowania określasz poziom, który działa jak filtr. Zgłaszane są tylko wiadomości o podanym lub wyższym poziomie ważności. Logowanie zapewnia dużą liczbę opcji konfiguracyjnych, głównie związanych z obsługą komunikatów w backendzie. Zwykle nie musisz o tym wiedzieć podczas pisania kodu aplikacji — używasz `debug()`, `info()`,

`warning()` i podobnych metod na danej instancji `Logger`. Podczas uruchamiania programu w specjalnej lokalizacji (takiej jak funkcja `main()` lub główny blok kodu), konfiguracja jest inna.

9.15.13. Moduł os

Moduł `os` zapewnia przenośny interfejs dla typowych funkcji systemu operacyjnego, zwykle związanych ze środowiskiem procesu, plikami, katalogami, uprawnieniami i tak dalej. Interfejs programistyczny ściśle odpowiada programowaniu w C i standardom takim jak POSIX.

Praktycznie rzecz biorąc, większość tego modułu jest prawdopodobnie zbyt niskopoziomowa, aby można go było bezpośrednio wykorzystać w typowej aplikacji. Jeśli jednak kiedykolwiek napotkasz problem z wykonaniem jakiejś niejasnej operacji systemu niskiego poziomu (takiej jak otwieranie TTY), istnieje duża szansa, że znajdziesz tutaj odpowiednią dla niego funkcjonalność.

9.15.14. Moduł os.path

Moduł `os.path` jest starszym modułem do manipulowania nazwami ścieżek i wykonywania typowych operacji na systemie plików. Jego funkcjonalność została w dużej mierze zastąpiona przez nowszy moduł `pathlib`, ale ponieważ jego użycie jest nadal rozpowszechnione, będziesz go widzieć w wielu kodach.

Jednym z podstawowych problemów rozwiązywanych przez ten moduł jest przenośna obsługa separatorów ścieżek w systemach UNIX (ang. *forward-slash*) i Windows (ang. *backslash*). Funkcje takie jak `os.path.join()` i `os.path.split()` są często używane do rozdzielania ścieżek plików i łączenia ich z powrotem:

```
>>> filename = '/Users/beazley/Desktop/old/data.csv'
>>> os.path.split()
('/Users/beazley/Desktop/old', 'data.csv')
>>> os.path.join('/Users/beazley/Desktop', 'out.txt')
'/Users/beazley/Desktop/out.txt'
>>>
```

Oto przykład kodu korzystającego z tych funkcji:

```
import os.path

def clean_line(line):
    # Ustaw linię (cokolwiek)
    return line.strip().upper() + '\n'

def clean_data(filename):
    dirname, basename = os.path.split()
    newname = os.path.join(dirname, basename+'.clean')
    with open(newname, 'w') as out_f:
        with open(filename, 'r') as in_f:
            for line in in_f:
                out_f.write(clean_line(line))
```

Moduł `os.path` zawiera również szereg funkcji, takich jak `isfile()`, `isdir()` i `getsize()`, służących do wykonywania testów systemu plików i pobierania metadanych plików. Na przykład ta funkcja zwraca całkowity rozmiar w bajtach prostego pliku lub wszystkich plików w katalogu:

```
import os.path

def compute_usage(filename):
    if os.path.isfile(filename):
        return os.path.getsize(filename)
    elif os.path.isdir(filename):
        return sum(compute_usage(os.path.join(filename, name))
                   for name in os.listdir(filename))
    else:
        raise RuntimeError('Nieobsługiwany rodzaj pliku')
```

9.15.15. Moduł pathlib

Moduł `pathlib` to nowoczesny sposób manipulowania nazwami ścieżek w sposób przenośny i wysokopoziomowy. Łączy wiele funkcji zorientowanych na pliki w jednym miejscu i wykorzystuje interfejs zorientowany obiektowo. Podstawowym obiektem jest klasa `Path`. Na przykład:

```
from pathlib import Path
filename = Path('/Users/beazley/old/data.csv')
```

Gdy masz już instancję klasy `Path` — `filename`, możesz wykonywać na niej różne operacje, aby manipulować nazwą pliku. Na przykład:

```
>>> filename.name
'data.csv'
>>> filename.parent
Path('/Users/beazley/old')
>>> filename.parent / 'newfile.csv'
Path('/Users/beazley/old/newfile.csv')
>>> filename.parts
('/', 'Users', 'beazley', 'old', 'data.csv')
>>> filename.with_suffix('.csv.clean')
Path('/Users/beazley/old/data.csv.clean')
>>>
```

Instancje `Path` mają również funkcje do uzyskiwania metadanych plików, list katalogów i innych podobnych funkcji. Oto reimplementacja funkcji `compute_usage()` z poprzedniej sekcji:

```
import pathlib

def compute_usage(filename):
    pathname = pathlib.Path(filename)
    if pathname.is_file():
        return pathname.stat().st_size
    elif pathname.is_dir():
        return sum(path.stat().st_size
                   for path in pathname.rglob('*')
                   if path.is_file())
        return pathname.stat().st_size
    else:
        raise RuntimeError('Nieobsługiwany rodzaj pliku')
```

9.15.16. Moduł re

Moduł re służy do wykonywania operacji dopasowywania, wyszukiwania i zamiany tekstu przy użyciu wyrażeń regularnych. Oto prosty przykład:

```
>>> text = 'Today is 3/27/2018. Tomorrow is 3/28/2018.'
>>> # Znajdź wszystkie wystąpienia daty
>>> import re
>>> re.findall(r'\d+/\d+/\d+', text)
['3/27/2018', '3/28/2018']
>>> # Zastąp wszystkie wystąpienia daty tekstem zastępczym
>>> re.sub(r'(\d+)/(\d+)/(\d+)', r'\3-\1-\2', text)
'Today is 2018-3-27. Tomorrow is 2018-3-28.'
>>>
```

Wyrażenia regularne są często znane ze swojej nieodgadnionej składni. W tym przykładzie \d+ jest interpretowane jako „jedna lub więcej cyfr". Więcej informacji na temat składni wzorca można znaleźć w oficjalnej dokumentacji modułu re.

9.15.17. Moduł shutil

Moduł shutil służy do wykonywania niektórych typowych zadań, które w innym przypadku mógłbyś wykonać w powłoce. Obejmują one kopiowanie i usuwanie plików, pracę z archiwami i tak dalej. Aby na przykład skopiować plik, wystarczy napisać:

```
import shutil

shutil.copy(srcfile, dstfile)
```

Aby przenieść plik, wpisz:

```
shutil.move(srcfile, dstfile)
```

Aby skopiować drzewo katalogów, zastosuj:

```
shutil.copytree(srcdir, dstdir)
```

Aby usunąć drzewo katalogów, użyj:

```
shutil.rmtree(pathname)
```

Moduł shutil jest często używany jako bezpieczniejsza i bardziej przenośna alternatywa dla bezpośredniego wykonywania poleceń powłoki za pomocą funkcji os.system().

9.15.18. Moduł select

Moduł select jest używany do prostego odpytywania wielu strumieni wejścia-wyjścia. Oznacza to, że może być wykorzystywany do obserwowania kolekcji deskryptorów plików dla danych przychodzących lub możliwości odbierania danych wychodzących. Poniższy przykład pokazuje typowe zastosowanie:

```
import select
```

```
# Kolekcje obiektów reprezentujących deskryptory plików. Muszą być liczbami całkowitymi lub obiektami z metodą fileno().
want_to_read = [ ... ]
want_to_write = [ ... ]
check_exceptions = [ ... ]

# Timeout (lub None)
timeout = None

# Odpytywanie I/O
can_read, can_write, have_exceptions = \
    select.select(want_to_read, want_to_write, check_exceptions, timeout)

# Wykonywanie operacji wejścia-wyjścia
for file in can_read:
    do_read(file)
for file in can_write:
    do_write(file)

# Obsługa wyjątków
for file in have_exceptions:
    handle_exception(file)
```

W tym kodzie konstruowane są trzy zestawy deskryptorów plików. Te zestawy odpowiadają czytaniu, zapisowi i wyjątkom. Są one przekazywane do `select()` wraz z opcjonalnym opóźnieniem (ang. *timeout*). `select()` zwraca trzy podzbiory przekazanych argumentów. Te podzbiory reprezentują pliki, na których można wykonać żądaną operację. Na przykład plik zwrócony w `can_read()` zawiera oczekujące dane przychodzące.

Funkcja `select()` to standardowe wywołanie systemowe niskiego poziomu, które jest powszechnie używane do śledzenia zdarzeń systemowych i implementowania asynchronicznych frameworków wejścia-wyjścia, takich jak wbudowany moduł `asyncio`.

Oprócz `select()` moduł `select` udostępnia także funkcje `poll()`, `epoll()`, `kqueue()` i podobne warianty operacji, które zapewniają podobną funkcjonalność. Dostępność tych funkcji zależy od systemu operacyjnego.

Moduł `selectors` zapewnia interfejs wyższego poziomu dla `select`, który może być przydatny w określonych kontekstach. Przykład podano wcześniej w sekcji 9.14.2.

9.15.19. Moduł smtplib

Moduł `smtplib` implementuje kliencką stronę SMTP, powszechnie używaną do wysyłania wiadomości e-mail. Typowym zastosowaniem modułu jest skrypt, który właśnie to robi — wysyła do kogoś wiadomość e-mail. Oto przykład:

```
import smtplib

fromaddr = "someone@some.com"
toaddrs = ["recipient@other.com"]
amount = 123.45
msg = f"""From: {fromaddr}
Zapłać {amount} bitcoinów lub w podobny sposób. Obserwujemy Cię.
"""

server = smtplib.SMTP('localhost')
```

```
serv.sendmail(fromaddr, toaddrs, msg)
serv.quit()
```

Istnieją dodatkowe funkcje do obsługi haseł, uwierzytelniania i innych spraw. Jeśli jednak uruchamiasz skrypt na komputerze, który jest skonfigurowany do obsługi poczty e-mail, powyższy przykład zwykle wykona zadanie.

9.15.20. Moduł socket

Moduł `socket` zapewnia dostęp niskiego poziomu do funkcji programowania sieciowego. Interfejs jest wzorowany na standardowym interfejsie gniazda BSD, powszechnie kojarzonym z programowaniem systemowym w języku C.

Poniższy przykład pokazuje, jak nawiązać połączenie wychodzące i otrzymać odpowiedź:

```
from socket import socket, AF_INET, SOCK_STREAM

sock = socket(AF_INET, SOCK_STREAM)
sock.connect(('python.org', 80))
sock.send(b'GET /index.html HTTP/1.0\r\n\r\n')
parts = []
while True:
    part = sock.recv(10000)
    if not part:
        break
    parts.append(part)
parts = b''.join(parts)
print(parts)
```

Poniższy przykład przedstawia podstawowy serwer echa, który akceptuje połączenia klientów i zwraca wszystkie odebrane dane. Aby przetestować ten serwer, uruchom go, a następnie połącz się z nim za pomocą polecenia takiego jak *telnet localhost 25000* lub *nc localhost 25000* w oddzielnej sesji terminala.

```
from socket import socket, AF_INET, SOCK_STREAM

def echo_server(address):
    sock = socket(AF_INET, SOCK_STREAM)
    sock.bind(address)
    sock.listen(1)
    while True:
        client, addr = sock.accept()
        echo_handler(client, addr)

def echo_handler(client, addr):
    print('Połączenie z:', addr)
    with client:
        while True:
            data = client.recv(10000)
            if not data:
                break
            client.sendall(data)
    print('Połączenie zamknięte')
```

```
if __name__ == '__main__':
    echo_server(('', 25000))
```

W przypadku serwerów UDP nie ma procesu połączenia. Serwer nadal jednak musi powiązać gniazdo ze znanym adresem. Oto typowy przykład tego, jak wyglądają serwer i klient UDP:

```
# udp.py

from socket import socket, AF_INET, SOCK_DGRAM

def run_server(address):
    sock = socket(AF_INET, SOCK_DGRAM)       # 1. Utwórz gniazdo UDP
    sock.bind(address)                       # 2. Powiąż z adresem/portem
    while True:
        msg, addr = sock.recvfrom(2000)      # 3. Otrzymaj wiadomość
        # Zrób coś
        response = b'world'
        sock.sendto(response, addr)          # 4. Odeślij odpowiedź

def run_client(address):
    sock = socket(AF_INET, SOCK_DGRAM)       # 1. Utwórz gniazdo UDP
    sock.sendto(b'hello', address)           # 2. Wyślij wiadomość
    response, addr = sock.recvfrom(2000)     # 3. Uzyskaj odpowiedź
    print("Otrzymano:", response)
    sock.close()

if __name__ == '__main__':
    import sys
    if len(sys.argv) != 4:
        raise SystemExit('Przykład użycia: udp.py [-client|-server] nazwa_hosta port')
    address = (sys.argv[2], int(sys.argv[3]))
    if sys.argv[1] == '-server':
        run_server(address)
    elif sys.argv[1] == '-client':
        run_client(address)
```

9.15.21. Moduł struct

Moduł `struct` służy do konwersji danych między Pythonem a binarnymi strukturami danych, reprezentowanymi jako łańcuchy bajtów Pythona. Te struktury danych są często używane podczas interakcji z funkcjami napisanymi w języku C, binarnymi formatami plików, protokołami sieciowymi lub binarną komunikacją przez porty szeregowe.

Załóżmy przykładowo, że musisz skonstruować wiadomość binarną o formacie opisanym przez strukturę danych w C:

```
# Format wiadomości: wszystkie wartości to "big endian"
struct Message {
    unsigned short msgid;    // 16-bitowa liczba całkowita bez znaku
    unsigned int sequence;   // 32-bitowy numer sekwencyjny
    float x;                 // 32-bitowy float
    float y;                 // 32-bitowy float
}
```

Oto jak to zrobić za pomocą modułu `struct`:

```
>>> import struct
>>> data = struct.pack('>HIff', 123, 456, 1.23, 4.56)
>>> data
b'\x00{\x00\x00\x00-?\x9dp\xa4@\x91\xeb\x85'
>>>
```

Aby zdekodować dane binarne, użyj `struct.unpack`:

```
>>> struct.unpack('>HIff', data)
(123, 456, 1.2300000190734863, 4.559999942779541)
>>>
```

Różnice w wartościach zmiennoprzecinkowych wynikają z utraty dokładności spowodowanej ich konwersją na wartości 32-bitowe. Python reprezentuje wartości zmiennoprzecinkowe jako 64-bitowe wartości o podwójnej precyzji.

9.15.22. Moduł subprocess

Moduł `subprocess` służy do wykonywania oddzielnego programu jako podprocesu, ale z kontrolą nad środowiskiem wykonawczym, w tym obsługą wejścia-wyjścia, zakończeniem i tak dalej. Istnieją dwa typowe zastosowania modułu.

Jeśli chcesz uruchomić oddzielny program i zebrać wszystkie jego dane wyjściowe naraz, użyj `check_output()`. Na przykład:

```
import subprocess

# Uruchom polecenie 'netstat -a' i zbierz jego dane wyjściowe
try:
    out = subprocess.check_output(['netstat', '-a'])
except subprocess.CalledProcessError as e:
    print("Wystąpił błąd:", e)
```

Dane zwracane przez `check_output()` są prezentowane w bajtach. Jeśli chcesz przekonwertować je na tekst, upewnij się, że stosujesz odpowiednie dekodowanie:

```
text = out.decode('utf-8')
```

Możliwe jest również skonfigurowanie potoku i bardziej szczegółowa interakcja z podprocesem. Aby to zrobić, użyj klasy `Popen` w ten sposób:

```
import subprocess

# wc to program, który zwraca liczbę wierszy, słów i bajtów
p = subprocess.Popen(['wc'],
                     stdin=subprocess.PIPE,
                     stdout=subprocess.PIPE)

# Wyślij dane do podprocesu
p.stdin.write(b'hello world\nthis is a test\n')
p.stdin.close()

# Odczytaj dane z powrotem
```

```
out = p.stdout.read()
print(out)
```

Wystąpienie instancji `Popen` ma atrybuty `stdin` i `stdout`, których można użyć do komunikacji z podprocesem.

9.15.23. Moduł tempfile

Moduł `tempfile` zapewnia obsługę tworzenia plików i katalogów tymczasowych. Oto przykład tworzenia pliku tymczasowego:

```
import tempfile

with tempfile.TemporaryFile() as f:
    f.write(b'Hello World')
    f.seek(0)
    data = f.read()
    print('Otrzymano:', data)
```

Domyślnie pliki tymczasowe są otwierane w trybie binarnym i umożliwiają zarówno odczyt, jak i zapis. Instrukcja `with` jest również powszechnie używana do określenia zakresu, w którym plik będzie wykorzystywany. Plik jest usuwany na końcu bloku `with`.

Jeśli chcesz utworzyć katalog tymczasowy, użyj tego:

```
with tempfile.TemporaryDirectory() as dirname:
    # Użyj katalogu dirname
    ...
```

Podobnie jak w przypadku pliku, katalog i cała jego zawartość zostaną usunięte na końcu bloku `with`.

9.15.24. Moduł textwrap

Moduł `textwrap` może być użyty do formatowania tekstu w celu dopasowania do określonej szerokości terminala. Moduł może być czasami przydatny do czyszczenia tekstu na wyjściu podczas tworzenia raportów. Interesujące są dwie funkcje.

`wrap()` pobiera tekst i zawija go tak, aby pasował do określonej szerokości kolumny. Funkcja zwraca listę ciągów. Na przykład:

```
import textwrap

text = """look into my eyes
look into my eyes
the eyes the eyes the eyes
not around the eyes
don't look around the eyes
look into my eyes you're under
"""

wrapped = textwrap.wrap(text, width=81)
print('\n'.join(wrapped))
```

```
# Tworzy:
# look into my eyes look into my eyes the
# eyes the eyes the eyes not around the
# eyes don't look around the eyes look
# into my eyes you're under
```

Funkcja `indent()` służy do robienia wcięć w bloku tekstu. Na przykład:

```
print(textwrap.indent(text, ' '))
# Tworzy:
#     look into my eyes
#     look into my eyes
#     the eyes the eyes the eyes
#     not around the eyes
#     don't look around the eyes
#     look into my eyes you're under
```

9.15.25. Moduł threading

Moduł `threading` służy do współbieżnego wykonywania kodu. Ten problem często pojawia się w przypadku obsługi wejścia-wyjścia w programach sieciowych. Programowanie wątków to obszerny temat, ale poniższe przykłady ilustrują rozwiązania typowych problemów.

Oto przykład uruchomienia wątku i czekania na niego:

```
import threading
import time

def countdown(n):
    while n > 0:
        print('T-minus', n)
        n -= 1
        time.sleep(1)

t = threading.Thread(target=countdown, args=[10])
t.start()
t.join()     # Poczekaj na zakończenie wątku
```

Jeśli nie zamierzasz czekać na zakończenie wątku, uczyń go demonicznym, dostarczając dodatkową flagę demona, taką jak ta:

```
t = threading.Thread(target=countdown, args=[10], daemon=True)
```

Jeśli chcesz, aby wątek się zakończył, musisz to zrobić jawnie za pomocą flagi lub jakiejś zmiennej służącej do tego celu. Dodatkowo wątek będzie musiał poprawnie to obsłużyć.

```
import threading
import time

must_stop = False

def countdown(n):
    while n > 0 and not must_stop:
        print('T-minus', n)
        n -= 1
```

```
        time.sleep(1)
```

Jeśli wątki będą udostępniać dane, ochroń je za pomocą blokady.

```
import threading

class Counter:
    def __init__(self):
        self.value = 0
        self.lock = threading.Lock()

    def increment(self):
        with self.lock:
            self.value += 1

    def decrement(self):
        with self.lock:
            self.value -= 1
```

Jeśli jeden wątek musi czekać, aż inny wątek coś zrobi, użyj funkcji `Event`.

```
import threading
import time

def step1(evt):
    print('Krok 1')
    time.sleep(5)
    evt.set()

def step2(evt):
    evt.wait()
    print('Krok 2')

evt = threading.Event()
threading.Thread(target=step1, args=[evt]).start()
threading.Thread(target=step2, args=[evt]).start()
```

Jeśli wątki będą się komunikować, użyj funkcji `Queue`:

```
import threading
import queue
import time

def producer(q):
    for i in range(10):
        print('Wytwarzanie:', i)
        q.put(i)
    print('Zrobione')
    q.put(None)

def consumer(q):
    while True:
        item = q.get()
        if item is None:
            break
        print('Przetwarzanie:', item)
```

```
    print('Do widzenia')

q = queue.Queue()
threading.Thread(target=producer, args=[q]).start()
threading.Thread(target=consumer, args=[q]).start()
```

9.15.26. Moduł time

Moduł `time` służy do uzyskiwania dostępu do funkcji systemowych związanych z czasem. Najbardziej przydatne są następujące funkcje:

`sleep(sekundy)`
: Uśpij aplikację Pythona przez określoną liczbę sekund, podaną jako liczba zmiennoprzecinkowa.

`time()`
: Zwróć bieżący czas systemowy w UTC jako liczbę zmiennoprzecinkową. Jest to liczba sekund od epoki (zwykle 1 stycznia 1970 dla systemów UNIX). Użyj `localtime()`, aby przekonwertować go na strukturę danych odpowiednią do wyodrębnienia przydatnych informacji.

`localtime([sekundy])`
: Zwróć obiekt `struct_time` reprezentujący czas lokalny w systemie lub czas reprezentowany przez wartość zmiennoprzecinkową `sekundy` przekazaną jako argument. Wynikowa struktura ma atrybuty `tm_year`, `tm_mon`, `tm_mday`, `tm_hour`, `tm_min`, `tm_sec`, `tm_wday`, `tm_yday` i `tm_isdst`.

`gmtime([sekundy])`
: To samo co `localtime()`, z wyjątkiem tego, że wynikowa struktura reprezentuje czas w UTC (lub czas uniwersalny — ang. *Greenwich Mean Time*).

`ctime([sekundy])`
: Konwertuj czas reprezentowany w sekundach na ciąg tekstowy odpowiedni do wyświetlenia. Przydatne do debugowania i rejestrowania.

`asctime(tm)`
: Konwertuj strukturę czasu reprezentowaną przez `localtime()` na ciąg tekstowy odpowiedni do wyświetlenia.

Moduł `datetime` jest ogólnie używany do reprezentowania dat i godzin w celu wykonywania obliczeń związanych z datą i radzenia sobie ze strefami czasowymi.

9.15.27. Pakiet urllib

Pakiet urllib służy do wysyłania żądań HTTP po stronie klienta. Być może najbardziej użyteczną funkcją jest urllib.request.urlopen(), której można używać do pobierania prostych stron internetowych. Na przykład:

```
>>> from urllib.request import urlopen
>>> u = urlopen('http://www.python.org')
>>> data = u.read()
>>>
```

Jeśli chcesz zakodować parametry formularza, możesz użyć `urllib.parse.urlencode()`, jak pokazano tutaj:

```
from urllib.parse import urlencode
from urllib.request import urlopen

form = {
'name': 'Mary A. Python',
'email': 'mary123@python.org'
}

data = urlencode(form)
u = urlopen('http://httpbin.org/post', data.encode('utf-8'))
response = u.read()
```

Funkcja `urlopen()` działa dobrze w przypadku podstawowych stron internetowych i interfejsów API obejmujących protokół HTTP lub HTTPS. Jednak korzystanie z niej staje się dość niewygodne, jeśli dostęp obejmuje również pliki cookie, zaawansowane schematy uwierzytelniania i inne warstwy. Szczerze mówiąc, większość programistów Pythona do obsługi takich sytuacji używałaby zewnętrznych bibliotek, takich jak `requests` lub `httpx`. Ty też powinieneś.

Podpakiet `urllib.parse` posiada dodatkowe funkcje do manipulowania samymi adresami URL. Na przykład funkcja `urlparse()` może zostać użyta do rozdzielenia adresu URL:

```
>>> url = 'http://httpbin.org/get?name=Dave&n=42'
>>> from urllib.parse import urlparse
>>> urlparse(url)
ParseResult(scheme='http', netloc='httpbin.org', path='/get', params='',
query='name=Dave&n=42', fragment='')
>>>
```

9.15.28. Moduł unicodedata

Moduł `unicodedata` jest używany do bardziej zaawansowanych operacji obejmujących ciągi tekstowe Unicode. Często istnieje wiele reprezentacji tego samego tekstu Unicode. Na przykład znak U+00F1 (ñ) może być w całości złożony jako pojedynczy znak U+00F1 lub rozłożony na sekwencję wieloznakową U+006e, U+0303 (n, ~). Może to powodować nieoczekiwane problemy w programach wymagających, by ciągi tekstowe renderujące tę samą zawartość wyglądały tak samo. Rozważmy następujący przykład dotyczący kluczy słownikowych:

```
>>> d = {}
>>> d['Jalape\xf1o'] = 'spicy'
>>> d['Jalapen\u0303o'] = 'mild'
>>> d
{'jalapeño': 'spicy', 'jalapeño': 'mild'}
>>>
```

Na pierwszy rzut oka wydaje się, że powinien wystąpić błąd operacyjny — jak słownik mógłby mieć dwa identyczne klucze? Na odpowiedź składa się fakt, że klucze są zbudowane z różnych sekwencji znaków Unicode.

Jeśli problemem jest spójne przetwarzanie identycznie renderowanych ciągów Unicode, należy je znormalizować. Funkcja `unicodedata.normalize()` może służyć do zapewnienia spójnej reprezentacji znaków. Na przykład `unicodedata.normalize('NFC', s)` upewni się, że wszystkie znaki w `s` są w pełni skomponowane i nie są reprezentowane jako sekwencja połączonych znaków. Użycie `unicodedata.normalize('NFD', s)` zapewni, że wszystkie znaki w `s` są w pełni zdekomponowane.

Moduł `unicodedata` posiada również funkcje do testowania właściwości znaków, takich jak wielkość liter, liczby i białe znaki. Ogólne właściwości znaków można uzyskać za pomocą funkcji `unicodedata.category(c)`. Na przykład `unicodedata.category('A')` zwraca `'Lu'`, co oznacza, że znak jest wielką literą. Więcej informacji na temat tych wartości można znaleźć w oficjalnej bazie danych znaków Unicode pod adresem *https://www.unicode.org/ucd*.

9.15.29. Pakiet xml

Pakiet `xml` to duży zbiór modułów do przetwarzania danych XML na różne sposoby.

Jeśli jednak Twoim głównym celem jest odczytanie dokumentu XML i wyodrębnienie z niego informacji, najprostszym sposobem na to jest użycie podpakietu `xml.etree`. Załóżmy, że masz plik w formacie XML — *recipe.xml*:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<recipe>
   <title>Famous Guacamole</title>
   <description>A southwest favorite!</description>
   <ingredients>
      <item num="4"> Large avocados, chopped </item>
      <item num="1"> Tomato, chopped </item>
      <item num="1/2" units="C"> White onion, chopped </item>
      <item num="2" units="tbl"> Fresh squeezed lemon juice </item>
      <item num="1"> Jalapeno pepper, diced </item>
      <item num="1" units="tbl"> Fresh cilantro, minced </item>
      <item num="1" units="tbl"> Garlic, minced </item>
      <item num="3" units="tsp"> Salt </item>
      <item num="12" units="bottles"> Ice-cold beer </item>
   </ingredients>
   <directions>
   Combine all ingredients and hand whisk to desired consistency.
   Serve and enjoy with ice-cold beers.
   </directions>
</recipe>
```

Oto jak z powyższego dokumentu wydobyć określone elementy:

```
from xml.etree.ElementTree import ElementTree

doc = ElementTree(file="recipe.xml")
title = doc.find('title')
print(title.text)
```

```
# Alternatywnie (po prostu pobierz tekst elementu)
print(doc.findtext('description'))

# Iteruj po wielu elementach
for item in doc.findall('ingredients/item'):
    num = item.get('num')
    units = item.get('units', '')
    text = item.text.strip()
    print(f'{num} {units} {text}')
```

9.16. Podsumowanie

Obsługa wejścia-wyjścia to podstawowy element tworzenia każdego użytecznego programu. Biorąc pod uwagę popularność Pythona, może on pracować z dosłownie każdym używanym formatem danych, kodowaniem lub strukturą dokumentu. Chociaż standardowa biblioteka może niektórych z tych rzeczy nie obsługiwać, prawie na pewno znajdziesz moduł innej firmy, który rozwiąże Twój problem.

Ogólnie rzecz biorąc, bardziej przydatne może być zastanowienie się nad elementami brzegowymi aplikacji. Na styku między programem a rzeczywistością często pojawiają się problemy związane z kodowaniem danych. Dotyczy to zwłaszcza danych tekstowych i Unicode. Duża część złożoności w obsłudze operacji wejścia-wyjścia w Pythonie — obsługa różnych kodowań, zasad obsługi błędów itd. — jest skierowana na ten konkretny problem. Należy również pamiętać, że dane tekstowe i dane binarne są ściśle oddzielone. Wiedza związana z konkretnym problemem, pomaga w zrozumieniu całościowego obrazu.

Drugorzędną kwestią obsługi operacji wejścia-wyjścia jest ogólny model przetwarzania. Kod Pythona jest obecnie podzielony na dwa światy: normalny kod synchroniczny i kod asynchroniczny, zwykle kojarzony z modułem `asyncio` (charakteryzującym się wykorzystaniem funkcji `async` i składni `async/await`). Kod asynchroniczny prawie zawsze wymaga użycia bibliotek, które mogą działać w tym środowisku. To z kolei wymusza pisanie kodu aplikacji również w stylu asynchronicznym. Szczerze mówiąc, prawdopodobnie powinieneś unikać kodowania asynchronicznego, chyba że wiesz, że absolutnie go potrzebujesz — ale jeśli nie jesteś naprawdę pewien, to go nie stosuj. Większość dobrze napisanych aplikacji w Pythonie — w standardowym, synchronicznym stylu — jest o wiele łatwiejsza do zrozumienia, debugowania i testowania. Powinieneś wybrać tę drogę.

10

Funkcje wbudowane i biblioteka standardowa

Ten rozdział jest zwięzłym odniesieniem do wbudowanych funkcji Pythona. Są one zawsze dostępne bez instrukcji `import`. Rozdział kończy się krótkim przeglądem kilku przydatnych standardowych modułów bibliotecznych.

10.1. Funkcje wbudowane

`abs(x)`

Zwraca wartość bezwzględną `x`.

`all(s)`

Zwraca `True`, jeśli wszystkie wartości w iterowalnym `s` mają wartość `True`. Zwraca `True`, jeśli `s` jest puste.

`any(s)`

Zwraca `True`, jeśli którakolwiek z wartości w iterowalnym `s` zostanie oceniona jako `True`. Zwraca `False`, jeśli `s` jest puste.

`ascii(x)`

Tworzy drukowalną reprezentację obiektu `x`, podobnie jak `repr()`, ale w wyniku używa tylko znaków ASCII. Znaki spoza zestawu ASCII są przekształcane w odpowiednie sekwencje specjalne. Może służyć do przeglądania ciągów Unicode w terminalu lub powłoce, które nie obsługują Unicode.

`bin(x)`

Zwraca ciąg z binarną reprezentacją liczby całkowitej `x`.

`bool([x])`

Typ reprezentujący wartości logiczne `True` i `False`. Użyty do przekonwertowania `x`, zwraca `True`, jeśli `x` zostanie wyliczony jako `True` przy wykorzystaniu zwykłej semantyki sprawdzania prawdziwości —czy liczba jest niezerowa, czy lista jest niepusta i tak dalej. W przeciwnym razie zwracane jest `False`. `False` jest również domyślnie zwracane, jeśli funkcja `bool()` zostanie wywołana bez żadnych argumentów. Klasa `bool` dziedziczy po `int`, więc wartości logiczne `True` i `False` mogą być używane jako liczby całkowite o wartościach `1` i `0` w obliczeniach matematycznych.

`breakpoint()`

Ustawia punkt przerwania ręcznego debugera. Po jego napotkaniu kontrola zostanie przeniesiona do `pdb` — debugera Pythona.

`bytearray([x])`

Typ reprezentujący zmienną tablicę bajtów. Podczas tworzenia instancji `x` może być iterowalną sekwencją liczb całkowitych z zakresu od 0 do 255, 8-bitowym ciągiem znaków lub literałem bajtów bądź liczbą całkowitą określającą rozmiar tablicy bajtów (w takim przypadku każdy wpis zostanie zainicjowany jako 0).

`bytearray(s, encoding)`

Alternatywna konwencja służąca do tworzenia instancji `bytearray` ze znaków w ciągu `s`, gdzie `encoding` określa kodowanie znaków, które ma być użyte podczas konwersji.

`bytes([x])`

Typ reprezentujący niezmienną tablicę bajtów.

`bytes(s, encoding)`

Alternatywna konwencja służąca do tworzenia bajtów z ciągu `s`, gdzie `encoding` określa kodowanie do użycia podczas konwersji.

Tabela 10.1 przedstawia operacje obsługiwane zarówno przez bajty, jak i tablice bajtów.

Tabela 10.1. *Operacje na bajtach i tablicach bajtów*

Operacja	Opis
`s + t`	Łączy, jeśli `t` reprezentuje bajty.
`s * n`	Replikuje, jeśli `n` jest liczbą całkowitą.
`s % x`	Formatuje bajty. `x` jest krotką.
`s[i]`	Zwraca `i`-ty element jako liczbę całkowitą.
`s[i:j]`	Zwraca wycinek.
`s[i:j:step]`	Zwraca rozszerzony wycinek.
`len(s)`	Liczba bajtów w `s`.
`s.capitalize()`	Zamienia pierwszy znak na wielką literę.
`s.center(width [, pad])`	Wyśrodkowuje napis w polu o długości `width`. `pad` to znak dopełniający.
`s.count(sub [, start [, end]])`	Zlicza wystąpienia określonego podłańcucha.
`s.decode([encoding [, errors]])`	Dekoduje ciąg bajtów na tekst (tylko typ bajtów).

Tabela 10.1. *Operacje na bajtach i tablicach bajtów (ciąg dalszy)*

Operacja	Opis
`s.endswith(suffix [, start [, end]])`	Sprawdza koniec łańcucha pod kątem przyrostka.
`s.expandtabs([tabsize])`	Zastępuje tabulatory spacjami.
`s.find(sub [, start [, end]])`	Znajduje pierwsze wystąpienie określonego podłańcucha.
`s.hex()`	Konwertuje na ciąg szesnastkowy.
`s.index(sub [, start [, end]])`	Znajduje pierwsze wystąpienie lub błąd w określonym podłańcuchu.
`s.isalnum()`	Sprawdza, czy wszystkie znaki są alfanumeryczne.
`s.isalpha()`	Sprawdza, czy wszystkie znaki są alfabetyczne.
`s.isascii()`	Sprawdza, czy wszystkie znaki są ASCII.
`s.isdigit()`	Sprawdza, czy wszystkie znaki są cyframi.
`s.islower()`	Sprawdza, czy wszystkie znaki są małymi literami.
`s.isspace()`	Sprawdza, czy wszystkie znaki są białymi znakami.
`s.istitle()`	Sprawdza, czy ciąg jest ciągiem pisanym wielkimi literami w tytule (pierwsza litera każdego słowa pisana wielką literą).
`s.isupper()`	Sprawdza, czy wszystkie znaki są wielkimi literami.
`s.join(t)`	Łączy ciąg łańcuchów `t` przy użyciu ogranicznika `s`.
`s.ljust(width [, fill])`	Wyrównuje `s` do lewej w łańcuchu o rozmiarze `width`.
`s.lower()`	Konwertuje na małe litery.
`s.lstrip([chrs])`	Usuwa początkowe białe znaki lub znaki podane w `chrs`.
`s.maketrans(x [, y [, z]])`	Tworzy tabelę translacji dla `s.translate()`.
`s.partition(sep)`	Dzieli łańcuch na partycje na podstawie separatora ciągu `sep`. Zwraca krotkę `(head, sep, tail)` lub `(s, '', '')`, jeśli nie znaleziono `sep`.
`s.removeprefix(prefix)`	Zwraca `s` z usuniętym przedrostkiem, jeśli jest obecny.
`s.removesuffix(suffix)`	Zwraca `s` z usuniętym przyrostkiem, jeśli jest obecny.
`s.replace(old, new [, maxreplace])`	Zastępuje podciąg.
`s.rfind(sub [, start [, end]])`	Znajduje ostatnie wystąpienie podciągu.
`s.rindex(sub [, start [, end]])`	Znajduje ostatnie wystąpienie lub zgłasza błąd.
`s.rjust(width [, fill])`	Wyrównuje `s` do prawej w ciągu o długości `width`.
`s.rpartition(sep)`	Partycje oparte na separatorze `sep`, ale wyszukiwanie przebiega od końca ciągu.

Tabela 10.1. *Operacje na bajtach i tablicach bajtów (ciąg dalszy)*

Operacja	Opis
`s.rsplit([sep [, maxsplit]])`	Oddziela ciąg od końca, używając `sep` jako separatora. `maxsplit` to maksymalna liczba podziałów do wykonania. Jeśli `maxsplit` zostanie pominięty, wynik jest identyczny z metodą `split()`.
`s.rstrip([chrs])`	Usuwa końcowe białe znaki lub znaki podane w `chrs`.
`s.split([sep [, maxsplit]])`	Dzieli łańcuch, używając `sep` jako separatora. `maxsplit` to maksymalna liczba podziałów do wykonania.
`s.splitlines([keepends])`	Dzieli łańcuch na listę linii. Jeśli `keepends` wynosi 1, zachowywane są końcowe znaki nowej linii.
`s.startswith(prefix [, start [, end]])`	Sprawdza, czy łańcuch zaczyna się od przedrostka.
`s.strip([chrs])`	Usuwa początkowe i końcowe białe znaki lub znaki podane w `chrs`.
`s.swapcase()`	Zamienia wielkie litery na małe i na odwrót.
`s.title()`	Zwraca wersję ciągu z pisanymi literami tytułu.
`s.translate(table [, deletechars])`	Tłumaczy ciąg znaków za pomocą tablicy translacji znaków, usuwając znaki z `deletechars`.
`s.upper()`	Konwertuje ciąg na wielkie litery.
`s.zfill(width)`	Dopełnia ciąg zerami po lewej stronie do określonej szerokości — `width`.

Tablice bajtowe dodatkowo obsługują metody z tabeli 10.2.

Tabela 10.2. *Dodatkowe operacje na tablicach bajtów*

Operacja	Opis
`s[i] = v`	Przypisuje element.
`s[i:j] = t`	Przypisuje wycinek.
`s[i:j:step] = t`	Rozszerzone przypisanie wycinka.
`del s[i]`	Usuwa element.
`del s[i:j]`	Usuwa wycinek.
`del s[i:j:step`	Rozszerzone usuwanie wycinka.
`s.append(x)`	Dodaje na końcu nowy bajt.
`s.clear()`	Czyści tablicę bajtów.
`s.copy()`	Tworzy kopię.
`s.extend(t)`	Rozszerza `s` o bajty od `t`.
`s.insert(n, x)`	Wstawia bajt `x` pod indeksem `n`.
`s.pop([n])`	Usuwa i zwraca bajt o indeksie `n`.
`s.remove(x)`	Usuwa pierwsze wystąpienie bajta `x`.
`s.reverse()`	Odwraca w miejscu tablicę bajtów.

`callable(obj)`

Zwraca `True`, jeśli `obj` można wywołać jako funkcję.

`chr(x)`

Konwertuje liczbę całkowitą `x` reprezentującą kod Unicode na ciąg jednoznakowy.

`classmethod(func)`

Ten dekorator tworzy metodę klasy dla funkcji `func`. Zwykle jest używany tylko w definicjach klas, w których jest wywoływany niejawnie za pomocą `@classmethod`. W przeciwieństwie do zwykłej metody, metoda klasy jako pierwszy argument otrzymuje klasę (a nie instancję).

`compile(string, filename, kind)`

Kompiluje `string` do obiektu kodu, który może być wykorzystany przez funkcję `exec()` lub `eval()`. `string` to ciąg znaków zawierający poprawny kod Pythona. Jeśli ten kod obejmuje wiele wierszy, muszą być one zakończone pojedynczym znakiem nowej linii (`'\n'`), a nie wariantami specyficznymi dla platformy (na przykład `'\r\n'` w systemie Windows). `filename` to ciąg znaków zawierający nazwę pliku, w którym ciąg został zdefiniowany (jeśli istnieje). `kind` przyjmuje wartość `'exec'` dla sekwencji instrukcji, `'eval'` dla pojedynczego wyrażenia i `'single'` dla pojedynczej instrukcji wykonywalnej. Wynikowy obiekt kodu, który jest zwracany, może być przekazany bezpośrednio do `exec()` lub `eval()`.

`complex([real [, imag]])`

Typ reprezentujący liczbę zespoloną ze składnikami rzeczywistymi i urojonymi, `real` i `imag`, które można podać jako dowolny typ liczbowy. Jeśli `imag` zostanie pominięty, składnik urojony zostanie ustawiony na zero. Jeśli wartość `real` jest przekazywana jako łańcuch znaków, jest on analizowany i konwertowany na liczbę zespoloną. W takim przypadku `imag` należy pominąć. Jeśli `real` jest jakimkolwiek innym rodzajem obiektu, zwracana jest wartość `real.__complex__()`. Jeśli nie podano argumentów, zwracane jest `0j`.

Tabela 10.3 przedstawia metody i atrybuty `complex`.

Tabela 10.3. *Atrybuty complex*

Atrybut/metoda	Opis
`z.real`	Składnik rzeczywisty
`z.imag`	Składnik urojony
`z.conjugate()`	Liczba sprzężona

`delattr(object, attr)`

Usuwa atrybut obiektu — `attr` (ciąg znaków). To samo co `del object.attr`.

`dict([m]) or dict(klucz1=wartość1, klucz2=wartość2, ...)`

Typ reprezentujący słownik. Jeśli nie podano argumentu, zwracany jest pusty słownik. Jeśli `m` jest obiektem mapującym (takim jak inny słownik), zwracany jest nowy słownik mający te same klucze i te same wartości co `m`. Jeśli na przykład `m` jest słownikiem, `dict(m)` tworzy jego płytką kopię. Jeśli `m` nie jest odwzorowaniem, musi obsługiwać iterację, w której

tworzona jest sekwencja par (`klucz, wartość`). Te pary służą do wypełniania słownika. `dict()` można również wywołać z argumentami słów kluczowych. Na przykład `dict(foo=3, bar=7)` tworzy słownik `{'foo': 3, 'bar': 7}`.

Tabela 10.4 przedstawia operacje obsługiwane przez słowniki.

Tabela 10.4. *Operacje na słownikach*

Operacja	Opis
`m | n`	Łączy `m` i `n` w jeden słownik.
`len(m)`	Zwraca liczbę elementów w `m`.
`m[k]`	Zwraca element z kluczem `k`.
`m[k]=x`	Ustawia wartość elementu `m[k]` na `x`.
`del m[k]`	Usuwa element `m[k]` z `m`.
`k in m`	Zwraca `True`, jeśli `k` jest kluczem w `m`.
`m.clear()`	Usuwa wszystkie elementy z `m`.
`m.copy()`	Tworzy płytką kopię `m`.
`m.fromkeys(s [, value])`	Tworzy nowy słownik z kluczami z sekwencji `s` i wartościami ustawionymi na `value`.
`m.get(k [, v])`	Zwraca element `m[k]`, jeśli znaleziono; w przeciwnym razie zwraca `v`.
`m.items()`	Zwraca pary (`klucz, wartość`).
`m.keys()`	Zwraca klucze.
`m.pop(k [, default])`	Zwraca element `m[k]`, jeśli został znaleziony, i usuwa go z `m`; w przeciwnym razie zwraca wartość domyślną, jeśli podano, lub zgłasza wyjątek `KeyError`, jeśli nie.
`m.popitem()`	Usuwa losową parę (`klucz, wartość`) z `m` i zwraca ją jako krotkę.
`m.setdefault(k [, v])`	Zwraca element `m[k]`, jeśli znaleziono; w przeciwnym razie zwraca `v` i ustawia `m[k] = v`.
`m.update(b)`	Dodaje wszystkie obiekty `b` do `m`.
`m.values()`	Zwraca wartości.

`dir([object])`

Zwraca posortowaną listę nazw atrybutów. Jeśli `object` jest modułem, zawiera listę symboli zdefiniowanych w tym module. Jeśli `object` jest obiektem typu lub klasy, zwraca listę nazw atrybutów. Nazwy są zazwyczaj uzyskiwane z atrybutu `__dict__` obiektu, jeśli jest zdefiniowany, ale można użyć innych źródeł. Jeśli nie podano argumentu, zwracane są nazwy z bieżącej lokalnej tablicy symboli. Należy zauważyć, że ta funkcja jest używana przede wszystkim w celach informacyjnych (na przykład stosowana interaktywnie w wierszu poleceń). Nie należy jej wykorzystywać do formalnej analizy programu, ponieważ uzyskane informacje mogą być niekompletne. Ponadto klasy zdefiniowane przez użytkownika mogą definiować specjalną metodę `__dir__()`, która zmienia wynik tej funkcji.

`divmod(a, b)`

Zwraca iloraz i resztę z dzielenia długiego jako krotkę. W przypadku liczb całkowitych zwracana jest wartość `(a // b, a % b)`. W przypadku elementów zmiennoprzecinkowych zwracane jest `(math.floor(a / b), a % b)`. Ta funkcja nie może być wywoływana z liczbami zespolonymi.

`enumerate(iter, start=0)`

Biorąc pod uwagę iterowalny obiekt `iter`, zwraca nowy iterator (typu `enumerate`), który generuje krotki zawierające liczbę i wartość utworzoną z `iter`. Jeśli na przykład `iter` daje `a`, `b`, `c`, to `enumerate(iter)` daje `(0,a)`, `(1,b)`, `(2,c)`. Opcjonalny `start` zmienia początkową wartość licznika.

`eval(expr [, globals [, locals]])`

Oblicza wyrażenie. `expr` to ciąg znaków lub obiekt kodu utworzony przez `compile()`. `globals` i `locals` to obiekty mapujące, które definiują odpowiednio globalną i lokalną przestrzeń nazw dla operacji. Jeśli zostaną pominięte, wyrażenie jest liczone przy użyciu wartości `globals()` i `locals()` wykonywanych w środowisku wywołującego. Najczęściej wartości `globals` i `locals` są określane jako słowniki, ale zaawansowane aplikacje mogą dostarczać niestandardowe obiekty mapowania.

`exec(expr [, globals [, locals]])`

Wykonuje instrukcje Pythona. `expr` to ciąg znaków, bajty lub obiekt kodu utworzony przez `compile()`. `globals` i `locals` definiują odpowiednio globalną i lokalną przestrzeń nazw dla operacji. Jeśli zostaną pominięte, kod jest wykonywany przy użyciu wartości `globals()` i `locals()`, tak jak jest wykonywany w środowisku wywołującego.

`filter(function, iterable)`

Tworzy iterator zwracający elementy w iteracji, dla których `function(item)` wylicza wartość `True`.

`float([x])`

Reprezentuje liczbę zmiennoprzecinkową. Jeśli `x` jest liczbą, jest konwertowana na liczbę zmiennoprzecinkową. Jeśli `x` jest łańcuchem znaków, jest przetwarzany na liczbę zmiennoprzecinkową. Dla wszystkich innych obiektów wywoływana jest funkcja `x.__float__()`. Jeśli nie podano argumentu, zwracane jest `0.0`.

Tabela 10.5 przedstawia metody i atrybuty typu `float`.

Tabela 10.5. *Metody i atrybuty float*

Atrybut/metoda	Opis
`x.real`	Składnik rzeczywisty, jeśli jest liczbą zespoloną.
`x.imag`	Składnik urojony, jeśli jest liczbą zespoloną.
`x.conjugate()`	Liczba sprzężona.
`x.as_integer_ratio()`	Konwertuje na parę licznik/mianownik.
`x.hex()`	Tworzy reprezentację szesnastkową.
`x.is_integer()`	Sprawdza, czy jest to dokładna wartość całkowita.
`float.fromhex(s)`	Tworzy `float` z ciągu szesnastkowego. Metoda klasy.

`format(value [, format_spec])`

Konwertuje `value` na sformatowany ciąg, zgodnie ze specyfikacją zawartą w `format_spec`. Ta operacja wywołuje metodę `value.__format__()`, która może swobodnie interpretować specyfikację formatu. W przypadku prostych typów danych specyfikator formatu (`format_spec`) zazwyczaj zawiera znak wyrównania `'<'`, `'>'` lub `'^'`, liczbę (wskazującą szerokość pola) oraz kod znaku `'d'`, `'f'` bądź `'s'` odpowiednio dla wartości całkowitych, zmiennoprzecinkowych lub łańcuchowych. Na przykład specyfikacja formatu `'d'` formatuje liczbę całkowitą, specyfikacja `'8d'` wyrównuje do prawej liczbę całkowitą w polu 8-znakowym, a `'<8d'` wyrównuje do lewej liczbę całkowitą w polu 8-znakowym. Więcej szczegółów na temat `format()` i specyfikatorów formatu można znaleźć w rozdziale 9.

`frozenset([items])`

Typ reprezentujący niezmienny obiekt zbioru wypełniony wartościami pobranymi z elementów, które muszą być iterowalne. Wartości muszą być również niezmienne. Jeśli nie podano argumentu, zwracany jest pusty zbiór. `frozenset` obsługuje wszystkie operacje zbiorów z wyjątkiem operacji, które zmieniają zbiór w miejscu.

`getattr(object, name [, default])`

Zwraca wartość nazwanego atrybutu obiektu. `name` to ciąg znaków zawierający nazwę atrybutu. `default` jest opcjonalną wartością do zwrócenia, jeśli taki atrybut nie istnieje; w przeciwnym razie zwracany jest wyjątek `AttributeError`. Działa tak samo jak operacja `object.name`.

`globals()`

Zwraca słownik bieżącego modułu, który reprezentuje globalną przestrzeń nazw. Wywoływany wewnątrz innej funkcji lub metody zwraca globalną przestrzeń nazw modułu, w którym funkcja lub metoda została zdefiniowana.

`hasattr(object, name)`

Zwraca `True`, jeśli `name` jest nazwą atrybutu obiektu. `name` to ciąg znaków.

`hash(object)`

Zwraca całkowitą wartość skrótu dla obiektu (jeśli to możliwe). Wartość skrótu jest używana głównie w implementacji słowników, zbiorów i innych obiektów mapujących. Wartość skrótu jest zawsze taka sama dla wszystkich obiektów, które są równe. Obiekty mutowalne zwykle nie definiują wartości skrótu, chociaż klasy zdefiniowane przez użytkownika mogą definiować metodę `__hash__()` do obsługi tej operacji.

`hex(x)`

Tworzy ciąg szesnastkowy z liczby całkowitej `x`.

`id(object)`

Zwraca unikalną tożsamość całkowitą obiektu. Nie powinieneś w żaden sposób interpretować zwracanej wartości (na przykład nie jest to lokalizacja w pamięci).

`input([prompt])`

Wypisuje znak zachęty na standardowe wyjście i odczytuje pojedynczy wiersz wejścia ze standardowego wejścia. Zwracana linia nie jest w żaden sposób modyfikowana. Nie zawiera znaku końca wiersza (na przykład `'\n'`).

`int(x [, base])`

Typ reprezentujący liczbę całkowitą. Jeśli `x` jest liczbą, jest konwertowana na liczbę całkowitą przez obcięcie miejsc zerowych. Jeśli jest to łańcuch znaków, jest przetwarzany na wartość całkowitą. `base` opcjonalnie określa podstawę podczas konwersji z łańcucha znaków.

Oprócz obsługi typowych operacji matematycznych liczby całkowite mają również szereg atrybutów i metod wymienionych w tabeli 10.6.

Tabela 10.6. *Metody i atrybuty liczb całkowitych*

Operacja	Opis
`x.numerator`	Licznik, jeśli liczba jest ułamkiem.
`x.denominator`	Mianownik, jeśli liczba jest ułamkiem.
`x.real`	Składnik rzeczywisty, jeśli to liczba zespolona.
`x.imag`	Składnik urojony, jeśli to liczba zespolona.
`x.conjugate()`	Liczba sprzężona.
`x.bit_length()`	Liczba bitów potrzebnych do przedstawienia wartości w postaci binarnej.
`x.to_bytes(bytes, byteorder, *, signed=False)`	Konwertuje na bajty.
`int.from_bytes(bytes, byteorder, *, signed=False)`	Konwertuje z bajtów. Metoda klasy.

`isinstance(object, classobj)`

Zwraca `True`, jeśli obiekt jest instancją `classobj` lub podklasą `classobj`. Parametr `classobj` może być również krotką możliwych typów lub klas. Na przykład `isinstance(s, (lista, krotka))` zwraca `True`, jeśli `s` jest krotką lub listą.

`issubclass(class1, class2)`

Zwraca `True`, jeśli `class1` jest podklasą (pochodną) `class2`. `class2` może być również krotką możliwych klas, w którym to przypadku każda klasa zostanie sprawdzona. Zauważ, że `issubclass(A, A)` zwraca wartość `True`.

`iter(object [, sentinel])`

Zwraca iterator do wytwarzania elementów w obiekcie `object`. Jeśli parametr `sentinel` zostanie pominięty, obiekt musi albo udostępniać metodę `__iter__()`, która tworzy iterator, albo implementować `__getitem__()`, która akceptuje argumenty w postaci liczb całkowitych zaczynających się od `0`. Jeśli określono wartość `sentinel`, obiekt jest interpretowany inaczej. Zamiast tego `object` powinien być obiektem możliwym do wywołania, który nie przyjmuje parametrów. Zwrócony obiekt iteratora będzie wywoływał funkcję (`__getitem__()` lub `__iter__()`) wielokrotnie, aż zwrócona wartość będzie równa `sentinel`, w którym to momencie iteracja zostanie zatrzymana. Wyjątek `TypeError` zostanie wygenerowany, jeśli obiekt nie obsługuje iteracji.

`len(s)`

Zwraca liczbę elementów zawartych w `s`. `s` powinien być pewnego rodzaju kontenerem, takim jak lista, krotka, ciąg, zbiór lub słownik.

`list([items])`

Typ reprezentujący listę. Wartościami `items` mogą być dowolne iterowalne obiekty, których wartości są używane do wypełniania listy. Jeśli `items` jest już listą, tworzona jest płytka kopia. Jeśli nie podano argumentu, zwracana jest pusta lista.

Tabela 10.7 przedstawia operacje zdefiniowane na listach.

Tabela 10.7. *Lista operatorów i metod*

Operacja	Opis
`s + t`	Konkatenacja, jeśli `t` jest listą.
`s * n`	Replikacja, jeśli `n` jest liczbą całkowitą.
`s[i]`	Zwraca `i`-ty element z `s`.
`s[i:j]`	Zwraca wycinek.
`s[i:j:step]`	Zwraca rozszerzony wycinek.
`s[i] = v`	Przypisuje element.
`s[i:j] = t`	Przypisuje wycinek.
`s[i:j:step] = t`	Rozszerzone przypisanie wycinka
`del s[i]`	Usuwa element.
`del s[i:j]`	Usuwa wycinek.
`del s[i:j:step`	Rozszerzone usuwanie wycinka.
`len(s)`	Zwraca liczbę elementów w `s`.
`s.append(x)`	Dołącza nowy element `x` na końcu `s`.
`s.extend(t)`	Dołącza nową listę (`t`) na końcu `s`.
`s.count(x)`	Zlicza wystąpienia `x` w `s`.
`s.index(x [, start [, stop]])`	Zwraca najmniejsze `i`, gdzie `s[i] == x`. Wartości `start` i `stop` opcjonalnie określają początkowy i końcowy indeks wyszukiwania.
`s.insert(i, x)`	Wstawia `x` pod indeksem `i`.
`s.pop([i])`	Zwraca element `i`, po czym usuwa go z listy. Jeśli pominięto `i`, zwracany jest ostatni element.
`s.remove(x)`	Wyszukuje `x` i usuwa je z `s`.
`s.reverse()`	Odwraca elementy `s` w miejscu.
`s.sort([key [, reverse]])`	Sortuje elementy `s` w miejscu. `key` jest funkcją klucza. `reverse` to flaga, która sortuje listę w odwrotnej kolejności. `key` i `reverse` powinny być zawsze podawane jako argumenty słów kluczowych.

`locals()`

Zwraca słownik odpowiadający lokalnej przestrzeni nazw dla funkcji wywołującej. Ten słownik powinien być używany tylko do sprawdzania środowiska wykonawczego — zmiany wprowadzone w słowniku nie mają żadnego wpływu na odpowiadające mu zmienne lokalne.

`map(function, items, ...)`

Tworzy iterator, który generuje wyniki zastosowania funkcji `function` do elementów `items`. Jeśli podano wiele sekwencji wejściowych, zakłada się, że funkcja przyjmuje tyle argumentów, a każdy argument pochodzi z innej sekwencji. W takim przypadku wynik jest tak długi jak najkrótsza sekwencja wejściowa.

`max(s [, args, ...], *, default=obj, key=func)`

Dla pojedynczego argumentu `s` funkcja zwraca maksymalną wartość elementów w `s`, która może być dowolnym obiektem iterowalnym. W przypadku wielu argumentów zwraca największy z nich. Jeśli podano domyślny argument (`default`), zawiera on wartość do zwrócenia, gdy `s` jest puste. Jeśli podano argument `key`, to zwracana jest wartość `v`, dla której `key(v)` zwraca maksymalną wartość.

`min(s [, args, ...], *, default=obj, key=func)`

To samo co `max(s)`, z tą różnicą, że zwracana jest wartość minimalna.

`next(s [, default])`

Zwraca następny element z iteratora. Jeśli iterator nie ma więcej elementów, zgłaszany jest wyjątek `StopIteration`, chyba że dla domyślnego argumentu (`default`) zostanie podana wartość. W takim przypadku zamiast wyjątku zwracana jest wartość `default`.

`object()`

Klasa bazowa dla wszystkich obiektów w Pythonie. Możesz ją wywołać, aby utworzyć instancję, ale wynik nie jest szczególnie interesujący.

`oct(x)`

Konwertuje liczbę całkowitą `x` na ciąg ósemkowy.

`open(filename [, mode [, bufsize [, encoding [, errors [, newline [, closefd]]]]]])`

Otwiera plik `filename` i zwraca obiekt pliku. Argumenty zostały szczegółowo opisane w rozdziale 9.

`ord(c)`

Zwraca całkowitą wartość porządkową pojedynczego znaku `c`. Ta wartość zwykle odpowiada wartości kodu znaku Unicode.

`pow(x, y [, z])`

Zwraca `x ** y`. Jeśli podano `z`, ta funkcja zwraca `(x ** y) % z`. Jeśli podane są wszystkie trzy argumenty, muszą być liczbami całkowitymi, a `y` musi być nieujemne.

`print(value, ... , *, sep=separator, end=ending, file=outfile)`

Wyświetla zbiór wartości. Jako dane wejściowe możesz podać dowolną liczbę wartości, z których wszystkie są wyświetlane w tym samym wierszu. Argument `sep` służy do określenia

innego znaku separatora (domyślnie jest to spacja). Argument `end` określa inne zakończenie linii (domyślnie `'\n'`). Argument `file` przekierowuje dane wyjściowe do obiektu pliku.

`property([fget [, fset [, fdel [, doc]]]])`

Tworzy atrybut właściwości dla klas. `fget` to funkcja, która zwraca wartość atrybutu, `fset` ustawia wartość atrybutu, a `fdel` usuwa atrybut. `doc` dostarcza wpis dokumentacyjny. Właściwości są często określane w postaci dekoratora:

```
class SomeClass:

            x = property(doc='This is property x')
            @x.getter
            def x(self):
                print('pobieranie x')

            @x.setter
            def x(self, value):
                print('ustawianie x na', value)

            @x.deleter
            def x(self):
                print('usuwanie x')
```

`range([start,] stop [, step])`

Tworzy obiekt `range`, który reprezentuje zakres wartości całkowitych od wartości `start` do `stop`. `step` wskazuje krok i jest ustawiany na `1`, jeśli jest pominięty. Jeśli wartość `start` jest pominięta (gdy `range()` jest wywoływana z jednym argumentem), domyślnie przyjmuje wartość `0`. Ujemny krok tworzy listę liczb w kolejności malejącej.

`repr(object)`

Zwraca ciąg znaków reprezentujący obiekt. W większości przypadków zwracany ciąg znaków jest wyrażeniem, które można przekazać do funkcji `eval()` w celu odtworzenia obiektu.

`reversed(s)`

Tworzy odwrotny iterator dla sekwencji `s`. Ta funkcja działa tylko wtedy, gdy `s` definiuje metodę `__reversed__()` lub implementuje metody sekwencji `__len__()` i `__getitem__()`. Nie działa z generatorami.

`round(x [, n])`

Zaokrągla liczbę zmiennoprzecinkową `x` do najbliższej wielokrotności `10` do potęgi minus `n`. Jeśli `n` zostanie pominięte, domyślnie przyjmuje wartość `0`. Jeśli dwie wielokrotności są równie bliskie, zaokrągla się w kierunku parzystego wyboru (na przykład `0.5` jest zaokrąglane do `0.0`, a `1.5` jest zaokrąglane do `2`).

`set([items])`

Tworzy zbiór wypełniony elementami pobranymi z iterowalnych obiektów `items`. Elementy muszą być niezmienne. Jeśli w `items` zawarte są inne zbiory, te muszą być typu `frozenset`. Jeśli `items` zostanie pominięte, zwracany jest pusty zbiór.

W tabeli 10.8 przedstawiono operacje na zbiorach.

Tabela 10.8. *Operacje i metody na zbiorach*

Operacja	Opis
`s | t`	Suma zbiorów.
`s & t`	Przecięcie.
`s - t`	Różnica.
`s ^ t`	Różnica symetryczna.
`len(s)`	Zwraca liczbę elementów w `s`.
`s.add(item)`	Dodaje element `item` do `s`. Nie działa, jeśli `item` jest już w `s`.
`s.clear()`	Usuwa wszystkie elementy z `s`.
`s.copy()`	Tworzy kopię `s`.
`s.difference(t)`	Różnica zbiorów. Zwraca wszystkie elementy w `s`, ale nie w `t`.
`s.difference_update(t)`	Usuwa wszystkie elementy z `s`, które są również w `t`.
`s.discard(item)`	Usuwa element `item` z `s`. Jeśli `item` nie należy do `s`, nic się nie dzieje.
`s.intersection(t)`	Przecięcie. Zwraca wszystkie elementy, które są zarówno w `s`, jak i w `t`.
`s.intersection_update(t)`	Oblicza część wspólną `s` oraz `t` i pozostawia wynik w `s`.
`s.isdisjoint(t)`	Zwraca `True`, jeśli `s` i `t` nie mają wspólnych elementów.
`s.issubset(t)`	Zwraca `True`, jeśli `s` jest podzbiorem `t`.
`s.issuperset(t)`	Zwraca `True`, jeśli `s` jest nadzbiorem `t`.
`s.pop()`	Zwraca dowolny element zbioru i usuwa go z `s`.
`s.remove(item)`	Usuwa element `item` z `s`. Jeśli element `item` nie należy do `s`, zgłaszany jest wyjątek `KeyError`.
`s.symmetric_difference(t)`	Różnica symetryczna. Zwraca wszystkie elementy, które są w `s` lub `t`, ale nie w obu zbiorach.
`s.symmetric_difference_update(t)`	Oblicza różnicę symetryczną `s` oraz `t` i pozostawia wynik w `s`.
`s.union(t)`	Suma. Zwraca wszystkie elementy w `s` lub `t`.
`s.update(t)`	Dodaje wszystkie elementy z `t` do `s`. `t` może być innym zbiorem, sekwencją lub dowolnym obiektem obsługującym iterację.

`setattr(object, name, value)`

Ustawia atrybut obiektu `object`. `name` to ciąg znaków. To samo co `object.name = value`.

`slice([start,] stop [, step])`

Zwraca obiekt wycinka reprezentujący liczby całkowite z określonego zakresu. Obiekty wycinka są również generowane przez rozszerzoną składnię wycinka `a[i:i:k]`.

`sorted(iterable, *, key=keyfunc, reverse=reverseflag)`

Tworzy posortowaną listę z elementów `iterable`. Słowo kluczowe `key` jest funkcją jednoargumentową, która przekształca wartości przed ich porównaniem. Argument `reverse` jest flagą logiczną, która określa, czy wynikowa lista jest sortowana w odwrotnej kolejności. Argumenty `key` i `reverse` muszą być określone za pomocą słów kluczowych — na przykład `sorted(a, key=get_name)`.

`staticmethod(func)`

Tworzy metodę statyczną do użytku w klasach. Ta funkcja jest zwykle używana jako dekorator `@staticmethod`.

`str([object])`

Typ reprezentujący ciąg znaków. Jeśli podano obiekt `object`, ciąg znaków reprezentujący jego wartość jest tworzony przez wywołanie jego metody `__str__()`. Jest to ten sam ciąg, który widzisz podczas wyświetlania obiektu. Jeśli nie podano argumentu, tworzony jest pusty ciąg.

Tabela 10.9 przedstawia metody zdefiniowane na ciągach znaków.

Tabela 10.9. *Operatory i metody zdefiniowane na ciągach znaków*

Operacja	Opis
`s + t`	Łączy ciągi znaków, jeśli `t` jest ciągiem znaków.
`s * n`	Replikuje ciąg znaków, jeśli `n` jest liczbą całkowitą.
`s % x`	Formatuje ciąg znaków. `x` jest krotką.
`s[i]`	Zwraca `i`-ty element ciągu znaków.
`s[i:j]`	Zwraca wycinek.
`s[i:j:step`	Zwraca rozszerzony wycinek.
`len(s)`	Liczba elementów w `s`.
`s.capitalize()`	Zamienia pierwszy znak na wielką literę.
`s.casefold()`	Konwertuje `s` na ciąg znaków, który można wykorzystać do porównania bez wielkości liter.
`s.center(width [, pad])`	Wyśrodkowuje napis w polu o długości `width`. `pad` to znak dopełniający.
`s.count(sub [, start [, end]])`	Zlicza wystąpienia określonego podciągu.
`s.decode([encoding [, errors]])`	Dekoduje ciąg bajtów na tekst (tylko dla typu `bytes`).
`s.encode([encoding [, errors]])`	Zwraca zakodowaną wersję ciągu (tylko dla typu `str`).
`s.endswith(suffix [, start [, end]])`	Sprawdza koniec ciągu pod kątem przyrostka.
`s.expandtabs([tabsize])`	Zastępuje tabulatory spacjami.
`s.find(sub [, start [, end]])`	Znajduje pierwsze wystąpienie określonego podciągu.
`s.format(*args, **kwargs)`	Formatuje `s` (tylko dla typu `str`).
`s.format_map(m)`	Formatuje `s` z podstawieniami z mapowania `m` (tylko dla typu `str`).

Tabela 10.9. *Operatory i metody zdefiniowane na ciągach znaków (ciąg dalszy)*

Operacja	Opis
`s.index(sub [, start [, end]])`	Znajduje pierwsze wystąpienie lub błąd w określonym podciągu.
`s.isalnum()`	Sprawdza, czy wszystkie znaki są alfanumeryczne.
`s.isalpha()`	Sprawdza, czy wszystkie znaki są alfabetyczne.
`s.isascii()`	Sprawdza, czy wszystkie znaki są ASCII.
`s.isdecimal()`	Sprawdza, czy wszystkie znaki są znakami dziesiętnymi. Nie działa dla indeksu górnego, indeksu dolnego ani innych cyfr specjalnych.
`s.isdigit()`	Sprawdza, czy wszystkie znaki są cyframi. Działa dla indeksu górnego i dolnego, ale nie dla ułamka prostego.
`s.isidentifier()`	Sprawdza, czy `s` jest prawidłowym identyfikatorem Pythona.
`s.islower()`	Sprawdza, czy wszystkie znaki są małymi literami.
`s.isnumeric()`	Sprawdza, czy wszystkie znaki są numeryczne. Dopasowuje wszystkie formy znaków numerycznych, takie jak ułamki proste, cyfry rzymskie itp.
`s.isprintable()`	Sprawdza, czy wszystkie znaki są drukowalne.
`s.isspace()`	Sprawdza, czy wszystkie znaki są białymi znakami.
`s.istitle()`	Sprawdza, czy ciąg jest ciągiem tytułu (pierwsza litera każdego słowa pisana wielką literą).
`s.isupper()`	Sprawdza, czy wszystkie znaki są wielkimi literami.
`s.join(t)`	Łączy ciąg znaków `t` przy użyciu ogranicznika `s`.
`s.ljust(width [, fill])`	Wyrównuje `s` do lewej w ciągu znaków o rozmiarze `width`.
`s.lower()`	Konwertuje na małe litery.
`s.lstrip([chrs])`	Usuwa początkowe białe znaki lub znaki podane w `chrs`.
`s.maketrans(x [, y [, z]])`	Tworzy tabelę translacji dla `s.translate()`.
`s.partition(sep)`	Dzieli ciąg znaków na partycje na podstawie separatora `sep`. Zwraca krotkę (`head, sep, tail`) lub (`s, '', ''`), jeśli nie znaleziono `sep`.
`s.removeprefix(prefix)`	Zwraca `s` z usuniętym podanym przedrostkiem, jeśli jest obecny.
`s.removesuffix(suffix)`	Zwraca `s` z usuniętym przyrostkiem, jeśli jest obecny.
`s.replace(old, new [, maxreplace])`	Zastępuje podciąg.
`s.rfind(sub [, start [, end]])`	Znajduje ostatnie wystąpienie podciągu.
`s.rindex(sub [, start [, end]])`	Znajduje ostatnie wystąpienie lub zgłasza błąd.

Tabela 10.9. *Operatory i metody zdefiniowane na ciągach znaków (ciąg dalszy)*

Operacja	Opis
`s.rjust(width [, fill])`	Wyrównuje `s` do prawej w ciągu o długości `width`.
`s.rpartition(sep)`	Partycje oparte na separatorze `sep`, ale wyszukiwanie przebiega od końca ciągu.
`s.rsplit([sep [, maxsplit]])`	Dzieli ciąg od końca, używając `sep` jako separatora. `maxsplit` to maksymalna liczba podziałów do wykonania. Jeśli `maxsplit` zostanie pominięty, wynik jest identyczny z metodą `split()`.
`s.rstrip([chrs])`	Usuwa końcowe białe znaki lub znaki podane w `chrs`.
`s.split([sep [, maxsplit]])`	Dzieli łańcuch, używając `sep` jako separatora. `maxsplit` to maksymalna liczba podziałów do wykonania.
`s.splitlines([keepends])`	Dzieli łańcuch na listę linii. Jeśli `keepends` wynosi 1, zachowywane są końcowe znaki nowej linii.
`s.startswith(prefix [, start [, end]])`	Sprawdza, czy ciąg znaków zaczyna się od prefiksu.
`s.strip([chrs])`	Usuwa początkowe i końcowe białe znaki lub znaki podane w `chrs`.
`s.swapcase()`	Zamienia wielkie litery na małe i na odwrót.
`s.title()`	Zwraca wersję ciągu z literami tytułu.
`s.translate(table [, deletechars])`	Zamienia ciąg znaków za pomocą tablicy translacji znaków, usuwając znaki w `deletechars`.
`s.upper()`	Konwertuje ciąg na wielkie litery.
`s.zfill(width)`	Dopełnia ciąg zerami po lewej stronie do określonej szerokości `width`.

`sum(items [, initial])`

Oblicza sumę sekwencji elementów pobranych z iterowalnych obiektów `items`. `initial` zapewnia wartość początkową i domyślnie wynosi 0. Ta funkcja zwykle działa tylko z liczbami.

`super()`

Zwraca obiekt, który reprezentuje kolektywne nadklasy klasy, w której jest używany. Podstawowym celem tego obiektu jest wywoływanie metod w klasach bazowych. Oto przykład:

```
class B(A):
    def foo(self):
        super().foo()          # Wywołaj foo() zdefiniowane przez nadklasy
```

`tuple([items])`

Typ reprezentujący krotkę. Jeśli podano, `items` jest obiektem iterowalnym, który jest używany do wypełniania krotki. Jeśli jednak elementy są już krotką, są zwracane w niezmienionej postaci. Jeśli nie podano argumentu, zwracana jest pusta krotka.

Tabela 10.10 przedstawia metody zdefiniowane na krotkach.

Tabela 10.10. *Operatory i metody krotek*

Operacja	Opis
`s + t`	Konkatenacja, jeśli `t` jest listą.
`s * n`	Replikacja, jeśli `n` jest liczbą całkowitą.
`s[i]`	Zwraca `i`-ty element z `s`.
`s[i:j]`	Zwraca wycinek.
`s[i:j:step]`	Zwraca rozszerzony wycinek.
`len(s)`	Liczba elementów w `s`.
`s.append(x)`	Dołącza nowy element `x` na końcu `s`.
`s.count(x)`	Zlicza wystąpienia `x` w `s`.
`s.index(x [, start [, stop]])`	Zwraca najmniejsze `i`, gdzie `s[i] == x`. `start` i `stop` opcjonalnie określają początkowy i końcowy indeks wyszukiwania.

`type(object)`

Klasa bazowa wszystkich typów w Pythonie. Wywoływana jako funkcja zwraca typ obiektu `object`. Ten typ jest taki sam jak klasa obiektu. W przypadku powszechnych typów, takich jak `int`, `float` i `list`, typ będzie się odnosić do jednej z innych wbudowanych klas, takich jak `int`, `float`, `list` i tak dalej. W przypadku obiektów zdefiniowanych przez użytkownika typem jest skojarzona klasa. W przypadku obiektów związanych z wewnętrznymi elementami Pythona zazwyczaj otrzymasz odniesienie do jednej z klas zdefiniowanych w module `types`.

`vars([object])`

Zwraca tablicę symboli obiektu `object` (zwykle znajdującą się w jego atrybucie `__dict__`). Jeśli nie podano argumentu, zwracany jest słownik odpowiadający lokalnej przestrzeni nazw. Należy przyjąć, że słownik zwrócony przez tę funkcję jest tylko do odczytu. Modyfikowanie jego zawartości nie jest bezpieczne.

`zip([s1 [, s2 [, ... ]]])`

Tworzy iterator, który buduje krotki zawierające jeden element z `s1`, `s2` i tak dalej. N-ta krotka to `(s1[n], s2[n], ...)`. Wynikowy iterator zatrzymuje się po wyczerpaniu elementów w najkrótszej danej wejściowej. Jeśli nie podano argumentów, iterator nie generuje żadnych wartości.

10.2. Wyjątki wbudowane

W tej sekcji opisano wyjątki wbudowane używane do zgłaszania różnego rodzaju błędów.

10.2.1. Klasy bazowe wyjątków

Następujące wyjątki służą jako klasy bazowe dla wszystkich innych wyjątków:

`BaseException`
: Klasa główna dla wszystkich wyjątków. Wszystkie wbudowane wyjątki pochodzą z tej klasy.

`Exception`
: Klasa bazowa dla wszystkich wyjątków związanych z aplikacją. Obejmuje wszystkie wbudowane wyjątki ale bez `SystemExit`, `GeneratorExit` i `KeyboardInterrupt`. Wyjątki zdefiniowane przez użytkownika powinny dziedziczyć po `Exception`.

`ArithmeticError`
: Klasa bazowa dla wyjątków arytmetycznych, w tym `OverflowError`, `ZeroDivisionError` i `FloatingPointError`.

`LookupError`
: Klasa bazowa dla indeksowania i błędów kluczy, w tym `IndexError` i `KeyError`.

`EnvironmentError`
: Klasa bazowa dla błędów występujących poza Pythonem. Jest synonimem `OSError`.

Powyższe wyjątki nigdy nie są zgłaszane wprost. Można ich jednak użyć do wyłapania pewnych klas błędów. Na przykład poniższy kod wychwyciłby każdy rodzaj błędu liczbowego:

```
try:
    # Kilka operacji
    ...
except ArithmeticError as e:
    # Błąd matematyczny
```

10.2.2. Atrybuty wyjątków

Instancje wyjątku e mają kilka standardowych atrybutów, które mogą być przydatne do sprawdzania i (lub) manipulowania nim w niektórych aplikacjach.

`e.args`
: Krotka argumentów dostarczana podczas zgłaszania wyjątku. W większości przypadków jest to krotka jednoelementowa z łańcuchem opisującym błąd. W przypadku wyjątków `EnvironmentError` wartość jest krotką dwu- lub trzyelementową, zawierającą całkowitą liczbę błędów, komunikat o błędzie i opcjonalną nazwę pliku. Zawartość tej krotki może być przydatna, jeśli musisz odtworzyć wyjątek w innym kontekście — na przykład aby zgłosić wyjątek w innym procesie interpretera Pythona.

`e.__cause__`
: Poprzedni wyjątek dla jawnie powiązanych wyjątków.

`e.__context__`
: Poprzedni wyjątek dla niejawnie powiązanych wyjątków.

`e.__traceback__`
: Obiekt `traceback` skojarzony z wyjątkiem.

10.2.3. Predefiniowane klasy wyjątków

Aplikacje zgłaszają następujące wyjątki:

`AssertionError`
: Nieudana instrukcja asercji.

`AttributeError`
: Nieudane odniesienie do atrybutu lub przypisanie.

`BufferError`
: Oczekiwany bufor pamięci.

`EOFError`
: Koniec pliku. Generowany przez wbudowane funkcje `input()` i `raw_input()`. Należy zauważyć, że większość innych operacji wejścia-wyjścia, takich jak metody plików `read()` i `readline()`, zamiast zgłaszać wyjątek (poprzez sygnalizowanie EOF), zwracają pusty ciąg znaków.

`FloatingPointError`
: Błąd operacji zmiennoprzecinkowej. Należy zauważyć, że obsługa wyjątków zmiennoprzecinkowych jest trudnym problemem i ten wyjątek jest zgłaszany tylko wtedy, gdy Python został skonfigurowany i zbudowany w sposób, który to umożliwia. Częściej zdarza się, że błędy zmiennoprzecinkowe po cichu generują takie wyniki jak `float('nan')` lub `float('inf')`. Podklasa `ArithmeticError`.

`GeneratorExit`
: Błąd zgłaszany wewnątrz funkcji generatora, aby zasygnalizować zakończenie jego działania. Dzieje się tak, gdy generator zostanie przedwcześnie zniszczony (zanim wszystkie wartości generatora zostaną zużyte) lub zostanie wywołana metoda `close()` generatora. Jeśli generator zignoruje ten wyjątek, generator zostanie zakończony, a wyjątek zostanie po cichu zignorowany.

`IOError`
: Nieudana operacja wejścia-wyjścia. Wartość jest instancją `IOError` z atrybutami `errno`, `strerror` i `filename`. `errno` to numer błędu, `strerror` to komunikat o błędzie, a `filename` to opcjonalna nazwa pliku. Podklasa `EnvironmentError`.

`ImportError`
: Wywoływany, gdy instrukcja `import` nie może znaleźć modułu lub gdy instrukcja `from` nie może znaleźć nazwy w module.

`IndentationError`
: Błąd wcięcia. Podklasa `SyntaxError`.

`IndexError`
: Indeks dolny sekwencji poza zakresem. Podklasa `LookupError`.

`KeyError`
: Nie znaleziono klucza w mapowaniu. Podklasa `LookupError`.

`KeyboardInterrupt`
: Wywoływany, gdy użytkownik naciśnie klawisz przerwania (zwykle *Ctrl+C*).

`MemoryError`
: Naprawialny błąd braku pamięci.

`ModuleNotFoundError`
: Nie można znaleźć modułu w instrukcji `import`.

`NameError`
: Nie znaleziono nazwy w lokalnych lub globalnych przestrzeniach nazw.

`NotImplementedError`
: Niezaimplementowana funkcja. Wyjątek może być zgłaszany przez klasy bazowe, które wymagają klas pochodnych do zaimplementowania niektórych metod. Podklasa `RuntimeError`.

`OSError`
: Błąd systemu operacyjnego. Wyjątek zgłaszany przede wszystkim przez funkcje w module `os`. Następujące wyjątki są podklasami: `BlockingIOError`, `BrokenPipeError`, `ChildProcessError`, `ConnectionAbortedError`, `ConnectionError`, `ConnectionRefusedError`, `ConnectionResetError`, `FileExistsError`, `FileNotFoundError`, `InterruptedError`, `IsADirectoryError`, `NotADirectoryError`, `PermissionError`, `ProcessLookupError`, `TimeoutError`.

`OverflowError`
: Zbyt duża wartość liczby całkowitej. Ten wyjątek zwykle występuje tylko wtedy, gdy duże wartości całkowite są przekazywane do obiektów, które wewnętrznie, w ich implementacji, opierają się na liczbach maszynowych o stałej precyzji. Na przykład błąd może wystąpić w przypadku obiektów `range` lub `xrange`, jeśli określisz wartości początkowe lub końcowe, które przekraczają rozmiar 32 bitów. Podklasa `ArithmeticError`.

`RecursionError`
: Przekroczono limit rekurencji.

`ReferenceError`
: Błąd powstały w wyniku dostępu do słabej referencji po zniszczeniu bazowego obiektu (zobacz moduł `weakref`).

`RuntimeError`
: Ogólny błąd nieobjęty żadną z pozostałych kategorii.

`StopIteration`
: Zgłaszany, aby zasygnalizować koniec iteracji. Zwykle dzieje się tak w metodzie `next()` obiektu lub w funkcji generatora.

`StopAsyncIteration`
: Zgłaszany, aby zasygnalizować koniec iteracji asynchronicznej. Ma zastosowanie tylko w kontekście funkcji `async` i generatorów.

`SyntaxError`
: Błąd składni parsera. Instancje mają atrybuty `filename`, `lineno`, `offset` i `text`, których można użyć do zebrania większej ilości informacji.

`SystemError`
: Błąd wewnętrzny interpretera. Wartość jest ciągiem znaków wskazującym na problem.

`SystemExit`
: Wywoływany przez funkcję `sys.exit()`. Wartość jest liczbą całkowitą wskazującą kod powrotu. Jeśli konieczne jest natychmiastowe zamknięcie, można użyć `os._exit()`.

`TabError`
: Niespójne użycie tabulacji. Generowany, gdy Python jest uruchamiany z opcją `-tt`. Podklasa `SyntaxError`.

`TypeError`
: Występuje, gdy operacja lub funkcja zostanie zastosowana do obiektu nieodpowiedniego typu.

`UnboundLocalError`
: Odwołanie do niezdefiniowanej zmiennej lokalnej. Ten błąd występuje, jeśli następuje odwołanie do zmiennej przed jej zdefiniowaniem w funkcji. Podklasa `NameError`.

`UnicodeError`
: Błąd kodowania lub dekodowania Unicode. Podklasa `ValueError`. Następujące wyjątki to podklasy: `UnicodeEncodeError`, `UnicodeDecodeError`, `UnicodeTranslateError`.

`ValueError`
: Generowany, gdy argument funkcji lub operacji ma właściwy typ, ale nieodpowiednią wartość.

`WindowsError`
: Generowany przez nieudane wywołania systemowe w systemie Windows. Podklasa `OSError`.

`ZeroDivisionError`
: Dzielenie przez zero. Podklasa `ArithmeticError`.

10.3. Biblioteka standardowa

Python jest dostarczany ze sporą biblioteką standardową. Wiele z tych modułów zostało już wcześniej opisanych w książce. Materiały referencyjne można znaleźć na stronie *https://docs.python.org/library*. Ten materiał nie jest tutaj powtarzany.

Wymienione poniżej moduły są godne uwagi, ponieważ są ogólnie przydatne dla wielu aplikacji w Pythonie.

10.3.1. Moduł collections

Moduł `collections` dodaje do Pythona szereg dodatkowych obiektów kontenerów, które mogą być bardzo przydatne do pracy z danymi — takimi jak kolejka z podwójnym zakończeniem (`deque`) czy słowniki, które automatycznie inicjują brakujące elementy (`defaultdict`) i liczniki do tabulacji (`Counter`).

10.3.2. Moduł datetime

Moduł `datetime` to miejsce, w którym znajdziesz funkcje związane z datami i godzinami oraz ich obliczaniem.

10.3.3. Moduł itertools

Moduł `itertools` zapewnia wiele przydatnych wzorców iteracji: łączenie iteracji, iterację po elementach zbiorów, permutacje, grupowanie i podobne operacje.

10.3.4. Moduł inspect

Moduł `inspect` udostępnia funkcje do sprawdzania elementów wewnętrznych związanych z kodem, takich jak funkcje, klasy, generatory i współprogramy. Jest powszechnie używany w metaprogramowaniu przez funkcje definiujące dekoratory i podobne funkcje.

10.3.5. Moduł math

Moduł `math` udostępnia typowe funkcje matematyczne, takie jak `sqrt()`, `cos()` i `sin()`.

10.3.6. Moduł os

Moduł `os` to miejsce, w którym znajdziesz funkcje niskiego poziomu związane z systemem operacyjnym hosta: procesy, pliki, potoki, uprawnienia i podobne funkcje.

10.3.7. Moduł random

Moduł `random` udostępnia różne funkcje związane z generowaniem liczb losowych.

10.3.8. Moduł re

Moduł `re` zapewnia obsługę pracy z tekstem poprzez dopasowanie wzorców wyrażeń regularnych.

10.3.9. Moduł shutil

Moduł `shutil` posiada funkcje do wykonywania typowych zadań związanych z powłoką, takich jak kopiowanie plików i katalogów.

10.3.10. Moduł statistics

Moduł `statistics` udostępnia funkcje do obliczania typowych wartości statystycznych, takich jak średnie, mediany i odchylenie standardowe.

10.3.11. Moduł sys

Moduł `sys` zawiera różne atrybuty i metody związane ze środowiskiem wykonawczym samego Pythona. Obejmuje to opcje wiersza poleceń, standardowe strumienie wejścia-wyjścia, ścieżkę importu i podobne funkcje.

10.3.12. Moduł time

Moduł `time` to miejsce, w którym można znaleźć różne funkcje związane z czasem systemowym, takie jak odczytywanie wartości zegara systemowego, uśpienie i sprawdzanie liczby sekund procesora, które upłynęły.

10.3.13. Moduł turtle

Grafika żółwia. Wiesz, dla dzieci.

10.3.14. Moduł unittest

Moduł `unittest` zapewnia wbudowaną obsługę pisania testów jednostkowych. Sam Python jest testowany za pomocą testu jednostkowego. Jednak wielu programistów woli używać do testowania zewnętrznych bibliotek, takich jak `pytest`. To rozsądne podejście.

10.4. Podsumowanie: korzystaj z wbudowanych elementów

We współczesnym świecie, w którym istnieją setki tysięcy pakietów Pythona, programiści mogą z łatwością szukać rozwiązań małych problemów w postaci zewnętrznych zależności pakietów. Jednak Python od dawna ma niezwykle przydatny zestaw wbudowanych funkcji i typów danych. W połączeniu z modułami z biblioteki standardowej szeroki zakres typowych problemów programistycznych można często rozwiązać za pomocą tego, co zapewnia sam Python.


PROGRAM
PARTNERSKI
GRUPY HELION
1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ
Zmień swoją stronę WWW w działający bankomat!
Dowiedz się więcej i dołącz już dzisiaj!
http://program-partnerski.helion.pl
GRUPA
Helion