

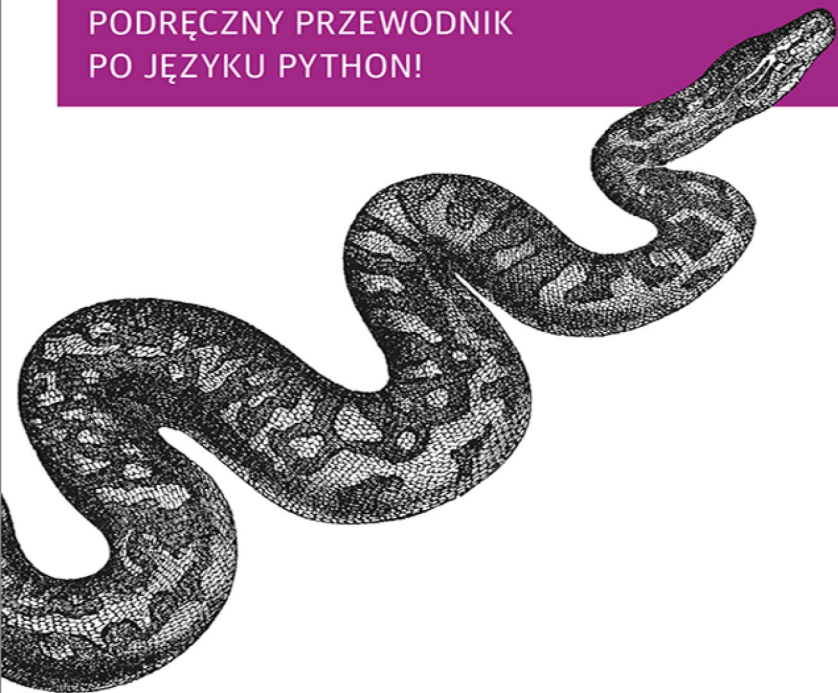
O'REILLY®

Wydanie V

Python

Leksykon kieszonkowy

PODRĘCZNY PRZEWODNIK
PO JĘZYKU PYTHON!



Helion 

Mark Lutz

Tytuł oryginału: Python Pocket Reference, Fifth Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-283-6036-5

© 2014, 2019 Helion S.A.

Authorized Polish translation of the English edition **Python Pocket Reference, 5th Edition** ISBN 9781449357016 © 2014 Mark Lutz.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek 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 Wydawnictwo HELION dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie bierze jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Wydawnictwo HELION nie ponosi również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION

ul. Kościuszki 1c, 44-100 GLIWICE

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie/pyIN5Y_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wprowadzenie	7
Konwencje	8
Opcje wiersza poleceń Pythona	9
Opcje poleceń Pythona	9
Specyfikacja programu w wierszu polecenia	11
Opcje poleceń Pythona 2.X	12
Zmienne środowiskowe	13
Zmienne operacyjne	13
Zmienne opcji wiersza poleceń	14
Python Launcher dla systemu Windows	15
Dyrektywy plikowe launchera	15
Wiersz poleceń launchera	16
Zmienne środowiskowe launchera	17
Wbudowane typy i operatory	17
Operatory i priorytet ich stosowania	17
Uwagi na temat stosowania operatorów	19
Operacje według kategorii	21
Uwagi na temat działań na sekwencjach	25
Specyficzne typy wbudowane	26
Liczby	26
Ciągi znaków	29
Łańcuchy znaków Unicode	46
Listy	50
Słowniki	56
Krotki	60
Pliki	61
Zbiory	66
Inne typy i konwersje	68

Instrukcje i ich składnia	70
Reguły składniowe	70
Reguły dotyczące nazw	72
Instrukcje	75
Instrukcja przypisania	75
Instrukcja wyrażeniowa	79
Instrukcja print	80
Instrukcja if	82
Instrukcja while	83
Instrukcja for	83
Instrukcja pass	84
Instrukcja break	84
Instrukcja continue	84
Instrukcja del	84
Instrukcja def	85
Instrukcja return	89
Instrukcja yield	89
Instrukcja global	91
Instrukcja nonlocal	91
Instrukcja import	92
Instrukcja from	95
Instrukcja class	97
Instrukcja try	99
Instrukcja raise	102
Instrukcja assert	104
Instrukcja with	104
Instrukcje w Pythonie 2.X	106
Przestrzenie nazw i reguły zasięgu	107
Nazwy kwalifikowane — przestrzenie nazw obiektów	107
Nazwy niekwalifikowane — zasięgi leksykalne	107
Zasięgi zagnieżdżone i domknięcia	109
Programowanie obiektowe	110
Klasy i egzemplarze	111
Atrybuty pseudoprywatne	112
Klasy nowego stylu	113
Formalne reguły dziedziczenia	114

Metody przeciążające operatory	118
Wszystkie typy	119
Kolekcje (sekwencje, mapy)	125
Liczby (operatory dwuargumentowe)	127
Liczby (inne działania)	130
Deskryptory	130
Menedżery kontekstu	131
Metody przeciążające operatory w Pythonie 2.X	132
Funkcje wbudowane	135
Funkcje wbudowane w Pythonie 2.X	157
Wbudowane wyjątki	163
Klasy bazowe (kategorie)	163
Wyjątki szczegółowe	165
Szczegółowe wyjątki OSError	169
Wyjątki kategorii ostrzeżeń	170
Framework ostrzeżeń	171
Wbudowane wyjątki w Pythonie 3.2	172
Wbudowane wyjątki w Pythonie 2.X	173
Wbudowane atrybuty	173
Standardowe moduły biblioteczne	174
Moduł sys	175
Moduł string	183
Funkcje i klasy modułu	183
Stałe	184
Moduł systemowy os	185
Narzędzia administracyjne	186
Stałe wykorzystywane do zapewnienia przenośności	187
Polecenia powłoki	188
Narzędzia do obsługi środowiska	190
Narzędzia do obsługi deskryptorów plików	191
Narzędzia do obsługi nazw ścieżek	194
Zarządzanie procesami	198
Moduł os.path	201
Moduł dopasowywania wzorców re	204
Funkcje modułu	204
Obiekty wyrażeń regularnych	206

Obiekty dopasowania	207
Składnia wzorców	208
Moduły utrwalania obiektów	210
Moduły dbm i shelve	212
Moduł pickle	214
Moduł GUI tkinter i narzędzia	217
Przykład użycia modułu tkinter	217
Podstawowe widgety modułu tkinter	218
Wywołania okien dialogowych	218
Dodatkowe klasy i narzędzia modułu tkinter	220
Porównanie biblioteki Tcl/Tk z modulem tkinter Pythona	220
Moduły i narzędzia do obsługi internetu	222
Inne standardowe moduły biblioteczne	224
Moduł math	225
Moduł time	225
Moduł timeit	227
Moduł datetime	228
Moduł random	228
Moduł json	228
Moduł subprocess	229
Moduł enum	230
Moduł struct	230
Moduły obsługi wątków	231
API baz danych Python SQL	232
Przykład użycia interfejsu API	233
Interfejs modułu	234
Obiekty połączeń	234
Obiekty kursora	235
Obiekty typów i konstruktory	236
Idiomy Pythona i dodatkowe wskazówki	236
Wskazówki dotyczące rdzenia języka	236
Wskazówki dotyczące środowiska	238
Wskazówki dotyczące użytkowania	239
Różne wskazówki	241
Skorowidz	243

Wprowadzenie

Python jest uniwersalnym, wieloparadygmatowym językiem programowania z otwartym dostępem do kodu źródłowego, zawierającym konstrukcje obiektowe, funkcyjne i proceduralne. Jest powszechnie używany zarówno do tworzenia samodzielnych programów, jak i aplikacji skryptowych o wielu różnych zastosowaniach. Jest jednym z najpowszechniej używanych języków programowania na świecie.

Spśród własności Pythona warto wymienić czytelność kodu i obszerną funkcjonalność bibliotek. Język został zaprojektowany z myślą o optymalizacji wydajności pracy programisty, jakości oprogramowania, zapewnieniu przenośności oprogramowania oraz integracji komponentów. Programy w Pythonie działają na większości powszechnie wykorzystywanych platform. Są obsługiwane w systemach Unix i Linux, Windows i Mac OS, a także na platformach Java, .NET, Android, iOS i wielu innych.

W *leksykonie kieszonkowym* opisano typy i instrukcje języka Python, specjalne nazwy metod, funkcje wbudowane i wyjątki, powszechnie używane standardowe moduły biblioteczne oraz inne istotne narzędzia. Podręcznik ten ma służyć jako zwarte kompendium i uzupełnienie wiadomości zawartych w innych książkach czy materiałach.

Piąte wydanie książki obejmuje wersje Pythona 3.X i 2.X. Skoncentrowano się tu głównie na Pythonie 3.X, choć opisano także różnice w stosunku do Pythona 2.X. Aktualne wydanie zostało zaktualizowane pod kątem Pythona w wersjach 3.3 i 2.7. Omówione są jednak także istotne ulepszenia, które są zapowiadane w wersji 3.4, chociaż większość treści tej książki koncentruje się na wcześniejszych oraz późniejszych wydaniach z linii 3.X oraz 2.X.

To wydanie dotyczy również wszystkich głównych implementacji Pythona — włącznie z CPython, PyPy, Jython, IronPython i Stackless.

Materiał zaktualizowano pod kątem ostatnich zmian w języku, bibliotekach i praktykach. Zmiany obejmują nowy opis MRO (ang. *Method Resolution Order*) i funkcji `super()`, formalnych algorytmów dziedziczenia, importowania, menedżerów kontekstu oraz wcięć blokowych, a także powszechnie używanych modułów bibliotecznych i narzędzi, włącznie z `json`, `timeit`, `random`, `subprocess` i `enum`. Opisano także nowy silnik uruchomieniowy dla systemu Windows.

Konwencje

W książce zastosowano następujące konwencje:

□

W formatach składni elementy w nawiasach kwadratowych są zazwyczaj opcjonalne. Nawiasy kwadratowe są również używane w niektórych elementach składni Pythona (np. w listach).

*

W formatach składni wyrażenie, za którym jest gwiazdka, może się powtarzać zero lub więcej razy. Gwiazdka jest również wykorzystywana w niektórych elementach składni Pythona (np. w mnożeniu).

a | *b*

W formatach składni elementy oddzielone poziomą kreską oznaczają alternatywę. Kreska jest również wykorzystywana w niektórych elementach składni Pythona (np. w uniach).

Kursywa

Oznacza nowe pojęcia, adresy URL, nazwy plików i narzędzi.

Czcionka o stałej szerokości

Oznacza kod, polecenia i opcje wiersza poleceń, a także nazwy modułów, funkcji, atrybutów, zmiennych i metod.

Czcionka o stałej szerokości – kursywa

W składni poleceń, wyrażeń, funkcji i metod oznacza nazwy parametrów, które można zastępować.

Funkcja()

Jeżeli nie zaznaczono inaczej, wywoływalne funkcje i metody są oznaczone końcowymi nawiasami, aby odróżnić je od innych typów atrybutów.

Patrz „Nagłówek podrozdziału”

Odniesienia do innych podrozdziałów w tej książce są oznaczone za pomocą tekstu nagłówek podrozdziału umieszczonego w cudzysłowie.

UWAGA

W tej książce oznaczenia „3.X” i „2.X” wskazują, że określony temat dotyczy wszystkich powszechnie używanych wydań w linii Pythona. Szczegółowe numery wydań są używane w odniesieniu do tematów o bardziej ograniczonym zakresie (np. „2.7” oznacza tylko 2.7). Ponieważ zmiany wprowadzone w przyszłych wersjach Pythona mogą podważyć zastosowanie określonego tematu do przyszłych wydań, warto się również zapoznać z dokumentami „What’s New” dostępnymi pod adresem <http://docs.python.org/3/whatsnew/index.html>. Można tam uzyskać informacje o wersjach Pythona wydanych po ukazaniu się tej książki.

Opcje wiersza poleceń Pythona

Wiersze poleceń służące do uruchamiania programów w Pythonie z poziomu powłoki systemowej mają następujący format:

```
python [opcja*]  
[ nazwaplikuskryptu | -c polecenie | -m moduł | - ] [arg*]
```

W tym zapisie *python* oznacza nazwę pliku wykonywalnego interpretera Pythona opisanego przez pełną ścieżkę do katalogu albo przez słowo *python* interpretowane przez powłokę systemową (np. za pośrednictwem zmiennej środowiskowej *PATH*). Opcje wiersza poleceń przeznaczone dla samego Pythona występują przed specyfikacją kodu programu, który ma być uruchomiony (*opcja*). Argumenty kodu występują za specyfikacją programu (*arg*).

Opcje poleceń Pythona

Fragment *opcja* w wierszu poleceń Pythona jest wykorzystywany przez samego Pythona. W Pythonie 3.X można zastosować dowolną spośród wymienionych poniżej opcji (różnice dotyczące wersji 2.X wymieniono w dalszej części, w podrozdziale „Opcje poleceń Pythona 2.X”):

-b

Generowanie ostrzeżeń w przypadku wywoływania funkcji *str()* z obiektami *bytes* lub *bytearray* i porównywania obiektu *bytes* lub *bytearray* z *str*. Opcja *-bb* powoduje generowanie błędów zamiast ostrzeżeń.

-B

Wyłączenie zapisywania plików kodu bajtowego *.pyc* lub *.pyo* podczas importowania.

- d Włączenie wyjścia diagnostycznego (dla programistów rdzenia Pythona).
- E Ignorowanie zmiennych środowiskowych Pythona opisanych w dalszej części tej książki (na przykład PYTHONPATH).
- h Wyświetlenie komunikatu pomocy i zakończenie działania.
- i Włączenie trybu interaktywnego po wykonaniu skryptu. Przydatne przy debugowaniu po awarii (w tzw. trybie „postmortem”). Zobacz też polecenie `pdb.pm` opisane w podręcznikach dotyczących bibliotek Pythona.
- O Optymalizacja generowanego kodu bajtowego (tworzenie i wykonywanie plików z kodem bajtowym *.pyo*). W bieżącej wersji nieznacznie poprawia wydajność.
- OO Działa podobnie jak opcja -O, opisana wcześniej, ale dodatkowo usuwa ciągi dokumentacyjne (ang. *docstring*) z kodu bajtowego.
- q Wyłączenie wyświetlania komunikatu z informacją o wersji i prawach autorskich dla interaktywnego uruchamiania (począwszy od Pythona 3.2).
- s Wyłączenie dodawania katalogu użytkownika do ścieżki wyszukiwania modułów `sys.path`.
- S Wyłączenie dedukowania „ośrodka importu” podczas inicjalizacji.
- u Wymuszenie braku buforowania i trybu binarnego dla urządzeń *stdout* i *stderr*.
- v Wyświetlenie każdorazowo podczas inicjowania modułu komunikatu zawierającego lokalizację, z której moduł jest ładowany. Aby uzyskać obszerniejszy wynik, należy powtórzyć tę flagę.

-V

Wyświetlenie numeru wersji Pythona i zakończenie działania aplikacji.

-W *arg*

Opcja ta steruje ostrzeżeniami. Argument *arg* ma postać *akcja:komunikat:kategoria:moduł:numerwiersza*. Więcej informacji można znaleźć poniżej w podrozdziałach „Framework ostrzeżeń” oraz „Wyjątki kategorii ostrzeżeń”, a także w podręczniku *Python Library Reference* (dostępnym pod adresem <http://www.python.org/doc/>), w rozdziale dotyczącym ostrzeżeń.

-x

Pominięcie pierwszego wiersza kodu źródłowego. Użycie tej opcji umożliwia wykorzystanie nieunixowskiej formy polecenia `#!cmd`.

-X *opcja*

Ustawia opcję specyficzną dla implementacji (od Pythona 3.2). Obsługiwane wartości opcji można znaleźć w dokumentacji implementacji.

Specyfikacja programu w wierszu polecenia

Kod do uruchomienia i argumenty wiersza polecenia przeznaczone do wysłania są w wierszach poleceń Pythona określone w następujący sposób:

nazwaplikuskryptu

Określa plik skryptu Pythona, z którego ma być uruchomiony program główny (np. *python main.py*). Nazwa skryptu może być określona za pomocą bezwzględnej lub względnej ścieżki do pliku (względem „.”) i jest dostępna w zmiennej `sys.argv[0]`. Na niektórych platformach wiersze poleceń mogą być również pozbawione składnika *python*, jeśli zaczynają się od nazwy pliku skryptu i nie zawierają opcji samego Pythona.

-c *polecenie*

Określa polecenie Pythona (w postaci ciągu znaków) do wykonania (np. *python -c "print('spam' * 8)"* powoduje uruchomienie instrukcji `print`). Zmienna `sys.argv[0]` jest ustawiana na wartość `-c`.

-m *moduł*

Uruchamia skrypt będący modulem bibliotecznym — wyszukuje *moduł* w ścieżce `sys.path` i uruchamia go jako plik najwyższego poziomu (np. polecenie *python -m pdb s.py* uruchamia moduł `pdb` debuggera Pythona znajdujący się w katalogu standardowej

biblioteki z argumentem `s.py`); *moduł* może być również nazwą pakietu (np. `idlelib.idle`). Zmienna `sys.argv[0]` przyjmuje wartość nazwy pełnej ścieżki modułu.

–

Odczytuje polecenia Pythona z urządzenia *stdin* (domyślnie). Jeśli urządzeniem *stdin* jest *tty* (urządzenie interaktywne), włącza tryb interaktywny. Zmienna `sys.argv[0]` jest ustawiana na wartość `-`.

*arg**

Wskazuje, że do pliku skryptu lub polecenia są przekazywane dodatkowe elementy (wartości te są dołączane do wbudowanej listy ciągów znaków `sys.argv[1:]`).

W przypadku braku elementów *nazwaplikuskryptu*, *polecenie* lub *moduł* Python wchodzi do trybu interaktywnego i odczytuje polecenia z urządzenia *stdin* (w roli urządzenia wejściowego wykorzystuje narzędzie GNU *readline*, o ile jest ono zainstalowane). Zmienna `sys.argv[0]` jest ustawiona na `' '` (pusty ciąg znaków), o ile Python nie został wywołany z opcją `-` z listy zamieszczonej powyżej.

Oprócz wykorzystywania tradycyjnych wierszy poleceń z poziomu powłoki systemowej można również uruchamiać programy Pythona, klikając nazwy ich plików z poziomu interfejsu GUI eksploratora. Można też wywoływać funkcje standardowej biblioteki Pythona (np. `os.popen()`) albo używać opcji menu uruchamiających programy w środowiskach IDE, takich jak IDLE, Komodo, Eclipse, NetBeans itp.

Opcje poleceń Pythona 2.X

Python 2.X ma taki sam format wiersza polecenia, ale nie obsługuje opcji `-b`, która jest związana ze zmianami w typie `string` wprowadzonymi w Pythonie 3.X, ani ostatnio dodanych do Pythona 3.X opcji `-q` i `-X`. W wersjach 2.6 i 2.7 obsługuje dodatkowe opcje (niektóre mogły być wymienione wcześniej):

`-t` oraz `-tt`

Generuje ostrzeżenia w przypadku niespójnego użycia mieszanki spacji i tabulacji we wcięciach. Opcja `-tt` zamiast ostrzeżeń generuje błędy. W Pythonie 3.X takie mieszanie spacji z tabulacjami zawsze jest traktowane jako błąd składniowy (patrz też „Reguły składniowe”).

-Q

Opcje związane z dzieleniem: `-Qold` (domyślna), `-Qwarn`, `-Qwarnall` oraz `-Qnew`. Opcje te zostały uwzględnione w nowym mechanizmie dzielenia wprowadzonym w Pythonie 3.X (zobacz też „Uwagi na temat stosowania operatorów”).

-3

Generuje ostrzeżenia dotyczące dowolnych niezgodności kodu z Pythonem 3.X, które nie mogą być w prosty sposób usunięte przez standardowe narzędzia instalacyjne Pythona 2 i 3.

-R

Włącza pseudolosowe ziarno do tworzenia wartości skrótów różnych typów, tak by były nieprzewidywalne pomiędzy kolejnymi wywołaniami interpretera. Ma to na celu obronę przed atakami *denial-of-service*. Nowość w Pythonie 2.6.8. Przełącznik ten jest również dostępny w Pythonie 3.X, począwszy od wersji 3.2.3, dla zapewnienia zgodności wstecz, ale losowość skrótów jest domyślnie włączona od wersji 3.3.

Zmienne środowiskowe

Zmienne środowiskowe to ustawienia systemowe dotyczące wielu programów i używane do globalnej konfiguracji.

Zmienne operacyjne

Poniższa lista zawiera najważniejsze skonfigurowane przez użytkownika zmienne środowiskowe związane z zachowaniem skryptów:

PYTHONPATH

Aktualizuje domyślną ścieżkę wyszukiwania importowanych plików modułów. Format zmiennej jest taki sam jak format zmiennej powłoki `PATH` — nazwy katalogów oddzielone od siebie dwukropkami (w systemie Windows średnikami). Podczas importowania modułów Python wyszukuje odpowiedni plik lub katalog w każdym z wymienionych katalogów — od lewej do prawej. Zmienna jest włączona w ustawienie `sys.path` — pełna ścieżka przeszukiwania modułów dotycząca występujących najbardziej z lewej strony komponentów na bezwzględnej liście importów — za katalogiem skryptów, a przed katalogami standardowych bibliotek. Zobacz też `sys.path` w podrozdziałach „Moduł `sys`” oraz „Instrukcja import”.

PYTHONSTARTUP

Jeśli zmienna ta zostanie ustawiona na nazwę pliku, z którego można czytać polecenia, to przed wyświetleniem pierwszego symbolu zachęty w trybie interaktywnym zostaną uruchomione polecenia Pythona znajdujące się w tym pliku.

PYTHONHOME

Ustawienie tej zmiennej spowoduje, że jej wartość będzie użyta w roli alternatywnego katalogu prefiksów dla modułów bibliotecznych (alternatywą są zmienne `sys.prefix` i `sys.exec_prefix`). W domyślnej ścieżce wyszukiwania modułów wykorzystywana jest zmienna `sys.prefix/lib`.

PYTHONCASEOK

Jeśli zmienna jest ustawiona, Python ignoruje wielkość liter w instrukcjach importowania (obecnie tylko w systemach Windows i OS X).

PYTHONIOENCODING

Zmienna ma format *nazwakodowania[:proceduraobsługibłędów]* i przesłania domyślne kodowanie Unicode (oraz opcjonalnie procedurę obsługi błędów) używane do transferu tekstu dla strumieni *stdin*, *stdout* oraz *stderr*. Ustawienie to może być wymagane dla tekstów niebędących tekstami ASCII dla niektórych powłok (jeśli drukowanie się nie powiedzie, można spróbować ustawić tę zmienną na wartość `utf8` lub `other`).

PYTHONHASHSEED

Ustawienie tej zmiennej na `random` powoduje, że w roli wartości używanej jako ziarno dla skrótów obiektów `str`, `byte` i `datetime` stosowana jest losowa wartość. Zmienna ta może być również ustawiona na wartość liczby całkowitej z zakresu `0...4,294,967,295`, aby uzyskać wartości skrótów z przewidywalnym ziarnem (w wersjach 3.2.3 i 2.6.8 Pythona).

PYTHONFAULTHANDLER

Ustawienie tej zmiennej powoduje, że Python w czasie uruchamiania rejestruje procedury obsługi do generowania zrzutów dla krytycznych błędów sygnałów (od Pythona 3.3; zmienna jest odpowiednikiem ustawienia `-X faulthandler`).

Zmienne opcji wiersza poleceń

Wymienione poniżej zmienne środowiskowe są synonimami niektórych opcji wiersza poleceń Pythona (zobacz „Opcje poleceń Pythona”):

PYTHONDEBUG

Jeśli zmienna nie jest pusta, to ma takie samo działanie jak opcja -d.

PYTHONDONTWRITEBYTECODE

Jeśli zmienna nie jest pusta, to ma takie samo działanie jak opcja -B.

PYTHONINSPECT

Jeśli zmienna nie jest pusta, to ma takie samo działanie jak opcja -i.

PYTHONNOUSERSITE

Jeśli zmienna nie jest pusta, to ma takie samo działanie jak opcja -s.

PYTHONOPTIMIZE

Jeśli zmienna nie jest pusta, to ma takie samo działanie jak opcja -O.

PYTHONUNBUFFERED

Jeśli zmienna nie jest pusta, to ma takie samo działanie jak opcja -u.

PYTHONVERBOSE

Jeśli zmienna nie jest pusta, to ma takie samo działanie jak opcja -v.

PYTHONWARNINGS

Jeśli zmienna nie jest pusta, to ma takie samo działanie jak opcja -W z tą samą wartością. Zmienna pozwala także na przekazanie ciągu rozdzielonego przecinkami jako odpowiednika wielu opcji -W (dostępna od Pythona w wersjach 3.2 i 2.7).

Python Launcher dla systemu Windows

W systemie Windows (wyłącznie), począwszy od Pythona 3.3, instalowane jest narzędzie do uruchamiania skryptów (tzw. launcher). Jest ono dostępne również osobno dla wcześniejszych wersji. Narzędzie to składa się z plików wykonywalnych *py.exe* (dla konsoli) oraz *pyw.exe* (działający w środowisku GUI). Pliki te można wywoływać bez ustawień PATH; są one zarejestrowane do uruchamiania plików Pythona za pośrednictwem asocjacji z nazwami plików. Za ich pomocą można wybierać wersje Pythona na trzy sposoby: za pomocą dyrektyw w stylu Uniksa `#!` umieszczanych na początku skryptów, przy użyciu argumentów wiersza polecenia oraz poprzez konfigurowalne wartości domyślne.

Dyrektywy plikowe launchera

Launcher rozpoznaje wiersze `#!` umieszczone na początku plików skryptów. Są w nich zapisane wersje Pythona w jednym z formatów wymienionych poniżej. Symbol `*` może przyjąć jedną z następujących

wartości: *pusty ciąg znaków* oznacza wersję domyślną (obecnie 2, jeśli taka wersja jest zainstalowana; ma to podobne działanie do ominięcia wiersza `#!`); *numer głównej wersji* (np. 3) do uruchomienia najnowszej zainstalowanej wersji; *kompletna specyfikacja główny.pomocniczy*, opcjonalnie z przyrostkiem -32 dla instalacji 32-bitowych (np. 3.1-32):

```
#!/usr/bin/env python*
#!/usr/bin/python*
#!/usr/local/bin/python*
#!/python*
```

Dowolne argumenty Pythona (programu *python.exe*) mogą być podane na końcu wiersza. Od Pythona 3.4 i w wersjach późniejszych w przypadku wierszy `#!`, w których występuje sama nazwa python bez podanego jawnie numeru wersji, może być przeszukiwana zmienna środowiskowa PATH.

Wiersz poleceń launchera

Launcher może również zostać wywołany z poziomu powłoki systemowej za pomocą wiersza polecenia w następującej postaci:

```
py [pyarg] [pythonarg*] script.py [scriptarg*]
```

Uogólniając, wszystko, co może wystąpić w poleceniu python za członem *python*, może się również pojawić za opcjonalnym członem *pyarg* w poleceniu py i zostanie w niezmienionej postaci przekazane do procesu Pythona. Obejmuje to specyfikacje programu w formie -m, -c oraz - (patrz „Opcje wiersza poleceń Pythona”).

Opcjonalny składnik *pyarg* launchera akceptuje argumenty w pokazanych niżej formach. Przypomina to składnię fragmentu oznaczonego * w wierszu `#!`:

```
-2      Uruchamia najnowszą zainstalowaną wersję 2.X
-3      Uruchamia najnowszą zainstalowaną wersję 3.X
-X.Y    Uruchamia określoną wersję (X może mieć wartość 2 bądź 3)
-X.Y-32 Uruchamia podaną wersję 32-bitową
```

Jeśli argumenty występują zarówno w wierszu polecenia, jak i w skryptach, w wierszu `#!`, to argumenty wiersza poleceń mają pierwszeństwo przed argumentami występującymi w wierszach `#!`. Zgodnie z instalacją wiersze `#!` mogą być stosowane w dodatkowych kontekstach (np. po kliknięciach ikon).

Zmienne środowiskowe launchera

Launcher interpretuje również opcjonalne ustawienia w zmiennych środowiskowych. Można je wykorzystać do spersonalizowania wyboru wersji w przypadkach domyślnych lub częściowych (np. kiedy w wierszu `#!` lub argumentie `py` nie została określona wersja lub jest tylko wersja główna):

<code>PY_PYTHON</code>	<i>Wersja wykorzystywana w przypadkach domyślnych (w pozostałych przypadkach 2)</i>
<code>PY_PYTHON3</code>	<i>Wersja wykorzystywana we fragmentach dotyczących Pythona 3 (np. 3.2)</i>
<code>PY_PYTHON2</code>	<i>Wersja wykorzystywana we fragmentach dotyczących Pythona 2 (np. 2.6)</i>

Te ustawienia są wykorzystywane tylko przez pliki wykonywalne launchera. Nie są stosowane, kiedy *python* jest wywoływany bezpośrednio.

Wbudowane typy i operatory

Operatory i priorytet ich stosowania

Operatory wyrażeniowe dostępne w Pythonie zestawiono w tabeli 1. Operatory zebrane w dalszych wierszach tej tabeli mają wyższy priorytet. Ma to znaczenie w przypadku stosowania ich w wyrażeniach zawierających wiele operatorów bez nawiasów.

Wyrażenia atomowe i dynamiczne określanie typów

Zastępowalne wyrażenia X , Y , Z , i , j i k w tabeli 1. mogą być:

- *nazwami zmiennych* — zastępowanymi przez ostatnio przypisaną wartość;
- *wyrażeniami literalnymi* — zdefiniowanymi w podrozdziale „Specyficzne typy wbudowane”;
- *wyrażeniami zagnieżdżonymi* — pobranymi z dowolnego wiersza z tej tabeli, czasami w nawiasach.

W przypadku *zmiennych* w Pythonie stosowany jest *dynamiczny model typów* — typy nie są deklarowane, a zmienne są tworzone w chwili przypisania. Wartościami zmiennych są referencje do obiektów. Zmienne mogą się odnosić do dowolnych typów obiektowych i muszą być przypisane przed wystąpieniem w wyrażeniach, ponieważ nie mają wartości domyślnej. W nazwach zmiennych wielkość liter zawsze ma znaczenie

(patrz „Reguły nazewnictwa”). *Obiekty*, do których odwołują się zmienne, są automatycznie tworzone i niszczone, jeśli nie są już potrzebne, przez *mechanizm odśmieciania* (ang. *garbage collector*). W implementacji CPython mechanizm ten wykorzystuje zliczanie referencji.

Zastępowalny człon *attr* w tabeli 1. musi być literalną (bez cudzysłówów i apostrofów) nazwą atrybutu; *args1* oznacza formalną listę argumentów zgodną z definicją opisaną w podrozdziale „Instrukcja def”, *args2* to lista argumentów wejściowych omówionych w podrozdziale „Instrukcja wyrażeniowa”, natomiast literal *...* jest wyrażeniem atomowym (wyłącznie w wersji 3.X).

Składnię literalów opisujących listy składane i struktury danych (krotki, listy, słowniki i zbiory) podane w ostatnich trzech wierszach tabeli 1. zdefiniowano w podrozdziale „Specyficzne typy wbudowane”.

Tabela 1. Operatory wyrażeniowe dostępne w Pythonie 3.X i priorytet ich stosowania

Operator	Opis
yield X	Wynik funkcji generatora (zwraca wartość send())
lambda args1: X	Operator tworzenia funkcji anonimowych (w wyniku wywołania zwraca X)
X if Y else Z	Operator trójargumentowy (wyrażenie X będzie obliczone tylko wtedy, gdy Y ma wartość true)
X or Y	Logiczna operacja OR: wyrażenie Y będzie obliczone tylko wtedy, gdy X ma wartość false
X and Y	Logiczna operacja AND: wyrażenie Y będzie obliczone tylko wtedy, gdy X ma wartość true
not X	Logiczna negacja
X in Y, X not in Y	Członkostwo: iteratory, zbiory
X is Y, X is not Y	Testy tożsamości obiektów
X < Y, X <= Y, X > Y, X >= Y	Porównywanie wielkości, ustawianie podzbiorów i nadzbiorów
X == Y, X != Y	Operatory równości
X Y	Bitowa operacja OR, unia zbiorów
X ^ Y	Bitowa operacja XOR, różnica symetryczna zbiorów
X & Y	Bitowa operacja AND, część wspólna zbiorów
X << Y, X >> Y	Przesunięcie X w lewo (prawo) o Y bitów
X + Y, X - Y	Dodawanie (konkatenacja), odejmowanie (różnica zbiorów)
X * Y, X % Y, X / Y, X // Y	Mnożenie (powtarzanie), reszta z dzielenia (formatowanie), dzielenie, dzielenie całkowite

Tabela 1. Operatory wyrażeniowe dostępne w Pythonie 3.X i priorytet ich stosowania — ciąg dalszy

Operator	Opis
<code>-X, +X</code>	Jednoargumentowa negacja, tożsamość
<code>~X</code>	Bitowa operacja NOT (inwersja)
<code>X ** Y</code>	Potęgowanie
<code>X[i]</code>	Indeksowanie (sekwencje, mapowanie, inne)
<code>X[i:j:k]</code>	Rozcinanie (ang. <i>slicing</i>) — wszystkie trzy granice opcjonalne
<code>X(args2)</code>	Wywołanie (funkcji, metody, klasy, innego obiektu wykonywalnego)
<code>X.attr</code>	Referencja atrybutu
<code>(...)</code>	Krotka, wyrażenie, wyrażenie generatora
<code>[...]</code>	Lista (lista składana)
<code>{...}</code>	Słownik, zbiór (słownik składany)

Uwagi na temat stosowania operatorów

- W Pythonie 2.X nierówność wartości można zapisać jako `X != Y` bądź `X <> Y`. W Pythonie 3.X tę drugą opcję usunięto, ponieważ jest nadmiarowa.
- W Pythonie 2.X wyrażenie ujęte w lewe apostrofy ``X`` działa tak samo jak `repr(X)` i konwertuje obiekty na postać ciągów do wyświetlenia. W Pythonie 3.X do tego celu służą czytelniejsze funkcje wbudowane: `str()` oraz `repr()`.
- Zarówno w Pythonie 3.X, jak i 2.X wyrażenie *dzielenia całkowitego* `X // Y` zawsze obcina resztę ułamkową i zwraca wynik całkowity.
- Wyrażenie `X / Y` w wersji 3.X realizuje *dzielenie rzeczywiste* (zawsze utrzymuje resztę w zmiennoprzecinkowym wyniku), natomiast w wersji 2.X *dzielenie klasyczne* (obcina resztę w przypadku liczb całkowitych), o ile w wersji 2.X nie włączono obsługi dzielenia rzeczywistego za pomocą instrukcji `from __future__ import division` lub przy użyciu opcji Pythona `-Qnew`.
- Konstrukcję `[...]` wykorzystuje się w odniesieniu do literalów listowych i wyrażeń z listami składanymi (ang. *list comprehension*). W drugim przypadku wykonywana jest pętla, w której wyniki wyrażenia są zbierane do nowej listy.

- Konstrukcję (...) wykorzystuje się w odniesieniu do krotek i wyrażeń, a także do wyrażeń generatora — rodzaju list składanych, które nie budują listy wyników, tylko tworzą wyniki na żądanie. We wszystkich trzech konstrukcjach czasami można pominąć nawiasy.
- Konstrukcja {...} jest używana do literałów słownikowych. W Pythonie 3.X i 2.7 jest ona również wykonywana w odniesieniu do literałów opisujących zbiory oraz zbiorów i słowników składanych; w wersji 2.6 i wersjach wcześniejszych do tego samego celu służy instrukcja `set()` oraz instrukcje pętli.
- Instrukcja `yield` oraz trójargumentowe wyrażenia wyboru `if/else` są dostępne w Pythonie 2.5 i w wersjach późniejszych. Pierwsza z nich zwraca argumenty funkcji `send()` w generatorach; druga to skrót wielowierszowej instrukcji `if`. Jeśli instrukcja `yield` nie występuje samotnie po prawej stronie instrukcji przypisania, wymaga użycia nawiasów.
- Operatory porównań można łączyć w łańcuchy: wyrażenie $X < Y < Z$ daje ten sam wynik co $X < Y$ i $Y < Z$, ale w postaci łańcucha podwyrażenie Y jest obliczane tylko raz.
- Wyrażenie rozcinające $X[i:j:k]$ jest równoważne indeksowaniu wykonywanemu dla obiektu `slice`: $X[\text{slice}(i, j, k)]$.
- W Pythonie 2.X jest dopuszczalne porównywanie wielkości mieszanych typów — konwertowanie liczb na wspólny typ, a także porządkowanie innych typów mieszanych zgodnie z nazwą typu. W Pythonie 3.X nienumeryczne porównywanie wielkości mieszanych typów nie jest dozwolone i powoduje wyjątki. Dotyczy to również sortowania przez obiekty proxy.
- Porównywanie wielkości dla słowników również nie jest już obsługiwane w Pythonie 3.X (choć testy równości są); jednym z dostępnych rozwiązań w Pythonie 3.X jest konstrukcja `sorted(dict.items())`.
- Wyrażenia wywołań umożliwiają stosowanie argumentów pozycyjnych oraz argumentów kluczowych. Argumenty te mogą być liczbami o dowolnej wielkości. Więcej informacji na temat składni wywołań można znaleźć w podrozdziałach „Instrukcja wyrażeniowa” oraz „Instrukcja `def`”.
- W Pythonie 3.X dozwolone jest stosowanie wielokropka (za pomocą literału `...` lub wbudowanej nazwy `Ellipsis`) w roli atomowego

wrażenia w dowolnym miejscu kodu źródłowego. Operator ten może być alternatywą dla instrukcji `pass`, a w niektórych kontekstach dla instrukcji `None` (np. pominięta treść funkcji, inicjalizacje zmiennych niezależne od typu).

- W Pythonie 3.5 oraz w wersjach późniejszych składnia z gwiazdką `*X` i `**X` w literałach opisujących struktury danych i konstrukcje składane *być może* zostanie uogólniona. Ma ona powodować rozpakowywanie kolekcji do postaci pojedynczych elementów, podobnie jak działa to obecnie w wywołaniach funkcji. Więcej informacji na ten temat można znaleźć w podrozdziale „Instrukcja przypisania”.

Operacje według kategorii

W tym podrozdziale dla zachowania zwięzłości pominięto końcowe nawiasy z nazw metod `__X__`. Wszystkie typy wbudowane obsługują *porównywanie* oraz *operacje logiczne* wymienione w tabeli 2. (choć Python 3.X nie obsługuje porównań wielkości dla słowników lub mieszanych typów nienumerycznych).

Wartość `true` typu `Boolean` oznacza dowolną liczbę niezerową lub dowolny niepusty obiekt kolekcji (listę, słownik itp.). Wszystkie obiekty mają wartość `Boolean`. Wbudowanym nazwom `True` i `False` są standardowo przypisywane wartości `true` i `false`. Wartości te zachowują się jak liczby całkowite 1 i 0 o niestandardowych formatach wyświetlania. Specjalny obiekt `None` ma wartość `false` i występuje w Pythonie w różnych kontekstach.

Operacje porównań zwracają wartości `True` lub `False`. W obiektach złożonych w celu obliczenia wyniku w miarę potrzeb są stosowane rekurencyjnie.

Operatory logiczne `and` i `or` zatrzymują obliczenia, kiedy znany jest wynik operacji, i zwracają jeden z obiektów będących operandami (z lewej lub prawej strony) — ich wartość `Boolean` daje wynik operacji.

W tabelach od 3. do 6. zdefiniowano operacje wspólne dla typów w trzech głównych kategoriach: *sekwencje* porządkowane pozycyjnie), *odzworowania* (dostęp za pomocą klucza) i *liczby* (wszystkie typy numeryczne), a także operacje dostępne dla typów *mutowalnych* (modyfikowalnych) Pythona. Poza tym większość typów eksportuje dodatkowe operacje specyficzne dla typu (tzn. metody). Opisano je w podrozdziale „Specyficzne typy wbudowane”.

Tabela 2. Operacje porównań i operacje logiczne

Operator	Opis
$X < Y$	Ścisłe mniejszy niż ¹
$X \leq Y$	Mniejszy lub równy
$X > Y$	Ścisłe większy niż
$X \geq Y$	Większy lub równy
$X == Y$	Równość (ta sama wartość)
$X != Y$	Różny (odpowiednik działania $X \neq Y$ dostępnego wyłącznie w Pythonie 2.X) ²
$X \text{ is } Y$	Ten sam obiekt
$X \text{ is not } Y$	Zanegowana tożsamość obiektu
$X < Y < Z$	Porównania łańcuchowe
<code>not X</code>	Jeśli X ma wartość fa l se, działanie zwraca True, w przeciwnym razie Fa l se
$X \text{ or } Y$	Jeśli X ma wartość fa l se, działanie zwraca Y , w przeciwnym razie X
$X \text{ and } Y$	Jeśli X ma wartość fa l se, działanie zwraca X , w przeciwnym razie Y

Tabela 3. Działania na sekwencjach (ciągach znaków, listach, krotkach, bajtach, tablicach bajtowych)

Działanie	Opis	Metoda klasy
$X \text{ in } S$ $X \text{ not in } S$	Test członkostwa	<code>__contains__</code> , <code>__iter__</code> , <code>__getitem__</code> ³
$S1 + S2$	Konkatenacja	<code>__add__</code>
$S * N, N * S$	Repetycja	<code>__mul__</code>
$S[i]$	Indeksowanie według przesunięcia	<code>__getitem__</code>

¹ W przypadku implementowania wyrażeń porównawczych warto się zapoznać zarówno z metodami klas tzw. bogatych porównań dostępnych w Pythonie 3.X i 2.X (np. `__lt__` dla klasy `<`), jak i z metodą ogólną `__cmp__`, opisaną w podrozdziale „Metody przeciążające operatory”.

² W Pythonie 2.X zarówno operator `!=`, jak i `<>` oznaczają różność wartości. W wersji 2.X preferowany jest operator `!=`, natomiast w wersji 3.X tylko on jest dostępny. Operator `is` służy do przeprowadzania testów tożsamości; operator `==` realizuje porównywanie wartości, w związku z czym jest bardziej uniwersalny.

³ Więcej informacji na temat tych metod można znaleźć w podrozdziale „Protokół iteracji”. Metoda `__contains__`, jeśli zostanie zdefiniowana, jest metodą bardziej preferowaną niż `__iter__`, a metoda `__iter__` ma wyższe preferencje niż `__getitem__`.

Tabela 3. Działania na sekwencjach (ciągach znaków, listach, krotkach, bajtach, tablicach bajtowych) — ciąg dalszy

Działanie	Opis	Metoda klasy
$S[i:j], S[i:j:k]$	Wycinanie: elementy w S od przesunięcia i do $j-1$ z opcjonalnym krokiem k	<code>__getitem__</code> ⁴
<code>len(S)</code>	Rozmiar	<code>__len__</code>
<code>min(S), max(S)</code>	Element minimalny (maksymalny)	<code>__iter__</code> , <code>__getitem__</code>
<code>iter(S)</code>	Protokół iteracji	<code>__iter__</code>
<code>for X in S:</code> , <code>[expr for X in S]</code> , <code>map(func, S)</code> itd.	Iteracja (wszystkie konteksty)	<code>__iter__</code> , <code>__getitem__</code>

Tabela 4. Działania na sekwencjach mutowalnych (listach, tablicach bajtowych)

Działanie	Opis	Metoda klasy
$S[i] = X$	Przypisanie indeksu: modyfikacja elementu pod istniejącym indeksem i , tak aby zawierał odwołanie do X	<code>__setitem__</code>
$S[i:j] = I$, $S[i:j:k] = I$	Przypisanie wycinka: elementy w S od przesunięcia i do $j-1$ z opcjonalnym krokiem k (może być pusty) są zastępowane przez wszystkie elementy z iteratora I	<code>__setitem__</code>
<code>del S[i]</code>	Usunięcie indeksu	<code>__delitem__</code>
<code>del S[i:j]</code> , <code>del S[i:j:k]</code>	Usunięcie wycinka	<code>__delitem__</code>

Tabela 5. Działania odwzorowań (słowniki)

Działanie	Opis	Metoda klasy
$D[k]$	Indeksowanie według klucza	<code>__getitem__</code>
$D[k] = X$	Przypisanie klucza: modyfikacja lub utworzenie elementu dla klucza k , tak by zawierał odwołanie do X	<code>__setitem__</code>

⁴ W Pythonie 2.X można również zdefiniować metody `__getslice__`, `__setslice__` i `__delslice__`, które służą do obsługi operacji podziału. W wersji 3.X metody te zostały usunięte. Zastąpiono je mechanizmem przekazywania obiektów `slice` do odpowiedników bazujących na elementach. Obiektów `slice` (tzw. wycinków) można używać jawnie w wyrażeniach indeksujących zamiast ograniczeń $i:j:k$.

Tabela 5. Działania odwzorowań (słowniki) — ciąg dalszy

Działanie	Opis	Metoda klasy
<code>del D[k]</code>	Usunięcie elementu na podstawie klucza	<code>__delitem__</code>
<code>len(D)</code>	Rozmiar (liczba kluczy)	<code>__len__</code>
<code>k in D</code>	Test członkostwa ⁵	Jak w tabeli 3.
<code>k not in D</code>	Odwrotność <code>k</code> w <code>D</code>	Jak w tabeli 3.
<code>iter(D)</code>	Obiekt iteratora dla kluczy <code>D</code>	Jak w tabeli 3.
<code>for k in D: itd.</code>	Iteracje według kluczy w <code>D</code> (wszystkie konteksty iteracyjne)	Jak w tabeli 3.

Tabela 6. Działania na liczbach (wszystkie typy liczbowe)

Działanie	Opis	Metoda klasy
<code>X + Y, X - Y</code>	Dodawanie, odejmowanie	<code>__add__</code> , <code>__sub__</code>
<code>X * Y, X / Y,</code> <code>X // Y, X % Y</code>	Mnożenie, dzielenie, dzielenie całkowite, reszta z dzielenia	<code>__mul__</code> , <code>__truediv__</code> ⁶ , <code>__floordiv__</code> , <code>__mod__</code>
<code>-X, +X</code>	Negacja, tożsamość	<code>__neg__</code> , <code>__pos__</code>
<code>X Y, X & Y,</code> <code>X ^ Y</code>	Bitowe operacje OR, AND, XOR (liczby całkowite)	<code>__or__</code> , <code>__and__</code> , <code>__xor__</code>
<code>X << N, X >> N</code>	Bitowe przesunięcie w lewo, bitowe przesunięcie w prawo o <code>N</code> pozycji (liczby całkowite)	<code>__lshift__</code> , <code>__rshift__</code>
<code>~X</code>	Bitowa negacja (liczby całkowite)	<code>__invert__</code>
<code>X ** Y</code>	Podniesienie <code>X</code> do potęgi <code>Y</code>	<code>__pow__</code>
<code>abs(X)</code>	Wartość bezwzględna	<code>__abs__</code>
<code>int(X)</code>	Konwersja na liczbę całkowitą ⁷	<code>__int__</code>

⁵ W Pythonie 2.X przynależność kluczy można kodować również jako `D.has_key(k)`. Ten sposób usunięto w Pythonie 3.X i zastąpiono go wyrażeniami. Ogólnie rzecz biorąc, wyrażenia są również preferowane w Pythonie w wersji 2.X. Patrz „Słowniki”.

⁶ Operator `/` wywołuje metodę `__truediv__` w Pythonie 3.X, ale metodę `__div__` w Pythonie 2.X, o ile rzeczywiste dzielenie nie jest włączone. Informacje na temat semantyki dzielenia można znaleźć w podrozdziale „Uwagi na temat stosowania operatorów”.

⁷ W Pythonie 2.X wbudowana funkcja `long()` wywołuje metodę klasy `__long__`. W Pythonie 3.X typ `int` obejmuje typ `long`. Ten ostatni usunięto.

Tabela 6. Działania na liczbach (wszystkie typy liczbowe) — ciąg dalszy

Działanie	Opis	Metoda klasy
<code>float(<i>X</i>)</code>	Konwersja na liczbę zmiennoprzecinkową	<code>__float__</code>
<code>complex(<i>X</i>), complex(<i>re</i>, <i>im</i>)</code>	Utworzenie liczby zespolonej	<code>__complex__</code>
<code>divmod(<i>X</i>, <i>Y</i>)</code>	Krotka (<i>X</i> / <i>Y</i> , <i>X</i> % <i>Y</i>)	<code>__divmod__</code>
<code>pow(<i>X</i>, <i>Y</i> [,<i>Z</i>])</code>	Podnoszenie do potęgi	<code>__pow__</code>

Uwagi na temat działań na sekwencjach

Oto lista przykładów i uwag dotyczących wybranych operacji na sekwencjach z tabel 3. i 4.:

Indeksowanie — `S[i]`

- Pobranie komponentów ze wskazanych przesunięć (pierwszy element znajduje się pod adresem przesunięcia równym 0).
- Indeksy ujemne oznaczają zliczanie wstecz od końca (ostatni element pod adresem przesunięcia -1).
- `S[0]` pobiera pierwszy element (`S[1]` pobiera drugi element).
- `S[-2]` pobiera przedostatni element (to samo co `S[len(S) - 2]`).

Wycinki proste — `S[i:j]`

- Wyodrębnia ciągłą sekcję z sekwencji od *i* do *j*-1.
- Domyślnymi granicami wycinków *i* oraz *j* są 0 oraz długość sekwencji `len(S)`.
- `S[1:3]` pobiera elementy sekwencji *S* od przesunięcia 1 do 3 (bez trzeciego).
- `S[1:]` pobiera elementy sekwencji od 1 do końca (`len(S)-1`).
- `S[: -1]` pobiera elementy sekwencji od przesunięcia 0 do przedostatniego elementu włącznie.
- `S[:]` tworzy płytką kopię obiektu sekwencji *S*.

Wycinki rozszerzone — `S[i:j:k]`

- Trzeci element *k* oznacza krok (domyślnie 1); jest on dodawany do przesunięcia każdego wyodrębnionego elementu.

- $S[:2]$ pobiera co drugi element sekwencji S .
- $S[::-1]$ to odwrócona sekwencja S .
- $S[4:1:-1]$ pobiera elementy od przesunięcia 4 do 1 (bez pierwszego) w odwróconej kolejności.

Przypisanie wycinka — $S[i:j:k] = I$

- Operacje przypisania wycinków przebiegają podobnie jak usuwanie elementów i późniejsze ich wstawianie.
- Iteratory przypisywane do prostych wycinków $S[i:j]$ muszą mieć taki sam rozmiar.
- Iteratory przypisywane do rozszerzonych wycinków $S[i:j:k]$ muszą mieć taki sam rozmiar.

Inne

- Operacje konkatencji, repetycji i wycinania zwracają nowe obiekty (w przypadku krotek nie zawsze).

Specyficzne typy wbudowane

W tym podrozdziale opisano liczby, ciągi znaków, listy, słowniki, krotki, pliki oraz inne zasadnicze typy wbudowane. Poniżej omówiono szczegóły wspólne dla wersji Pythona 3.X i 2.X. Złożone typy danych (np. listy, słowniki i krotki) mogą być dowolnie zagnieżdżane wewnątrz siebie tak głęboko, jak potrzeba. Zagnieżdżać można także zbiory, ale mogą one zawierać tylko niemutowalne obiekty.

Liczby

Liczby są zawsze *niemutowalnymi* (niezmiennymi) wartościami umożliwiającymi wykonywanie działań liczbowych. Poniżej opisano podstawowe typy liczbowe (całkowite, zmiennoprzecinkowe), a także bardziej zaawansowane typy danych (liczby zespolone, dziesiętne oraz ułamki).

Literały i ich tworzenie

Liczby zapisuje się w formie różnych form literałów. Tworzy się je za pomocą określonych wbudowanych działań:

1234, -24, +42, 0

Liczby całkowite (nieograniczona precyzja)⁸.

1.23, 3.14e-10, 4E210, 4.0e+210, 1., .1

Liczby zmiennoprzecinkowe (zwykle implementowane jako typ `double` języka C w implementacji CPython).

0o177, 0x9ff, 0b1111

Ósemkowe, szesnastkowe i binarne literały liczb całkowitych⁹.

3+4j, 3.0+4.0j, 3j

Liczby zespolone.

`decimal.Decimal('1.33')`, `fractions.Fraction(4, 3)`

Typy bazujące na modułach: liczby dziesiętne, ułamki.

`int(9.1)`, `int('-9')`, `int('1111', 2)`, `int('0b1111', 0)`,

`float(9)`, `float('1e2')`, `float('-1')`, `complex(3, 4.0)`

Tworzenie liczb na podstawie innych obiektów lub ciągów znaków z możliwością podania podstawy konwersji. Dla porównania `hex(N)`, `oct(N)` i `bin(N)` tworzą ciągi cyfr dla liczb całkowitych, natomiast ogólne ciągi dla liczb tworzy się za pomocą funkcji formatowania ciągów znaków. Więcej informacji na ten temat można znaleźć w podrozdziałach „Formatowanie łańcuchów znaków”, „Konwersje typów” i „Funkcje wbudowane”.

Działania

Typy liczbowe obsługują wszystkie *działania liczbowe* (patrz tabela 6.). W wyrażeniach z typami mieszanymi Python konwertuje operandy w górę, do „najwyższego” typu. W tej hierarchii liczby całkowite (`integer`) znajdują się niżej od zmiennoprzecinkowych, a te z kolei są niżej od liczb zespolonych. W wersjach Pythona 3.0 i 2.6 liczby

⁸ W Pythonie 2.X dla liczb całkowitych o nieograniczonej precyzji występuje osobny typ o nazwie `long`. Typ `int` odpowiada zwykłemu liczbom całkowitym o precyzji ograniczonej zazwyczaj do 32 bitów. Obiekty typu `long` mogą być kodowane z przyrostkiem „L” (np. 99999L). Zwykle liczby całkowite są automatycznie promowane do typu `long`, w przypadku gdy wymagają specjalnej dokładności. W Pythonie 3.X typ `int` zapewnia nieograniczoną precyzję, a zatem zawiera w sobie zarówno typ `int`, jak i `long` z Pythona 2.X. W wersji 3.X Pythona usunięto oznaczenie „L” z literałów liczbowych.

⁹ W Pythonie 2.X literały ósemkowe można zapisywać z wiodącym zerem — zapisy `-0777` i `0o777` są sobie równoważne. W wersji 3.X dla literałów ósemkowych dostępna jest tylko druga postać.

całkowite i zmiennoprzecinkowe udostępniają również szereg *metod* oraz innych *atrybutów*. Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference*.

```
>>> (2.5).as_integer_ratio()           # atrybuty liczb zmiennoprzecinkowych
(5, 2)
>>> (2.5).is_integer()
False

>>> (2).numerator, (2).denominator     # atrybuty liczb całkowitych
(2, 1)
>>> (255).bit_length(), bin(255)      # 3.1+ bit_length()
(8, '0b11111111')
```

Liczby dziesiętne i ułamki

W Pythonie dostępne są dwa dodatkowe typy liczbowe w standardowych modułach bibliotecznych: *liczby dziesiętne* (ang. *decimal*), czyli liczby zmiennoprzecinkowe stałej precyzji, oraz *ułamki* (ang. *fraction*), a zatem typ liczbowy, który jawnie przechowuje licznik i mianownik. Oba wymienione typy można wykorzystać w celu uniknięcia niedokładności w wynikach działań arytmetyki zmiennoprzecinkowej.

```
>>> 0.1 - 0.3
-0.19999999999999998
>>> from decimal import Decimal
>>> Decimal('0.1') - Decimal('0.3')
Decimal('-0.2')

>>> from fractions import Fraction
>>> Fraction(1, 10) - Fraction(3, 10)
Fraction(-1, 5)

>>> Fraction(1, 3) + Fraction(7, 6)
Fraction(3, 2)
```

Typ ułamkowy zapewnia automatyczne skracanie wyników. Dzięki stałej precyzji oraz obsłudze różnorodnych protokołów obcinania i przybliżania liczby dziesiętne przydają się do obliczeń walutowych. Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference*.

Inne typy numeryczne

W Pythonie jest dostępny również typ *set* (opisano go w podrozdziale „Zbiory”). Dodatkowe typy numeryczne, takie jak wektory i macierze, są dostępne za pośrednictwem zewnętrznych rozszerzeń *open source* (np. w pakiecie *NumPy* — <http://www.numpy.org>). Dostępne są

również zewnętrzne pakiety do wizualizacji, obliczeń statystycznych, rozszerzonej arytmetyki zmiennoprzecinkowej i wiele innych (można je znaleźć w internecie).

Ciągi znaków

Standardowy obiekt `str` udostępniający własności ciągu znaków to *niemutowalna* (tzn. niezmienna) *sekwencja* znaków, do której dostęp można uzyskać za pomocą *przesunięcia* (pozycji). Znaki to wartości kodów z odpowiedniego zestawu znaków. Poszczególne znaki są obiektami łańcuchowymi o długości 1.

Pełny model obiektowy łańcucha znaków różni się w poszczególnych liniach.

W *Pythonie 3.X* dostępne są trzy typy łańcuchowe o bardzo podobnych interfejsach:

`str`

Niemutowalna sekwencja znaków używana w odniesieniu do wszystkich rodzajów tekstu — zarówno ASCII, jak i Unicode.

`bytes`

Niemutowalna sekwencja liczb całkowitych `short` używana do reprezentacji binarnych danych bajtowych.

`bytearray`

Mutowalna wersja literału `bytes`.

W *Pythonie 2.X* dostępne są dwa typy łańcuchowe o bardzo podobnych interfejsach:

`str`

Niemutowalna sekwencja znaków używana zarówno w odniesieniu do tekstu bajtowego (8-bitowego), jak i danych binarnych.

`unicode`

Niemutowalna sekwencja znaków używana w odniesieniu do potencjalnie bogatszego tekstu Unicode.

W *Pythonie 2.X* (począwszy od wersji 2.6) dostępny jest także typ `bytearray` z *Pythona 3.X* jako port z wersji 3.X, ale jego stosowanie nie nakłada konieczności ostrego rozróżniania danych tekstowych od binarnych (w wersji 2.X można go dowolnie łączyć z danymi tekstowymi).

Informacje o wsparciu dla Unicode w wersjach 3.X i 2.X można znaleźć w podrozdziale „Łańcuchy znaków Unicode”. Większość materiału w tym podrozdziale dotyczy wszystkich typów łańcuchowych.

Więcej informacji na temat typów `bytes` i `bytearray` można znaleźć w podrozdziałach „Metody klasy `String`”, „Łańcuchy znaków Unicode” oraz „Funkcje wbudowane”.

Literały i ich tworzenie

Łańcuchy znaków zapisuje się w postaci serii znaków ujętych w cudzysłowy (apostrofy). Opcjonalnie można je poprzedzić znakiem desygatora. We wszystkich formach literałów znakowych pusty ciąg jest kodowany jako sąsiednie znaki cudzysłowu (apostrofu). Różne operacje wbudowane zwracają także nowe ciągi znaków:

```
'Python's', "Python's"
```

Apostrofy i cudzysłowy można stosować zamiennie. Wewnątrz apostrofów można umieszczać cudzysłowy i na odwrót — wewnątrz cudzysłowów apostrofy — bez potrzeby poprzedzania ich znakiem lewego ukośnika.

```
"""To jest blok wielowierszowy"""
```

Bloki umieszczone w potrójnych cudzysłowach (apostrofach) pozwalają na prezentowanie wielu wierszy tekstu w jednym łańcuchu. Pomiędzy wierszami są wstawiane znaczniki końca wiersza (`\n`).

```
'M\c Donalds\n'
```

Sekwencje specjalne poprzedzone lewym ukośnikiem (patrz tabela 7.) są zastępowane przez specjalne wartości bajtowe (np. `'\n'` to znak ASCII o dziesiętnej wartości kodu 10).

```
"To" "będzie" "scalone"
```

Sąsiednie stałe łańcuchowe są scalane. Wskazówka: jeśli zostaną ujęte w nawiasy, mogą obejmować wiele wierszy.

```
r'surowy\łańcuch znaków', R'kolejny\przykład'
```

Tzw. „surowe” łańcuchy znaków (ang. *raw strings*) — znaki lewego ukośnika występujące wewnątrz ciągów są interpretowane literalnie (z wyjątkiem przypadków, kiedy występują na końcu łańcucha). Mechanizm ten przydaje się do prezentowania wyrażeń regularnych oraz ścieżek dostępu w systemie DOS, np. `r'c:\katalog1\plik'`.

```
hex(), oct(), bin()
```

Ciągi znaków zawierających tekstowe reprezentacje liczb w formatach ósemkowym, szesnastkowym i dwójkowym. Więcej informacji można znaleźć w podrozdziałach „Liczby” i „Funkcje wbudowane”.

Literały znakowe opisane poniżej tworzą specjalizowane łańcuchy znaków omówione w podrozdziale „Łańcuchy znaków Unicode”:

b'...'

Bajtowy literal tekstowy w Pythonie 3.X — sekwencja wartości 8-bitowych bajtów reprezentujących „surowe” dane binarne. Dla zachowania zgodności z wersją 3.X ta forma jest dostępna również w Pythonie w wersjach 2.6 i 2.7, gdzie tworzy po prostu zwykły ciąg str. Więcej informacji na ten temat można znaleźć w podrozdziałach „Metody klasy String”, „Łańcuchy znaków Unicode” i „Funkcje wbudowane”.

bytearray(...)

Tekstowy literal bytearray — mutowalna wersja literału bytes. Dostępny w Pythonie 3.X, a także w Pythonie 2.X, począwszy od wersji 2.6. Więcej informacji na ten temat można znaleźć w podrozdziałach „Metody klasy String”, „Łańcuchy znaków Unicode” i „Funkcje wbudowane”.

u'...'

Literal łańcuchowy Unicode dostępny wyłącznie w Pythonie 2.X — sekwencja kodów znaków Unicode. Dla zachowania zgodności z wersją 2.X ta forma jest dostępna również w Pythonie 3.X, począwszy od wersji 3.3, gdzie tworzy po prostu zwykły ciąg str (ale w Pythonie 3.X normalne literały łańcuchowe i ciągi str obsługują tekst Unicode). Więcej informacji na ten temat można znaleźć w podrozdziale „Łańcuchy znaków Unicode”.

str(), bytes(), bytearray() (i unicode() — tylko w 2.X)

Ciągi znaków tworzone na podstawie obiektów. W Pythonie 3.X dostępna jest opcja kodowania (dekodowania) Unicode. Więcej informacji można znaleźć w podrozdziale „Funkcje wbudowane”.

Literały łańcuchowe mogą zawierać sekwencje specjalne reprezentujące bajty o specjalnym znaczeniu. Zestawiono je w tabeli 7.

Tabela 7. Sekwencje specjalne dostępne dla stałych tekstowych

Sekwencja specjalna	Znaczenie	Sekwencja specjalna	Znaczenie
\nowywiersz	Kontynuacja w nowym wierszu	\t	Tabulacja pozioma
\\	Lewy ukośnik (\)	\v	Tabulacja pionowa
\'	Apostrof (')	\N{id}	Znak o kodzie Unicode <i>id</i>
\"	Cudzysłów (")	\uhhhh	16-bitowy szesnastkowy kod Unicode

Tabela 7. Sekwencje specjalne dostępne dla stałych tekstowych — ciąg dalszy

Sekwencja specjalna	Znaczenie	Sekwencja specjalna	Znaczenie
\a	Dzwonek (Bell)	\Uhhhhhhhh	32-bitowy szesnastkowy kod Unicode ¹⁰
\b	Backspace	\xhh	Liczba szesnastkowa (co najwyżej 2 cyfry)
\f	Wysuw strony	\ooo	Liczba ósemkowa (maksymalnie 3 znaki)
\n	Wysuw wiersza	\0	Znak Null (nie jest to koniec łańcucha znaków)
\r	Powrót karetki	\inne	To nie są sekwencje specjalne

Działania

Wszystkie typy łańcuchowe obsługują wszystkie *działania na sekwencjach* (patrz tabela 3.) oraz dodatkowo *metody* specyficzne dla tekstu (opisane w podrozdziale „Metody klasy String”). Dodatkowo typ str obsługuje wyrażenia *formatowania ciągów znaków* % oraz zastępowanie szablonów (omówione w następnym podrozdziale). Ponadto typ bytearray obsługuje *działania na sekwencjach mutowalnych* (zestawione w tabeli 4.) oraz dodatkowe metody przetwarzania list. Warto się również zapoznać z modułem dopasowywania wzorców tekstowych *re* (jego opis znajduje się w podrozdziale „Moduł dopasowywania wzorców re”), a także z tekstowymi funkcjami wbudowanymi omówionymi w podrozdziale „Funkcje wbudowane”.

Formatowanie łańcuchów znaków

Zarówno w Pythonie 3.X, jak i 2.X (począwszy od wersji 3.0 i 2.6) standardowe łańcuchy znaków str obsługują dwa różne rodzaje formatowania łańcuchów znaków — działania formatujące obiekty zgodnie z opisowymi ciągami formatującymi:

- podstawowe wyrażenie formatujące (wszystkie wersje Pythona) zakodowane za pomocą operatora %: *fmt % (wartości)*,
- nowy sposób (wersje 3.0, 2.6 i nowsze) kodowania za pomocą wywołania o następującej składni: *fmt.format(wartości)*.

¹⁰ \Uhhhhhhhh zajmuje dokładnie osiem cyfr szesnastkowych (h); zarówno sekwencja \u, jak i \U mogą być używane wyłącznie w literałach tekstowych Unicode.

Oba te rodzaje tworzą nowe ciągi znaków na podstawie kodów zastąpień, które mogą być specyficzne dla różnych typów. Uzyskane wyniki mogą być wyświetlane lub przypisywane do zmiennych w celu ich późniejszego wykorzystania:

```
>>> '%s, %s, %.2f' % (42, 'spam', 1 / 3.0)
'42, spam, 0.33'
```

```
>>> '{0}, {1}, {2:.2f}'.format(42, 'spam', 1 / 3.0)
'42, spam, 0.33'
```

Chociaż sposób bazujący na wywołaniu metody w ostatnich latach rozwijał się dość dynamicznie, to wyrażenia formatujące są powszechnie wykorzystywane w istniejącym kodzie. W związku z tym obie formy w dalszym ciągu są w pełni obsługiwane. Co więcej, choć niektórzy postrzegają postać bazującą na wywołaniu metody jako odrobinę bardziej opisową i spójną, wyrażenia są często prostsze i bardziej zwarte. Ponieważ opisane dwie formy to w zasadzie niewiele różniące się odmiany o równoważnej sobie funkcjonalności i złożoności, nie ma dziś istotnych powodów, aby w jakiś szczególny sposób rekomendować jedną z nich.

Wyrażenie formatujące łańcuchy znaków

Działanie *wyrażeń* formatujących łańcuchy znaków bazuje na zastępowaniu tzw. celów (ang. *targets*) operatora %, występujących po jego lewej stronie, wartościami znajdującymi się po stronie prawej (podobnie do instrukcji `sprintf` języka C). Jeśli trzeba zastąpić więcej niż jedną wartość, należy je zakodować w postaci krotki po prawej stronie operatora %. Jeśli zastąpiona ma być tylko jedna wartość, można ją zakodować jako pojedynczą wartość lub jednoelementową krotkę (aby zakodować krotkę, należy zastosować krotkę zagnieżdżoną). Jeśli po lewej stronie wyrażenia zostaną użyte nazwy kluczy, po prawej należy umieścić słownik. Symbol * pozwala na dynamiczne przekazywanie szerokości i precyzji:

```
>>> 'Ryccerze, którzy mówią %s!' % 'Ni'
'Ryccerze, którzy mówią Ni!'
>>> '%d %s, %d you' % (1, 'spam', 4.0)
'1 spam, 4 you'
>>> '%(n)d named %(x)s' % {'n': 1, 'x': "spam"}
'1 named spam'
>>> '%(n).0E => [% (x)-6s]' % dict(n=100, x='spam')
'1E+02 => [spam ]'
>>> '%f, %.2f, %+.f' % (1/3.0, 1/3.0, 4, 1/3.0)
'0.333333, 0.33, +0.3333'
```

Składnia wyrażenia formatującego łańcuchy znaków

Cele zastępowania występujące po lewej stronie operatora % w wyrażeniu formatującym mają ogólny format zamieszczony poniżej. Wszystkie człony z wyjątkiem ostatniego są opcjonalne (tekst na zewnątrz celów zastępowania jest interpretowany dosłownie):

`%[(nazwakłucza)][flagi][szerokość][.precyzja]kodotypu`

Oto znaczenie poszczególnych członów powyższego celu zastępowania:

nazwakłucza

Odwołuje się do elementu w odpowiednim słowniku; w nawiasach.

flagi

Zbiór znaków o następującym znaczeniu: - (wyrównanie do lewej), + (znak liczby), spacja (pozostawienie pustego miejsca przed liczbami dodatnimi) oraz 0 (wypełnianie zerami).

szerokość

Oznacza całkowitą szerokość pola (aby wyliczyć ją na podstawie wartości, należy użyć symbolu *).

precyzja

Oznacza liczbę cyfr po kropce (tzn. precyzję) — aby wyliczyć ją na podstawie wartości, należy użyć symbolu *.

kodotypu

Znak z tabeli 8.

Człony *szerokość* i *precyzja* można zakodować jako *. W tym przypadku ich wartości będą pobrane z następnego elementu występującego po prawej stronie operatora %. Takie kodowanie można wykorzystać, gdy rozmiar nie jest znany do czasu wykonania programu. Wskazówka: %s przekształca dowolny obiekt na postać jego reprezentacji tekstowej.

Metoda formatująca

Efekt wywołania *metody* formatującej jest podobny do opisanego w poprzednim podrozdziale sposobu bazującego na wyrażeniach, ale wykorzystuje standardową składnię wywołania metody obiektu formatowania łańcucha znaków. Cele zastępowania są natomiast identyfikowane za pomocą nawiasów klamrowych ({}), a nie operatora %.

Tabela 8. Kody typu formatowania łańcuchów znaków

Kod	Znaczenie	Kod	Znaczenie
s	łańcuch znaków (lub dowolny obiekt, wykorzystuje <code>str()</code>)	X	łańcuch x zapisany wielkimi literami
r	To samo co s, ale wykorzystuje metodę <code>repr()</code> zamiast <code>str()</code>	e	Wykładnik liczby zmiennoprzecinkowej
c	Character (int lub str)	E	e wielkimi literami
d	Decimal (dziesiętna liczba całkowita)	f	Dziesiętna liczba zmiennoprzecinkowa
i	Liczba całkowita	F	f zapisany wielkimi literami
u	To samo co d (przestarzałe)	g	Zmiennoprzecinkowy typ e lub f
o	Octal (ósemkowa liczba całkowita)	G	Zmiennoprzecinkowy typ E lub F
x	Hex (szesnastkowa liczba całkowita)	%	Litera ' % ' (kodowany jako %%)

Cele zastępowania występujące w łańcuchu formatującym mogą się odwoływać do argumentów wywołania metody za pomocą pozycji lub nazwy słowa kluczowego. Dodatkowo można się w nich odwoływać do atrybutów argumentów, kluczy i przesunąć. Dozwolone jest posługiwanie się domyślnym formatowaniem lub jawne wprowadzanie kodów typu. Można również zagnieżdżać cele, dzięki czemu możliwe jest pobieranie wartości z list argumentów:

```
>>> 'Rycerze, którzy mówią {0}!'.format('Ni')
'Rycerze, którzy mówią Ni!'
>>> '{0} {1}, {2:.0f} you'.format(1, 'spam', 4.0)
'1 spam, 4 you'
>>> '{n} named {x:s}'.format(n=1, x="spam")
'1 named spam'
>>> '{n:.0E} => [{x:<6s}]'.format(
    **dict(n=100, x='spam'))
'1E+02 => [spam ]'
>>> '{:f}, {:.2f}, {:+.{}f}'.format(
    1/3.0, 1/3.0, 1/3.0, 4)
'0.333333, 0.33, +0.3333'
```

Większość z pokazanych wywołań ma odpowiedniki we wzorcach użycia wyrażeń z operatorem `%` (np. odwołania do klucza i wartości `*`), chociaż w przypadku metody niektóre działania mogą być kodowane wewnątrz samego łańcucha formatującego.

```
>>> import sys # Metoda versus wyrażenie: attr, key, index

>>> fmt = '{0.platform} {1[x]} {2[0]}'
>>> fmt.format(sys, dict(x='ham'), 'AB')
'win32 ham A'
```

```
>>> fmt = '%s %s %s'
>>> fmt % (sys.platform, dict(x='ham')['x'], 'AB'[0])
'win32 ham A'
```

W Pythonie 3.1 i 2.7 znak `,` (przecinek) poprzedzający liczbę całkowitą lub zmiennoprzecinkową wewnątrz członu *kodtypu* (formalnie opisano to w podrozdziale „Składnia metody formatującej”) powoduje wstawianie separatorów tysięcy (przecinków), natomiast człon *kodtypu* o wartości `%` formatuje znak procenta (narzędzia nie występują w samym wyrażeniu formatującym, ale bezpośrednio w kodzie jako funkcje wielokrotnego użytku):

```
>>> '{0:,d}'.format(1000000)
'1,000,000'
>>> '{0:13,.2f}'.format(1000000)
'1,000,000.00'
>>> '{0:%} {1:,.2%}'.format(1.23, 1234)
'123.000000% 123,400.00%'
```

Ponadto, począwszy od Pythona 3.1 i 2.7, w przypadku pominięcia pól w komponencie *nazwapola* numery pól są automatycznie numerowane sekwencyjnie (to także opisano w podrozdziale „Składnia metody formatującej”) — poniższe trzy instrukcje przynoszą ten sam efekt, choć w przypadku występowania wielu pól pola numerowane automatycznie mogą być mniej czytelne:

```
>>> '{0}/{1}/{2}'.format('usr', 'home', 'bob')
'usr/home/bob'
>>> '{}/{}/{}'.format('usr', 'home', 'bob') # Auto
'usr/home/bob'
>>> '%s/%s/%s' % ('usr', 'home', 'bob')    # Wyrażenie
'usr/home/bob'
```

Pojedyncze obiekty można również formatować za pomocą wbudowanej funkcji `format(obiekt, specyfikacjaformatu)` (więcej informacji można znaleźć w podrozdziale „Funkcje wbudowane”). Sposób ten jest wykorzystywany przez metodę formatującą ciągu. Zachowanie tej funkcji można zaimplementować w klasach za pomocą metody przeciążania operatorów `__format__` (patrz „Metody przeciążające operatory”).

Składnia metody formatującej

Cele zastępowania występujące w wywołaniach metody formatującej mają zaprezentowany poniżej ogólny format. Wszystkie cztery członki są opcjonalne i nie mogą pomiędzy nimi występować spacje (tutaj użyto ich dla przejrzystości):

```
{nazwapola komponent !flagakonwersji :specyfikacjaformatu}
```

Oto znaczenie poszczególnych członów powyższego celu zastępowania:

nazwapola

Opcjonalna liczba lub słowo kluczowe identyfikujące argument. W wersjach Pythona 2.7, 3.1 i późniejszych można pominąć ten element, aby zastosować względne numerowanie argumentów.

komponent

Ciąg znaków składający się z zera bądź większej liczby odwołań *.name* lub *[index]* używanych do pobierania atrybutów oraz indeksowanych wartości argumentów. Można je pominąć, aby wyzyszczyć całą wartość argumentu.

flagakonwersji

Jeśli flaga występuje, to poprzedza ją symbol `!`. Za nim znajdują się symbole `r`, `s` lub `a`. Wywołują one odpowiednio metody `repr()`, `str()` lub `ascii()` dla wartości.

specyfikacjaformatu

Jeśli specyfikacja występuje, poprzedza ją symbol `:`. Składa się na nią tekst określający sposób prezentacji wartości, włącznie z takimi szczegółami jak szerokość pola, wyrównania, wypełnienia, precyzji dziesiętnej itp. Ostatnim elementem jest opcjonalny kod typu danych.

Komponent *specyfikacjaformatu* występujący za znakiem dwukropka można formalnie opisać w poniższy sposób (nawiasy prostokątne oznaczają komponenty opcjonalne):

```
[[wypełnienie] [wyrównanie] [znak] [#] [0] [[szerokość]] [,] [[.precyzja]] [kodtypu]
```

Oto znaczenie poszczególnych członów powyższej składni zagnieźdzonej *specyfikacjiformatu*:

wypełnienie

Dowolny znak wypełnienia z wyjątkiem `{` lub `}`.

wyrównanie

Może być jednym ze znaków: `<`, `>`, `=` lub `^`, co oznacza odpowiednio wyrównanie do lewej, wyrównanie do prawej, uzupełnienie za symbolem znaku lub wyrównanie do środka.

znak

Może być symbolem `+`, `-` lub spacją.

, (przecinek)

Żądanie użycia przecinka w roli separatora tysięcy. Dostępny od Pythona w wersjach 3.1 i 2.7.

szerokość i precyzja

Ogólnie rzecz biorąc, mają takie samo znaczenie jak w przypadku wyrażen z operatorem %. Komponent *specyfikacjaformatu* może także zawierać zagnieżdżone ciągi formatujące {}, w których znajduje się wyłącznie człon *nazwapola*. Powoduje to dynamiczne pobieranie wartości z listy argumentów (podobnie jak symbol * w wyrażeniach formatujących). Znak 0 poprzedzający człon *szerokość* włącza funkcję uzupełniania zerami z rozpoznawaniem znaku (podobnie jak w przypadku członu *wypełnienie*), natomiast # — alternatywną konwersję (o ile jest dostępna).

kodtypu

Uogólniając, ma takie samo znaczenie jak w przypadku wyrażen z operatorem % z tabeli 8., ale metoda formatująca obsługuje dodatkowy kod typu b służący do wyświetlania liczb całkowitych w formacie dwójkowym (działa podobnie do wbudowanej funkcji bin()). Ponadto obsługuje dodatkowy kod typu %, który od wersji 3.1 i 2.7 pozwala na formatowanie wartości procentowych. Dla dziesiętnych liczb całkowitych wykorzystywany jest wyłącznie kod typu d (kody i oraz u nie są wykorzystywane).

Warto zwrócić uwagę, że w odróżnieniu od generycznego %s dla wyrażenia, kod typu s dla metody wymaga argumentu w postaci obiektu string. Pominięcie kodu typu sprawia, że metoda akceptuje wszystkie typy.

Zastępowanie szablonów w łańcuchach znaków

W Pythonie 2.4 i późniejszych wersjach jako alternatywa wyrażen formatujących łańcuchy znaków oraz metod (zagadnienia te zostały opisane w poprzednich podrozdziałach) dostępny jest inny mechanizm zastępowania ciągów. W przypadku pełnego formatowania zastępowanie uzyskuje się za pomocą operatora % lub metody str.format() (wszystkie poniższe cztery sposoby zwracają '2: PR5E'):

```
%(strona)i: %(tytul)s' % {'strona': 2, 'tytul': 'PR5E'}
'%(strona)i: %(tytul)s' % dict(strona=2, tytul='PR5E')

'{strona}: {tytul}'.format(**dict(strona=2, tytul='PR5E'))
'{strona}: {tytul}'.format(strona=2, tytul='PR5E')
```

W przypadku prostych zadań zastępowania klasa Template w module string wykorzystuje symbol \$ do wskazania zastąpienia:

```
>>> import string
>>> t = string.Template('$strona: $tytul')
```

```
>>> t.substitute({'strona': 2, 'tytul': 'PR5E'})
'2: PR5E'
```

Wartości do zastąpienia można podawać w postaci argumentów kluczowych lub kluczy słownikowych:

```
>>> s = string.Template('$kto lubi $co')
>>> s.substitute(kto='bogdan', co=3.14)
'bogdan lubi 3.14'
>>> s.substitute(dict(kto='bogdan', co='ciasto'))
'bogdan lubi ciasto'
```

Metoda `safe_substitute` w przypadku brakujących kluczy ignoruje je i nie zgłasza wyjątku:

```
>>> t = string.Template('$strona: $tytul')
>>> t.safe_substitute({'strona': 3})
'3: $tytul'
```

Metody klasy String

Oprócz opisanej wcześniej metody `format()` dostępne są inne metody klasy `String`. Są to wysokopoziomowe narzędzia przetwarzania tekstu zapewniające większe możliwości od wyrażeń łańcuchowych. Listę dostępnych metod klasy `String` zamieszczono w tabeli 9. `s` oznacza dowolny obiekt `string` (z technicznego punktu widzenia jest to obiekt `str` Pythona 3.X). Metody klasy `String`, które modyfikują tekst, zawsze zwracają nowy łańcuch tekstowy. Nigdy nie modyfikują obiektu „w miejscu” (obiekty `String` są niemutowalne).

Tabela 9. Metody klasy `String` dostępne w Pythonie 3.X

```
S.capitalize()
S.casefold() (począwszy od Pythona 3.3)
S.center(width, [, fill])
S.count(sub [, start [, end]])
S.encode([encoding [, errors]])
S.endswith(suffix [, start [, end]])
S.expandtabs([tabsize])
S.find(sub [, start [, end]])
S.format(*args, **kwargs)
S.format_map(mapping) (począwszy od Pythona 3.2)
S.index(sub [, start [, end]])
S.isalnum()
S.isalpha()
S.isdecimal()
```

Tabela 9. Metody klasy *String* dostępne w Pythonie 3.X — ciąg dalszy

```
S.isdigit()
S.isidentifier()
S.islower()
S.isnumeric()
S.isprintable()
S.isspace()
S.istitle()
S.isupper()
S.join(iterable)
S.ljust(width [, fill])
S.lower()
S.lstrip([chars])
S.maketrans(x [, y [, z]])
S.partition(sep)
S.replace(old, new [, count])
S.rfind(sub [, start [, end]])
S.rindex(sub [, start [, end]])
S.rjust(width [, fill])
S.rpartition(sep)
S.rsplit([sep [, maxsplit]])
S.rstrip([chars])
S.split([sep [, maxsplit]])
S.splitlines([keepends])
S.startswith(prefix [, start [, end]])
S.strip([chars])
S.swapcase()
S.title()
S.translate(map)
S.upper()
S.zfill(width)
```

Więcej informacji na ten temat można znaleźć w podrozdziałach poświęconych opisowi funkcji w dalszej części tej książki. W trybie interaktywnym można też uruchomić polecenie `help(str.metoda)`. Wskazówka: lista zaprezentowana poniżej może być różna dla różnych wersji Pythona. Aby ją wyświetlić, należy skorzystać z polecenia:

```
sorted(x for x in dir(str) if not x.startswith('__'))
```

Warto się również zapoznać z modulem dopasowywania wzorców tekstowych *re* (jego opis znajduje się w podrozdziale „Moduł dopasowywania wzorców *re*”). Można tam znaleźć bazujące na wzorcach odpowiedniki niektórych metod klasy String.

Metody typów bytes i bytearray

Występujące w Pythonie 3.X typy łańcuchowe bytes i bytearray mają podobne, ale nie identyczne zbiory metod jak standardowy typ łańcuchowy str, który został zaprezentowany w poprzednim podrozdziale. Typy te spełniają jednak różne role, dlatego też zbiory metod nie są identyczne (str to typ tekstowy Unicode, bytes to surowe dane binarne, natomiast typ bytearray jest mutowalny). Na poniższej liście (aby ją wyświetlić, w Pythonie 3.3 należy użyć polecenia `set(dir(X))` – `set(dir(Y))` wyliczono unikatowe atrybuty zbioru *X*:

```
>>> set(dir(str)) - set(dir(bytes))
{'__rmod__', 'encode', 'isnumeric', 'format',
 'isidentifier', 'isprintable', 'isdecimal',
 'format_map', '__mod__', 'casefold'}

>>> set(dir(bytes)) - set(dir(str))
{'decode', 'fromhex'}

>>> set(dir(bytearray)) - set(dir(bytes))
{'extend', 'remove', 'insert', 'append', 'pop',
 '__iadd__', 'reverse', 'clear', '__imul__',
 'copy', '__setitem__', '__alloc__', '__delitem__'}
```

Uwaga:

- typ str nie obsługuje *dekodowania* Unicode (ten tekst już jest zdekodowany), ale można go *zakodować* do postaci typu bytes;
- typy bytes i bytearray nie obsługują *kodowania* Unicode (są „surowymi” bajtami zawierającymi zarówno nośnik, jak i już zakodowany tekst), ale mogą być *zdekodowane* do typu str;
- typy bytes i bytearray nie obsługują *formatowania* łańcuchów znaków (zaimplementowanego za pomocą metody str.format oraz atrybutów `__mod__` i `__rmod__` operatora %);
- typ bytearray dostarcza unikatowych, mutowalnych metod przetwarzania „w miejscu”, a także operatorów podobnych do tych, które są dostępne dla list (np. append, +=).

Więcej informacji na temat operacji na łańcuchach znaków bytes można znaleźć w podrozdziale „Łańcuchy znaków bytes i bytearray”. Aby

dowiedzieć się więcej o modelach typu string, można zajrzeć do podrödziału „Łańcuchy znaków Unicode”, natomiast więcej informacji o wywołaniach konstrukcyjnych można znaleźć w podrödziale „Funkcje wbudowane”.

UWAGA

Zbiór metod typu string w *Pythonie 2.X* jest nieco inny (np. istnieje metoda `decode` dla innego modelu typu Unicode charakterystycznego dla wersji 2.X). Typ `unicode` Pythona 2.X ma niemal identyczny interfejs jak obiekty `str` z tej wersji Pythona. Więcej informacji na ten temat można znaleźć w podröczniku *Python 2.X Library Reference*. W trybie interaktywnym można też uruchomić polecenia `dir(str)` oraz `help(str.metoda)`.

W kilku poniöszych podrödziałach opisano szczegółowo niektóre z metod wymienionych w tabeli 9. Pogrupowano je według obszarów funkcjonalnych. We wszystkich metodach zwracających wynik w postaci łańcucha znaków jest to *nowy łańcuch znaków* (ze względu na to, że łańcuchy znaków są niemutowalne, nigdy nie są modyfikowane w miejscu). Określenie „białe spacje” oznacza spacje, tabulacje oraz znaki przejścia do nowego wiersza (wszystkie wartości z `string.whitespace`).

Wyszukiwanie

`S.find(sub [, start [, end]])`

Zwraca indeks (przesunięcie) pierwszego wystąpienia łańcucha `sub` w łańcuchu `S` pomiędzy indeksami `start` i `end` (domyślnie mają one wartości 0 oraz `len(S)` — cały ciąg znaków). Jeśli łańcuch nie zostanie znaleziony, metoda zwraca `-1`. Wskazówka: do testowania przynależności łańcucha znaków w innym łańcuchu można także wykorzystać operator `in` (z tabeli 3.).

`S.rfind(sub [, start [, end]])`

Podobna do metody `find`, ale skanuje ciągi od końca (od prawej do lewej).

`S.index(sub [, start [, end]])`

Podobna do metody `find`, ale w przypadku niepowodzenia zamiast zwracać `-1`, zgłasza wyjątek `ValueError`.

`S.rindex(sub [, start [, end]])`

Podobna do metody `rfind`, ale w przypadku niepowodzenia zamiast zwracać `-1`, zgłasza wyjątek `ValueError`.

`S.count(sub [, start [, end]])`

Zlicza liczbę nienakładających się na siebie wystąpień ciągu *sub* wewnątrz ciągu *S* — od indeksu *start* do indeksu *end* (wartości domyślne: 0, len(*S*)).

`S.startswith(sub [, start [, end]])`

Zwraca True, jeśli ciąg *S* rozpoczyna się od ciągu *sub*. Parametry *start* i *end* określają opcjonalny początkowy i końcowy indeks dopasowywanego ciągu *sub*.

`S.endswith(sub [, start [, end]])`

Zwraca True, jeśli ciąg *S* kończy się ciągiem *sub*. Parametry *start* i *end* określają opcjonalny początkowy i końcowy indeks dopasowywanego ciągu *sub*.

Rozdzielanie i łączenie

`S.split([sep [, maxsplit]])`

Zwraca listę słów w ciągu *S*, wykorzystując *sep* w roli separatora. Jeśli występuje argument *maxsplit*, wykonywanych jest co najwyżej *maxsplit* podziałów. Jeśli argument *sep* nie zostanie określony lub ma wartość None, rolę separatora spełnia dowolny znak białej spacji. Wywołanie `'a*b'.split('*')` zwraca zbiór `['a', 'b']`. Wskazówka: aby przekształcić łańcuch na listę znaków (np. `['a', '*', 'b']`), można skorzystać z wywołania `list(S)`.

`S.join(iterable)`

Scala argument *iterable* (np. listę lub krotkę) składający się z łańcuchów znaków. Tworzy pojedynczy łańcuch znaków, wstawiając ciąg *S* pomiędzy poszczególne elementy. *S* może mieć wartość "" (pustego ciągu). W takim przypadku metodę można wykorzystać do konwersji listy znaków na łańcuch znaków (wywołanie `'*'.join(['a', 'b'])` zwraca ciąg `'a*b'`).

`S.replace(old, new [, count])`

Zwraca kopię łańcucha znaków *S*, w którym wszystkie wystąpienia ciągu *old* są zastąpione ciągiem *new*. Jeśli podano argument *count*, metoda zastępuje tylko *count* wystąpień. Metoda działa tak jak kombinacja instrukcji `x=S.split(old)` oraz `new.join(x)`.

`S.splitlines([keepends])`

Rozdziela ciąg znaków *S* na osobne wiersze i zwraca listę wierszy. Jeśli argument *keepends* ma wartość różną od true, wynik nie zawiera znaków przejścia do następnego wiersza.

Formatowanie

`S.format(*args, **kwargs)`, `S.format_map(mapping)`

Patrz podrozdział „Formatowanie łańcuchów znaków”. W Pythonie 3.2 i nowszych wersjach wywołanie `S.format_map(M)` działa tak jak `S.format(**M)`, ale `M` nie jest kopiowane.

`S.capitalize()`

Zamienia pierwszy znak w łańcuchu `S` na wielką literę, a pozostałe na małe.

`S.expandtabs([tabsize])`

Zastępuje tabulacje w łańcuchu `S` spacjami w liczbie `tabsize` (domyślnie 8).

`S.strip([chars])`

Usuwa wiodące i końcowe białe spacje z łańcucha `S` (lub znaki ze zbioru `chars`, jeśli ten argument występuje).

`S.lstrip([chars])`

Usuwa z łańcucha `S` wiodące białe spacje (lub znaki ze zbioru `chars`, jeśli ten argument występuje).

`S.rstrip([chars])`

Usuwa z łańcucha `S` końcowe białe spacje (lub znaki ze zbioru `chars`, jeśli ten argument występuje).

`S.swapcase()`

Zamienia wszystkie małe litery w łańcuchu znaków na wielkie, a wielkie na małe.

`S.upper()`

Zamienia wszystkie litery w łańcuchu znaków na wielkie.

`S.lower()`

Zamienia wszystkie litery w łańcuchu znaków na małe.

`S.casefold()`

W Pythonie 3.3 i późniejszych wersjach zwraca wersję łańcucha znaków `S` odpowiednią do porównywania bez rozróżniania wielkich i małych liter. Działa podobnie jak `S.lower()`, ale dodatkowo inteligentnie zamienia na małe litery niektóre znaki Unicode.

`S.ljust(width [, fill])`

Wyrównuje do lewej łańcuch `S` w polu podanej szerokości `width`; wypełnia łańcuch z prawej strony znakiem `fill` (domyślnie spacjami). Podobny efekt można osiągnąć za pomocą wyrażenia formatującego oraz metody formatującej klasy `String`.

`S.rjust(width [, fill])`

Wyrównuje do prawej łańcuch *S* w polu podanej szerokości *width*; wypełnia łańcuch z lewej strony znakiem *fill* (domyślnie spacjami). Podobny efekt można osiągnąć za pomocą wyrażenia formatującego oraz metody formatującej klasy `String`.

`S.center(width [, fill])`

Wyrównuje do środka łańcuch *S* w polu podanej szerokości *width*; wypełnia łańcuch z lewej i prawej strony znakiem *fill* (domyślnie spacjami). Podobny efekt można osiągnąć za pomocą mechanizmów formatowania łańcuchów znaków.

`S.zfill(width)`

Wypełnia łańcuch *S* z lewej strony zerami, tak aby powstał wynikowy łańcuch tekstowy o pożądanej szerokości *width* (taki sam efekt można osiągnąć za pomocą mechanizmów formatowania łańcuchów znaków).

`S.translate(table [, deletechars])`

Usuwa z łańcucha *S* wszystkie znaki ze zbioru *deletechars* (o ile ten argument podano), a następnie „tłumaczy” kolejne znaki łańcucha *S*, bazując na pozycji w 256-elementowym ciągu znaków zawierającym tłumaczenia poszczególnych wartości znaków.

`S.title()`

Zwraca wersję łańcucha *S* o pisowni właściwej dla tytułów — kolejne słowa zaczynają się wielkimi literami; pozostałe znaki są zapisane małymi literami.

Testy zawartości

`S.is*()`

Boole’owskie testy `is*()` działają w odniesieniu do łańcuchów znaków dowolnej długości. Ich działanie polega na testowaniu zawartości łańcuchów znaków różnych kategorii (dla pustego łańcucha znaków zawsze zwracają wartość `False`).

Standardowy moduł `string`

Począwszy od Pythona 2.0, większość funkcji przetwarzania łańcuchów znaków, które wcześniej znajdowały się w standardowym module `string`, stała się dostępna w postaci metod obiektów `string`. Jeśli *x* jest referencją do obiektu `string`, to wywołaniu funkcji modułu `string` postaci:

```
import string
res = string.replace(X, 'span', 'spam')
```

w Pythonie 2.0 zazwyczaj odpowiada wywołanie metody obiektu `string` następującej postaci:

```
res = X.replace('span', 'spam')
```

Postać wywołania metody jest jednak preferowanym i szybszym sposobem. Dodatkowo wywołania metod nie wymagają importowania modułu. Warto zwrócić uwagę, że działanie `string.join(obiekt_iterowalny, delim)` przekształcono na metodę separatora (ang. *delimiter*) — `delim.join(obiekt_iterowalny)`. Wszystkie te funkcje usunięto z modułu `string` w Pythonie 3.X; zamiast nich należy używać odpowiednich metod obiektu `string`. Opis pozostałej zawartości tego modułu można znaleźć w podrozdziale „Moduł `string`”.

Łańcuchy znaków Unicode

Każdy tekst jest tekstem Unicode. Jest nim również prosty tekst, w którym każdy znak zajmuje jeden bajt (8 bitów) w schemacie kodowania ASCII. Python obsługuje bogatsze zestawy znaków i schematy kodowania *Unicode*, gdzie każdy znak może być reprezentowany za pomocą wielu bajtów w pamięci. Zawiera również mechanizmy pozwalające na tłumaczenie tekstu w plikach z różnych schematów kodowania i na różne schematy kodowania. Sposób działania tych mechanizmów jest różny w poszczególnych wersjach Pythona. Python 3.X traktuje dowolny tekst jako Unicode i oddzielnie tworzy reprezentację danych binarnych. Z kolei Python 2.X odróżnia 8-bitowy tekst (i dane) od szerszego tekstu Unicode.

Obsługa Unicode w Pythonie 3.X

Standardowy typ `str` oraz literał `'ccc'` reprezentują wszystkie rodzaje tekstu — zarówno 8-bitowy tekst, jak i bogatszy tekst Unicode. Typ `str` jest niemutowalną sekwencją *znaków* — w pamięci reprezentowanych jako zdekodowane punkty kodowe Unicode (identyfikatory porządkowe).

Osobny typ `bytes` oraz literał `b'ccc'` reprezentują binarne dane bajtowe, w tym dane multimedialne, oraz zakodowany tekst Unicode. Typ `bytes` jest niemutowalną sekwencją krótkich *liczb całkowitych* (8-bitowych bajtów), ale obsługuje większość działań typu `str`, a jeśli to możliwe, wyświetla tekst ASCII. Dodatkowy typ `bytearray` jest mutowalną odmianą typu `bytes`. Typ ten obsługuje

dotatkowe metody, podobne do metod przetwarzania list, służące do wprowadzania zmian „w miejscu”.

W Pythonie 3.X normalne *pliki* tworzone za pomocą poleceń `open()` zawierają obiekty `str` i `bytes` w przypadku treści — odpowiednio — w trybie tekstowym i binarnym. W trybie tekstowym pliki są automatycznie kodowane na wyjściu i dekodowane na wejściu.

Począwszy od Pythona 3.3, dostępna jest również forma literału Unicode `u'ccc'` z wersji 2.X. Wprowadzono ją w celu zachowania wstecznej zgodności z kodem 2.X (tworzy obiekty `str` właściwe dla wersji 3.X).

Obsługa Unicode w Pythonie 2.X

Standardowy typ `str` oraz literał `'ccc'` reprezentują wartości bajtowe — zarówno 8-bitowy tekst, jak i dane binarne. Dostępny jest także osobny typ `unicode` i literał `u'ccc'`, które reprezentują punkty kodowe potencjalnie szerszego tekstu Unicode. Oba typy łańcuchowe są sekwencjami niemutowalnymi i zapewniają wykonywanie niemal identycznych operacji.

Także w wersji 2.X normalne *pliki* tworzone za pomocą operacji `open()` zawierają znaki bajtowe, natomiast metoda `codecs.open()` umożliwia czytanie i zapisywanie plików zawierających tekst Unicode. Kodowanie i dekodowanie odbywa się podczas transferu.

Począwszy od Pythona 2.6, dostępna jest również forma literału bajtowego `b'ccc'` z wersji 3.X. Wprowadzono ją w celu zachowania zgodności „w przód” z kodem 3.X (tworzy obiekty `str` właściwe dla wersji 2.X). Dostępny jest również mutowalny typ `bytearray` z wersji 3.X, który jednak jest w mniejszym stopniu specyficzny dla typów.

Obsługa Unicode w Pythonie 3.X

Python 3.X pozwala na kodowanie znaków spoza tablicy ASCII szesnastkowo (`\x`), a także w 16- i 32-bitowej wersji Unicode (sekwencje `\u` i `\U`). Dodatkowo kody znaków Unicode obsługuje funkcja `chr()`:

```
>>> 'A\xE4B'
'AäB'
>>> 'A\u00E4B'
'AäB'
>>> 'A\U000000E4B'
'AäB'
>>> chr(0xe4)
'ä'
```

Zwykle łańcuchy znaków można kodować na postać „surowych” bajtów, które z kolei można dekodować na zwykłe łańcuchy znaków. Do tego celu wykorzystuje się domyślne lub jawne kodowanie (oraz opcjonalnie strategię obsługi błędów). Więcej informacji można znaleźć w opisie metody `str()` w podrozdziale „Funkcje wbudowane”.

```
>>> 'A\xE4B'.encode('latin-1')
b'A\xe4B'
>>> 'A\xE4B'.encode()
b'A\xc3\xa4B'
>>> 'A\xE4B'.encode('utf-8')
b'A\xc3\xa4B'

>>> b'A\xc3\xa4B'.decode('utf-8')
'AäB'
```

Obiekty `File` również automatycznie dekodują i kodują dane podczas operacji wejścia i wyjścia w trybie tekstowym (ale nie w trybie binarnym). Umożliwiają one podanie nazwy kodowania w celu przesłonięcia kodowania domyślnego (patrz opis funkcji `open()` w podrozdziale „Funkcje wbudowane”):

```
>>> S = 'A\xE4B'
>>> open('uni.txt', 'w', encoding='utf-8').write(S)
3
>>> open('uni.txt', 'rb').read()
b'A\xc3\xa4B'
>>>
>>> open('uni.txt', 'r', encoding='utf-8').read()
'AäB'
```

Począwszy od Pythona 3.3, dostępna jest również forma literału `Unicode` `u'ccc'` z wersji 2.X dodana w celu zachowania wstecznej zgodności z kodem 3.X. Jest ona jednak synonimem literału `'ccc'` i tworzy standardowy łańcuch `str` wersji 3.X.

Zarówno w wersji 3.X, jak i 2.X można bezpośrednio osadzać treść `Unicode` w plikach źródłowych programów. Aby przesłonić domyślne dla Pythona kodowanie UTF-8, należy w pierwszym bądź drugim wierszu pliku wprowadzić następującą instrukcję:

```
# -*- coding: latin-1 -*-
```

Łańcuchy znaków `bytes` i `bytearray`

Obiekty łańcuchowe `bytes` i `bytearray` Pythona 3.X reprezentują 8-bitowe dane binarne (włącznie z kodowanym tekstem `Unicode`), wyświetlają znaki ASCII (jeśli to możliwe) oraz obsługują standardowe operacje łańcuchowe właściwe dla obiektów `str`, w tym metody i działania na sekwencjach (ale bez metod formatowania):

```

>>> B = b'spam'
>>> B
b'spam'
>>> B[0]                                # działania na sekwencjach
115
>>> B + b'abc'
b'spamabc'
>>> B.split(b'a')                       # metody
[b'sp', b'm']
>>> list(B)                             # sekwencja złożona z danych typu int
[115, 112, 97, 109]

```

Typ bytearray dodatkowo obsługuje działania mutowalne podobne do działań na listach:

```

>>> BA = bytearray(b'spam')
>>> BA
bytearray(b'spam')
>>> BA[0]
115
>>> BA + b'abc'
bytearray(b'spamabc')
>>> BA[0] = 116                         # mutowalność
>>> BA.append(115)                      # metody listowe
>>> BA
bytearray(b'tpams')

```

Z formalnego punktu widzenia zarówno typ bytes, jak i bytearray obsługują *działania na sekwencjach* (patrz tabela 3.) oraz dodatkowo *metody* specyficzne dla typów, opisane w podrozdziale „Metody typów bytes i bytearray”. Typ bytearray obsługuje ponadto *mutowalne operacje na sekwencjach* (patrz tabela 4.). Warto się również zapoznać z opisem wywoływania konstruktorów typów w podrozdziale „Funkcje wbudowane”.

W Pythonie w wersjach 2.6 i 2.7 jest dostępny typ bytearray, ale nie ma typu bytes. Literał b'ccc' z wersji 3.X jest dostępny w celu zapewnienia zgodności „w przód”, ale jest tylko synonimem literału 'ccc' i tworzy standardowe dla wersji 2.X łańcuchy znaków str.

Obsługa Unicode w Pythonie 2.X

W Pythonie 2.X ciągi znaków Unicode zapisuje się za pomocą literału u'ccc', który tworzy obiekt typu unicode (w Pythonie 3.X dla łańcuchów Unicode wykorzystywany jest standardowy typ łańcuchowy i skojarzony z nim literał). Za pomocą specjalnej sekwencji \uHHHH można zapisać dowolne znaki Unicode, przy czym HHHH oznacza czterocyfrową liczbę szesnastkową z zakresu od 0000 do FFFF. Można także skorzystać z tradycyjnej sekwencji specjalnej \xHH. Dla znaków o kodach

do +01FF (co odpowiada wartości \777) można też wykorzystać sekwencje ósemkowe.

Typ `unicode` obsługuje zarówno metody typowe dla klasy `String`, jak i *działania na sekwencjach* (patrz tabela 3.). W Pythonie 2.X można dowolnie łączyć zwykłe obiekty tekstowe z obiektami `Unicode`. Łączenie łańcuchów 8-bitowych i `Unicode` zawsze wymusza wynik `Unicode` o domyślnym kodowaniu ASCII (np. wynikiem działania `'a'+u'bc'` jest `u'abc'`). W działaniach z mieszanymi typami zakłada się, że 8-bitowe łańcuchy znaków zawierają 7-bitowe dane ASCII (znaki spoza tablicy ASCII powodują zgłoszenie błędu). Do konwersji pomiędzy zwykłymi łańcuchami tekstowymi a łańcuchami `Unicode` można wykorzystać wbudowane funkcje `str()` i `unicode()`, natomiast metody `encode()` i `decode()` powodują — odpowiednio — zastosowanie kodowania `Unicode` oraz wycofanie się z niego.

W modułach dostępna jest między innymi wbudowana funkcja `codecs.open()`, która realizuje tłumaczenie kodowania `Unicode` na poziomie plików, podobnie jak jest w przypadku wbudowanej funkcji `open()` z wersji 3.X.

Listy

Listy to mutowalne (modyfikowalne) sekwencje złożone z referencji do obiektów, do których dostęp uzyskuje się za pośrednictwem pozycji (przesunięcia).

Literały i ich tworzenie

Literały list zapisuje się w formie rozdzielanego przecinkami ciągu wartości ujętych w nawiasy kwadratowe. Istnieje też szereg operacji, w wyniku których listy są tworzone dynamicznie.

```
[]
```

Pusta lista.

```
[0, 1, 2, 3]
```

Lista czteroelementowa: indeksy od 0 do 3.

```
L = ['spam', [42, 3.1415], 1.23, {}]
```

Zagnieżdżone podlisty: instrukcja `L[1][0]` pobiera 42.

```
L = list('spam')
```

Tworzy listę złożoną z wszystkich elementów dowolnego iteratora poprzez wywołanie funkcji konstruktora typu.

```
L = [x ** 2 for x in range(9)]
```

Tworzy listę poprzez pobranie wyników wyrażenia podczas iteracji (lista składana).

Działania

Dla list dostępne są wszystkie *działania na sekwencjach* (patrz tabela 3.), wszystkie *mutowalne działania na sekwencjach* (patrz tabela 4.) oraz dodatkowo wymienione poniżej *metody* specyficzne dla list. We wszystkich tych metodach *L* oznacza dowolny obiekt listy:

L.append(x)

Wstawia pojedynczy obiekt *x* na koniec listy *L*, powodując modyfikację listy w miejscu.

L.extend(I)

Wstawia każdy element dowolnego iteratora *I* na koniec listy *L* w miejscu (działanie + w miejscu). Ma ono podobny efekt do działania *L[len(L):] = I*. Wskazówka: aby dołączyć wszystkie elementy z *I* na początku listy, należy skorzystać z instrukcji *L[:0] = I*.

L.sort(key=None, reverse=False)

Sortuje listę *L* w miejscu, domyślnie w porządku rosnącym. Jeśli występuje argument *key*, oznacza on funkcję jednoargumentową wykorzystywaną w celu pobrania bądź obliczenia wartości do porównań z każdego elementu listy. Jeśli zostanie przekazany argument *reverse*, który ma wartość *true*, elementy listy zostaną posortowane w taki sposób, jakby każde porównanie zwracało odwrotny wynik. Na przykład: *L.sort(key=str.lower, reverse=True)*. Więcej informacji można znaleźć w opisie metody *sorted()* w podrödziale „Funkcje wbudowane”.

L.reverse()

Odwraca elementy listy *L* w miejscu. Więcej informacji można znaleźć w opisie metody *reversed()* w podrödziale „Funkcje wbudowane”.

L.index(x [, i [, j]])

Zwraca indeks pierwszego wystąpienia obiektu *x* na liście *L*. Jeśli obiekt nie zostanie znaleziony, zgłasza wyjątek. Jest to metoda wyszukiwania. Jeśli zostaną przekazane argumenty *i* oraz *j*, metoda zwraca najmniejszy indeks *k*, dla którego *L[k] == x* oraz *i <= k < j*. Domyślnie *j* ma wartość *len(L)*.

`L.insert(i, x)`

Wstawia pojedynczy obiekt x do listy L na pozycji i (podobnie do instrukcji $L[i:i] = [x]$ dla dodatnich bądź ujemnych i). Wskazówka: aby wstawić wszystkie elementy z I na pozycji i , należy skorzystać z instrukcji $L[i:i] = I$.

`L.count(x)`

Zwraca liczbę wystąpień obiektu x na liście L .

`L.remove(x)`

Usuwa pierwsze wystąpienie obiektu x z listy L . W przypadku gdy obiekt nie zostanie znaleziony, zgłasza wyjątek. Metoda działa identycznie jak instrukcja `del L[L.index(x)]`.

`L.pop([i])`

Zwraca ostatni element listy L (lub element spod indeksu i) i usuwa go z niej. Razem z metodą `append` może służyć do zaimplementowania stosu. Metoda działa identycznie jak sekwencja instrukcji $x=L[i]$; `del L[i]`; `return x`, gdzie i ma domyślną wartość -1 — ostatni element.

`L.clear()`

Usuwa wszystkie elementy z listy L . Dostępna wyłącznie w Pythonie 3.X, począwszy od wersji 3.3.

`L.copy()`

Wykonuje kopię najwyższego poziomu (tzw. płytką kopię) listy L . Dostępna wyłącznie w Pythonie 3.X, począwszy od wersji 3.3.

UWAGA

W *Pythonie 2.X* sygnatura metody sortującej listę to

`L.sort(cmp=None, key=None, reverse=False),`

gdzie `cmp` jest dwuargumentową funkcją porównawczą, która zwraca wartość mniejszą od zera, zero lub większą od zera w celu zaprezentowania wyniku mniejszy niż, równy bądź większy. W Pythonie 3.X funkcję porównawczą usunięto, ponieważ zwykle była używana do mapowania wartości sortowania oraz odwracania porządku sortowania — przypadków użycia wspieranych przez pozostałe dwa argumenty.

Listy składane

Literał listowy ujęty w nawiasy prostokątne (`[...]`) może oznaczać prostą listę wyrażeń bądź listę składaną (wyrażenie listowe) o następującej postaci:

```
[wyrażenie for wyr1 in iterator1 [if warunek1]
 for wyr2 in iterator2 [if warunek2] ...
 for wyrN in iteratorN [if warunekN] ]
```

Listy składane konstruuja listy wyników — pobierają wszystkie wartości wyrażenia *wyrażenie* dla każdej iteracji wszystkich zagnieżdżonych pętli *for*, dla których każdy z opcjonalnych warunków *warunek* jest prawdziwy. Pętle *for* od drugiej do *n*-tej, a także wszystkie człony *if* są opcjonalne. W argumentach *wyrażenie* i *warunek* można wykorzystywać zmienne przypisane w zagnieżdżonych pętlach *for*. Nazwy związane (przypisane) wewnątrz wyrażenia listowego są tworzone w zasięgu, w którym rezyduje ta lista składana. Listy składane można dowolnie zagnieżdżać.

Listy składane działają podobnie jak wbudowana funkcja `map()` (w wersji 3.X funkcja `map()` wymaga użycia funkcji `list()` w celu wymuszenia generowania wyników do wyświetlania; wynika to stąd, że funkcja ta zarówno iteruje, jak i sama jest iterowana; w wersji 2.X funkcja `map()` zwraca listę):

```
>>> [ord(x) for x in 'spam']
[115, 112, 97, 109]
>>> list(map(ord, 'spam')) # Wykorzystanie funkcji list() w Pythonie 3.X
[115, 112, 97, 109]
```

Zastosowanie list składanych pozwala jednak uniknąć konieczności tworzenia tymczasowej funkcji pomocniczej:

```
>>> [x ** 2 for x in range(5)]
[0, 1, 4, 9, 16]
>>> list(map((lambda x: x ** 2), range(5)))
[0, 1, 4, 9, 16]
```

Listy składane z warunkami działają podobnie do metody `filter` (w Pythonie 3.X jest ona również iterowalna):

```
>>> [x for x in range(5) if x % 2 == 0]
[0, 2, 4]
>>> list(filter((lambda x: x % 2 == 0), range(5)))
[0, 2, 4]
```

Listy składane z zagnieżdżonymi pętlami *for* działają podobnie jak zwykła instrukcja *for*:

```
>>> [x + y for x in range(3) for y in [10, 20, 30]]
[10, 20, 30, 11, 21, 31, 12, 22, 32]

>>> res = []
>>> for x in range(3):
...     for y in [10, 20, 30]:
...         res.append(x + y)
... 
```

```
>>> res
[10, 20, 30, 11, 21, 31, 12, 22, 32]
```

Protokół iteracji

Protokół iteracji określa zestaw obiektów i metod stosowanych we wszystkich *kontekstach iteracji* — listach składanych, instrukcjach pętli `for` oraz funkcjach wbudowanych, takich jak `map()` i `filter()` — w celu automatycznego przetwarzania elementów kolekcji lub wyników generowanych na żądanie. Zasada działania iteracji jest następująca:

- Konteksty iteracji wykonują działania na *obiekтах iterowalnych* (ang. *iterable*) — obiektach, które implementują metodę `__iter__()`.
- Po wywołaniu metody `__iter__()` obiektu iterowalnego metoda ta zwraca *iterator* — obiekt implementujący metodę `__next__()` (może to być ten sam obiekt).
- Po wywołaniu metody `__next__()` iteratora metoda ta zwraca następny element iteracji albo zgłasza wyjątek `StopIteration` w celu zakończenia iteracji.

Ponadto wbudowana funkcja `iter(X)` wywołuje metodę `X.__iter__()` obiektu iterowalnego, natomiast wbudowana funkcja `next(I)` wywołuje metodę `I.__next__()` iteratora — w celu uproszczenia ręcznych pętli iteracji i spełnienia roli warstwy zapewniającej przenośność. Niektóre mechanizmy, takie jak wbudowana funkcja `map()` oraz *wyrażenia generatorowe*, są zarówno kontekstami iteracji (dla swojej zawartości), jak i obiektami iterowalnymi (dla swoich wyników). Więcej informacji można znaleźć w poprzednim i następnym podrozdziale.

Klasy mogą dostarczać metodę `__iter__()`, która przechwytyje wbudowaną operację `iter(X)`. Jeśli zdefiniowano taką metodę w klasie, jej wynik dostarcza metodę `__next__()`, która służy do przetwarzania wyników (iterowania po wynikach) w kontekstach iteracyjnych. Jeśli nie zdefiniowano metody `__iter__()`, wykorzystywana jest metoda indeksująca `__getitem__()`, której działanie polega na iterowaniu do chwili zgłoszenia wyjątku `IndexError`.

W *Pythonie 2.X* metoda obiektów iteratora `I.__next__()` ma nazwę `I.next()`, ale poza tym mechanizm iteracji działa identycznie. W *Pythonie* w wersjach 2.6 i 2.7 wbudowana funkcja `next(I)` wywołuje metodę `I.next()` zamiast metody `I.__next__()`. Dzięki temu funkcja ta przydaje się do zapewnienia zgodności zarówno z 3.X w wersji 2.X, jak i zgodności z 2.X w wersji 3.X.

Wyrażenia generatorowe

Począwszy od Pythona 2.4, wyrażenia generatorowe pozwalają na uzyskanie wyników podobnych do list składanych, ale nie generują fizycznej listy do przechowywania wyników. Wyrażenia generatorowe definiują zbiór wyników, ale dla zaoszczędzenia pamięci nie tworzą całej listy. Zamiast tego tworzą *obiekt generatora*, który zwraca elementy pojedynczo w kontekstach iteracji poprzez automatyczną obsługę protokołu iteracji opisanego w poprzednim podrozdziale. Na przykład:

```
ords = (ord(x) for x in aString if x not in skipStr)
for o in ords:
    ...
```

Wyrażenia generatorowe zapisuje się wewnątrz nawiasów okrągłych zamiast prostokątnych, ale poza tym obsługują one wszystkie konstrukcje składniowe list składanych. Podczas tworzenia obiektu iterowalnego, który ma być przekazany do funkcji, wystarczą nawiasy okrągłe użyte dla funkcji z pojedynczym argumentem:

```
sum(ord(x) for x in aString)
```

Zmienne pętli wyrażen generatorowych (w powyższym przykładzie *x*) nie są dostępne na zewnątrz wyrażen generatorowych ani w Pythonie 2.X, ani w 3.X. W Pythonie 2.X po stworzeniu listy składanej zmienna sterująca pętlą ma wartość odpowiadającą ostatniej wartości, ale wszystkie pozostałe obiekty składane lokalizują zmienną zgodnie z wyrażeniem. W Pythonie 3.X zmienne pętli są lokalizowane zgodnie z wyrażeniem we wszystkich formach obiektów składanych.

W celu iterowania po wynikach poza kontekstami iteracji, na przykład w pętlach `for`, w Pythonie 3.X należy skorzystać z metody `I.__next__()` protokołu iteracji, a w Pythonie 2.X z metody `I.next()` protokołu iteracji albo z wbudowanej funkcji `next(I)` dostępnej zarówno w Pythonie 2.X, jak i 3.X. Ten ostatni sposób zapewnia przenośność przy wywoływaniu odpowiedniej metody. Jeśli jest taka potrzeba, można skorzystać z wywołania `list()` w celu wygenerowania wszystkich pozostałych wyników jednocześnie (ze względu na to, że generatory są swoimi własnymi iteratorami, wywoływanie ich metod `__iter__()` jest nieszkodliwe, ale nie jest wymagane):

```
>>> squares = (x ** 2 for x in range(5))
>>> squares
<generator object <genexpr> at 0x027C1AF8>

>>> iter(squares) is squares # __iter__() opcjonalnie
True
>>> squares.__next__() # Metoda (.next w Pythonie 2.X)
```

```

0
>>> next(squares) # Funkcja wbudowana (3.X, 2.6+)
1
>>> list(squares) # Do momentu zgłoszenia StopIteration
[4, 9, 16]

```

Więcej informacji na temat mechanizmu wykorzystywanego w wyrażeniach generatorowych można znaleźć w podrozdziale „Protokół iteracji”, natomiast o *funkcji generatora*, która jednocześnie tworzy obiekt generatora, można poczytać w podrozdziale „Instrukcja yield”.

Inne wyrażenia składane

Warto się zapoznać również ze słownikami składanymi (ang. *dictionary comprehension*) oraz zbiorami składanymi (ang. *set comprehension*) opisanymi w innym miejscu tej książki (podrozdziały „Słowniki” i „Zbiory”). Słowniki składane i zbiory składane to wyrażenia podobne do list składanych, które tworzą słowniki i zbiory. Konstrukcje te mają składnię identyczną jak listy składane i wyrażenia generatorowe, ale są kodowane wewnątrz pary nawiasów klamrowych (`{}`), a słowniki składane rozpoczynają się od pary *klucz:wartość*:

```

>>> [x * x for x in range(10)] # lista składana
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

>>> (x * x for x in range(10)) # wyrażenie generatorowe
<generator object <genexpr> at 0x009E7328>

>>> {x * x for x in range(10)} # zbiór składany: 3.X, 2.7
{0, 1, 4, 81, 64, 9, 16, 49, 25, 36}

>>> {x: x * x for x in range(10)} # słownik: 3.X, 2.7
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49,
8: 64, 9: 81}

```

Słowniki

Słowniki to *mutowalne* (modyfikowalne) *odwzorowania* złożone z referencji do obiektów, do których dostęp uzyskuje się za pośrednictwem *klucza*, a nie pozycji. Są to nieuporządkowane kolekcje odwzorowujące klucze na wartości, które są wewnętrznie zaimplementowane w postaci dynamicznie rozszerzalnych tablic asocjacyjnych. W Pythonie 3.X słowniki uległy znaczącym zmianom:

- W *Pythonie 2.X* metody `keys()`, `values()` i `items()` zwracają listy. Dostępna jest metoda wyszukiwania `has_key()` oraz odrębne metody iteratorów `iterkeys()`, `itervalues()` i `iteritems()`. Słowniki

można porównywać bezpośrednio. Począwszy od Pythona 2.7, słowniki składane wersji 3.X są dostępne jako poprawki (ang. *backport*), natomiast widoki w stylu 3.X są obsługiwane za pomocą metod `viewkeys()`, `viewvalues()` i `viewitems()`.

- W Pythonie 3.X metody `keys()`, `values()` i `items()` zwracają iterowalne obiekty *widoków* zamiast list. Metodę wyszukiwania `has_key()` usunięto na rzecz wyrażenia `in`. Metody obiektów iterowalnych Pythona 2.X usunięto na rzecz iteracji *widoku*. Słowników nie można porównywać bezpośrednio, ale można porównywać ich wyrażenia `sorted(D.items())`. Ponadto wprowadzono nowe wyrażenie słownika składanego.
- Obiekty *widoków* w Pythonie 3.X tworzą wyniki na żądanie, zachowują pierwotny porządek w słowniku, uwzględniają przyszłe zmiany w słowniku oraz pozwalają na wykonywanie działań właściwych dla *zbiorów*. Widoki kluczy zawsze mają charakter zbiorów, natomiast widoki wartości nigdy. Widoki elementów mają charakter zbiorów wtedy, gdy wszystkie ich elementy są unikatowe i niezmiennie (ang. *hashable*). Więcej informacji na temat wyrażenia właściwych dla zbiorów, które można zastosować w odniesieniu do niektórych widoków, można znaleźć w podrozdziale „Zbiory”. Aby wymusić wygenerowanie wszystkich wyników widoku (na przykład w celu wyświetlenia lub wykonania metody `L.sort()`), można przekazać widok do wywołania `list()`.

Literały i ich tworzenie

Literały słownikowe zapisuje się w postaci ciągu par *klucz:wartość* wewnątrz nawiasów klamrowych. Wbudowana funkcja `dict()` obsługuje inne wzorce tworzenia, a w Pythonie 3.X i 2.7 słowniki składane wykorzystują iterację. Przypisywanie wartości do nowych kluczy powoduje generowanie nowych pozycji słownika.

Kluczami mogą być dowolne *niemutowalne* obiekty (np. łańcuchy znaków, liczby, krotki), a instancje klas mogą być kluczami, w przypadku gdy dziedziczą metody protokołu haszowania (patrz opis metody `__hash__()` w podrozdziale „Metody przeciążające operatory”). Klucze krotek mogą być wartościami złożonymi (np. `adict[(M,D,Y)]` z opcjonalnymi nawiasami):

```
{}
```

Pusty słownik.

```
{'mielonka': 2, 'jajka': 3}
```

Słownik dwuelementowy: klucze 'mielonka' i 'jajka', wartości 2 i 3.

```
D = {'info': {42: 1, type(''): 2 }, 'mielonka': []}
```

Słowniki zagnieżdżone: instrukcja `D['info'][42]` pobiera 1.

```
D = dict(name='Bogdan', wiek=45, stanowisko=('kier', 'inf'))
```

Tworzy słownik poprzez przekazanie argumentów kluczowych do konstruktora typu.

```
D = dict(zip('abc', [1, 2, 3]))
```

Tworzy słownik poprzez przekazanie listy krotek klucz-wartość do konstruktora typu.

```
D = dict(['a', 1], ['b', 2], ['c', 3])
```

Instrukcja daje ten sam efekt co ta umieszczona w poprzednim wierszu — można jej użyć w przypadku dowolnego obiektu iterowalnego złożonego z kluczy i wartości.

```
D = {c.upper(): ord(c) for c in 'mielonka'}
```

Wyrażenie słownika składanego (w Pythonie 3.X i 2.7). Z jego składnią można się zapoznać w podrozdziale „Listy składane”.

Działania

Słowniki obsługują wszystkie *odwzorowania* (patrz tabela 5.) oraz dodatkowo następujące *metody* specyficzne dla słowników. We wszystkich tych metodach *D* oznacza dowolny obiekt słownika:

D.keys()

Wszystkie klucze słownika *D*. W Pythonie 2.X zwraca listę. W Pythonie 3.X zwraca opisany wcześniej iterowalny obiekt widoku. Instrukcja `for K in D` także obsługuje iterację kluczy.

D.values()

Wszystkie wartości zapisane w słowniku *D*. W Pythonie 2.X zwraca listę. W Pythonie 3.X zwraca opisany wcześniej iterowalny obiekt widoku.

D.items()

Krotki (*klucz, wartość*) — po jednej dla każdego elementu słownika *D*. W Pythonie 2.X zwraca listę. W Pythonie 3.X zwraca opisany wcześniej iterowalny obiekt widoku.

D.clear()

Usuwa wszystkie elementy ze słownika *D*.

D.copy()

Zwraca płytką kopię (najwyższego poziomu) słownika *D*.

D.update(D2)

Scala wszystkie elementy słownika *D2* z elementami słownika *D* w miejscu. Jest to działanie podobne do instrukcji `for (k, v) in D2.items(): D[k] = v`. W Pythonie 2.4 i wersjach późniejszych metoda akceptuje dowolny obiekt iterowalny złożony z par klucz-wartość, a także argumenty kluczowe (*D.update(k1=v1, k2=v2)*).

D.get(K [, default])

Działa podobnie jak metoda *D[K]* dla klucza *K*, ale gdy klucz *K* nie zostanie znaleziony w słowniku *D*, zwraca wartość *default* (lub *None* w przypadku braku wartości *default*), zamiast zgłaszać wyjątek.

D.setdefault(K [, default])

Działa identycznie jak metoda *D.get(K, default)*, ale dodatkowo przypisuje klucz *K* do wartości *default*, jeśli klucz ten nie zostanie odnaleziony w słowniku *D*.

D.popitem()

Zwraca dowolną parę (*klucz, wartość*) i usuwa ją ze słownika.

D.pop(K [, default])

Zwraca wartość *D[K]*, w przypadku gdy klucz *K* znajduje się w *D* (jednocześnie usuwa *K*). W przeciwnym razie zwraca *default*, o ile podano ten argument, albo zgłasza wyjątek *KeyError*, jeśli argumentu *default* nie podano.

dict.fromkeys(I [, value])

Tworzy nowy słownik o kluczach należących do obiektu iterowalnego *I* oraz wartościach ustawionych na *value* (domyślnie *None*). Metodę można wywołać dla egzemplarza słownika *D* albo nazwy typu *dict*.

Poniższe metody są dostępne wyłącznie w *Pythonie 2.X*:

D.has_key(K)

Zwraca *True*, jeśli w słowniku *D* istnieje klucz *K*, lub *False* w przeciwnym razie. W Pythonie 2.X metoda ta jest równoważna instrukcji *K in D*, ale ogólnie rzecz biorąc, nie jest zalecana — w Pythonie 3.X została ona usunięta.

D.iteritems(), D.iterkeys(), D.itervalues()

Zwracają obiekty iterowalne dla par klucz-wartość, samych kluczy lub samych wartości. W Pythonie 3.X usunięto je, ponieważ metody *items()*, *keys()* i *values()* zwracają iterowalny obiekt widoku.

D.viewitems(), D.viewkeys(), D.viewvalues()

Metody dostępne, począwszy od Pythona 2.7. Zwracają iterowalne obiekty widoku dla par klucz-wartość, samych kluczy bądź

samych wartości. Metody mają na celu emulację obiektów widoku zwracanych przez metody `items()`, `keys()` i `values()` z Pythona 3.X.

Poniższe działania opisano w tabeli 5., ale dotyczą metod omówionych powyżej:

`K in D`

Zwraca `True`, jeśli w słowniku `D` istnieje klucz `K`, lub `False` w przeciwnym razie. Zastępuje metodę `has_key()` w Pythonie 3.X.

`for K in D`

Wykonuje iterację według kluczy `K` w słowniku `D` (wszystkie konteksty iteracyjne). Słowniki pozwalają na bezpośrednie iteracje. Instrukcja `for K in D` działa podobnie jak instrukcja `for K in D.keys()`. Pierwsza z nich wykorzystuje iterator według kluczy obiektu słownika. W Pythonie 2.X metoda `keys()` zwraca listę, co wprowadza pewien narzut. W Pythonie 3.X metoda `keys()` zamiast listy fizycznie rezydującej w pamięci zwraca iterowalny obiekt widoku. Dzięki temu obie formy są równoważne.

Krotki

Krotki to *niemutowalne* (niezmienne) *sekwencje* złożone z referencji do obiektów, do których dostęp uzyskuje się za pośrednictwem *przesunięcia* (pozycji).

Literały i ich tworzenie

Krotki zapisuje się w formie rozdzielanego przecinkami ciągu wartości ujętych w nawiasy okrągłe. Nawiasy można czasami pominąć (np. w nagłówkach pętli `for` oraz instrukcjach przypisania =):

`()`

Pusta krotka.

`(0,)`

Krotka jednoelementowa (w odróżnieniu od prostego wyrażenia).

`(0, 1, 2, 3)`

Krotka czteroelementowa.

`0, 1, 2, 3`

Inna postać krotki czteroelementowej (taka sama jak w poprzednim wierszu). Nie jest prawidłowa w kontekstach, w których przecinek lub nawiasy mają inne znaczenie (np. w argumentach funkcji lub wywołaniach funkcji `print` Pythona 2.X).

```
T = ('mielonka', (42, 'jajka'))
```

Krotki zagnieżdżone — instrukcja `T[1][1]` pobiera ciąg `'jajka'`.

```
T = tuple('mielonka')
```

Tworzy krotkę złożoną ze wszystkich elementów dowolnego obiektu iterowalnego poprzez wywołanie konstruktora typu.

Działania

Krotki obsługują wszystkie *działania na sekwencjach* (patrz tabela 3.) oraz dodatkowo poniższe *metody* specyficzne dla krotek, które są dostępne w Pythonie 2.6, 3.0 i wersjach późniejszych.

```
T.index(x [, i [, j]])
```

Zwraca indeks pierwszego wystąpienia obiektu `x` w krotce `T`. Jeśli obiekt nie zostanie znaleziony, zgłasza wyjątek. Jest to metoda wyszukiwania. Jeśli zostaną przekazane argumenty `i` oraz `j`, metoda zwraca najmniejszy indeks `k`, dla którego `T[k] == x` oraz `i <= k < j`. Domyślnie `j` ma wartość `len(T)`.

```
T.count(x)
```

Zwraca liczbę wystąpień obiektu `x` wewnątrz krotki `T`.

Pliki

Wbudowana funkcja `open()` tworzy *obiekt pliku* — najpopularniejszy interfejs do zewnętrznych plików. Obiekty plików eksportują metody transferu danych opisane w następnych podrozdziałach. Zawartość pliku jest reprezentowana przez ciągi znaków Pythona. Poniższa lista nie zawiera wszystkich metod — informacji na temat mniej popularnych wywołań i atrybutów należy szukać w podręcznikach dotyczących Pythona.

W Pythonie 2.X podczas tworzenia obiektu pliku można użyć nazwy `file()` jako synonimu metody `open()`, ale ogólnie rzecz biorąc, zalecane jest korzystanie z metody `open()`. W Pythonie 3.X metoda `file()` nie jest dostępna (do personalizacji działań na plikach służą klasy modułu `io`).

Więcej informacji na temat tworzenia plików można znaleźć w podrozdziale „Funkcje wbudowane”. Warto się również zapoznać z podrozdziałem „Łańcuchy znaków Unicode”, w którym opisano różnice pomiędzy plikami tekstowymi a binarnymi oraz pomiędzy typami łańcuchowymi w Pythonie 2.X i Pythonie 3.X.

Powiązane z plikami narzędzia są opisane w dalszej części książki — zobacz opis modułów `dbm`, `shelve` i `pickle` w podrozdziale „Moduły

utrwalania obiektów”; funkcje plikowe bazujące na deskryptorach z modułu `os`, a także narzędzia przetwarzania ścieżek katalogów `os.path` są omówione w podrozdziale „Moduł systemowy `os`”; zagadnienia związane z przechowywaniem plików JSON opisano w podrozdziale „Moduł `json`”, natomiast opis interfejsu API baz danych SQL zawarto w podrozdziale „API baz danych Python SQL”.

Warto także sięgnąć do podręczników Pythona, aby zapoznać się z opisami metody `objektgniazda.makefile()` — pozwalającej na konwersję gniazda na obiekt podobny do pliku — oraz metod `io.StringIO(str)` i `io.BytesIO(bytes)` (`StringIO.StringIO(str)` w Pythonie 2.X), które służą do konwertowania łańcuchów znaków na obiekt podobny do pliku. Wywołania te są zgodne z interfejsami API oczekującymi interfejsu obiektu plikowego opisanego w tym podrozdziale.

Pliki wejściowe

```
infile = open(filename, 'r')
```

Tworzy plik wejściowy powiązany z zewnętrznym plikiem o podanej nazwie. Argument *filename* jest zazwyczaj ciągiem znaków (np. `'data.txt'`). Wskazuje on na bieżący katalog roboczy, o ile nie zawiera prefiksu ze ścieżką do katalogu (np. `r'c:\dir\data.txt'`). Drugi argument oznacza *tryb* otwarcia pliku: `'r'` to odczyt tekstu, natomiast `'rb'` — odczyt danych binarnych bez translacji znaków przejścia do nowego wiersza. Argument oznaczający tryb otwarcia pliku jest opcjonalny, a jego wartość domyślna to `'r'`. Metoda `open()` w Pythonie 3.X akceptuje także opcjonalną nazwę kodowania Unicode w trybie tekstowym. Dla metody `codecs.open()` z Pythona 2.X istnieją podobne narzędzia.

```
infile.read()
```

Czyta zawartość całego pliku i zwraca ją w postaci pojedynczego łańcucha znaków. W trybie tekstowym (`'r'`) znaki zakończenia wiersza są przekształcane na `'\n'`. W trybie binarnym (`'rb'`) wynikowy łańcuch znaków może zawierać znaki niedrukowalne (np. `'\0'`). W Pythonie 3.X w trybie tekstowym metoda *dekoduje* tekst Unicode na postać obiektu `str`, natomiast w trybie binarnym zwraca nieprzetworzoną zawartość w postaci obiektu `bytes`.

```
infile.read(N)
```

Czyta najwyżej *N* kolejnych bajtów (1 lub więcej). W przypadku osiągnięcia końca pliku zwraca pusty ciąg.

`infile.readline()`

Czyta następny wiersz (wiersze są rozdzielane znacznikiem końca wiersza). W przypadku osiągnięcia końca pliku zwraca pusty ciąg.

`infile.readlines()`

Czyta cały plik do listy złożonej z łańcuchów znaków odpowiadających kolejnym wierszom. Patrz także opis iteratorów wierszy obiektów plikowych zawarty w następnej pozycji listy.

for *line* in *infile*

Wykorzystuje *iterator wierszy* obiektu pliku *infile* do automatycznego przeglądania krok po kroku wierszy pliku. Instrukcja dostępna we wszystkich kontekstach iteracyjnych, włącznie z pętlami `for`, funkcją `map()` oraz obiektami składanymi (np. `[line.rstrip() for line in open('nazwapliku')]`). Iteracja w postaci `for line in infile` daje efekt podobny do instrukcji `for line in infile.readlines()`, ale wersja iteratora wierszy pobiera wiersze na żądanie, zamiast ładować cały plik do pamięci, w związku z czym jest wydajniejsza.

Pliki wyjściowe

`outfile = open(filename, 'w')`

Tworzy obiekt pliku wyjściowego podłączony do zewnętrznego pliku o nazwie *filename* (zdefiniowanego w poprzednim punkcie). Tryb `'w'` oznacza zapis tekstu, natomiast `'wb'` — zapis danych binarnych bez translacji znaków przejścia do nowego wiersza. Metoda `open()` w Pythonie 3.X akceptuje także opcjonalną nazwę kodowania Unicode w trybie tekstowym. Dla metody `codecs.open()` z Pythona 2.X istnieją podobne narzędzia.

`outfile.write(S)`

Zapisuje całą zawartość łańcucha *S* do pliku (bez stosowania formatowania). W trybie tekstowym sekwencja `'\n'` jest domyślnie przekształcana na ciąg odpowiadający znakowi końca wiersza dla specyficznej platformy. W trybie binarnym łańcuch znaków może zawierać znaki niedrukowalne (np. aby zapisać ciąg złożony z pięciu bajtów, z których dwa to binarne bajty 0, można użyć sekwencji `'a\0b\0c'`). W Pythonie 3.X tryb tekstowy wymaga łańcuchów `str` Unicode, dlatego metoda *koduje* łańcuchy Unicode podczas zapisu, natomiast w trybie binarnym metoda oczekuje łańcuchów w nieprzetworzonej formie w postaci obiektu `bytes` i w taki sposób je zapisuje.

`outfile.writelines(I)`

Zapisuje do pliku wszystkie łańcuchy w obiekcie iterowalnym *I*. Nie dodaje automatycznie znaków zakończenia wierszy.

Pliki dowolnego rodzaju

`file.close()`

Ręczne zamknięcie pliku w celu zwolnienia zasobów (choć implementacja CPython w bieżącej wersji automatycznie zamyka otwarte pliki w procesie odśmiecania — ang. *garbage collection*). Patrz także opis menedżerów kontekstu obiektu `file` w podrödziale „Menedżery kontekstu plików”.

`file.tell()`

Zwraca bieżącą pozycję w pliku.

`file.seek(offset [, whence])`

Ustawia bieżącą pozycję pliku na *offset* dla operacji losowego dostępu. Argument *whence* może mieć wartość 0 (przesunięcie od początku), 1 (przesunięcie w przód lub w tył od bieżącej pozycji) lub 2 (przesunięcie od końca). Domyślna wartość argumentu *whence* wynosi 0.

`file.isatty()`

Zwraca `True`, w przypadku gdy plik jest połączony z interaktywnym urządzeniem typu `tty`, lub `False` w przeciwnym razie (w starszych wersjach Pythona może zwracać 1 lub 0).

`file.flush()`

Opróżnia bufor pliku `stdio`. Metoda przydaje się do obsługi buforowanych potoków, w przypadku gdy z pliku czyta inny proces (lub człowiek). Jest przydatna także w odniesieniu do plików, które są tworzone i czytane w tym samym procesie.

`file.truncate([size])`

Obcina plik do rozmiaru co najwyżej *size* bajtów (lub na bieżącej pozycji, jeśli argument *size* nie zostanie podany). Metoda nie jest dostępna na wszystkich platformach.

`file.fileno()`

Pobiera numer pliku (deskryptor w postaci liczby całkowitej). Metoda ta konwertuje obiekty `file` na deskryptory, które można przekazywać jako argumenty do narzędzi modułu `os`. Wskazówka: aby dokonać konwersji deskryptora pliku na obiekt pliku, można skorzystać z metody `os.fdopen()` bądź metody `open()` Pythona 3.X.

Inne atrybuty plików (część tylko do odczytu)

`file.closed`

True, jeśli plik został zamknięty.

`file.mode`

Ciąg znaków oznaczający tryb otwarcia pliku (np. 'r') przekazany do funkcji `open`.

`file.name`

Nazwa zewnętrznego pliku w postaci łańcucha znaków.

Menedżery kontekstu plików

W standardowym Pythonie (CPython) otwarte pliki są automatycznie zamykane w procesie odświeżania. Z tego względu pliki tymczasowe (np. `open('name').read()`) nie muszą być jawnie zamykane. Obiekty plikowe są natychmiast zwalniane i zamykane. Jednak w innych implementacjach Pythona (np. Jython) pliki mogą być zbierane i zamykane w sposób mniej jawny.

Aby zagwarantować zamknięcie pliku po zakończeniu działania bloku kodu, należy użyć instrukcji `try/finally` i ręcznie zamknąć pliki, niezależnie od tego, czy określony blok zgłasza wyjątek, czy nie:

```
myfile = open(r'C:\misc\script', 'w')
try:
    ...skorzystanie ze zmiennej myfile...
finally:
    myfile.close()
```

Można też skorzystać z instrukcji `with/as` dostępnej w Pythonie 3.X i 2.X (od wersji 2.6 i 3.0):

```
with open(r'C:\misc\script', 'w') as myfile:
    ...skorzystanie ze zmiennej myfile...
```

W pierwszym przypadku następuje wstawienie wywołania `close` w bloku zamykającym. Drugi sposób polega na wykorzystaniu *menedżerów kontekstu*, które gwarantują automatyczne zamknięcie pliku w momencie zakończenia bloku kodu. Warto zapoznać się z instrukcjami `try` i `with` w podrozdziale „Instrukcje i ich składnia”.

Uwagi na temat korzystania z plików

- Niektóre tryby otwarcia plików (np. 'r+') umożliwiają wykorzystanie pliku zarówno jako wejściowego, jak i wyjściowego. Inne tryby (np. 'rb') oznaczają transfer danych w trybie binarnym

w celu wyłączenia konwersji znaczników końca wiersza (oraz kodowania Unicode w Pythonie 3.X). Więcej informacji można znaleźć w opisie metody `open()` w podrozdziale „Funkcje wbudowane”.

- Operacje transferu danych z plików są wykonywane od bieżącej pozycji w pliku. Za pomocą metody `seek()` można zmienić tę pozycję, co pozwala na losowy dostęp do danych w pliku.
- Dla operacji transferu danych z pliku można wyłączyć *buforowanie*. Więcej informacji można znaleźć w opisie metody `open()` w podrozdziale „Funkcje wbudowane”, a także w opisie flagi wiersza poleceń `-u` w podrozdziale „Opcje wiersza poleceń Pythona”.
- W Pythonie 2.X dostępna jest także metoda `xreadlines()` obiektu `file`, która działa tak samo jak automatyczny iterator wierszowy obiektu `file`. Metodę tę usunięto z Pythona 3.0 ze względu na redundancję.

Zbiory

Zbiory to *mutowalne* (modyfikowalne) i nieuporządkowane kolekcje *unikatowych i niemutowalnych* obiektów. Ten typ danych pozwala na wykonywanie matematycznych działań na zbiorach, takich jak wyznaczanie sumy oraz części wspólnej. Zbiory nie są sekwencjami (są nieuporządkowane), a także nie są odwzorowaniami (nie realizują odwzorowania wartości na klucze), ale obsługują iteracje i funkcje podobnie jak słowniki bez wartości (złożone wyłącznie z kluczy).

Literały i ich tworzenie

W Pythonie 2.X i 3.X zbiory można tworzyć poprzez wywołanie wbudowanej funkcji `set()` z argumentem w postaci obiektu iterowalnego. Jego elementy staną się elementami utworzonego zbioru. W Pythonie 3.X i 2.7 zbiory można również tworzyć za pomocą literału `{...}` oraz wyrażenia zbiorów składanych, chociaż do tworzenia pustego zbioru (`{}` oznacza pusty słownik) i do budowania zbiorów na podstawie istniejących obiektów jest używane wywołanie `set()`.

Zbiory są mutowalne, ale elementy zbiorów muszą być niemutowalne. Wbudowana funkcja `frozenset()` tworzy niemutowalny zbiór, który można zagnieźdzać wewnątrz innego zbioru.

`set()`

Pusty zbiór (`{}` oznacza pusty słownik).

`S = set('spam')`

Czteroelementowy zbiór — wartości `'s'`, `'p'`, `'a'`, `'m'` (funkcja pozwala na przekazanie dowolnego obiektu iterowalnego).

`S = {'s', 'p', 'a', 'm'}`

Czteroelementowy zbiór, taki sam jak ten, który utworzono za pomocą instrukcji z poprzedniego wiersza (w Pythonie 3.X i 2.7).

`S = {ord(c) for c in 'spam'}`

Wyrażenie tworzenia zbioru składanego (w Pythonie 3.X i 2.7). Z jego składnią można się zapoznać w podrozdziale „Listy składane”.

`S = frozenset(range(-5, 5))`

Zamrożony (niemutowalny) zbiór 10 liczb całkowitych `-5...4`.

Działania

Poniżej opisano najważniejsze działania na zbiorach. *S*, *S1* i *S2* oznaczają dowolny zbiór. Większość operatorów stosowanych w wyrażeniach wymaga dwóch zbiorów, ale ich odpowiedniki w postaci metod akceptują dowolne *obiekty iterowalne* (w poniższym opisie oznaczane jako *inny*). Na przykład wyrażenie `{1, 2} | [2, 3]` jest nieprawidłowe, ale wywołanie `{1, 2}.union([2, 3])` jest już poprawne. Poniższa lista jest reprezentatywna, ale nie jest kompletna. Wyczerpującą listę wyrażzeń przetwarzania zbiorów oraz dostępnych metod można znaleźć w podręczniku *Python's Library Reference*.

`x in S`

Członkostwo — zwraca `True`, jeżeli zbiór *S* zawiera *x*.

`S1 - S2, S1.difference(inny)`

Różnica — nowy zbiór zawierający elementy, które należą do zbioru *S1*, ale nie należą do zbioru *S2* (lub zbioru *inny*).

`S1 | S2, S1.union(inny)`

Unia — nowy zbiór zawierający elementy, które należą do zbioru *S1* lub do zbioru *S2* (lub zbioru *inny*) bez duplikatów.

`S1 & S2, S1.intersection(inny)`

Część wspólna — nowy zbiór zawierający elementy, które należą zarówno do zbioru *S1*, jak i do zbioru *S2* (lub zbioru *inny*).

- `S1 <= S2, S1.issubset(inny)`
Podzbiór — sprawdza, czy każdy element należący do zbioru *S1* należy także do zbioru *S2* (lub do zbioru *inny*).
- `S1 >= S2, S1.issuperset(inny)`
Nadzbior — sprawdza, czy każdy element należący do zbioru *S2* (lub do zbioru *inny*) należy także do zbioru *S1*.
- `S1 < S2, S1 > S2`
Rzeczywisty podzbiór i nadzbiór — jednocześnie sprawdza, czy zbiory *S1* i *S2* nie są identyczne.
- `S1 ^ S2, S1.symmetric_difference(inny)`
Różnica symetryczna — nowy zbiór zawierający elementy, które należą do zbioru *S1* lub do zbioru *S2* (albo zbioru *inny*), ale nie należą do obu jednocześnie.
- `S1 |= S2, S1.update(inny)`
Aktualizuje zbiór (metoda niedozwolona dla zbiorów zamrożonych) — dodaje elementy ze zbioru *S2* (lub zbioru *inny*) do zbioru *S1*.
- `S.add(x), S.remove(x), S.discard(x), S.pop(), S.clear()`
Aktualizuje zbiór (metoda niedozwolona dla zbiorów zamrożonych) — dodaje element, usuwa element o podanej wartości, usuwa element, jeśli istnieje, usuwa i zwraca dowolny element, usuwa wszystkie elementy.
- `len(S)`
Długość — liczba elementów w zbiorze.
- `for x in S`
Iteracja — wszystkie konteksty iteracyjne.
- `S.copy()`
Tworzy kopię najwyższego poziomu (płytką) zbioru *S*. Ma działanie identyczne jak metoda `set(S)`.

Inne typy i konwersje

Do zbioru zasadniczych typów Pythona należą również *typ logiczny* (opisany w następnym podrozdziale), `None` (obiekt-wypełniacz); `NotImplemented` (używany w metodach przeciążania operatorów), `Ellipsis` (tworzony za pomocą literału `...` w Pythonie 3.X), *typy* dostępne za pośrednictwem wbudowanej funkcji `type()`, które w Pythonie 3.X

zawsze są klasami, a także typy odpowiadające *jednostkom programu*, takie jak funkcje, moduły i klasy (wszystkie obiekty runtime i obiekty pierwszego rzędu Pythona).

Boolean

Logiczny typ danych o nazwie `bool` dostarcza dwie predefiniowane stałe `True` i `False` (typ `bool` jest dostępny od wersji 2.3). W większości zastosowań stałe te można traktować tak, jakby były liczbami całkowitymi odpowiednio 1 i 0 (np. wyrażenie `True + 3` ma wartość 4). Typ `bool` jest jednak podklasą typu `int`, a jego instancje wyświetlają się inaczej niż instancje typu `int` (wartość `True` wyświetla się jako „True”, a nie „1”, i można ją wykorzystywać w roli wbudowanej nazwy w testach logicznych).

Konwersje typów

W tabelach 10. i 11. zdefiniowano popularne wbudowane narzędzia służące do konwersji z jednego typu na inny. Wszystkie one tworzą *nowe* obiekty (nie są to konwertery działające w miejscu). W Pythonie 2.X są dodatkowo dostępne konwertery `long(S)` na `long` oraz ``X`` na `string`; konwertery te usunięto w Pythonie 3.X. Więcej informacji na temat niektórych narzędzi wymienionych w poniższych tabelach można znaleźć w podrozdziałach „Liczby” i „Formatowanie łańcuchów znaków”.

Tabela 10. Konwertery typów sekwencyjnych

Konwerter	Konwersja z	Konwersja na
<code>list(X)</code> , <code>[n for n in X]</code> ¹¹	łańcuch znaków, krotka, dowolny obiekt iterowalny	Lista
<code>tuple(X)</code>	łańcuch znaków, lista, dowolny obiekt iterowalny	Krotka
<code>''.join(X)</code>	Obiekt iterowalny złożony ze znaków	łańcuch znaków

¹¹ Forma listy składanej może działać wolniej od metody `list()`. W tym konkretnym kontekście iteracji zastosowanie listy składanej nie jest najlepszą praktyką. W Pythonie 2.X wywołanie `map(None, X)` przynosi w tym kontekście ten sam skutek co wywołanie `list(X)`, ale tę formę wywołania metody `map()` usunięto w Pythonie 3.X.

Tabela 11. Konwertery typów łańcuchowych (obiektów)

Konwerter	Konwersja z	Konwersja na
<code>eval(S)</code>	Łańcuch znaków	Dowolny obiekt o składni wyrażenia
<code>int(S [, base])</code> ¹² , <code>float(S)</code>	Łańcuch znaków lub liczba	<code>integer</code> , <code>float</code>
<code>repr(X)</code> , <code>str(X)</code>	Dowolny obiekt Pythona	Łańcuch znaków (<code>repr</code> to reprezentacja w postaci kodów znaków, postać <code>str</code> jest bardziej przyjazna dla użytkownika)
<code>F % X</code> , <code>F.format(X)</code> , <code>format(X, [F])</code>	Obiekty z kodami formatu	Łańcuch znaków
<code>hex(X)</code> , <code>oct(X)</code> , <code>bin(X)</code> , <code>str(X)</code>	Typy całkowite	Ciągi znaków zawierających tekstowe reprezentacje liczb w formacie szesnastkowym, ósemkowym, dwójkowym i dziesiętnym
<code>ord(C)</code> , <code>chr(I)</code>	Znak, kod znaku (<code>integer</code>)	Kod znaku (<code>integer</code>), znak

Instrukcje i ich składnia

W tym podrozdziale opisano reguły składni oraz zasady dotyczące nazewnictwa zmiennych.

Reguły składniowe

Poniżej zestawiono ogólne zasady pisania programów w Pythonie:

Przepływ sterowania

Instrukcje uruchamiają się kolejno, jedna po drugiej, o ile nie zostaną użyte instrukcje sterujące (`if`, `while`, `for`, `raise`, wywołania itp.).

¹² W wersji 2.2 i wersjach późniejszych funkcje konwerterów (np. `int()`, `float()`, `str()`) służą także jako konstruktory klas i można na ich podstawie tworzyć podklasy. W Pythonie 3.X wszystkie typy są klasami, a wszystkie klasy są instancjami klasy `type`.

Zagnieżdżony blok wyróżnia się poprzez wcięcie wszystkich jego instrukcji o tę samą liczbę spacji bądź tabulacji. Zagnieżdżony blok może także występować w tym samym wierszu co jego nagłówek instrukcji, za znakiem: w nagłówku, o ile składa się tylko z prostych (niezłożonych) instrukcji.

Obowiązuje ogólna zasada, zgodnie z którą określony blok powinien wykorzystywać wszystkie tabulacje lub wszystkie spacje dla wcięcia. W przypadku użycia zarówno tabulacji, jak i spacji obowiązują następujące dwie reguły: (1) tabulacja jest liczona jako wystarczająca liczba spacji do przesunięcia kolumny o liczbę pozycji równą kolejnej wielokrotności liczby 8 oraz (2) w przypadku wykrycia kolejnych niespójności każda tabulacja jest liczona jako pojedyncza spacja.

W *Pythonie 2.X* kombinacje tabulacji i spacji są dozwolone, o ile spełniają tylko pierwszą regułę, jednak mieszanie tabulacji ze spacjami nie jest zalecane, ponieważ zwiększa podatność na błędy i obniża czytelność. Do oznaczenia kombinacji uznanych za niespójne z drugą regułą można użyć opcji `-t` bądź `-tt` (patrz „Opcje wiersza poleceń Pythona”). W *Pythonie 3.X* kombinacje tabulacji i spacji są nadal dozwolone, jeśli są one zgodne z *obydwoma* regułami, ale w innym przypadku zawsze są uznawane za błędy (tak samo jak w przypadku zastosowania opcji `-tt` w *Pythonie 2.X*).

Na przykład zarówno w *Pythonie 3.X*, jak i *2.X* blok zewnętrzny wcięty za pomocą dwóch spacji, jednej tabulacji i dwóch spacji (reguła 1.: 10, reguła 2.: 5) pozwala na wcięcie wewnętrznego bloku za pomocą jednej tabulacji i pięciu spacji (reguła 1.: 13, reguła 2.: 6). Wewnętrzny blok z dwoma tabulacjami i jedną spacją (reguła 1.: 17, reguła 2.: 3) jest domyślnie prawidłowy w *Pythonie 2.X* (spełnia regułę 1.), ale nie jest poprawny w *Pythonie 3.X* (reguła 2.). W kodzie, który powinien być łatwy w utrzymaniu, ogólnie rzecz biorąc, nie należy polegać na tych subtelnych regułach — należy używać albo samych tabulacji, albo samych spacji.

Instrukcje

Instrukcja kończy się wraz z końcem wiersza, ale może być kontynuowana w kolejnych wierszach, kiedy fizyczny wiersz kończy się ukośnikiem (`\`), niezamkniętym nawiasem okrągłym, kwadratowym bądź klamrowym albo niezamkniętym łańcuchem

znaków z potrójnym apostrofem (cudzysłowem). W jednym wierszu może występować wiele prostych instrukcji, pod warunkiem że zostaną rozdzielone średnikiem (;).

Komentarze

Komentarze rozpoczynają się od znaku # (ale nie wtedy, gdy znak ten występuje w stałej łańcuchowej), a kończą wraz z końcem wiersza.

Ciągi dokumentacyjne

Jeśli funkcja, plik modułu bądź klasa rozpoczynają się literałem łańcuchowym (ewentualnie za komentarzem #), to jest on zapisywany w atrybucie `__doc__` obiektu. Więcej informacji na temat automatycznego wydobywania dokumentacji oraz narzędzi do jej wyświetlania można znaleźć w opisie funkcji `help()` w podrödziale „Funkcje wbudowane” oraz w opisie modułu `pydoc` oraz skryptów w podröczniku *Python Library Reference*. Wskazówka: począwszy od Pythona w wersji 3.2, polecenie `python -m pydoc -b` uruchamia interfejs *PyDoc* w przeglądarce (aby włączyć tryb GUI we wcześniejszych wydaniach, należy zamiast opcji `-b` użyć opcji `-g`).

Białe spacje

Ogólnie rzecz biorąc, białe spacje mają znaczenie tylko z lewej strony kodu, w miejscach, gdzie do grupowania bloków stosuje się wcięcia. W pozostałych przypadkach puste wiersze i spacje są ignorowane, chyba że pełnią funkcję separatorów znaczników lub występują wewnątrz stałych łańcuchowych.

Reguły dotyczące nazw

Poniżej zawarto reguły dotyczące występujących w programach nazw definiowanych przez użytkownika (tzn. zmiennych).

Format nazwy

Struktura

Nazwy definiowane przez użytkowników mogą się rozpoczynać literą bądź znakiem podkreślenia (`_`). Dalej może występować dowolna liczba liter, cyfr bądź znaków podkreślenia.

Słowa zarezerwowane

Żadna z nazw definiowanych przez użytkownika nie może być taka sama jak dowolne z zarezerwowanych słów Pythona zestawionych w tabeli 12.¹³

Tabela 12. Słowa zarezerwowane w Pythonie 3.X

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Rozróżnianie wielkich i małych liter

W nazwach definiowanych przez użytkowników oraz słowach zarezerwowanych wielkość liter zawsze ma znaczenie: *SPAM*, *spam* i *Spam* są różnymi nazwami.

Nieużywane znaczniki

W konstrukcjach składniowych Pythona nie wykorzystuje się znaków \$ i ?, choć mogą się one pojawić w stałych łańcuchowych i komentarzach. Znak \$ w łańcuchach znaków ma specjalne znaczenie w mechanizmie zastępowania wzorców (patrz „Zastępowanie szablonów w łańcuchach znaków”), natomiast znaki \$ i ? spełniają specjalne role w mechanizmie dopasowywania wzorców (patrz „Moduł dopasowywania wzorców re”).

Tworzenie

Nazwy definiowane przez użytkowników tworzy się poprzez przypisanie, ale w momencie odwoływania się do nich muszą być zdefiniowane (np. liczniki muszą być jawnie zainicjowane na wartość zero). Patrz podrozdziały „Wyrażenia atomowe i dynamiczne określanie typów” oraz „Przestrzenie nazw i reguły zasięgu”.

¹³ Jednak zasada ta nie może być ani absolutna, ani surowo przestrzegana poza implementacją CPython. Na przykład w implementacji Pythona w Javie — Jython — w niektórych kontekstach słowa zarezerwowane mogą być wykorzystywane jako zmienne.

UWAGA

Ponieważ w *Pythonie 2.X* `print` i `exec` mają postać instrukcji, a nie funkcji wbudowanych, oba te słowa są zastrzeżone. Z kolei słowa `nonlocal`, `True` i `False` w *Pythonie 2.X* nie są zarezerwowane. Pierwsze z nich jest niedostępne, natomiast dwa pozostałe są po prostu nazwami wbudowanymi. Słowa `with` i `as` są zarezerwowane od wersji 2.6 i 3.0, ale nie są zarezerwowane we wcześniejszych wydaniach 2.X, o ile nie zostaną jawnie włączone menedżery kontekstu. Słowo `yield` jest zarezerwowane, począwszy od wersji 2.3. W późniejszych wersjach zostało ono przekształcone z instrukcji na wyrażenie, ale nadal jest słowem zarezerwowanym.

Konwencje nazewnictwa

- Nazwy rozpoczynające się i kończące dwoma znakami podkreślenia (na przykład `__init__`) mają dla interpretera specjalne znaczenie, ale nie są słowami zarezerwowanymi.
- Nazwy rozpoczynające się od jednego znaku podkreślenia (np. `_x`), którym są przypisywane wartości na najwyższym poziomie modułu, nie są kopiowane w wyniku wykonywania operacji importowania `from...*` (warto także zapoznać się z listą `__all__` nazw eksportów modułu, o której mowa w podrozdziałach „Instrukcja `from`” oraz „Atrybuty pseudoprywatne”). W innych kontekstach jest to nieformalna konwencja dla nazw wewnętrznych.
- Nazwy rozpoczynające się dwoma podkreśleniami (ale niezakończone taką sekwencją) — np. `__x` — w obrębie instrukcji `class` są poprzedzone prefiksem w postaci nazwy klasy, w której je zdefiniowano (patrz podrozdział „Atrybuty pseudoprywatne”).
- Nazwy składającej się z jednego znaku podkreślenia (`_`) używa się w interpreterze interaktywnym (wyłącznie) do zapisania wyników ostatniej operacji obliczenia wartości.
- Nazwy funkcji wbudowanych i wyjątków (np. `open`, `SyntaxError`) nie są słowami zarezerwowanymi. Nazwy te „żyją” w ostatnio przeszukiwanym zasięgu i mogą być przypisane powtórnie w celu ukrycia (zwanego również *zacienianiem* — ang. *shadow*) wbudowanego znaczenia w bieżącym zasięgu (np. `open = myfunction`).
- Nazwy klas zwykle są zapisywane wielką literą (np. `MyClass`), a nazwy modułów — małą (np. `mymodule`).

- Pierwszy argument funkcji, która jest metodą klasy, zwykle ma nazwę `self`. Jest to powszechnie przyjęta konwencja.
- Nazwy modułów są rozwiązywane zgodnie z regułą skanowania ścieżki przeszukiwania katalogów — nazwy zlokalizowane wcześniej na ścieżce mogą ukryć inne o tej samej nazwie, niezależnie od tego, czy zostały one użyte po wcięciu, czy nie (patrz podrozdział „Instrukcja `import`”).

Instrukcje

W kolejnych podrozdziałach zostaną opisane wszystkie instrukcje występujące w Pythonie. W każdym podrozdziale zamieszczono format składni instrukcji, a za nim szczegółowe informacje na temat jej wykorzystania. W przypadku instrukcji złożonych każde wystąpienie ciągu *grupa* wewnątrz formatu instrukcji oznacza jedną lub więcej innych instrukcji. Instrukcje te mogą tworzyć blok pod wierszem nagłówka. Dla grupy trzeba zastosować wcięcie pod nagłówkiem, jeśli zawiera inną instrukcję złożoną (`if`, `while` itp.). W przeciwnym razie może się znaleźć w tym samym wierszu co nagłówki instrukcji. Obie poniższe konstrukcje są prawidłowe:

```
if x < 42:
    print(x)
    while x: x = x - 1

if x < 42: print(x)
```

Poniżej zamieszczono szczegółowe informacje wspólne dla wersji Pythona 3.X i 2.X. Szczegóły obowiązujące jedynie w wersji 2.X zestawiono na końcu podrozdziału „Instrukcje w Pythonie 2.X”.

Instrukcja przypisania

```
cel = wyrażenie
cell = cel2 = wyrażenie
cell, cel2 = wyrażenie1, wyrażenie2
cell += wyrażenie

cell, cel2, ... = obiekt_iterowalny-o-tej-samej-długości
(cell1, cel2, ...) = obiekt_iterowalny-o-tej-samej-długości
[cell1, cel2, ...] = obiekt_iterowalny-o-tej-samej-długości
cell, *cel2, ... = obiekt_iterowalny-o-odpowiedniej-długości
```

W instrukcjach przypisania cele zawierają *referencje* do obiektów. Instrukcje przypisania powinny mieć jawny format zamieszczony powyżej, w którym:

- *wyrażenia* generują obiekty;
- *cele* mogą być prostymi nazwami (x), kwalifikowanymi atrybutami ($x.attr$) albo indeksami i wycinkami ($x[i]$, $x[i:j:k]$);
- *zmienne* wewnątrz celów nie są deklarowane wcześniej, ale przed użyciem ich w wyrażeniu trzeba nadać im wartość (patrz podrozdział „Wyrażenia atomowe i dynamiczne określanie typów”).

Pierwszy format wymieniony powyżej to *proste* przypisanie. Drugi z formatów — przypisanie *wielocelowe* (ang. *multiple-target*) — służy do przypisania tego samego wyrażenia do każdego z celów. Trzeci format — przypisanie *krotek* — łączy w pary cele z wyrażeniami, od lewej do prawej. Czwarty format — *przypisanie z aktualizacją* — jest skrótem dla połączenia operacji dodawania z przypisaniem (patrz następny podrozdział).

Ostatnie cztery formaty to *przypisanie sekwencji*, które służy do przypisywania komponentów dowolnej sekwencji (bądź innego obiektu iterowalnego) do odpowiadających im celów — od lewej do prawej. Sekwencja lub obiekt iterowalny występujące po prawej stronie mogą być dowolnego typu, ale muszą być tej samej długości, chyba że wśród celów z lewej strony znajdzie się nazwa rozpoczynająca się gwiazdką ($*x$), tak jak w ostatnim formacie. Ten ostatni format jest znany jako *rozszerzone przypisanie sekwencji*. Instrukcja o takim formacie jest dostępna wyłącznie w Pythonie 3.X. Pozwala ona na pobranie dowolnej liczby elementów (patrz podrozdział „Rozszerzone instrukcje przypisania sekwencji (3.X)”).

Przypisania mogą również występować *niejawnie* w innych kontekstach w Pythonie (np. zmienne pętli `for` oraz mechanizm przekazywania argumentów funkcji). Niektóre z formatów instrukcji przypisania można również stosować w innych miejscach (np. przypisania sekwencji w instrukcjach `for`).

Przypisanie z aktualizacją

W Pythonie dostępny jest zbiór dodatkowych formatów instrukcji przypisania. Zestawiono je w tabeli 13. Są to tzw. *przypisania z aktualizacją* — formaty te implikują wyrażenie dwuargumentowe razem z przypisaniem. Na przykład poniższe dwa formaty w przybliżeniu są sobie równoważne:

```
 $x = x + y$ 
 $x += y$ 
```

Tabela 13. Instrukcje przypisania z aktualizacją

$X += Y$	$X \&= Y$	$X -= Y$	$X = Y$
$X *= Y$	$X \wedge= Y$	$X /= Y$	$X >>= Y$
$X \%= Y$	$X <=<= Y$	$X **= Y$	$X // = Y$

Jednak wartość celu X w instrukcji drugiego formatu może być wyznaczona *tylko raz*. Dodatkowo format ten pozwala na zastosowanie działań *w miejscu* dla typów mutowalnych (np. wyrażenie `list1 += list2` automatycznie wywołuje metodę `list1.extend(list2)` zamiast wolniejszej operacji konkatenacji implikowanej przez operator `+`). W klasach przypisania w miejscu mogą być przeciążane za pomocą nazw metod rozpoczynających się od `i` (np. `__iadd__()` dla operatora `+=`, `__add__()` dla operatora `+`). W Pythonie w wersji 2.2 wprowadzono nowy format: $X // = Y$ (dzielenie całkowite).

Zwykłe instrukcje przypisania sekwencji

W Pythonie 2.X i 3.X dowolną sekwencję (bądź też inny obiekt iterowalny) złożoną z wartości można przypisać do dowolnej sekwencji nazw, pod warunkiem że ich długości są takie same. Podstawowa forma instrukcji przypisania sekwencji działa w większości kontekstów przypisania:

```
>>> a, b, c, d = [1, 2, 3, 4]
>>> a, d
(1, 4)

>>> for (a, b, c) in [[1, 2, 3], [4, 5, 6]]:
...     print(a, b, c)
...
1 2 3
4 5 6
```

Rozszerzone instrukcje przypisania sekwencji (3.X)

W Pythonie 3.X (wyłącznie) instrukcja przypisania sekwencji została rozszerzona, by umożliwić obsługę przypisania sekwencji o dowolnej liczbie elementów — wystarczy poprzedzić gwiazdką jedną ze zmiennych celu przypisania. W takim przypadku długości sekwencji nie muszą być identyczne, natomiast nazwa poprzedzona gwiazdką pobiera niepasujące elementy do nowej listy:

```
>>> a, *b = [1, 2, 3, 4]
>>> a, b
(1, [2, 3, 4])
```

```
>>> a, *b, c = (1, 2, 3, 4)
>>> a, b, c
(1, [2, 3], 4)

>>> *a, b = 'spam'
>>> a, b
(['s', 'p', 'a'], 'm')

>>> for (a, *b) in [[1, 2, 3], [4, 5, 6]]:
...     print(a, b)
...
1 [2, 3]
4 [5, 6]
```

UWAGA

Uogólnienie gwiazdki w Pythonie 3.5 lub w wersjach późniejszych?

W Pythonie 3.3 i wcześniejszych wersjach specjalne formy składni `*X` i `**X` mogą występować w trzech miejscach: w *instrukcjach przypisania*, gdzie `*X` pobiera niedopasowane elementy w instrukcjach przypisania sekwencji; w *nagłówkach funkcji*, w których te dwie formy służą do zbierania niepasujących argumentów pozycyjnych i argumentów w postaci słów kluczowych; oraz w *wywołaniach funkcji*, gdzie wymienione dwie formy rozpakowują obiekty iterowalne i słowniki do indywidualnych elementów (argumentów).

W Pythonie 3.4 deweloperzy planowali uogólnić składnię z gwiazdką w taki sposób, by można jej było używać także w *literalach opisujących struktury danych*, w których miałyby ona powodować rozpakowywanie kolekcji do postaci pojedynczych elementów, podobnie jak działa to obecnie w wywołaniach funkcji. Rozpakowująca składnia z gwiazdką będzie mogła być wykorzystywana dla *krotek*, *list*, *zbiorów*, *słowników* i *obiektów składanych*. Na przykład:

```
[x, *iter]           # rozpakowanie elementów obiektu iterowalnego iter: lista
(x, *iter), {x, *iter} # to samo dla krotki, zbiór
{'x': 1, **dict}      # rozpakowanie elementów słownika: słowniki
[*iter for iter in x]  # rozpakowanie elementów obiektu iterowalnego:
                       # obiekty składane
```

Są to dodatkowe miejsca zastosowania składni z gwiazdką oprócz instrukcji przypisania, nagłówków funkcji i wywołań funkcji. Dodatkowo być może zostaną zniesione niektóre obecne ograniczenia w zakresie korzystania z tej składni. Proponowana zmiana została przesunięta na wydanie po wersji 3.4, tuż przed ukazaniem się niniejszej książki, i pozostaje niepewna. W istocie dyskutowano o niej od 2008 roku. Zmiana ta nie będzie rozpatrywana do wydania Pythona 3.5 lub wersji późniejszej i być może nie pojawi się w ogóle. Szczegółowe informacje można znaleźć w dokumencie „What’s New”.

Instrukcja wyrażeniowa

wyrażenie

funkcja([wartość, nazwa=wartość, *nazwa, **nazwa...])

obiekt.metoda([wartość, nazwa=wartość, *nazwa, **nazwa...])

Dowolne wyrażenie może być instrukcją (np. występujące samodzielnie w wierszu). Z drugiej strony instrukcje nie mogą występować w żadnym innym kontekście wyrażenia (np. instrukcje przypisania nie zwracają wyników i nie mogą być zagnieżdżane).

Wyrażenia są powszechnie używane do wywoływania funkcji i metod, które nie zwracają wartości, oraz do wyświetlania w trybie interaktywnym. Instrukcje wyrażeniowe są także najpopularniejszym sposobem kodowania dla wyrażeń `yield` oraz wywołań wbudowanej funkcji `print()` z Pythona 3.X (choć w tej książce instrukcje te opisano osobno).

Składnia wywołań

W wywołaniach funkcji i metod aktualne argumenty wywołania są od siebie oddzielone przecinkami i zwykle są dopasowywane według pozycji do argumentów w nagłówkach `def` funkcji. W wywołaniach można opcjonalnie wymienić specyficzne nazwy argumentów, do których będą przekazywane wartości. W tym celu należy skorzystać ze składni argumentu kluczowego *nazwa=wartość*. Argumenty kluczowe są dopasowywane według nazwy zamiast pozycji.

Składnia dowolnych argumentów wywołań

W listach argumentów wywołania funkcji i metod można zastosować specjalną składnię w celu *rozpakowania* kolekcji do dowolnej liczby argumentów. Jeśli *pargs* i *kargs* to odpowiednio sekwencja (lub inny obiekt iterowalny) oraz słownik:

```
f(*pargs, **kargs)
```

to instrukcja o tym formacie wywoła funkcję `f` z argumentami *pozycyjnymi* z obiektu iterowalnego *pargs* oraz z *argumentami kluczowymi* ze słownika *kargs*. Na przykład:

```
>>> def f(a, b, c, d): print(a, b, c, d)
...
>>> f(*[1, 2], **dict(c=3, d=4))
1 2 3 4
```

Składnię tę dodano po to, by była symetryczna ze składnią argumentów nagłówka funkcji, na przykład `def f(*pargs, **kargs)`, która *pobiera*

niedopasowane argumenty. W wywołaniach elementy poprzedzone gwiazdkami są rozpakowywane do pojedynczych argumentów i mogą być łączone z innymi argumentami pozycyjnymi i kluczowymi zgodnie z regułami porządkowania (np. `g(1, 2, foo=3, bar=4, *pargs, **kargs)`).

W Pythonie 2.X podobny efekt można osiągnąć za pomocą wbudowanej funkcji `apply()`, która w Pythonie 3.X została usunięta:

```
apply(f, pargs, kargs)
```

Więcej informacji na temat składni wywołania można znaleźć w podrozdziale „Instrukcja `def`” (włącznie z tabelą 15.).

Instrukcja `print`

W Pythonie 3.X wyświetlanie tekstu do standardowego strumienia wyjściowego przyjęło postać wywołania wbudowanej funkcji. Zwykle jest ona kodowana jako instrukcja wyrażeniowa (w oddzielnym wierszu). Jej sygnatura wywołania jest następująca:

```
print([wartość [, wartość]*]  
      [, sep=łańcuch-znaków] [, end=łańcuch-znaków]  
      [, file=obiekt] [, flush=bool])
```

Każda *wartość* jest wyrażeniem generującym obiekt, dla którego ma być wyświetlony łańcuch znaków `str()`. To wywołanie konfiguruje się za pomocą trzech argumentów, które mogą być wyłącznie argumentami kluczowymi (jeśli argument będzie pominięty lub zostanie przekazana wartość `None`, przyjmie on wartość domyślną):

`sep`

Łańcuch znaków, który ma być umieszczony pomiędzy wartościami (domyślnie spacja: `' '`).

`end`

Łańcuch znaków do umieszczenia na końcu wyświetlanego tekstu (domyślnie znak nowego wiersza: `'\n'`).

`file`

Obiekt postaci pliku, do którego jest zapisywany tekst (domyślnie standardowe wyjście: `sys.stdout`).

`flush`

Wartość `true` lub `false` umożliwiająca włączenie lub wyłączenie wymuszonego opróżnienia strumienia wyjściowego (wprowadzone w Pythonie 3.3 — domyślnie `False`).

Aby wyłączyć separatory w postaci spacji oraz znaki wysuwu wiersza, można przekazać puste lub dostosowane do własnych potrzeb argumenty `sep` i `end`. W celu przekierowania wyjścia można przekazać nazwę pliku za pomocą argumentu `file` (patrz także podrozdział „Pliki”):

```
>>> print(2 ** 32, 'spam')
4294967296 spam

>>> print(2 ** 32, 'spam', sep='')
4294967296spam

>>> print(2 ** 32, 'spam', end=' '); print(1, 2, 3)
4294967296 spam 1 2 3

>>> print(2 ** 32, 'spam', sep='',
...       file=open('out', 'w'))
>>> open('out').read()
'4294967296spam\n'
```

Ponieważ domyślnie funkcja `print` po prostu wywołuje metodę `write()` obiektu, do którego w danym momencie odwołuje się urządzenie `sys.stdout`, to poniższa sekwencja jest równoważna wywołaniu `print(X)`:

```
import sys
sys.stdout.write(str(X) + '\n')
```

Aby przekierować polecenie wyświetlania tekstu do plików bądź obiektów klasy, należy albo przekazać dowolny obiekt z implementacją metody `write()` do argumentu kluczowego `file`, tak jak pokazano wcześniej, albo przypisać urządzenie `sys.stdout` do dowolnego takiego obiektu (patrz także podrozdział „Pliki”):

```
sys.stdout = open('log', 'a') # dowolny obiekt z metodą write()
print('Ostrzeżenie: spam!')   # wywołanie jest kierowane do metody write() obiektu
```

Ponieważ do urządzenia wyjściowego `sys.stdout` można przypisać nową wartość, argument kluczowy `file` nie jest konieczny. Często jednak pozwala on na uniknięcie zarówno jawnych wywołań metody `write()`, jak i zapisywania i odtwarzania oryginalnej wartości `sys.stdout` wokół wywołania metody `print`, w przypadku gdy pierwotny strumień danych jest w dalszym ciągu wymagany. Więcej informacji na temat działania instrukcji `print()` w Pythonie 3.X można znaleźć w podrozdziale „Funkcje wbudowane”.

Instrukcja `print` w Pythonie 2.X

W Pythonie 2.X wyświetlanie jest instrukcją, a nie funkcją wbudowaną. Jest to instrukcja o następującej postaci:

```
print [wartość [, wartość]* [,]]
print >> plik [, wartość [, wartość]* [,]]
```

Instrukcja `print` z Pythona 2.X wyświetla drukowalne reprezentacje wartości w strumieniu `stdout` (używając bieżącego ustawienia `sys.stdout`) oraz dodaje spacje pomiędzy wartościami. Dodanie przecinka na końcu powoduje wyłączenie znaku wysuwu wiersza, który jest standardowo dodawany na końcu listy. Jest to równoważne użyciu klauzuli `end=' '` w Pythonie 3.X:

```
>>> print 2 ** 32, 'spam'
4294967296 spam

>>> print 2 ** 32, 'spam',; print 1, 2, 3
4294967296 spam 1 2 3
```

Instrukcja `print` z Pythona 2.X pozwala także na podanie nazwy pliku wyjściowego (lub podobnego mu obiektu), który będzie pełnił funkcję celu dla wyświetlanego tekstu zamiast strumienia `sys.stdout`.

```
fileobj = open('log', 'a')
print >> fileobj, "Ostrzeżenie: spam!"
```

Jeśli obiekt pliku ma wartość `None`, wykorzystywany jest strumień `sys.stdout`. Składnia Pythona 2.X `>>` jest równoważna użyciu argumentu kluczowego `file=F` z Pythona 3.X. W instrukcji `print` Pythona 2.X nie istnieje odpowiednik argumentu `sep=S`, chociaż wiersze mogą być sformatowane i wyświetlone jako pojedynczy element.

W instrukcji `print` Pythona 2.X można używać nawiasów okrągłych. W przypadku wielu elementów tworzą one krotki. Aby skorzystać z funkcji wyświetlania Pythona 3.X w Pythonie 2.X, należy uruchomić poniższe instrukcje (w interaktywnej sesji lub na początku skryptu) — z tego mechanizmu można skorzystać zarówno w *Pythonie 2.X* (w celu zachowania zgodności w przód z wersją 3.X), jak i w *Pythonie 3.X* (w celu zachowania wstecznej zgodności z Pythonem 2.X):

```
from __future__ import print_function
```

Instrukcja if

```
if warunek:
    grupa
[elif warunek:
    grupa]*
[else:
    grupa]
```

Instrukcja `if` pozwala na wybór jednego lub kilku działań (bloków instrukcji) i uruchamia grupę instrukcji powiązaną z pierwszym warun-

kiem `if` lub `elif`, który jest prawdziwy, albo wykonuje grupę `else`, jeśli wszystkie warunki `if` (`elif`) mają wartość `false`. Człony `elif` i `else` są opcjonalne.

Instrukcja `while`

```
while warunek:
    grupa
[else:
    grupa]
```

Pętla `while` to instrukcja pętli ogólnego przeznaczenia, która wykonuje pierwszą grupę instrukcji, gdy warunek na początku instrukcji jest prawdziwy. Instrukcja uruchamia grupę `else`, w przypadku gdy nastąpi zakończenie działania pętli bez wykonania instrukcji `break`.

Instrukcja `for`

```
for cel in obiekt_iterowalny:
    grupa
[else:
    grupa]
```

Pętla `for` realizuje iterację po sekwencji (bądź innym obiekcie iterowalnym). Przypisuje elementy obiektu iterowalnego do zmiennej `cel` i w każdej iteracji wykonuje pierwszą grupę instrukcji. Instrukcja `for` uruchamia opcjonalną grupę `else`, w przypadku gdy nastąpi zakończenie działania pętli bez wykonania instrukcji `break`. Zmienną `cel` instrukcji `for` może być dowolny obiekt, który może się znaleźć po lewej stronie instrukcji przypisania (np. `for (x, y) in lista_krotek`).

Od Pythona w wersji 2.2 instrukcja `for` najpierw próbuje uzyskać obiekt *iteratora* `I` za pomocą metody `iter(obiekt_iterowalny)`, a następnie wielokrotnie wywołuje metodę `I.__next__()` obiektu — do czasu wystąpienia wyjątku `StopIteration` (w Pythonie 2.X metoda `I.__next__()` ma nazwę `I.next()`). Jeśli nie można uzyskać żadnego obiektu iteratora (np. nie zdefiniowano metody `__iter__`), to instrukcja `for` działa poprzez wielokrotne indeksowanie obiektu `obiekt_iterowalny` z użyciem coraz większych wartości przesunięcia, aż zostanie zgłoszony wyjątek `IndexError`.

Iteracja w Pythonie odbywa się w wielu kontekstach, włącznie z instrukcjami pętli `for`, obiektami składanymi i instrukcjami `map()`. Więcej informacji na temat mechanizmów używanych przez pętlę `for` oraz w pozostałych kontekstach iteracji można znaleźć w podrozdziale „Protokół iteracji”.

Instrukcja pass

`pass`

Instrukcja-wypełniacz, która nie wykonuje żadnych działań. Używa się jej wtedy, gdy jest to syntaktycznie konieczne (np. w odniesieniu do namiastek funkcji). W Pythonie 3.X podobny efekt można uzyskać za pomocą wielokropka (...).

Instrukcja break

`break`

Powoduje natychmiastowe zakończenie najbliższej instrukcji pętli `while` lub `for` z pominięciem powiązanych z nimi grup `else` (o ile takie istnieją). Wskazówka: w celu wyjścia z wielopoziomowych pętli można skorzystać z instrukcji `raise` i `try`.

Instrukcja continue

`continue`

Powoduje natychmiastowe przejście na początek najbliższej instrukcji pętli `while` lub `for` i wznowienie wykonywania grupy instrukcji w pętli.

Instrukcja del

```
del nazwa
del nazwa[i]
del nazwa[i:j:k]
del nazwa.atrybut
```

Instrukcja `del` usuwa nazwy, elementy, wycinki i atrybuty, a także powiązania. W pierwszej postaci *nazwa* jest dosłowną nazwą zmiennej. W ostatnich trzech formach instrukcji *nazwa* może być dowolnym wyrażeniem (z użyciem nawiasów, jeśli są potrzebne do określenia priorytetu). Na przykład: `del a.b()[1].c.d`.

Instrukcja ta służy przede wszystkim do struktur danych, a nie zarządzania pamięcią. Usuwa również referencje do obiektów wywoływanych wcześniej. Może spowodować ich usunięcie przez mechanizm odśmiecania, w przypadku gdy nie będzie do nich odwołania w innych miejscach. Proces odśmiecania zachodzi jednak automatycznie i nie musi być wymuszany za pomocą instrukcji `del`.

Instrukcja def

```
[dekorator]
def nazwa([arg, ... arg=wartość, ... *arg, **arg]):
    grupa
```

Instrukcja `def` służy do tworzenia nowych funkcji, które mogą również pełnić rolę metod w klasach. Jej działanie polega na utworzeniu obiektu funkcji i przypisaniu do niego zmiennej *nazwa*. Każde wywołanie do obiektu funkcji generuje nowy, lokalny zasięg, w którym przypisywane nazwy są domyślnie lokalne dla wywołania (o ile nie zostaną zadeklarowane jako `global` lub — w Pythonie 3.X — `nonlocal`). Więcej informacji na temat zasięgów można znaleźć w podrozdziale „Przestrzenie nazw i reguły zasięgu”.

Argumenty przekazuje się przez przypisanie. W nagłówku instrukcji `def` można je definiować, wykorzystując dowolny z czterech formatów zestawionych w tabeli 14. Argumenty w postaci pokazanej w tabeli 14. mogą być również wykorzystywane w wywołaniach funkcji. Są tam interpretowane w sposób pokazany w tabeli 15. (więcej informacji na temat składni wywołań funkcji można znaleźć w podrozdziale „Instrukcja wyrażeniowa”).

Tabela 14. Formaty argumentów w definicjach

Format argumentu	Interpretacja
<i>nazwa</i>	Dopasowywany według nazwy lub pozycji
<i>nazwa</i> =wartość	Wartość domyślna, jeśli argument <i>nazwa</i> nie zostanie przekazany
* <i>nazwa</i>	Pobiera dodatkowe argumenty pozycyjne jako nową krotkę <i>nazwa</i>
** <i>nazwa</i>	Pobiera dodatkowe argumenty kluczowe jako nowy słownik <i>nazwa</i>
inne, <i>nazwa</i> [=wartość]	Za gwiazdką () mogą wystąpić wyłącznie argumenty kluczowe (Python 3.X)
*, <i>nazwa</i> [=wartość]	Taki sam efekt jak w poprzednim wierszu

Tabela 15. Formaty argumentów w wywołaniach

Format argumentu	Interpretacja
<i>wartość</i>	Argument pozycyjny
<i>nazwa</i> =wartość	Argument kluczowy (dopasowywany według nazwy)
* <i>obiekt_iterowalny</i>	Rozpakowanie sekwencji bądź innego obiektu iterowalnego złożonego z argumentów pozycyjnych
** <i>słownik</i>	Rozpakowanie słownika złożonego z argumentów kluczowych

Argumenty kluczowe Pythona 3.X

W Pythonie 3.X (wyłącznie) uogólniono definicję funkcji. Pozwalają one na przekazywanie jedynie kluczowych argumentów. Muszą one być przekazywane przez klucz i są obowiązkowe, jeśli nie zostaną zakodowane z określeniem wartości domyślnej. Argumenty wyłącznie kluczowe koduje się za pomocą *. Symbol gwiazdki może występować bez nazwy, jeśli istnieją argumenty wyłącznie kluczowe, których pozycja jest ustalona:

```
>>> def f(a, *b, c): print(a, b, c) # c – obowiązkowy argument kluczowy
...
>>> f(1, 2, c=3)
1 (2,) 3

>>> def f(a, *, c=None): print(a, b, c) # c – opcjonalny argument kluczowy
...
>>> f(1)
1 None
>>> f(1, c='spam')
1 spam
```

Adnotacje funkcji w Pythonie 3.X

W Pythonie 3.X (wyłącznie) uogólniono definicję funkcji również przez wprowadzenie możliwości opisywania argumentów i zwracanych wartości za pomocą wartości obiektów, co może zostać wykorzystane w rozszerzeniach. Adnotacje koduje się jako :wartość za nazwą argumentu, a przed wartością domyślną, a także jako ->wartość za listą argumentów. Adnotacje są zbierane do atrybutu __annotations__, ale poza tym Python nie traktuje ich w specjalny sposób:

```
>>> def f(a:99, b:'spam'=None) -> float:
...     print(a, b)
...
>>> f(88)
88 None
>>> f.__annotations__
{'a': 99, 'b': 'spam', 'return': <class 'float'>}
```

Wyrażenia lambda

Funkcje można również tworzyć z wykorzystaniem formy wyrażenia lambda. Wyrażenie lambda nie przypisuje obiektu funkcji do nazwy, ale tworzy nowy obiekt funkcji i zwraca go, dzięki czemu można go później wywołać:

```
lambda arg, arg, ...: wyrażenie
```

W wyrażeniu `lambda` argumenty *arg* mają takie samo znaczenie jak w instrukcji `def` (tabela 14.), a *wyrażenie* jest domniemaną zwracaną wartością późniejszych wywołań. Kod wewnątrz wyrażenia *wyrażenie* jest „odroczony” do chwili wywołania:

```
>>> L = lambda a, b=2, *c, **d: [a, b, c, d]
>>> L(1, 2, 3, 4, x=1, y=2)
[1, 2, (3, 4), {'y': 2, 'x': 1}]
```

Ponieważ `lambda` jest wyrażeniem, a nie instrukcją, można je wykorzystać w miejscach, gdzie nie da się skorzystać z instrukcji `def` (np. wewnątrz wyrażenia literału słownikowego, listy argumentów lub wywołania funkcji). Wyrażenie `lambda` nie uruchamia instrukcji, tylko oblicza pojedyncze wyrażenie, więc nie jest przeznaczone do wykorzystania w funkcjach złożonych (w takich przypadkach należy skorzystać z instrukcji `def`).

Wartości domyślne i atrybuty

Mutowalne wartości argumentów domyślnych są obliczane raz — w momencie uruchamiania instrukcji `def` — a nie przy każdym wywołaniu. Z tego względu mogą zachowywać stan pomiędzy wywołaniami. Niektórzy uważają jednak takie działanie za ograniczenie; lepszymi narzędziami zachowywania stanów są często klasy oraz zasięgi. Aby uniknąć modyfikowania argumentów, w przypadku argumentów mutowalnych oraz w jawnych testach należy używać wartości domyślnej `None`. Zaprezentowano to w poniższych komentarzach:

```
>>> def grow(a, b=[]): # def grow(a, b=None):
...     b.append(a)    # if b == None: b = []
...     print(b)      # ...
...
>>> grow(1); grow(2)
[1]
[1, 2]
```

Zarówno w Pythonie 2.X, jak i 3.X dozwolone jest dołączanie dowolnych *atrybutów* do funkcji. Jest to kolejna forma zachowywania stanów (ale atrybuty zachowują stan tylko na poziomie obiektu funkcji; w przypadku gdy każde wywołanie generuje nową funkcję, oznacza to zachowywanie stanu tylko dla wywołania):

```
>>> grow.food = 'spam'
>>> grow.food
'spam'
```

Dekoratory funkcji i metod

Począwszy od Pythona w wersji 2.4, definicje funkcji mogą być poprzedzane deklaracjami opisującymi funkcję. Są to tzw. *dekoratory*, które koduje się za pomocą znaku @. Deklaracje te dostarczają jawną składnię dla technik funkcyjnych. Zastosowanie składni dekoratora funkcji:

```
@decorator
def F():
    ...
```

jest równoważne następującemu ręcznemu wiązaniu nazwy:

```
def F():
    ...
F = decorator(F)
```

Efekt tej instrukcji to powiązanie nazwy funkcji z wynikiem przekazania funkcji przez obiekt wywoływalny *decorator*. Dekoratory funkcji można wykorzystywać do zarządzania funkcjami lub kierowania do nich wywołań (poprzez wykorzystanie obiektów proxy). Dekoratory można stosować w odniesieniu do dowolnych definicji funkcji, w tym także do metod wewnątrz klas:

```
class C:
    @decorator
    def M():          # to samo co M = decorator(M)
    ...
```

Ogólnie zastosowanie poniższych zagnieżdżonych dekoratorów:

```
@A
@B
@C
def f(): ...
```

jest równoważne poniższemu kodowi bez dekoratorów:

```
def f(): ...
f = A(B(C(f)))
```

Do dekoratorów można również przekazywać listy argumentów:

```
@spam(1, 2, 3)
def f(): ...
```

W tym przypadku spam musi być funkcją, która zwraca inną funkcję (tzw. *funkcja-fabryka*). Jej wynik jest wykorzystywany jako właściwy dekorator. W miarę potrzeb może on zachowywać stan argumentów. Dekoratory muszą występować w wierszu poprzedzającym definicję funkcji. Nie mogą się znaleźć w tym samym wierszu (np. składnia @A def f(): ... jest nieprawidłowa).

Ponieważ dekoratory w roli argumentów wykorzystują obiekty wywołalne, a także je zwracają, jako dekoratory funkcji można wykorzystać niektóre funkcje wbudowane, na przykład `property()`, `staticmethod()` oraz `classmethod()` (więcej informacji na ten temat można znaleźć w podrozdziale „Funkcje wbudowane”). Składnię dekoratorów można także stosować w odniesieniu do *klas* w Pythonie 2.6, 3.0 i późniejszych wersjach; patrz podrozdział „Instrukcja `class`”.

Instrukcja `return`

```
return [wyrażenie]
```

Instrukcja `return` powoduje zakończenie funkcji, w której ją wywołano, oraz zwraca *wyrażenie* jako wynik wywołania tej funkcji. Jeśli pominięto argument *wyrażenie*, domyślnie zwracana jest wartość `None`. Jest to również domyślna zwracana wartość dla funkcji, które kończą działanie bez uruchamiania instrukcji `return`. Wskazówka: aby zwrócić wynik funkcji złożony z wielu wartości, można wykorzystać krotkę. Informacje dotyczące specjalnej semantyki instrukcji `return` w przypadku jej użycia w funkcji generatora można znaleźć w podrozdziale „Instrukcja `yield`”.

Instrukcja `yield`

```
yield wyrażenie          # Wszystkie wersje Pythona
yield from obiekt_iterowalny # Python 3.3 i wersje późniejsze
```

Wyrażenie `yield` w Pythonie 2.X i 3.X definiuje *funkcję generatora*, która tworzy wyniki na żądanie. Funkcje zawierające instrukcję `yield` są kompilowane w specjalny sposób. Kiedy zostaną wywołane, tworzą i zwracają *obiekt generatora* — iterowalny obiekt, który automatycznie obsługuje *protokół iteracji* w celu zwracania wyników w kontekstach iteracji.

Wyrażenie `yield` zwykle koduje się w postaci instrukcji wyrażeniowej (w osobnym wierszu). Zapisuje ono stan funkcji i zwraca *wyrażenie*. W następnej iteracji następuje odtworzenie wcześniejszego stanu funkcji, a sterowanie jest przekazywane do instrukcji występującej bezpośrednio za instrukcją `yield`.

Aby zakończyć iterację, należy użyć instrukcji `return` bez przekazywania wartości lub przeskoczyć do instrukcji kończącej funkcję. Funkcja generatora w wersjach Pythona wcześniejszych niż 3.3 nie może zwracać wartości, natomiast wartości mogą być zwracane w Pythonie

w wersji 3.3 lub wersjach późniejszych. Wartość ta jest przechowywana jako atrybut obiektu wyjątku (zobacz podrozdział „Zmiany w funkcji generatora w Pythonie 3.3”):

```
def generateSquares(N):
    for i in range(N):
        yield i ** 2

>>> G = generateSquares(5) # zawiera wywołania __init__, __next__
>>> list(G)                 # wymuszenie generowania wyniku
[0, 1, 4, 9, 16]
```

W przypadku użycia instrukcji `yield` jako wyrażenia zwraca ona obiekt przekazany do metody `send()` generatora wywołanej na rzecz obiektu wywołującego (np. `A = yield X`). Wyrażenie `yield` musi być ujęte w nawiasy, o ile nie jest jedynym elementem występującym po prawej stronie (np. `A = (yield X) + 42`). W tym trybie wartości są przesyłane do generatora poprzez wywołanie metody `send(wartość)`. Następuje wznowienie działania generatora i wyrażenie `yield` zwraca *wartość*. Jeśli wcześniej została wywołana metoda `__next__()` lub wbudowana funkcja `next()`, wyrażenie `yield` zwraca `None`.

Generatory mają również metodę `throw(typ)`, która zgłasza wyjątek wewnątrz generatora w momencie wywołania ostatniej instrukcji `yield`, oraz metodę `close()`, zgłaszającą wyjątek `GeneratorExit` wewnątrz generatora w celu zakończenia iteracji. Instrukcja `yield` występuje standardowo w Pythonie od wersji 2.3. Metody generatora `send()`, `throw()` i `close()` są dostępne w Pythonie od wersji 2.5.

Metoda klasy `__iter__()` zawierająca instrukcję `yield` zwraca generator z automatycznie utworzoną metodą `__next__()`. Więcej informacji na temat mechanizmów używanych przez funkcje generatora można znaleźć w podrozdziale „Protokół iteracji”. Warto również zajrzeć do podrozdziału „Wyrażenia generatorowe” w celu zapoznania się z pokrewnym narzędziem, które także tworzy obiekt generatora.

Zmiany w funkcji generatora w Pythonie 3.3

Począwszy od wersji 3.3, Python 3.X (wyłącznie) obsługuje w tej instrukcji klauzulę `from`. W podstawowym zastosowaniu klauzula ta jest podobna do pętli `for`, która iteruje po elementach wewnątrz *obektu_iterowalnego*. W bardziej zaawansowanych rolach rozszerzenie to pozwala podgeneratorom otrzymywać wysyłane i zwracane wartości bezpośrednio z wyższego zasięgu wywołującego:

```
for i in range(N): yield i # Wszystkie wersje Pythona
yield from range(N)       # Opcja w wersji 3.3 i wersjach późniejszych
```

Także od wersji 3.3 w przypadku jeśli funkcja generatora zakończy iterację i działanie za pomocą wywołanej jawnie instrukcji `return`, to wartości przekazane w instrukcji `return` są dostępne za pośrednictwem atrybutu `value` niejawnie utworzonego i zgłoszonego egzemplarza obiektu wyjątku `StopIteration`. Wartość ta jest ignorowana przez automatyczne iteracje, ale może być odpytywana w ręcznych iteracjach lub innym kodzie, który ma dostęp do wyjątku (zobacz podrozdział „Wbudowane wyjątki”). W Pythonie 2.X oraz w wersjach 3.X wcześniejszych niż 3.3 instrukcja `return` z przekazaną wartością w funkcji generatora jest traktowana jak błąd składniowy.

Instrukcja `global`

`global nazwa [, nazwa]*`

Instrukcja `global` jest deklaracją przestrzeni nazw — jeśli użyje się jej wewnątrz klasy lub instrukcji definicji funkcji, to powoduje ona, że wszystkie wystąpienia ciągu *`nazwa`* w wybranym kontekście będą traktowane jak referencje do zmiennej globalnej (poziomu modułu) zmiennej *`nazwa`* — niezależnie od tego, czy do zmiennej *`nazwa`* zostanie przypisana wartość i czy zmienna *`nazwa`* jest już zdefiniowana.

Instrukcja ta umożliwia definiowanie bądź modyfikowanie zmiennych globalnych wewnątrz funkcji bądź klas. Ze względu na reguły zasięgów obowiązujące w Pythonie istnieje obowiązek deklarowania tylko globalnych nazw, które zostały *przypisane*. W momencie przypisania niezadeklarowane nazwy są traktowane jak lokalne, natomiast referencje globalne są automatycznie wyszukiwane w bieżącym module. Więcej informacji można znaleźć w podrozdziale „Przestrzenie nazw i reguły zasięgu”.

Instrukcja `nonlocal`

`nonlocal nazwa [, nazwa]*`

Dostępna wyłącznie w Pythonie 3.X.

Instrukcja `nonlocal` jest deklaracją przestrzeni nazw — jeśli użyje się jej wewnątrz zagnieżdżonej funkcji, to wszystkie wystąpienia ciągu *`nazwa`* w wybranym kontekście będą traktowane jak referencje do zmiennej lokalnej o tej nazwie w bieżącym zasięgu funkcji okalającej — niezależnie od tego, czy zmiennej *`nazwa`* została przypisana wartość, czy nie.

Zmienna *`nazwa`* musi być zdefiniowana w funkcji okalającej. Instrukcja `nonlocal` pozwala na modyfikowanie jej przez funkcję zagnieżdżoną.

Ze względu na obowiązujące w Pythonie reguły zasięgów istnieje obowiązek deklarowania tylko tych nielokalnych nazw, do których są przypisywane wartości. W momencie przypisania niezadeklarowane nazwy są traktowane jak lokalne, natomiast referencje nielocalne są automatycznie wyszukiwane w okalającej funkcji. Więcej informacji można znaleźć w podrozdziale „Przestrzenie nazw i reguły zasięgu”.

Instrukcja import

```
import [pakiet.]* moduł [as nazwa]
[, [pakiet.]* moduł [as nazwa]]*
```

Instrukcja `import` daje dostęp do modułów — importuje moduł jako całość. Z kolei moduły zawierają nazwy pobierane według kwalifikacji (np. `moduł.atrybut`). Przypisania na najwyższym poziomie pliku Pythona tworzą atrybuty obiektu modułu. Opcjonalna klauzula `as` powoduje przypisanie zmiennej *nazwa* do importowanego obiektu modułu i usuwa oryginalną nazwę modułu (jest to przydatne ze względu na to, że dostarcza krótszych synonimów dla długich nazw modułów). Opcjonalne prefiksy *pakiet* oznaczają ścieżki do katalogów zawierających pakiety (opisano je w następnym podrozdziale).

Argument *moduł* oznacza nazwę modułu docelowego — zazwyczaj jest to plik z kodem źródłowym w Pythonie lub moduł skompilowany. Argument *moduł* podaje się jako nazwę pliku bez rozszerzenia (*.py* lub inne rozszerzenia są pomijane). Plik ten musi się znajdować w katalogu umieszczonym w ścieżce wyszukiwania modułów, o ile nie zostanie zagnieżdżony w ścieżce *pakiet*.

Dla komponentów *moduł* lub *pakiet* występujących w pierwszej kolejności na bezwzględnej ścieżce importu *ścieżka wyszukiwania modułów* jest zgodna ze zmienną `sys.path` — listą katalogów inicjowaną na podstawie katalogu głównego programu, ustawienia `PYTHONPATH`, zawartości pliku *.pth* oraz wartości domyślnych obowiązujących w środowisku Pythona. W przypadku zagnieżdżonych komponentów pakietu moduły mogą być umieszczone w jednym katalogu pakietu (patrz „Importowanie pakietów”), natomiast importy względne definiuje się w instrukcjach `from` (patrz „Importowanie względem katalogu pakietów”). Od wersji Pythona 3.3 ścieżki wyszukiwania pakietów przestrzeni nazw mogą obejmować dowolne katalogi (patrz „Pakiety przestrzeni nazw Pythona 3.3”).

Jeśli jest taka potrzeba, to operacje importu powodują kompilację kodu źródłowego do postaci kodu bajtowego (i jeśli to możliwe, zapisują go

w pliku *.pyc*). Następnie uruchamiają skompilowany kod od góry do dołu w celu wygenerowania atrybutów obiektu modułu przez przypisanie. W Pythonie 2.X i 3.1 oraz wersjach wcześniejszych pliki z kodem bajtowym są zapisywane w katalogu zawierającym pliki z kodem źródłowym. Mają one tę samą nazwę bazową (np. *moduł.pyc*). W Pythonie 3.2 oraz wersjach późniejszych pliki z kodem bajtowym są zapisywane w podkatalogu *__pycache__* katalogu z kodem źródłowym. Mają one nazwę bazową zawierającą informacje umożliwiające zidentyfikowanie wersji (np. *moduł.cpython-33.pyc*).

W kolejnych operacjach importu wykorzystywane są moduły zaimportowane wcześniej, ale funkcja `imp.reload()` (`reload()` w wersji 2.X) wymusza ponowny import załadowanych modułów. Aby przeprowadzić import na podstawie nazwy podanej w formie łańcucha znaków, warto zapoznać się ze sposobem wykorzystania metody `__import__` używanej przez instrukcję `import`. Informacje na ten temat można znaleźć w podrozdziale „Funkcje wbudowane” oraz w opisie funkcji `importlib.import_module(nazwa_modułu)` ze standardowej biblioteki.

W standardowej implementacji CPython w operacjach importowania można także łądować skompilowane rozszerzenia języków C i C++ razem z atrybutami odpowiadającymi nazwom zewnętrznych języków. W innych implementacjach w operacjach importu mogą również występować nazwy bibliotek klas innych języków (np. implementacja Jython może generować wrapper modułu Pythona komunikujący się z biblioteką Javy).

Importowanie pakietów

Jeśli w instrukcji `import` zostaną użyte nazwy prefiksów *pakiet*, to oznaczają one nazwy okalających katalogów, natomiast ścieżki modułów rozdzielone kropkami odzwierciedlają hierarchię katalogów. Instrukcja importu w postaci `import dir1.dir2.mod` ładuje plik modułu z katalogu *dir1/dir2/mod.py*, przy czym katalog *dir1* musi istnieć w katalogu zdefiniowanym w ścieżce wyszukiwania modułów (`sys.path` dla importów bezwzględnych), natomiast katalog *dir2* musi być umieszczony wewnątrz katalogu *dir1* (poza ścieżką `sys.path`).

W przypadku standardowych pakietów w każdym katalogu wymienionym w instrukcji `import` musi istnieć (choć może być pusty) plik *__init__.py*, który odgrywa rolę przestrzeni nazw modułu na poziomie katalogu. Ten plik jest uruchamiany przy okazji pierwszej operacji importu z katalogu, a wszystkie nazwy przypisane w plikach *__init__.py*

stają się atrybutami obiektu modułu katalogu. Pakiety katalogów pozwalają na rozwiązywanie konfliktów spowodowanych liniową naturą ścieżki `PYTHONPATH`.

Warto się zapoznać z podrozdziałem „Importowanie względem katalogu pakietów”, gdzie zamieszczono więcej informacji o odwołaniach do pakietów w instrukcjach `from`, oraz z podrozdziałem „Pakiety przestrzeni nazw Pythona 3.3”, w którym opisano alternatywny typ pakietów, niewymagający stosowania pliku `__init__.py`.

Pakiety przestrzeni nazw Pythona 3.3

Od Pythona 3.3 operację importowania rozszerzono o możliwość rozpoznawania *pakietów przestrzeni nazw* — pakietów modułów, które są wirtualną konkatenacją jednego lub większej liczby katalogów zagnieżdżonych w elementach ścieżki wyszukiwania modułów.

Pakiety przestrzeni nazw nie zawierają (i nie mogą zawierać) pliku `__init__.py`. Służą one jako opcja awaryjna i rozszerzenie standardowych modułów oraz pakietów. Są rozpoznawane tylko wtedy, gdy nazwa nie zostanie zlokalizowana w inny sposób, ale pasuje do jednego bądź większej liczby katalogów znalezionych podczas skanowania ścieżki wyszukiwania. Własność tę uaktywnia zarówno instrukcja `import`, jak i `from`.

Algorytm importu

Po wprowadzeniu pakietów przestrzeni nazw początkowe etapy importowania przebiegają tak jak wcześniej (tzn. sprawdzane są już zaimportowane moduły i pliki z kodem bajtowym), ale sposób wyszukiwania modułu rozszerzono w sposób opisany poniżej.

Podczas importu Python iteruje po katalogach umieszczonych w *ścieżce wyszukiwania modułów* — zdefiniowanej za pomocą zmiennej `sys.path` w przypadku skrajnych komponentów importowania bezwzględnego oraz za pomocą lokalizacji pakietu w przypadku importowania względnego oraz komponentów zagnieżdżonych w ścieżkach pakietów. Począwszy od wersji 3.3, podczas wyszukiwania importowanego modułu lub pakietu o nazwie *spam* dla każdego katalogu w ścieżce wyszukiwania modułów Python sprawdza kryteria dopasowywania w następującej kolejności:

1. Jeśli zostanie znaleziony plik `katalog\spam__init__.py`, zaimportowany i zwrócony będzie standardowy pakiet.

2. Jeśli zostanie znaleziony plik `katalog\spam.{py, pyc lub inne rozszerzenie modułu}`, zaimportowany i zwrócony będzie prosty moduł.
3. Jeśli zostanie znaleziony katalog `katalog\spam`, zostanie on zapisany, a skanowanie będzie kontynuowane od następnego katalogu w ścieżce wyszukiwania.
4. Jeśli żaden z powyższych obiektów nie zostanie znaleziony, to operacja skanowania przechodzi do następnego katalogu w ścieżce wyszukiwania.

Jeśli skanowanie ścieżki wyszukiwania zakończy się bez zwrócenia modułu bądź pakietu, zgodnie z krokami 1. i 2., oraz jeśli w kroku 3. będzie zarejestrowany co najmniej jeden *katalog*, to natychmiast zostanie utworzony *pakiet przestrzeni nazw*. Nowy pakiet przestrzeni nazw ma atrybut `__path__` ustawiony na wartość obiektu iterowalnego zawierającego znalezione i zarejestrowane w kroku 3. ciągi ścieżki katalogów, ale nie ma ustawionego atrybutu `__file__`.

Atrybut `__path__` jest używany później do przeszukiwania wszystkich komponentów pakietu w przypadku żądania zagnieżdżonych elementów, podobnie jak dla jedynego katalogu standardowego pakietu. Atrybut ten spełnia dla komponentów niższego poziomu tę samą rolę, jaką zmienna `sys.path` spełnia dla komponentów najwyższego poziomu w odniesieniu do skrajnych lewych komponentów bezwzględnych ścieżek importu. Pełni on rolę ścieżki rodzica, która umożliwia dostęp do pozycji niższego poziomu przy użyciu tego samego czteroetapowego algorytmu.

Instrukcja from

```
from [pakiet.]* moduł import  
    [(nazwa [as innanazwa]  
    [, nazwa [as innanazwa])* []]
```

```
from [pakiet.]* moduł import *
```

Instrukcja `from` importuje moduł podobnie jak instrukcja `import` (zobacz poprzedni podrozdział), ale jednocześnie kopiuje nazwy zmiennych z modułu. Dzięki temu można później używać tych nazw bez potrzeby kwalifikowania ich za pomocą odpowiedniego atrybutu. Instrukcja w drugim formacie (`from ... import *`) kopiuje *wszystkie* nazwy przypisane na najwyższym poziomie modułu z wyjątkiem nazw zaczynających się pojedynczym znakiem podkreślenia lub nazw, których nie wymieniono w atrybucie opisującym listę łańcuchów znaków modułu `__all__` (jeśli go zdefiniowano).

Klauzula `as` służy do tworzenia synonimów (podobnie jak w instrukcji `import`) i działa dla dowolnego komponentu *nazwa*. Ścieżki importowania *pakiet* działają tak samo jak w instrukcji `import` (np. `from dir1.dir2.mod import x`) zarówno dla pakietów standardowych, jak i pakietów przestrzeni nazw z Pythona 3.3, ale ścieżka do pakietu musi być wymieniona tylko raz, w samej instrukcji `from` (nie trzeba jej wymieniać w każdym odwołaniu do atrybutu). Od Pythona 2.4 nazwy importowane z modułu można ująć w nawiasy. Dzięki temu instrukcja importu może obejmować kilka wierszy i nie wymaga użycia lewych ukośników (jest to specjalna składnia obowiązująca tylko dla instrukcji `from`).

W Pythonie 3.X instrukcja w postaci `from ... import *` jest niedozwolona wewnątrz funkcji lub klasy, ponieważ uniemożliwia sklasyfikowanie zasięgów nazw w momencie definicji. Ze względu na reguły dotyczące zasięgów użycie formatu `*` generuje również ostrzeżenia w Pythonie 2.X, począwszy od wersji 2.2, jeśli symbol ten zostanie zagnieźdżony wewnątrz funkcji lub klasy.

Instrukcja `from` jest także wykorzystywana w celu umożliwienia wprowadzania do języka eksperymentalnych dodatków. Robi się to za pomocą instrukcji `from __future__ import nazwa_własności`. Instrukcja w takim formacie może się znaleźć wyłącznie na początku pliku modułu (poprzedzona tylko przez opcjonalny ciąg dokumentacyjny lub komentarz). Można jej również użyć w dowolnym momencie podczas sesji interaktywnej.

Importowanie względem katalogu pakietów

W Pythonie 3.X i 2.X w instrukcji `from` (ale *nie* `import`) można wykorzystywać wiodące kropki w nazwach modułów w celu określenia wewnątrzpakietowych odwołań do modułów — importów realizowanych *względem* katalogu pakietów, w którym rezyduje moduł importujący. We względnych operacjach importów wstępna ścieżka wyszukiwania modułów jest ograniczona do katalogu z pakietem. Pozostałe importy są *bezwzględne*. Moduły są wyszukiwane zgodnie z ustawieniami zmiennej `sys.path`. Oto ogólne wzorce składni:

```
from źródło import nazwa [, nazwa]*      # sys.path: bezwzględna

from . import moduł [, moduł]*            # tylko katalog pakietu: względna
from .źródło import nazwa [, nazwa]*      # tylko katalog pakietu: względna

from .. import moduł [, moduł]*           # katalog nadrzędny pakietu
from ..źródło import nazwa [, nazwa]*     # katalog nadrzędny pakietu
```

W tej formie instrukcji *from źródło* może być prostym identyfikatorem lub rozdzielaną kropkami ścieżką pakietów. Wiodące kropki identyfikują operację importowania jako operację względem pakietów. Rozszerzenie zmiany nazwy *as* (tutaj go nie pokazano) także działa w tej formie instrukcji podobnie jak w standardowej instrukcji *from* zarówno dla argumentu *nazwa*, jak i *moduł*.

Składnia z wiodącymi kropkami pozwala na jawne wskazanie, że import jest realizowany względem pakietu. Jest ona dostępna zarówno w Pythonie 3.X, jak i 2.X. W instrukcjach importu *bez kropek* w Pythonie 2.X (ale nie w Pythonie 3.X) najpierw jest przeszukiwany katalog, w którym znajduje się pakiet. Aby włączyć stosowanie semantyki importu pakietów Pythona 3.X w Pythonie 2.6 lub wersji późniejszej, należy skorzystać z instrukcji:

```
from __future__ import absolute_import
```

Ponieważ bezwzględne importowanie pakietów w odniesieniu do ścieżki *sys.path* może pokrywać więcej przypadków użycia, jest to preferowany sposób. Jest polecany zarówno zamiast niejawnych importów względem ścieżek pakietów w Pythonie 2.X, jak i jawnych importów względem ścieżki pakietów w Pythonie 2.X i 3.X.

Instrukcja `class`

```
[dekorator]  
class nazwa [ (klasanadrzędna [, klasanadrzędna]* [, metaclass=M] ) ]:  
    grupa
```

Instrukcja `class` tworzy nowe obiekty klasy będące fabrykami dla egzemplarzy (instancji) obiektów. Nowy obiekt klasy dziedziczy z każdej z wymienionych klas nadrzędnych w podanym porządku i jest przypisywany do zmiennej *nazwa*. Instrukcja `class` wprowadza nowy lokalny zasięg nazw. Wszystkie nazwy przypisane wewnątrz instrukcji `class` generują atrybuty obiektu klasy współdzielone przez wszystkie egzemplarze tej klasy.

Poniżej zamieszczono listę istotnych własności klas (więcej informacji na temat technik programowania obiektowego można także znaleźć w podrozdziałach „Programowanie obiektowe” oraz „Metody przeciążające operatory”).

- *Klasy nadrzędne* (zwane także klasami bazowymi), z których nowa klasa dziedziczy atrybuty, są wymienione w nawiasach wewnątrz nagłówka (np. `class Podrzedna(Nadrzedna1, Nadrzedna2)`).

- Instrukcje przypisania wewnątrz grupy generują *atrybuty klas* dziedziczone przez egzemplarze — zagnieżdżone instrukcje `def` tworzą *metody*, natomiast instrukcje przypisania tworzą proste składowe klasy itp.
- Wywołanie klasy generuje *obiekty egzemplarzy*. Każdy obiekt egzemplarza może mieć własne atrybuty, a przy tym dziedziczy atrybuty klasy oraz jej wszystkich klas nadrzędnych.
- *Funkcje będące metodami* otrzymują specjalny pierwszy argument, zwykle o nazwie `self`, który oznacza obiekt egzemplarza będący domniemanym podmiotem wywoływanych metod. Argument ten daje dostęp do stanu egzemplarza oraz jego atrybutów.
- Funkcje wbudowane `staticmethod()` i `classmethod()` dostarczają dodatkowy rodzaj metod. Metody Pythona 3.X wywoływane za pośrednictwem klasy mogą być traktowane jak zwykłe funkcje.
- Definicje metod *przeciążających operatory* o specjalnych nazwach w postaci `__x__` przechwytyują działania wbudowane.
- W uzasadnionych przypadkach klasy zapewniają utrzymanie stanu i struktury programu oraz wspierają *wielokrotne wykorzystanie kodu* poprzez dostosowanie do specyficznych wymagań w nowych klasach.

Dekoratory klas z Pythona 3.X, 2.6 i 2.7

W Pythonie 2.6, 3.0 i w wersjach późniejszych obu linii Pythona dekoratory można stosować nie tylko w odniesieniu do definicji funkcji, ale także instrukcji `class`.

Składnia dekoratora klasy:

```
@dekorator
class C:
    def meth():
    ...
```

jest równoważna następującemu ręcznemu wiązaniu nazwy:

```
class C:
    def meth():
    ...
C = dekorator(C)
```

Efekt tej instrukcji to powiązanie nazwy klasy z wynikiem przekazania klasy przez obiekt wywoływalny *dekorator*. Dekoratory klas, podobnie jak dekoratory funkcji, można zagnieżdżać. Można również stosować

argumenty. Dekoratory klas można wykorzystywać do zarządzania klasami lub późniejszymi wywołaniami tworzenia egzemplarzy (poprzez wykorzystanie obiektów proxy).

Metaklasy

Metaklasy to klasy, które ogólnie rzecz biorąc, są podklasami ułatwiającymi tworzenie samych obiektów klas. Na przykład:

```
class Meta(type):
    def __new__(meta, cname, supers, cdict):
        # poniższa instrukcja oraz __init__ są uruchamiane przez metodę type.__call__
        c = type.__new__(meta, cname, supers, cdict)
        return c;
```

W Pythonie 3.X klasy definiują swoje metaklasy za pomocą argumentów kluczowych w nagłówku instrukcji class:

```
class C(metaclass=Meta): ...
```

W Pythonie 2.X do tego celu wykorzystuje się atrybuty klas:

```
class C(object):
    __metaclass__ = Meta
    ...
```

Kod metaklasy jest uruchamiany po zakończeniu działania instrukcji class (podobnie jak w przypadku dekoratorów klas). Warto się również zapoznać z opisem funkcji type() w podrozdziale „Funkcje wbudowane”. Można tam znaleźć informacje na temat odwzorowań instrukcji class na metody metaklasy.

Instrukcja try

```
try:
    grupa
except [typ [as wartość]]:      # lub [, wartość] w Pythonie 2.X
    grupa
[except [typ [as wartość]]:
    grupa]*
[else:
    grupa]
[finally:
    grupa]

try:
    grupa
finally:
    grupa
```

Instrukcja `try` służy do przechwytywania wyjątków. Instrukcje `try` pozwalają na definiowanie klauzul `except` zawierających grupy spełniające rolę procedur obsługi wyjątków zgłaszanych podczas wykonywania grupy `try`. W instrukcji można również zdefiniować klauzule `else`, które są wykonywane, w przypadku gdy podczas działania grupy `try` nie wystąpi żaden wyjątek. Można też zdefiniować klauzulę `finally`, która uruchomi się niezależnie od tego, czy wyjątek wystąpi, czy nie. Klauzule `except` przechwytyują wyjątki i wykonują działania zaradcze, natomiast klauzule `finally` służą do definiowania działań końcowych.

Wyjątki mogą być zgłaszane automatycznie przez Pythona lub jawnie za pomocą instrukcji `raise` (warto się również zapoznać z opisem instrukcji `raise` w następnym podrozdziale, „Instrukcja `raise`”). W klauzulach `except` argument *typ* jest wyrażeniem służącym do przekazywania klasy wyjątku do obsłużenia. Dodatkową zmienną o nazwie *wartość* można wykorzystać do przechwycenia instancji klasy zgłoszonego wyjątku. Wszystkie klauzule, które mogą się znaleźć w instrukcji `try`, zestawiono w tabeli 16.

Tabela 16. Formaty klauzul instrukcji `try`

Format klauzuli	Interpretacja
<code>except:</code>	Przechwytywanie wszystkich (lub wszystkich niewymienionych) wyjątków
<code>except typ:</code>	Przechwytywanie tylko specyficznych wyjątków
<code>except typ as wartość:</code>	Przechwytywanie wyjątku i jego instancji
<code>except (typ1, typ2):</code>	Przechwytywanie dowolnego z wymienionych wyjątków
<code>except (typ1, typ2) as wartość:</code>	Przechwytywanie egzemplarza dowolnego z wymienionych wyjątków
<code>else:</code>	Grupa instrukcji uruchamianych, w przypadku gdy nie zostaną zgłoszone wyjątki
<code>finally:</code>	Blok instrukcji wykonywany zawsze na zakończenie

W instrukcji `try` musi wystąpić klauzula `except`, klauzula `finally` lub obie. Kolejność tych elementów powinna być następująca: `try`→`except`→`else`→`finally`, przy czym klauzule `else` i `finally` są opcjonalne, natomiast klauzul `except` może być zero lub więcej. Jeśli jednak występuje klauzula `else`, to konieczna jest co najmniej jedna klauzula `except`. Klauzula `finally` prawidłowo działa z instrukcjami `return`, `break` i `continue` (jeśli dowolna z tych instrukcji przekazuje sterowanie poza blok `try`, to „po drodze” jest wykonywany blok instrukcji `finally`).

Oto kilka popularnych wariantów klauzuli `except`:

`except nazwaklasy as X:`

Przechwycenie wyjątku klasy i przypisanie *X* do zgłoszonego egzemplarza wyjątku. *X* daje dostęp do atrybutów zawierających informacje o stanie i do ciągów znaków do wyświetlania oraz pozwala na wywoływanie metod egzemplarza zgłaszanego wyjątku. W przypadku starszych wyjątków tekstowych do zmiennej *X* są przypisywane dodatkowe dane, które są przekazywane razem z ciągiem znaków (wyjątki tekstowe usunięto z Pythona 3.X i 2.X w wersjach 3.0 i 2.6).

`except (typ1, typ2, typ3) as X:`

Przechwycenie wyjątku dowolnego z typów wymienionych w krotce i przypisanie *X* do dodatkowych danych.

W Pythonie 3.X nazwa *X* w klauzuli `as` jest *lokalna* dla bloku `except` i jest ona usuwana po wyjściu sterowania poza ten blok. W Pythonie 2.X nazwa ta nie jest lokalna dla tego bloku. Warto się zapoznać z opisem wywołania `sys.exc_info()` w podrozdziale „Moduł `sys`”. Można tam znaleźć informacje dotyczące ogólnego dostępu do klas oraz egzemplarzy wyjątków (tzn. *typu* i *wartości*) po zgłoszeniu wyjątku.

Instrukcja `try` w Pythonie 2.X

W Pythonie 2.X instrukcja `try` działa w sposób opisany powyżej, ale klauzula `as`, używana w procedurach obsługi `except` w celu uzyskania dostępu do egzemplarza zgłoszonego wyjątku, jest kodowana za pomocą przecinka. Zarówno klauzula `as`, jak i przecinek działają w Pythonie 2.6 i 2.7 (w celu zachowania zgodności z wersją 3.X), ale klauzula `as` nie jest dostępna we wcześniejszych wersjach linii 2.X:

`except nazwaklasy, X:`

Przechwycenie wyjątku klasy i przypisanie *X* do zgłoszonego egzemplarza wyjątku (w wersjach nowszych niż 2.5 zamiast przecinka należy używać klauzuli `as`).

`except (nazwa1, nazwa2, nazwa3), X:`

Przechwycenie wyjątku dowolnego z typów wymienionych w krotce i przypisanie *X* do dodatkowych danych (w wersjach nowszych niż 2.5 zamiast przecinka należy używać klauzuli `as`).

Instrukcja raise

W Pythonie 3.X instrukcja `raise` może mieć jedną z następujących postaci:

```
raise egzemplarz [from (innywyjatek | None)]
raise klasa [from (innywyjatek | None)]
raise
```

Instrukcja w pierwszej formie powoduje zgłoszenie ręcznie utworzonego egzemplarza klasy (np. `raise Error(argumenty)`). Instrukcja w drugiej postaci powoduje utworzenie i zgłoszenie nowego egzemplarza klasy `klasa` (odpowiednik instrukcji `raise klasa()`). Z kolei instrukcja w trzeciej postaci powoduje ponowne zgłoszenie ostatnio zgłoszonego wyjątku. Informacje na temat opcjonalnej klauzuli `from` można znaleźć w następnym podrozdziale („Łańcuchy wyjątków w Pythonie 3.X”).

Instrukcja `raise` służy do zgłaszania wyjątków. Można ją wykorzystać do jawnego zgłoszenia wbudowanych wyjątków bądź wyjątków zdefiniowanych przez użytkownika. Więcej informacji na temat predefiniowanych wyjątków w Pythonie można znaleźć w podrozdziale „Wbudowane wyjątki”.

W momencie wykonania instrukcji `raise` sterowanie przechodzi do odpowiedniej klauzuli `except` ostatniej instrukcji `try` lub na najwyższy poziom procesu (w takim przypadku następuje zakończenie programu i wyświetlenie standardowego komunikatu o błędzie; „po drodze” wykonywane są napotkane klauzule `finally`). Klauzula `except` jest uważana za dopasowaną, jeśli wymieniono w niej klasę zgłoszonego egzemplarza wyjątku lub jedną z jej klas nadrzędnych (patrz podrozdział „Klasy wyjątków”). Egzemplarz zgłoszonego wyjątku jest przypisywany do zmiennej `as` w dopasowanej klauzuli `except` (o ile klauzula `as` w niej występuje).

Łańcuchy wyjątków w Pythonie 3.X

W Pythonie 3.X (wyłącznie) opcjonalna klauzula `from` umożliwia tworzenie łańcuchów wyjątków — argument `innywyjatek` to inna klasa wyjątku bądź egzemplarz wyjątku dołączony do atrybutu `__cause__` zgłaszanych wyjątków. Jeśli zgłoszony wyjątek nie zostanie przechwycony, Python w standardowym komunikacie o błędzie wyświetla oba wyjątki.

```
try:
    ...
except Exception as E:
    raise TypeError('Żle') from E
```

Począwszy od Pythona 3.3, forma instrukcji `raise from` pozwala również na podanie wartości `None` w celu anulowania łańcuchów wyjątków akumulowanych do chwili uruchomienia instrukcji:

```
raise TypeError('Źle') from None
```

Klasy wyjątków

W Pythonie 3.0 i 2.6 wszystkie wyjątki są definiowane przez klasy, które muszą być pochodnymi wbudowanej klasy `Exception` (w wersji 2.6 to wymaganie dotyczy tylko klas nowego stylu). Klasa bazowa `Exception` dostarcza domyślne komunikaty oraz argumenty konstruktora przechowywane w krotce `args`.

Klasy wyjątków dzielą się na *kategorie*, które można łatwo rozszerzać. Dzięki temu, że instrukcje `try` wywołują wszystkie podklasy, w przypadku gdy zawierają nazwę klasy bazowej, to modyfikowanie zbioru podklas pozwala zmieniać kategorie wyjątków bez potrzeby modyfikowania istniejących instrukcji `try`. Zgłaszany egzemplarz zawiera również kontener pozwalający na przechowywanie dodatkowych informacji o wyjątku:

```
class General(Exception):
    def __init__(self, x):
        self.data = x

class Specific1(General): pass
class Specific2(General): pass

try:
    raise Specific1('spam')
except General as X:
    print(X.data)           # wyświetla 'spam'
```

Instrukcja `raise` w Pythonie 2.X

Zanim wydano Pythona 2.6, w wersji 2.X można było identyfikować wyjątki, używając zarówno łańcuchów znaków, jak i klas. Z tego powodu instrukcje `raise` w tych wersjach mogą przyjmować formy pokazane poniżej (wiele z nich istnieje w celu zapewnienia wstecznej zgodności):

<code>raise łańcuchznaków</code>	<code># Dopasowuje obiekt tekstowy</code>
<code>raise łańcuchznaków, dane</code>	<code># Przypisuje obiekt dane do zmiennej wyjątku</code>
<code>raise klasa, egzemplarz</code>	<code># Dopasowuje klasę lub dowolną klasę bazową</code>
<code>raise egzemplarz</code>	<code># = egz.__class__, egz</code>
<code>raise klasa</code>	<code># = class()</code>

```
raise klasa, arg          # = class(arg), noninst
raise klasa, (arg [, arg]*) # = class(arg, arg, ...)
raise                    # Ponowne zgłoszenie bieżącego wyjątku
```

Od Pythona w wersji 2.5 wyjątki tekstowe nie są zalecane (zgłaszają ostrzeżenia). W Pythonie 2.X w instrukcjach `raise` można podać trzeci element. Jest to obiekt opisujący miejsce, w którym wystąpił wyjątek używany zamiast bieżącej lokalizacji.

Instrukcja `assert`

```
assert wyrażenie [, komunikat]
```

Instrukcja `assert` służy do przeprowadzania testów diagnostycznych. Jeśli *wyrażenie* ma wartość `false`, zgłasza wyjątek `AssertionError` i przekazuje *komunikat* jako dodatkowe dane (o ile ten argument występuje). Flaga wiersza poleceń `-O` wyłącza asercje (ich testy nie są uruchamiane).

Instrukcja `with`

```
with wyrażenie [as zmienna]:  # 3.0/2.6 +
    grupa

with wyrażenie [as zmienna]
    [, wyrażenie [as zmienna]]*:  # 3.1/2.7 +
    grupa
```

Instrukcja `with` opakowuje zagnieżdżony blok kodu w menedżer kontekstu (opisany w dalszej części książki). To daje pewność uruchomienia działań zapisanych w blokach wejściowym i zamykającym niezależnie od tego, czy wyjątki zostaną zgłoszone, czy nie. Jest to alternatywa dla działań końcowych konstrukcji `try/finally`, ale tylko dla obiektów wyposażonych w menedżery kontekstu.

Fraza *wyrażenie* powinna zwracać obiekt, który obsługuje protokół zarządzania kontekstem. Obiekt ten może także zwracać wartość, która zostanie przypisana do zmiennej *zmienna*, w przypadku gdy występuje w niej opcjonalna klauzula `as`. Klasy mogą definiować niestandardowe menedżery kontekstu, a niektóre typy wbudowane, na przykład pliki i wątki, dostarczają menedżery kontekstu wraz z operacjami końcowymi, zamykającymi pliki, zwalnającymi blokady wątków itp.:

```
with open(r'C:\misc\script', 'w') as myfile:
    ...przetwarzanie pliku myfile zamykanego automatycznie przy zakończeniu
    grupy instrukcji...
```

Więcej informacji na temat zastosowania menedżerów kontekstu można znaleźć w podrozdziale „Pliki”. Warto również zajrzeć do dokumentacji Pythona i zapoznać się z innymi typami wbudowanymi obsługującymi ten protokół i tę instrukcję.

Instrukcja `with` jest dostępna od Pythona w wersji 2.6 i 3.0. W wersji 2.5 można ją włączyć za pomocą następującej instrukcji:

```
from __future__ import with_statement
```

Wiele menedżerów kontekstu w Pythonie 3.1 i 2.7

Od Pythona 3.1 i 2.7 instrukcja `with` umożliwia podanie kilku menedżerów kontekstu (tzw. *menedżerów zagnieżdżonych*). Można oddzielić przecinkami dowolną liczbę elementów menedżera kontekstu; podanie wielu elementów ma takie samo działanie jak zagnieżdżone instrukcje `with`. Ogólnie rzecz biorąc, kod Pythona w wersji 3.1, 2.7 i późniejszych w postaci:

```
with A() as a, B() as b:
    ...instrukcje...
```

jest równoważny poniższemu kodowi działającemu w wersjach 3.0 i 2.6:

```
with A() as a:
    with B() as b:
        ...instrukcje...
```

Na przykład w poniższym fragmencie kodu operacje zamykające obu plików są wykonywane automatycznie w momencie wyjścia z bloku instrukcji, niezależnie od zgłaszanych wyjątków:

```
with open('data') as fin, open('res', 'w') as fout:
    for line in fin:
        fout.write(transform(line))
```

Protokół menedżera kontekstu

Obiekty integrują się z instrukcją `with` zgodnie z opisanym poniżej modelem wywołań metody. Więcej informacji można znaleźć w podrozdziale „Menedżery kontekstu”.

1. Na podstawie oceny wartości wyrażenia *wyrażenie* powstaje obiekt znany jako menedżer kontekstu, który musi definiować metody `__enter__` i `__exit__`.
2. Wywołanie metody `__enter__()` menedżera kontekstu. Wartość zwracana przez tę metodę jest przypisywana do zmiennej *zmienna*, o ile ją podano, a w przeciwnym przypadku jest ignorowana.

3. Uruchomienie kodu w zagnieżdżonym bloku instrukcji *grupa*.
4. Jeśli *grupa* zgłosi wyjątek, następuje wywołanie metody `__exit__` ↪ (*typ*, *wartość*, *ślad*) ze szczegółami wyjątku. Jeśli ta metoda zwróci wartość `false`, następuje ponowne zgłoszenie wyjątku. W przeciwnym razie wyjątek jest niszczone.
5. Jeśli *grupa* nie zgłosi wyjątku, to wywołanie metody `__exit__` również następuje, ale wszystkie trzy argumenty są przekazywane jako `None`.

Instrukcje w Pythonie 2.X

Python 2.X obsługuje instrukcję `print` opisaną powyżej, ale nie obsługuje instrukcji `nonlocal` ani (do wersji 2.6) instrukcji `with`. Poza tym instrukcje `raise`, `try` i `def` mają w Pythonie 2.X nieco inne składnie, o czym wspomniano wyżej. Ponadto semantyka oznaczona w poprzednim podrozdziale jako specyficzna dla wersji 3.X (np. pakiety przestrzeni nazw), ogólnie rzecz biorąc, nie ma zastosowania do wersji 2.X.

Poniższa dodatkowa instrukcja jest dostępna wyłącznie w Pythonie 2.X:

```
exec łańcuchkodu [in globalny_słownik [, lokalny_słownik]]
```

Instrukcja `exec` kompiluje i uruchamia łańcuchy kodu. Argument *łańcuchkodu* to dowolna instrukcja Pythona (lub wiele instrukcji oddzielonych od siebie znakami przejścia do nowego wiersza) w postaci tekstowej. Kod uruchamia się w przestrzeni nazw zawierającej instrukcję `exec` albo w przestrzeniach nazw słownika lokalnego bądź globalnego, jeśli je podano (domyślną wartością słownika *lokalny_słownik* jest *globalny_słownik*). Argument *łańcuchkodu* może być również skompilowanym obiektem kodu. Warto się również zapoznać z opisem funkcji `compile()` i `eval()`, a także funkcją Pythona 2.X `execfile()` w podrozdziale „Funkcje wbudowane”.

W Pythonie 3.X funkcję tę zastąpiono funkcją `exec()` (patrz podrozdział „Funkcje wbudowane”). W Pythonie 2.X można również używać składni `exec(a, b, c)`, która zapewnia zgodność wstecz i w przód. Wskazówka: nie należy używać tej instrukcji do sprawdzania niezauważalnych ciągów kodu, ponieważ są one uruchamiane tak samo jak kod programu.

Przestrzenie nazw i reguły zasięgu

W tym podrozdziale zostaną omówione reguły wiązania i wyszukiwania nazw (patrz także „Format nazwy”, „Konwencje nazewnictwa” oraz „Wyrażenia atomowe i dynamiczne określanie typów”). We wszystkich przypadkach nazwy są tworzone przy pierwszym przypisaniu, ale w momencie odwoływania się do nich muszą już istnieć. Nazwy kwalifikowane są rozwiązywane inaczej niż niekwalifikowane.

Nazwy kwalifikowane

— przestrzenie nazw obiektów

Nazwy kwalifikowane (x w nazwie *obiekt.x*) to tzw. *atrybuty*, które istnieją w przestrzeniach nazw obiektów. Instrukcje przypisania w niektórych zasięgach leksykalnych¹⁴ inicjują przestrzenie nazw obiektów (moduły, klasy):

Przypisanie: obiekt.x = wartość

Tworzy bądź modyfikuje atrybut o nazwie x w przestrzeni nazw obiektu *obiekt*. To standardowy przykład; szczegółowe informacje można znaleźć w dalszej części tej książki, w podrozdziale „Formalne reguły dziedziczenia”.

Referencja: obiekt.x

Poszukuje atrybutu o nazwie x wewnątrz obiektu *obiekt*, a następnie we wszystkich dostępnych klasach nadrzędnych (dotyczy egzemplarzy i klas). To definicja dziedziczenia — szczegółowe informacje można znaleźć w podrozdziale „Formalne reguły dziedziczenia”.

Nazwy niekwalifikowane — zasięgi leksykalne

Podczas stosowania nazw niekwalifikowanych — x na początku wyrażenia — obowiązują leksykalne reguły zasięgu. Instrukcja przypisania wiąże taką nazwę z lokalnym zasięgiem, o ile nazwa ta nie zostanie zadeklarowana jako globalna lub `nonlocal` w Pythonie 3.X.

¹⁴ Zasięgi leksykalne odnoszą się do fizycznie (syntaktycznie) zagnieżdżonych struktur kodu w kodzie źródłowym programów.

Przypisanie: x = wartość

Domyślnie tworzy nazwę x jako lokalną — tworzy lub modyfikuje nazwę x w bieżącym zasięgu lokalnym. W przypadku zadeklarowania x z użyciem słowa kluczowego `global` zostanie utworzona nazwa x w zasięgu okalającego modułu. Jeśli x zostanie zadeklarowana z użyciem słowa kluczowego `nonlocal` w Pythonie 3.X, spowoduje to zmianę nazwy x w zasięgu okalającej funkcji. Aby zapewnić szybki dostęp do zmiennych lokalnych w fazie wykonywania programu, są one zapisywane na stosie wywołań. Zmienne lokalne są bezpośrednio widoczne tylko dla kodu z tego samego zasięgu.

Referencja: x

Wyszukiwanie nazwy x w co najwyżej czterech kategoriach zasięgu w następującej kolejności:

- a. w bieżącym zasięgu *lokalnym* (najgłębszej funkcji okalającej);
- b. w zasięgach lokalnych wszystkich leksykalnych *funkcji okalających* (innych warstwach funkcji — od wewnątrz w kierunku na zewnątrz);
- c. w bieżącym zasięgu *globalnym* (okalającym module);
- d. w zasięgu *funkcji wbudowanych* (odpowiadającym metodzie `builtins` modułu w Pythonie 3.X oraz funkcji `__builtin__` w Pythonie 2.X).

Lokalne i globalne konteksty zasięgu zdefiniowano w tabeli 17. Deklaracje `global` powodują, że wyszukiwanie rozpoczyna się w zasięgu globalnym, natomiast deklaracje `nonlocal` w Pythonie 3.X ograniczają wyszukiwanie do okalających funkcji.

Tabela 17. Zasięgi niekwalifikowanych nazw

Kontekst kodu	Zasięg globalny	Zasięg lokalny
Moduł	Taki sam jak lokalny	Sam moduł
Funkcja, metoda	Okalający moduł	Definicja (wywołanie) funkcji
Klasa	Okalający moduł	Instrukcja <code>class</code>
Skrypt, tryb interaktywny	Taki sam jak lokalny	Moduł <code>__main__</code>
<code>exec()</code> , <code>eval()</code>	Zasięg globalny obiektu wywołującego (lub przekazanego)	Zasięg lokalny obiektu wywołującego (lub przekazanego)

Przypadki specjalne: obiekty składane, wyjątki

W Pythonie 3.X zmienne pętli we wszystkich *obiektych składanych* są lokalne (w Pythonie 2.X obowiązują te same zasady dla wszystkich obiektów składanych z wyjątkiem list). W Pythonie 3.X zmienna wyjątku staje się lokalna i jest usuwana z klauzuli `except` instrukcji `try` (w Pythonie 2.X ta nazwa nie jest lokalna). Patrz także podrozdział „Instrukcja `try`”.

Zasięgi zagnieżdżone i domknięcia

Wyszukiwanie odwołań do nazw w *funkcjach okalających* (punkt b, „Referencja”, z reguł opisanych w poprzednim podrozdziale) określa się terminem *zasięgu zagnieżdżonego statycznie*. Zasięgi tego rodzaju włączono do standardu w wersji 2.2. Na przykład poniższa funkcja jest prawidłowa, ponieważ odwołanie do `x` wewnątrz funkcji `f2` ma dostęp do zasięgu funkcji `f1`:

```
def f1():
    x = 42
    def f2():
        print(x)      # Zachowuje x w zasięgu funkcji f1
    return f2          # Do późniejszego wywołania: f1()=>42
```

Funkcje zagnieżdżone, które zachowują referencje zasięgu okalającego (np. `f2` w kodzie zamieszczonym powyżej), to tzw. *domknięcia* — narzędzia do zapamiętywania stanów, które czasami są alternatywą lub uzupełnieniem dla klas. Bardziej przydatne stały się one w wersji 3.X razem ze słowem kluczowym `nonlocal` (patrz „Instrukcja `nonlocal`”). Zasięgi mogą być dowolnie zagnieżdżane, ale przeszukiwane są tylko okalające funkcje (klasy już nie):

```
def f1():
    x = 42
    def f2():
        def f3():
            print(x)      # Znajduje x w zasięgu funkcji f1
            f3()           # f1() wyświetla 42
        f2()
```

Zasięgi okalające i wartości domyślne

W Pythonie w wersjach wcześniejszych niż 2.2 próba wykonania funkcji z poprzedniego podrozdziału nie powiodłaby się, ponieważ nazwa `x` nie jest lokalna (w zasięgu zagnieżdżonej funkcji) ani globalna (w module zawierającym funkcję `f1`), ani wbudowana. W celu obsługi takich

sytuacji w Pythonie w wersji poprzedzającej 2.2 *domyślne argumenty* przechowywały wartości z bezpośrednio okalającego zasięgu (wartości domyślne są obliczane przed wykonaniem instrukcji def):

```
def f1():
    x = 42
    def f2(x=x):
        print(x)      # f1()() wyświetla 42
    return f2
```

Powyższa technika w dalszym ciągu działa w nowszych wersjach Pythona. Dotyczy to również wyrażeń lambda, które zakładają istnienie zasięgu zagnieżdżonego, podobnie jak w przypadku instrukcji def, a w praktyce są znacznie częściej zagnieżdżane:

```
def func(x):
    action = (lambda n: x ** n)      # działa, począwszy od wersji 2.2
    return action                    # func(2)(4)=16

def func(x):
    action = (lambda n, x=x: x ** n) # wartości domyślne — alternatywa
    return action                     # func(2)(4)=16
```

Choć obecnie w większości ról wartości domyślne są w dużej mierze przestarzałe, ciągle bywają potrzebne do odwoływania się do *zmiennych pętli* podczas tworzenia funkcji wewnątrz pętli (odzwierciedlają *ostatyczną* wartość pętli):

```
for I in range(N):
    actions.append(lambda I=I: F(I)) # Bieżąca wartość I
```

Programowanie obiektowe

Głównym narzędziem programowania obiektowego (ang. *object-oriented programming*, OOP) w Pythonie są klasy. Pozwalają one na tworzenie wielu egzemplarzy, dziedziczenie atrybutów oraz przeciążanie operatorów. Python umożliwia również stosowanie technik *programowania funkcyjnego* — daje dostęp do takich narzędzi jak generatory, wyrażenia lambda, listy składane, mapy, domknięcia, dekoratory oraz obiekty funkcyjne pierwszego rzędu. Techniki te mogą służyć jako uzupełnienie, a w niektórych kontekstach jako alternatywa dla technik OOP.

Klasy i egzemplarze

Obiekty klas definiują działania domyślne

- Instrukcja `class` tworzy obiekt *klasy* i przypisuje go do nazwy.
- Instrukcje przypisania wewnątrz instrukcji `class` tworzą *atrybuty klasy*. Są to stany i zachowania obiektu, które mogą być dziedziczone.
- *Metody klasy* to zagnieżdżone instrukcje `def`, które mają specjalny pierwszy argument pozwalający na przekazanie egzemplarza.

Egzemplarze są generowane na podstawie klas

- Wywołanie obiektu klasy — tak jak funkcji — tworzy nowy egzemplarz.
- Każdy egzemplarz dziedziczy atrybuty klas i otrzymuje własny atrybut — *przestrzeń nazw*.
- Instrukcje przypisania do atrybutów pierwszego argumentu wewnątrz metod (np. `self.x = v`) powodują utworzenie *atrybutów egzemplarzy*.

Reguły dziedziczenia

- Dziedziczenie następuje w czasie odwoływania się do atrybutu — w momencie wywołania `obiekt.atrybut`, w przypadku gdy *obiekt* jest klasą lub egzemplarzem.
- Klasy dziedziczą atrybuty po wszystkich klasach wymienionych w wierszu nagłówka ich instrukcji `class` (klas nadrzędnych). Jeśli jest wymieniona więcej niż jedna klasa, oznacza to *dziedziczenie wielokrotne*.
- Egzemplarze dziedziczą atrybuty po klasie, na podstawie której zostały wygenerowane, oraz dodatkowo po wszystkich klasach nadrzędnych tej klasy.
- Mechanizm dziedziczenia polega na przeszukiwaniu egzemplarza, następnie jego klasy oraz wszystkich dostępnych klas nadrzędnych i użyciu pierwszej wersji atrybutu o podanej nazwie. Klasy nadrzędne są najpierw przeszukiwane w głąb, a następnie od lewej do prawej (choć klasy nowego stylu wykonują przeszukiwanie wzdłuż, a następnie w górę).

Więcej informacji na temat dziedziczenia można znaleźć w podrozdziale „Formalne reguły dziedziczenia”.

Atrybuty pseudoprywatne

Domyślnie wszystkie nazwy atrybutów w modułach i klasach są wszędzie widoczne. Istnieją specjalne konwencje pozwalające na ograniczone ukrywanie danych, ale w większości są one przeznaczone do zapobiegania kolizjom nazw (patrz także podrozdział „Konwencje nazewnictwa”).

Prywatne atrybuty modułów

Występujące w modułach nazwy poprzedzone pojedynczym znakiem podkreślenia (np. `_x`) oraz te, których nie wymieniono na liście `__all__` modułu, nie są kopiowane w momencie wywołania instrukcji `from moduł import *`. Nie jest to jednak ścisła prywatność, ponieważ dostęp do takich nazw w dalszym ciągu można uzyskać za pomocą instrukcji `import` w innej postaci.

Prywatne atrybuty klas

Nazwy zdefiniowane w dowolnym miejscu klasy rozpoczynające się dwoma znakami podkreślenia (np. `__x`) są przetwarzane w czasie kompilacji w taki sposób, by zawierały prefiks w postaci nazwy klasy okalającej (np. `_Klasa__x`). Dzięki dodaniu tego prefiksu nazwy są przypisywane do klasy okalającej. Tym samym nazwa staje się unikatowa zarówno dla egzemplarza obiektu `self`, jak i w obrębie hierarchii klas.

Zapobiega to konfliktom, które mogą powstać w przypadku występowania metod o tych samych nazwach. Podobne konflikty mogą się również pojawić w przypadku występowania atrybutów pojedynczego obiektu egzemplarza na dole łańcucha dziedziczenia (wszystkie instrukcje przypisania do atrybutu `self.atrybut` w dowolnym miejscu modyfikują przestrzeń nazw pojedynczego egzemplarza). Nie jest to jednak ścisła prywatność, ponieważ dostęp do takich nazw w dalszym ciągu można uzyskać za pomocą nazw z dodanymi prefiksami.

Kontrolę dostępu zapewniającą prywatność można również zaimplementować za pomocą klas proxy, które walidują dostęp do atrybutów w metodach `__getattr__()` i `__setattr__()` (patrz podrozdział „Metody przeciążające operatory”).

Klasy nowego stylu

W Pythonie 3.X istnieje jeden model klas — wszystkie klasy są uważane za klasy *nowego stylu*, niezależnie od tego, czy dziedziczą z klasy `object`, czy nie. W Pythonie 2.X istnieją dwa modele klas: *klasyczne* (domyślne we wszystkich wersjach 2.X) oraz *nowego stylu*, dostępne od wersji 2.2 i w wersjach późniejszych (dziedziczące po typie wbudowanym lub typie `object`, np. `class A(object)`).

Klasy nowego stylu (wszystkie klasy w Pythonie 3.X) różnią się od klasycznych następującymi cechami:

- Wzorce diamentów (ang. *diamond patterns*) dziedziczenia wielokrotnego charakteryzują się nieco innym porządkiem wyszukiwania — ogólnie rzecz biorąc, przeszukiwanie odbywa się najpierw wzdłuż, a potem w górę. Oznacza to, że najpierw następuje przeszukiwanie wszerz, a nie w głąb. Jest to zgodne z nowym stylem `__mro__` (patrz „Formalne reguły dziedziczenia”).
- Klasy są typami, a typy są klasami. Wbudowana funkcja `type(I)` zwraca klasę, na podstawie której powstał egzemplarz `I`, a nie generyczny typ egzemplarza. Identyczny wynik daje wywołanie metody `I.__class__`. Istnieje możliwość tworzenia klas podrzędnych klasy `type`, co pozwala na personalizację tworzenia klas. Wszystkie klasy dziedziczą po klasie `object`.
- Metody `__getattr__` i `__getattribute__` nie działają w odniesieniu do atrybutów pobieranych niejawnie przez operacje wbudowane. Nie są one wywoływane w celu uzyskania nazw metod przeciążających operatory `__x__`. Wyszukiwanie takich nazw rozpoczyna się od klas, a nie egzemplarzy. W celu przechwycenia i delegowania dostępu do nazw takich metod należy je ponownie zdefiniować w klasach opakowujących (pośredniczących).
- Klasy nowego stylu dysponują zbiorem nowych narzędzi, takich jak gniazda (ang. *slots*), właściwości, deskryptory, a także metodą `__getattribute__()`. Większość z nich służy do tworzenia narzędzi. Opis metod `__slots__`, `__getattribute__()` oraz metod deskryptorów `__get__()`, `__set__()` i `__delete__()` można znaleźć w podrozdziale „Metody przeciążające operatory”. O funkcji `property()` można dowiedzieć się więcej w podrozdziale „Funkcje wbudowane”.

Formalne reguły dziedziczenia

Dziedziczenie następuje w chwili odwołania się do nazwy atrybutu — w momencie wyszukiwania *obiekt.nazwa* w kodzie obiektowym — za każdym razem, kiedy *obiekt* wywodzi się z klasy. Różni się ono w klasach klasycznych oraz nowego stylu, chociaż typowy kod zawsze działa tak samo w obu modelach.

Klasyczne klasy — DFLR

W klasycznych klasach (domyślnych w Pythonie 2.X) nazwy są rozwiązywane zgodnie z następującymi regułami dziedziczenia:

1. *egzemplarz*,
2. *klasa*,
3. wszystkie *klasy bazowe* tej klasy — najpierw w głąb, a następnie od lewej do prawej.

Używane jest pierwsze znalezione wystąpienie. Ta kolejność jest znana jako *DFLR* (ang. *depth first left right*).

Wyszukiwanie referencji może być zainicjowane z poziomu egzemplarza bądź klasy; operacje *przypisania* atrybutów są standardowo przechowywane w obiekcie docelowym, bez wyszukiwania. Są również specjalne przypadki dla metod `__getattr__()` (uruchamia się, kiedy wyszukiwanie nazwy się nie powiedzie) oraz `__setattr__()` (uruchamianej dla operacji przypisania dla wszystkich atrybutów).

Klasy nowego stylu — MRO

Dziedziczenie w klasach nowego stylu (standardowe w Pythonie 3.X i opcja w Pythonie 2.X) bazuje na *MRO* — liniowej ścieżce przez drzewo klasy oraz zagnieżdżonym komponencie dziedziczenia dostępnym za pośrednictwem atrybutu `__mro__`. Porządek MRO jest w przybliżeniu obliczany w następujący sposób:

1. Stworzenie listy wszystkich klas, które egzemplarz dziedziczy za pomocą klasycznej reguły wyszukiwania DFLR, z włączeniem klasy kilkakrotnie, jeżeli została użyta więcej niż raz.
2. Skanowanie listy wynikowej w celu wyszukania duplikatów klas. Usunięcie z listy wszystkich duplikatów z wyjątkiem ostatniego (skrajnego prawego).

Powstała sekwencja MRO dla danej klasy obejmuje klasę, jej klasy nadrzędne i wszystkie klasy nadrzędne wyższego rzędu, włącznie z występującą jawnie lub niejawnie główną klasą `object` na szczycie drzewa. Sekwencja ta jest uporządkowana w taki sposób, że każda klasa pojawia się przed swoimi rodzicami, a w przypadku wielu rodziców zachowana jest kolejność, w jakiej klasy rodzice występują w krotce `__bases__` klasy nadrzędnej.

Ponieważ wspólni rodzice w *diamentach* pojawiają się tylko w miejscu ich *ostatniego* wystąpienia w MRO, to podczas korzystania z listy MRO przez mechanizm dziedziczenia atrybutów najpierw są przeszukiwane niższe klasy (w związku z tym w obrębie samych diamentów przeszukiwanie odbywa się bardziej w szerz niż w głąb), a każda klasa występuje tylko raz, a zatem tylko raz jest sprawdzana, niezależnie od tego, ile klas do niej prowadzi.

Porządek MRO jest wykorzystywany zarówno przez mechanizm dziedziczenia (co opisano powyżej), jak i przez wywołanie `super()` — funkcję wbudowaną, która zawsze wywołuje *następną* klasę w MRO (względem punktu wywołania). Nie musi to być klasa nadrzędna, ale można ją stosować do dystrybucji wywołań metod po całym drzewie klas, tak aby każda klasa była odwiedzona tylko raz.

Przykład: bez diamentów

```
class D:      attr = 3      # D:3      E:2
class B(D):   pass          # |          |
class E:      attr = 2      # B          C:1
class C(E):   attr = 1      # \          /
class A(B, C): pass         #   A
X = A()       #           |
print(X.attr) #           X

# DFLR = [X, A, B, D, C, E]
# MRO = [X, A, B, D, C, E, object]
# Wyświetla "3" zarówno w Pythonie 3.X, jak i 2.X (zawsze)
```

Przykład: diamenty

```
class D:      attr = 3      # D:3 D:3
class B(D):   pass          # |      |
class C(D):   attr = 1      # B      C:1
class A(B, C): pass         # \      /
X = A()       #           A
print(X.attr) #           |
              #           X

# DFLR = [X, A, B, D, C, D]
# MRO = [X, A, B, C, D, object] (zachowuje tylko ostatnią klasę D)
# Wyświetla "1" w Pythonie 3.X, "3" w Pythonie 2.X ("1", jeśli D(object))
```

Algorytm dziedziczenia nowego stylu

W zależności od kodu klasy nowy styl dziedziczenia może obejmować deskryptory, metaklasy i sekwencje MRO w opisany poniżej sposób (źródła nazw w tej procedurze są sprawdzane po kolei — według numeracji, a w połączeniach „or” według porządku od lewej do prawej).

Wyszukiwanie nazwy atrybutu

1. Zaczynając od *egzemplarza I*, poszukaj egzemplarza, jego klasy i jego klas bazowych w następujący sposób:
 - a. Odszukaj atrybut `__dict__` wszystkich klas dla `__mro__` znalezionej w atrybucie `__class__` egzemplarza *I*.
 - b. Jeśli w kroku *a* został znaleziony deskryptor *data*, wywołaj jego metodę `__get__()` i zakończ działanie.
 - c. Jeśli nie, zwróć wartość w atrybucie `__dict__` egzemplarza *I*.
 - d. W pozostałych przypadkach wywołaj deskryptor *nondata* lub zwróć wartość znaną w kroku *a*.
2. Zaczynając od *klasy C*, poszukaj klasy, jej klas bazowych i metaklas w następujący sposób:
 - a. Odszukaj atrybut `__dict__` wszystkich metaklas dla `__mro__` znalezionej w atrybucie `__class__` klasy *C*.
 - b. Jeśli w kroku *a* został znaleziony deskryptor *data*, wywołaj jego metodę `__get__()` i zakończ działanie.
 - c. Jeśli nie, wywołaj deskryptor lub zwróć wartość w atrybucie `__dict__` własnego atrybutu `__mro__` klasy *C*.
 - d. W pozostałych przypadkach wywołaj deskryptor *nondata* lub zwróć wartość znaną w kroku *a*.
3. Zarówno w regule 1., jak i 2. operacje *wbudowane* (np. wyrażenia) w istocie wykorzystują źródła kroku *a* w celu jawnego wyszukiwania nazw metod, natomiast wyszukiwanie wywołania `super()` jest spersonalizowane.

Poza tym jeśli atrybut nie zostanie znaleziony, można uruchomić metodę `__getattr__()` (o ile ją zdefiniowano). Metodę `__getattribute__()` można uruchomić dla każdej operacji pobrania atrybutu, natomiast domniemana klasa bazowa `object` dostarcza atrybutów domyślnych na szczycie drzewa wszystkich klas i metaklas (tzn. na końcu każdego MRO).

W szczególnych przypadkach wbudowane operacje pomijają źródła nazw, tak jak zostało to opisane w regule 3., natomiast wbudowana funkcja `super()` uniemożliwia normalne dziedziczenie. Dla obiektów zwracanych przez funkcję `super()` atrybuty są rozwiązywane w wyniku specjalnego kontekstowego skanowania ograniczonej części MRO samej klasy. Zamiast uruchamiania pełnego dziedziczenia wybierany jest pierwszy znaleziony deskryptor lub wartość (pełny algorytm jest wykorzystywany tylko w odniesieniu do samego obiektu `super` wyłącznie wtedy, gdy skanowanie zawiedzie). Więcej informacji na ten temat można znaleźć w opisie funkcji `super()` w podrozdziale „Funkcje wbudowane”.

Przypisywanie nazwy atrybutu

Do przypisywania atrybutów stosowany jest podzbiór procedury wyszukiwania:

- Jeśli zastosujemy przypisania do *egzemplarza*, tego rodzaju przypisania są realizowane zgodnie z podpunktami od *a* do *c* reguły 1. Przeszukiwane jest drzewo klasy egzemplarza, choć w kroku *b* wywoływana jest metoda `__set__()` zamiast `__get__()`, natomiast w kroku *c* procedura się zatrzymuje i zapisuje wartość w egzemplarzu, zamiast podjąć próbę pobrania wartości.
- Jeśli procedura zostanie użyta w odniesieniu do *klasy*, to tego rodzaju przypisania uruchamiają taką samą procedurę jak dla drzewa metaklasy: w przybliżeniu zgodnie z regułą 2., ale w kroku *c* następuje zatrzymanie procedury i zapisanie wartości w klasie.

Metoda `__setattr__()` w dalszym ciągu przechwytuje wszystkie przypisania atrybutów, tak jak było wcześniej, ale w tej metodzie mniej przydatne jest wykorzystanie egzemplarza `__dict__` do przypisania nazw, ponieważ niektóre rozszerzenia nowego stylu, takie jak sloty, właściwości i deskryptory, implementują atrybuty na poziomie klasy — jest to rodzaj „wirtualnego” mechanizmu danych egzemplarza. Niektóre egzemplarze w ogóle nie zawierają atrybutu `__dict__`. Jest tak, jeśli są wykorzystywane sloty (w celu optymalizacji).

Zasady pierwszeństwa i kontekst w klasach nowego stylu

Procedury dziedziczenia według nowego stylu skutecznie narzucają zasady pierwszeństwa w przypadku podstawowych operacji dotyczących rozwiązywania nazw. Opisano je poniżej (w nawiasach podano odpowiedniki kroków algorytmu dziedziczenia).

W przypadku egzemplarzy próbuj:

1. Deskryptorów *data* drzewa klasy (1b).
2. Wartości obiektów egzemplarzy (1c).
3. Deskryptorów *nondata* drzewa klasy (1d).
4. Wartości drzewa klasy (1d).

W przypadku klas próbuj:

1. Deskryptorów *data* drzewa metaklasy (2b).
2. Deskryptorów drzewa klasy (2c).
3. Wartości drzewa klasy (2c).
4. Deskryptorów *nondata* drzewa metaklasy (2d).
5. Wartości drzewa metaklasy (2d).

Python uruchamia co najwyżej jedną (dla reguły 1.) bądź dwie (dla reguły 2.) operacje przeszukiwania drzewa na jedno wyszukiwanie nazwy, mimo że istnieją cztery lub pięć źródeł nazw. Warto się również zapoznać z opisem specjalnego przypadku procedury wyszukiwania dla obiektów zwracanych przez wbudowaną funkcję `super()` nowego stylu.

Patrz także „Deskryptory” oraz „Metaklasy”. W podrozdziale „Metody przeciążające operatory” można znaleźć szczegółowe informacje dotyczące użycia metod `__setattr__()`, `__getattr__()` i `__getattribute__()`. Warto też zajrzeć do plików kodu źródłowego *object.c* i *typeobject.c*, które zawierają implementacje odpowiednich egzemplarzy i klas (w dystrybucji z kodem źródłowym Pythona).

Metody przeciążające operatory

Klasy przechwytyują i implementują działania wbudowane poprzez metody o specjalnych nazwach. Wszystkie one rozpoczynają się i kończą dwoma znakami podkreślenia. Nazwy te nie są zarezerwowane i mogą być dziedziczone z klas nadrzędnych w zwykły sposób. Python wyszukuje i wywołuje co najmniej jedną taką metodę w każdej operacji.

Python automatycznie wywołuje metody przeciążające klasy, w przypadku gdy egzemplarze znajdują się w wyrażeniach oraz innych kontekstach. Jeśli na przykład klasa definiuje metodę o nazwie `__getitem__`, a *x* jest egzemplarzem tej klasy, to wyrażenie *x*[*i*] jest równoważne

wywołaniu metody `x.__getitem__(i)` (choćaż obecnie stosowanie wywołania metody na ogół nie poprawia szybkości, a czasami nawet ją pogarsza).

Nazwy przeciążanych metod mogą być dowolne — metoda `__add__` klasy nie musi realizować dodawania lub konkatencji (choć w zasadzie powinna pełnić podobną funkcję). Co więcej, w klasach dopuszczalne jest mieszanie metod przetwarzających liczby i kolekcje oraz działań mutowalnych i niemutowalnych. Większość nazw przeciążających operatory nie ma wartości domyślnych, a odpowiadające im działania zgłaszają wyjątek, jeśli określona metoda nie jest zdefiniowana (np. `+` bez `__add__`).

W poniższych podrozdziałach zestawiono dostępne metody działań. W tym podrozdziale końcowe nawiasy w nazwach metod `__x__` pominięto w celu zapewnienia zwięzłości, ponieważ ich kontekst jest domniemany. Ten podrozdział koncentruje się na Pythonie 3.X, ale informacje dotyczące przeciążania operatorów są wspólne dla większości wersji Pythona. Informacje specyficzne dla Pythona 2.X można znaleźć w podrozdziale „Metody przeciążające operatory w Pythonie 2.X”.

Wszystkie typy

`__new__(klasa [, argumenty]*)`

Metoda wywoływana w celu utworzenia i zwrócenia nowego egzemplarza klasy *klasa*. Metoda otrzymuje argumenty konstruktora przekazane do klasy. Jeśli zwróci egzemplarz klasy *klasa*, to następuje wywołanie metody `__init__` nowego egzemplarza *self* z takimi samymi argumentami jak argumenty konstruktora. W przeciwnym razie metoda `__init__` nie jest uruchamiana. Zwykle metoda jest kodowana w celu wywołania metody `__new__` klasy nadrzędnej za pośrednictwem jawnie podanej nazwy klasy nadrzędnej albo za pomocą funkcji `super()` (patrz podrozdział „Funkcje wbudowane”), a także do zarządzania i zwrócenia uzyskanego w ten sposób egzemplarza. Jest to metoda automatycznie statyczna.

Metoda nie jest używana w standardowych klasach. Spełnia dwa cele: pozwala klasom podrzędnym niezmiennych typów spersonalizować tworzenie egzemplarzy, a *metaklasy* mogą dzięki niej tworzyć klasy w specyficzny dla siebie sposób. Warto się również zapoznać z opisem funkcji `type()` w podrozdziale „Funkcje

wbudowane”. Można tam znaleźć informacje na temat innego przypadku użycia tej metody z argumentami tworzenia klasy.

`__init__(self [, argumenty]*)`

Metoda wywoływana w momencie odwołania *klasa(argumenty...)*. Jest to metoda *konstruktora*, która inicjuje nowy egzemplarz *self*. W przypadku uruchomienia dla wywołań nazwy klasy argument *self* jest dostarczany automatycznie. Do nazwy klasy są przekazywane argumenty, które mogą przyjąć dowolną postać właściwą dla argumentów definiowania funkcji (więcej informacji na ten temat można znaleźć w podrozdziałach „Instrukcja wyrażeniowa” oraz „Instrukcja def”, włącznie z tabelą 14.).

Chociaż z technicznego punktu widzenia metoda `__init__` jest wywoływana po metodzie `__new__`, jest to preferowany sposób konfigurowania nowych obiektów we wszystkich klasach poziomu aplikacji. Metoda nie może zwracać wartości, a jeśli jest taka potrzeba, to powinna ręcznie wywoływać metodę `__init__` klasy nadrzędnej za pomocą jawnie podanej nazwy klasy nadrzędnej albo przy użyciu funkcji `super()` (patrz podrozdział „Funkcje wbudowane”), przekazując egzemplarz za pomocą argumentu *self*. Python automatycznie wywołuje tylko jedną metodę `__init__`.

`__del__(self)`

Wywoływana w czasie przeprowadzania odśmiecania na poziomie egzemplarza. Ta metoda *destruktor* wykonuje zadania „sprzątania” w momencie zwalniania egzemplarza *self*. Obiekty osadzone są automatycznie zwalniane w momencie zwalniania obiektu nadrzędnego (o ile nie nastąpi do nich odwołanie z innego miejsca). Wyjątki, które powstaną podczas działania tej metody, są ignorowane i wyświetlają komunikaty na urządzeniu `sys.stderr`. Wskazówka: bardziej przewidywalne działania końcowe dla bloku kodu zapewnia instrukcja `try/finally`. Instrukcja `with` dostarcza podobny mechanizm dla obsługiwanych typów obiektowych.

`__repr__(self)`

Wywoływana w momencie wywoływania funkcji `repr(self)`, echa interaktywnego i zagnieżdżonych wystąpień (a także odwołania do ``self`` — tylko w Pythonie 2.X). Jest wywoływana także podczas wykonywania funkcji `str(self)` i `print(self)`, w przypadku gdy nie zdefiniowano metody `__str__`. Metoda zwraca niskopoziomą tekstową reprezentację egzemplarza *self* w postaci kodu.

`__str__(self)`

Metoda wywoływana podczas wykonywania funkcji `str(self)` i `print(self)` (wykorzystuje pomocniczo metodę `__repr__`, jeśli ją zdefiniowano). Metoda zwraca wysokopoziomową tekstową reprezentację egzemplarza `self` w postaci przyjaznej dla użytkownika.

`__format__(self, specyfikacjaformatu)`

Wywoływana przez wbudowaną funkcję `format()` (oraz jako rozszerzenie metody `str.format()` obiektów tekstowych) w celu utworzenia „sformatowanej” tekstowej reprezentacji obiektu zgodnie z argumentem *specyfikacjaformatu*. Składnia tego argumentu dla typów wbudowanych jest taka sama jak dla komponentu o identycznej nazwie w metodzie `str.format()`. Więcej informacji można znaleźć w podrozdziałach „Składnia metody formatującej”, „Formatowanie łańcuchów znaków” oraz „Funkcje wbudowane”. Metoda jest nowością wprowadzoną w Pythonie 2.6 i 3.0.

`__bytes__(self)`

Metoda wywoływana przez funkcję `bytes()` w celu zwrócenia tekstowej reprezentacji bajtów obiektu `self`. Dostępna wyłącznie w Pythonie 3.X.

`__hash__(self)`

Wywoływana w momencie odwołań `dictionary[self]` oraz `hash ↪(self)`, a także w czasie przeprowadzania innych operacji na kolekcjach, w których są wykorzystywane skróty (ang. *hash*) — w tym również tych, które dotyczą typu obiektu `set`. Metoda zwraca unikatowy i niezmienny klucz skrótu będący liczbą całkowitą. Metoda subtelnie współdziała z metodą `__eq__`. Obie te metody mają wartości domyślne, które zapewniają, że podczas porównywania obiektów tylko porównanie z samym sobą daje wynik „równy”. Więcej informacji na ten temat można znaleźć w dokumentacji Pythona.

`__bool__(self)`

Wywoływana podczas przeprowadzania testów na prawdziwość wartości oraz w trakcie wykonywania wbudowanej funkcji `bool()`. Zwraca wartości `False` bądź `True`. Jeśli nie zdefiniowano metody `__bool__`, wywoływana jest metoda `__len__()` — wartość `True` odpowiada niezerowej długości. Jeśli nie zostanie zdefiniowana ani metoda `__len__`, ani `__bool__`, to wszystkie egzemplarze takiej klasy są wartościowane jako `true`. Metoda wprowadzona w Pythonie 3.X. W Pythonie 2.X metoda o analogicznym działaniu nosi nazwę `__nonzero__`.

`__call__(self [, argumenty]*)`

Wywoływana w momencie wywołania `self(argumenty...)` — tzn. gdy egzemplarz zostanie wywołany tak jak funkcja. *argumenty* mogą przyjąć dowolną formę prawidłową w definicjach funkcji. Na przykład obie instrukcje:

```
def __call__(self, a, b, c, d=5):  
def __call__(self, *pargs, **kargs):
```

odpowiadają następującym wywołaniom:

```
self(1, 2, 3, 4)  
self(1, *(2,), c=3, **dict(d=4))
```

Więcej informacji na temat opcji argumentu *argumenty* można znaleźć w podrozdziale „Instrukcja def” (włącznie z tabelą 14.).

`__getattr__(self, nazwa)`

Wywoływana w momencie odwołania `self.nazwa`, gdzie *nazwa* oznacza nieokreślony dostęp do atrybutu (metoda nie jest wywoływana, jeśli *nazwa* istnieje lub została odziedziczona przez egzemplarz `self`). *nazwa* jest łańcuchem znaków. Metoda zwraca obiekt lub zgłasza wyjątek `AttributeError`.

Metoda jest dostępna zarówno dla klasycznych klas, jak i dla klas nowego stylu. W Pythonie 3.X i w klasach nowego stylu w wersji 2.X metoda ta nie jest niejawnie wywoływana dla odwołań do atrybutów `__x__` pobieranych przez operacje wbudowane (np. wyrażenia). Takie nazwy należy ponownie zdefiniować w klasach opakowujących (klasach proxy) lub w klasach nadrzędnych. Patrz także opis metody `__dir__` na tej liście.

`__setattr__(self, nazwa, wartość)`

Wywoływana przy okazji odwołań `self.nazwa=wartość` (wszystkich operacji przypisania atrybutów). Wskazówka: aby uniknąć pętli rekurencyjnych, przypisania powinny się odbywać za pośrednictwem klucza `__dict__` lub klasy nadrzędnej (np. `object`); instrukcja `self.attr=x` w obrębie wywołań metody `__setattr__` ponownie wywołuje metodę `__setattr__`, ale wywołanie `self.__dict__['attr'] = x` tego nie robi.

Rekurencji można także uniknąć dzięki jawnemu wywołaniu wersji z klasy bazowej dla klas nowego stylu: `object.__setattr__(self, nazwa, wartość)`. Może to być korzystne lub konieczne dla drzew klas, które implementują „wirtualne” atrybuty egzemplarza na poziomie klasy, takie jak *sloty*, *właściwości* lub *deskryptory* (np. sloty mogą wykluczać metodę `__dict__` egzemplarza).

`__delattr__(self, nazwa)`

Wywoływana przy okazji wykonywania instrukcji `del self.nazwa` (wszystkich operacji usuwania atrybutów). Wskazówka: należy unikać rekurencyjnych pętli poprzez kierowanie operacji usunięcia atrybutów przez metodę `__dict__` lub klasę nadrzędną — podobnie do metody `__setattr__`.

`__getattr__(self, nazwa)`

Wywoływana bezwarunkowo w celu zaimplementowania dostępu atrybutów do egzemplarza klasy. Jeśli klasa jednocześnie definiuje metodę `__getattr__`, ta metoda nigdy nie zostanie wywołana (o ile nie wywoła się jej jawnie). Metoda powinna zwrócić (obliczoną) wartość atrybutu lub zgłosić wyjątek `AttributeError`. W celu uniknięcia nieskończonych rekurencji w tej metodzie podczas sięgania do atrybutów jej implementacja zawsze powinna wywoływać metodę klasy nadrzędnej o takiej samej nazwie (np. `object.__getattr__(self, nazwa)`).

Dostępna w Pythonie 3.X oraz 2.X wyłącznie dla klas *nowego stylu*. Zarówno w Pythonie 3.X, jak i w *klasach nowego stylu* w wersji 2.X metoda ta nie jest niejawnie wywoływana dla odwołań do atrybutów `__x__` pobieranych niejawnie przez operacje *wbudowane* (np. wyrażenia). Takie nazwy należy ponownie zdefiniować w klasach opakowujących (klasach proxy). Patrz także opis metody `__dir__` na tej liście.

`__lt__(self, inny)`

`__le__(self, inny)`

`__eq__(self, inny)`

`__ne__(self, inny)`

`__gt__(self, inny)`

`__ge__(self, inny)`

Metody używane podczas porównań `self < inny`, `self <= inny`, `self == inny`, `self != inny`, `self > inny` oraz `self >= inny`. Wprowadzono je w wersji 2.1. Są to tzw. metody *bogatych porównań*, które w Pythonie 3.X są wywoływane w odniesieniu do wszystkich wyrażen wykorzystujących porównania. Na przykład porównanie `x < y` wywołuje metodę `x.__lt__(y)` (o ile ją zdefiniowano). W Pythonie 2.X metody te są preferowane zamiast metody `__cmp__`, a dla wyrażenia `self <> inny` wywoływana jest również metoda `__ne__`.

Metody mogą *zwrócić* dowolną wartość, ale jeśli operator porównania zostanie użyty w kontekście operacji logicznych, to zwrócona wartość zostanie zinterpretowana jako logiczny wynik (typu

Boolean) działania operatora. Metody te mogą również zwracać (choć nie zgłaszają wyjątku) specjalny obiekt `NotImplemented`, w przypadku gdy operandy ich nie obsługują (efekt jest taki, jakby metoda w ogóle nie została zdefiniowana; w Pythonie 2.X taka sytuacja zmusza do sięgnięcia do ogólnej metody `__cmp__`, o ile ją zdefiniowano).

Nie istnieją domniemane relacje pomiędzy operatorami porównań. Na przykład z faktu, że wyrażenie `X == Y` ma wartość `true`, nie wynika, że wyrażenie `X != Y` ma wartość `false` — aby operatory działały symetrycznie, należy zdefiniować metodę `__ne__` razem z metodą `__eq__`. Nie istnieją również prawostronne (o zamienionych argumentach) wersje tych metod, które można by wykorzystać w sytuacji, kiedy lewy argument nie obsługuje określonego działania, a prawy je obsługuje. Metody `__lt__` i `__gt__` są swoimi wzajemnymi odbiciami, `__le__` i `__ge__` są wzajemnymi odbiciami, natomiast `__eq__` i `__ne__` są odbiciami dla samych siebie. W Pythonie 3.X w operacji sortowania należy używać metody `__lt__`. Na temat roli metody `__eq__` w tworzeniu skrótów można przeczytać w dokumentacji Pythona.

`__slots__`

Do tego atrybutu klasy można przypisać łańcuch znaków, sekwencję albo inny obiekt iterowalny złożony z łańcuchów znaków opisujących nazwy atrybutów egzemplarza klasy. Jeśli atrybut `__slots__` zostanie zdefiniowany w klasie *nowego stylu* (wszystkich klasach w Pythonie 3.X), to wygeneruje deskryptor zarządzania poziomą klasy (patrz „Deskryptory”). W takim przypadku zarezerwuje miejsce dla zadeklarowanych atrybutów w egzemplarzach i zabezpieczy przed automatycznym tworzeniem atrybutu `__dict__` dla każdego egzemplarza (jeśli w ramach atrybutu `__slots__` nie zostanie uwzględniony ciąg `'__dict__'`, egzemplarze również będą miały atrybut `__dict__`, atrybuty niewymienione w `__slots__` będą mogły być dodane automatycznie).

Ponieważ sloty mogą zablokować atrybut `__dict__` na poziomie egzemplarza, ich zastosowanie może zoptymalizować wykorzystanie miejsca. Niemniej poza uzasadnionymi przypadkami na ogół nie są one zalecane zarówno ze względu na zwiększenie ryzyka złamania pewnych typów kodu, jak i na skomplikowane ograniczenia użytkowania (szczegółowe informacje można znaleźć w dokumentacji Pythona).

W celu obsłużenia klas z atrybutem `__slots__` narzędzia, które wyświetlają atrybuty bądź korzystają z nich na podstawie nazwy

tekstowej, muszą używać narzędzi neutralnych pod względem pamięci masowej, takich jak `getattr()`, `setattr()` i `dir()`, mających zastosowanie zarówno do atrybutu `__dict__`, jak i `__slots__`.

`__instancecheck__(self, egzemplarz)`

Zwraca `true` dla metody `isinstance()`, w przypadku gdy *egzemplarz* może być uznany za bezpośredni lub pośredni egzemplarz klasy. Wprowadzona w Pythonie 3.X i 2.6. Informacje na temat sposobu użycia można znaleźć w dokumentacji Pythona.

`__subclasscheck__(self, podklasa)`

Zwraca `true` dla metody `issubclass()`, w przypadku gdy *podklasa* może być uznana za bezpośrednią lub pośrednią podklasę klasy. Wprowadzona w Pythonie 3.X i 2.6. Informacje na temat sposobu użycia można znaleźć w dokumentacji Pythona.

`__dir__(self)`

Wywoływana podczas działania funkcji wbudowanej `dir(self)` (patrz „Funkcje wbudowane”). Zwraca sekwencję nazw atrybutów. Pozwala, aby atrybuty niektórych klas były znane w czasie introspekcji za pomocą funkcji `dir()`, gdy atrybuty te są obliczane dynamicznie za pomocą takich narzędzi jak `__getattr__`, przy założeniu, że są one znane dla samej klasy. Przypadki dynamicznego użycia mogą nie być kwalifikowane bezpośrednio, ale *proxy ogólnego przeznaczenia* mogą delegować takie wywołania do innych obiektów w celu obsługi narzędzi atrybutów. Nowość w Pythonie 3.X. Wprowadzona również jako *backport* do wersji Pythona 2.6 i 2.7.

Kolekcje (sekwencje, mapy)

`__len__(self)`

Wywoływana podczas wykonywania funkcji wbudowanej `len` `↪(self)` oraz ewentualnie w trakcie wykonywania sprawdzenia, czy obiekt ma wartość `true`. Metoda zwraca rozmiar kolekcji. W testach logicznych Python najpierw poszukuje metody `__bool__`, następnie `__len__` i na koniec uznaje, że obiekt ma wartość `true` (atrybut `__bool__` w Pythonie 2.X ma nazwę `__nonzero__`). Zerowa długość oznacza `false`.

`__contains__(self, element)`

Wywoływana podczas odwołania *element* in *self* testującego przynależność (w przeciwnym razie w teście przynależności wykorzystywana jest metoda `__iter__`, o ile zostanie zdefiniowana, bądź `__getitem__`). Metoda zwraca `true` lub `false`.

`__iter__(self)`

Wywoływana podczas odwołania `iter(self)`. Metodę tę wprowadzono w wersji 2.2. Jest ona częścią *protokołu iteracji*. Zwraca obiekt za pomocą metody `__next__` (ewentualnie `self`). Następnie we wszystkich kontekstach iteracyjnych (np. pętlach `for`) jest kolejno wywoływana metoda `__next__()`. Powinna ona zwrócić kolejny wynik lub zgłosić wyjątek `StopIteration` w celu zakończenia progresji wyników.

Jeśli metoda `__iter__` nie zostanie zdefiniowana, w iteracjach wykorzystywana jest metoda `__getitem__`. Metodę `__iter__` klasy można również zakodować za pomocą osadzonej instrukcji `yield` zwracającej generator z automatycznie utworzoną metodą `__next__`. W Pythonie 2.X metoda `__next__` nosi nazwę `next`. Zobacz też podrozdziały „Instrukcja `for`” i „Protokół iteracji”.

`__next__(self)`

Wywoływana przy uruchomieniu wbudowanej funkcji `next(self)` oraz we wszystkich kontekstach iteracyjnych do przeglądania wyników. Metoda ta jest częścią *protokołu iteracji*. Więcej informacji na temat wykorzystania tej metody można znaleźć w opisie metody `__iter__` na tej liście. Metoda wprowadzona w Pythonie 3.X. W Pythonie 2.X metoda o analogicznym działaniu nosi nazwę `next`.

`__getitem__(self, klucz)`

Wywoływana w momencie odwołań `self[klucz]`, `self[i:j:k]`, `x in self`, a także we wszystkich kontekstach iteracyjnych. Metoda ta implementuje wszystkie operacje związane z indeksowaniem, włącznie z tymi, które dotyczą sekwencji i map. W kontekstach iteracyjnych (np. `in` oraz w instrukcji `for`) następuje wielokrotne indeksowanie od 0 do wartości `IndexError`, o ile nie zostanie zdefiniowana metoda `__iter__`. Metody `__getitem__` i `__len__` tworzą *protokół sekwencji*.

W Pythonie 3.X ta oraz dwie kolejne metody są także wywoływane w operacjach *wycinania*. W tym przypadku argument `klucz` oznacza obiekt wycinka. Obiekty wycinków można propagować do innych wyrażeń wycinających. Mają one atrybuty `start`, `stop` i `step`. Każdy z nich może mieć wartość `None`. Więcej informacji na ten temat można znaleźć w opisie funkcji `slice()` w podrozdziale „Funkcje wbudowane”.

`__setitem__(self, klucz, wartość)`

Wywoływana podczas odwołań `self[klucz]=wartość` oraz `self[i:j:k]=wartość`. Ta metoda jest wywoływana w celu przypisania do klucza, indeksu kolekcji albo do wycinka sekwencji.

- `__delitem__(self, klucz)`
Wywoływana podczas odwołań `del self[klucz]` oraz `del self` \hookrightarrow `[i:j:k]`. Metoda wywoływana podczas usuwania indeksów (kluczy) oraz wycinków sekwencji.
- `__reversed__(self)`
Jeśli ta metoda jest zdefiniowana, to jest wywoływana przez wbudowaną funkcję `reversed()` w celu implementacji niestandardowych iteracji odwracających. Zwraca nowy obiekt iterowalny, który realizuje iterację dla wszystkich obiektów w kontenerze w odwróconym porządku. Jeśli metoda `__reversed__` nie zostanie zdefiniowana, funkcja `reversed()` wykorzystuje protokół sekwencji (metody `__len__` i `__getitem__`).

Liczby (operatory dwuargumentowe)

Jeśli któraś z metod działań na liczbach (oraz porównań) nie obsługuje działania dla przekazanych argumentów, to powinna zwrócić (a nie zgłosić) wbudowany obiekt `NotImplemented` (który działa tak, jakby metoda w ogóle nie była zdefiniowana). Operacje nieobsługiwane dla żadnych typów operandów powinny pozostać niezdefiniowane.

Przykładowe role dla operatorów w typach wbudowanych można znaleźć w tabeli 1., chociaż znaczenie operatorów jest określone w klasach przeciążających. Na przykład metoda `__add__` jest wywoływana w przypadku użycia operatora `+` zarówno dla operacji dodawania liczb, jak i konkatencji sekwencji, ale może mieć dowolną semantykę w nowych klasach.

Podstawowe metody działań dwuargumentowych

- `__add__(self, inny)`
Wywoływana podczas odwołań `self + inny`.
- `__sub__(self, inny)`
Wywoływana podczas odwołań `self - inny`.
- `__mul__(self, inny)`
Wywoływana podczas odwołań `self * inny`.
- `__truediv__(self, inny)`
W Pythonie 3.X wywoływana podczas odwołań `self / inny`. W Pythonie 2.X operator `/` wywołuje metodę `__div__`, pod warunkiem że włączono obsługę dzielenia rzeczywistego (patrz podrozdział „Uwagi na temat stosowania operatorów”).

`__floordiv__(self, inny)`
Wywoływana podczas odwołań `self // inny`.

`__mod__(self, inny)`
Wywoływana podczas odwołań `self % inny`.

`__divmod__(self, inny)`
Wywoływana podczas odwołań `divmod(self, inny)`.

`__pow__(self, inny [, modulo])`
Wywoływana podczas odwołań `pow(self, inny [, modulo])` oraz `self ** inny`.

`__lshift__(self, inny)`
Wywoływana podczas odwołań `self << inny`.

`__rshift__(self, inny)`
Wywoływana podczas odwołań `self >> inny`.

`__and__(self, inny)`
Wywoływana podczas odwołań `self & inny`.

`__xor__(self, inny)`
Wywoływana podczas odwołań `self ^ inny`.

`__or__(self, inny)`
Wywoływana podczas odwołań `self | inny`.

Prawostronne metody działań dwuargumentowych

`__radd__(self, inny)`
`__rsub__(self, inny)`
`__rmul__(self, inny)`
`__rtruediv__(self, inny)`
`__rfloordiv__(self, inny)`
`__rmod__(self, inny)`
`__rdivmod__(self, inny)`
`__rpow__(self, inny)`
`__rlshift__(self, inny)`
`__rrshift__(self, inny)`
`__rand__(self, inny)`
`__rxor__(self, inny)`
`__ror__(self, inny)`

Są to prawostronne odpowiedniki operatorów dwuargumentowych, opisanych w poprzednim podrozdziale. Metody operatorów dwuargumentowych mają odmiany prawostronne. Ich nazwy rozpoczynają się od prefiksu `r` (np. `__add__` i `__radd__`). Odmiany prawostronne mają takie same listy argumentów, ale argument

self występuje po prawej stronie operatora. Na przykład działanie *self* + *inny* wywołuje metodę *self.__add__(inny)*, natomiast *inny* + *self* wywołuje metodę *self.__radd__(inny)*.

Metody prawostronne (r) są wywoływane tylko wtedy, kiedy egzemplarz klasy znajduje się po prawej stronie, a lewy operand nie jest egzemplarzem klasy, która implementuje działanie:

- *egzemplarz* + *inny obiekt* uruchamia metodę *__add__*;
- *egzemplarz* + *egzemplarz* uruchamia metodę *__add__*;
- *inny obiekt* + *egzemplarz* uruchamia metodę *__radd__*.

Jeśli w działaniu występują obiekty dwóch klas przeciążających działanie, to preferowana jest klasa argumentu występującego po lewej stronie. Metoda *__radd__* często jest implementowana w ten sposób, że zamienia kolejność operandów i wywołuje metodę *__add__*.

Metody działań dwuargumentowych z aktualizacją w miejscu

```
__iadd__(self, inny)  
__isub__(self, inny)  
__imul__(self, inny)  
__itruediv__(self, inny)  
__ifloordiv__(self, inny)  
__imod__(self, inny)  
__ipow__(self, inny [, modulo])  
__ilshift__(self, inny)  
__irshift__(self, inny)  
__iand__(self, inny)  
__ixor__(self, inny)  
__ior__(self, inny)
```

Są to metody przypisania z aktualizacją (w miejscu). Są wywoływane odpowiednio dla następujących formatów instrukcji przypisania: +=, -=, *=, /=, //=, %=, **=, <=<=, >=>=, &=, ^= i |=. Metody te powinny podejmować próbę wykonania działania w miejscu (z modyfikacją egzemplarza *self*) i zwracać wynik (którym może być egzemplarz *self*). Jeśli metoda jest niezdefiniowana, to działanie z aktualizacją sięga do zwykłych metod. W celu obliczenia wartości wyrażenia *X* += *Y*, gdzie *X* jest egzemplarzem klasy ze zdefiniowaną metodą *__iadd__*, wywoływana jest metoda *X.__iadd__(Y)*. W przeciwnym razie wykorzystywane są metody *__add__* i *__radd__*.

Liczby (inne działania)

- `__neg__(self)`
Wywoływana przy wyznaczaniu wartości wyrażenia `-self`.
- `__pos__(self)`
Wywoływana przy wyznaczaniu wartości wyrażenia `+self`.
- `__abs__(self)`
Wywoływana przy uruchomieniu funkcji `abs(self)`.
- `__invert__(self)`
Wywoływana przy wyznaczaniu wartości wyrażenia `~self`.
- `__complex__(self)`
Wywoływana przy uruchomieniu funkcji `complex(self)`.
- `__int__(self)`
Wywoływana przy uruchomieniu funkcji `int(self)`.
- `__float__(self)`
Wywoływana przy uruchomieniu funkcji `float(self)`.
- `__round__(self [, n])`
Wywoływana przy uruchomieniu funkcji `round(self [, n])`.
Nowość w Pythonie 3.X.
- `__index__(self)`
Wywoływana w celu implementacji funkcji `operator.index()`.
Wywoływana także w innych kontekstach, w których Python wymaga obiektu `integer`. Obejmuje to takie wystąpienia egzemplarzy jak indeksy, granice segmentów oraz argumenty do wbudowanych funkcji `bin()`, `hex()` i `oct()`. Musi zwracać liczbę całkowitą.

Działa podobnie w Pythonie 3.X i 2.X, ale nie jest wywoływana w Pythonie 2.X dla funkcji `hex()` i `oct()` (te funkcje w Pythonie 2.X wymagają metod `__hex__` i `__oct__`). W Pythonie 3.X `__index__` obejmuje i zastępuje metody `__oct__` i `__hex__` z Pythona 2.X, a zwrócona liczba `integer` jest formatowana automatycznie.

Deskryptory

Metody opisane w tym podrozdziale stosuje się tylko wtedy, kiedy egzemplarz klasy definiujący metodę (*klasa-deskryptor*) zostanie przypisany do atrybutu innej klasy (znanej jako *klasa-właściciel*). Metody te są wywoływane w celu uzyskania dostępu do atrybutu wewnątrz klasy-właściciela oraz jej egzemplarzy.

`__get__(self, egzemplarz, właściciel)`

Wywoływana w celu uzyskania atrybutu klasy *właściciel* lub egzemplarza tej klasy. Argument *właściciel* zawsze oznacza klasę właściciela, a argument *egzemplarz* — egzemplarz, do którego atrybutu sięgamy. Może mieć wartość `None`, kiedy żądamy dostępu do atrybutu bezpośrednio poprzez klasę właściciela. Atrybut *self* oznacza egzemplarz klasy deskryptora. Zwraca wartość atrybutu albo zgłasza wyjątek `AttributeError`. Zarówno argument *self*, jak i *egzemplarz* mogą zawierać informacje o stanie.

`__set__(self, egzemplarz, wartość)`

Metoda wywoływana w celu ustawienia atrybutu egzemplarza klasy właściciela na nową wartość.

`__delete__(self, egzemplarz)`

Metoda wywoływana w celu usunięcia atrybutu egzemplarza klasy właściciela.

Deskryptory i ich metody są dostępne dla klas *nowego stylu*, do których zalicza się wszystkie klasy w Pythonie 3.X. W wersji 2.X działają w pełni tylko wtedy, gdy zarówno klasa deskryptora, jak i właściciela są klasami nowego stylu. Deskryptor zawierający metodę `__set__` to tzw. *deskryptor danych*. Deskryptor tego rodzaju ma pierwszeństwo w stosunku do innych nazw w operacji dziedziczenia (patrz „Formalne reguły dziedziczenia”).

UWAGA

Określenie „deskryptor” w znaczeniu używanym w tym rozdziale różni się od „deskryptorów plików” (opisanych w podrozdziałach „Pliki” oraz „Narzędzia do obsługi deskryptorów plików”).

Menedżery kontekstu

Poniższe metody implementują protokół menedżera kontekstu używany w instrukcji `with` (zobacz „Instrukcja `with`”).

`__enter__(self)`

Wejście do kontekstu runtime związanego z tym obiektem. Instrukcja `with` przypisuje wartość zwracaną przez tę metodę do obiektu docelowego, określonego w klauzuli `as` instrukcji (o ile taka klauzula istnieje).

`__exit__(self, typ_wyjątku, wartość_wyjątku, ślad)`

Wyjście z kontekstu runtime związanego z tym obiektem. Parametry za argumentem *self* opisują wyjątek, który spowodował wyjście z kontekstu. Jeśli wyjście z kontekstu nastąpiło bez zgłaszania wyjątku, wszystkie trzy argumenty mają wartość `None`. W przeciwnym razie argumenty mają taką samą wartość jak wyniki funkcji `sys.exc_info()` (patrz „Moduł `sys`”).

Zwrócenie wartości `true` przeciwdziała propagacji zgłoszonego wyjątku przez obiekt wywołujący.

Metody przeciążające operatory w Pythonie 2.X

W poprzednim podrozdziale wyszczególniono różnice semantyczne pomiędzy metodami przeciążania operatorów dostępnymi *zarówno* w Pythonie 3.X, jak i 2.X. Tu opisano różnice w treści występujące pomiędzy dwoma liniami wersji Pythona.

Niektóre z metod opisanych w poprzednim podrozdziale działają w Pythonie 2.X tylko dla *klas nowego stylu* — opcjonalnego rozszerzenia dla Pythona linii 2.X. Do tej grupy można zaliczyć metody `__getattr__`, `__slots__` oraz metody deskryptorów. Inne metody mogą zachowywać się inaczej w Pythonie 2.X dla klas nowego stylu (np. metoda `__getattr__` dla funkcji wbudowanych). Niektóre metody są dostępne tylko w późniejszych wydaniach linii 2.X (np. `__dir__`, `__instancecheck__`, `__subclasscheck__`). Poniżej zestawiono metody uniikatowe dla każdej z linii Pythona.

Metody dostępne wyłącznie w Pythonie 3.X

Poniższe metody są dostępne w Pythonie 3.X, ale nie w Pythonie 2.X:

- `__round__`,
- `__bytes__`,
- `__bool__` (w Pythonie 2.X należy wykorzystać metody `__nonzero__` lub `__len__`),
- `__next__` (w Pythonie 2.X należy wykorzystać metodę o nazwie `next`),
- `__truediv__` (dostępna w Pythonie 2.X tylko wtedy, gdy włączono opcję rzeczywistego dzielenia — patrz „Uwagi na temat stosowania operatorów”),

- `__index__` dla funkcji `oct()`, `hex()` (w Pythonie 2.X należy użyć metod `__oct__` i `__hex__`).

Metody dostępne wyłącznie w Pythonie 2.X

Poniższe metody są dostępne w Pythonie 2.X, ale nie w Pythonie 3.X:

`__cmp__(self, inny)` (oraz `__rcmp__`)

Wywoływana w momencie odwołania `self > inny`, `inny == self`, `cmp(self, inny)` itp. Metoda jest wywoływana dla wszystkich porównań, dla których nie zdefiniowano innych specyficznych metod (takich jak `__lt__`) lub jeśli nie zostały one odziedziczone. Metoda zwraca wartości odpowiednio -1, 0 bądź 1 dla przypadków, gdy egzemplarz `self` jest mniejszy, równy bądź większy od egzemplarza `inny`. Jeśli nie zdefiniowano metod porównania lub metody `__cmp__`, egzemplarze klas są porównywane według swojej tożsamości (adresu w pamięci). Od Pythona w wersji 2.1 prawostronna metoda `__rcmp__` nie jest obsługiwana.

W Pythonie 3.X należy wykorzystać bardziej specyficzne metody porównań, opisane wcześniej: `__lt__`, `__ge__`, `__eq__` itp. Do sortowania należy zaś wykorzystać metodę `__lt__`.

`__nonzero__(self)`

Wywoływana przy odwołaniach do wartości `true` (w przeciwnym razie wykorzystywana jest metoda `__len__`, o ile ją zdefiniowano).

W Pythonie 3.X zmieniono nazwę tej metody na `__bool__`.

`__getslice__(self, dolny, górny)`

Wywoływana podczas odwołań `self[dolny:górny]` do realizacji wycinania sekwencji. Jeśli metoda `__getslice__` nie zostanie znaleziona (oraz w przypadku tworzenia rozszerzonych wycinków trójelementowych), obiekt *wycinka* jest tworzony i przekazywany do metody `__getitem__`.

W Pythonie 2.X ta oraz dwie kolejne metody nie są zalecane, ale w dalszym ciągu są dostępne — jeśli je zdefiniowano, są wywoływane w wyrażeniach z wycinkami i mają pierwszeństwo w stosunku do swoich odpowiedników bazujących na elementach wycinka. W Pythonie 3.X te trzy metody całkowicie usunięto — wycinki zawsze wywołują metody `__getitem__`, `__setitem__` bądź `__delitem__` z argumentem w postaci obiektu wycinka. Więcej informacji na ten temat można znaleźć w opisie funkcji `slice()` w podrozdziale „Funkcje wbudowane”.

`__setslice__(self, dolny, górny, wartość)`

Wywoływana podczas odwołań `self[dolny:górny]=wartość` w celu przypisania sekwencji wycinka.

Zobacz wcześniejszą uwagę o wycofywaniu metody `__getitem__`.

`__delslice__(self, dolny, górny)`

Wywoływana podczas odwołań `self[dolny:górny]` do usuwania sekwencji wycinka.

Zobacz wcześniejszą uwagę o wycofywaniu metody `__getitem__`.

`__div__(self, inny)` (oraz `__rdiv__`, `__idiv__`)

Wywoływana przy odwołaniach `self / inny`, o ile nie zostanie włączona operacja rzeczywistego dzielenia za pomocą instrukcji `from` (w takim przypadku wykorzystywana jest metoda `__true__div__`). W Pythonie 3.X wymienione metody zostały zastąpione metodami `__truediv__`, `__rtruediv__` oraz `__itruediv__`, ponieważ operator `/` zawsze oznacza rzeczywiste dzielenie. Patrz „Uwagi na temat stosowania operatorów”. Wskazówka: aby obsługiwać oba modele w jednej metodzie, należy skorzystać z przypisania `__truediv__ = __div__`.

`__long__(self)`

Wywoływana przy uruchomieniu funkcji `long(self)`. W Pythonie 3.X typ `int` całkowicie obejmuje typ `long`, dlatego tę metodę usunięto.

`__oct__(self)`

Wywoływana przy uruchomieniu funkcji `oct(self)`. Metoda zwraca tekstową reprezentację liczby ósemkowej. W Pythonie 3.X zamiast niej należy zwrócić liczbę całkowitą dla metody `__index__()`.

`__hex__(self)`

Wywoływana przy uruchomieniu funkcji `hex(self)`. Metoda zwraca tekstową reprezentację liczby szesnastkowej. W Pythonie 3.X zamiast niej należy zwrócić liczbę całkowitą dla metody `__index__()`.

`__coerce__(self, inny)`

Wywoływana podczas wartościowania mieszanych wyrażeń arytmetycznych `coerce()`. Metoda zwraca krotkę `(self, inny)` przekonwertowaną na wspólny typ. Jeśli zdefiniowano metodę `__coerce__`, to jest ona wywoływana, zanim zostaną wywołane metody „właściwych” operatorów (np. przed metodą `__add__`). Metoda powinna zwracać krotkę zawierającą operandy przekonwertowane na wspólny typ danych (lub wartości `None`, jeśli kon-

wersja jest niemożliwa). Więcej informacji na temat reguł koercji można znaleźć w podręczniku *Python Language Reference*.

`__unicode__(self)`

Metoda wywoływana w Pythonie 2.X przez funkcję `unicode(self)` w celu zwrócenia łańcucha Unicode (patrz „Funkcje wbudowane Pythona 2.X”). Metoda jest odpowiednikiem Unicode metody `__str__`.

`__metaclass__`

Atrybut klasy przekazany do metaklasy związanej z klasą. W Pythonie 3.X należy użyć składni argumentu kluczowego `metaclass = M` w wierszu nagłówka klasy (patrz „Metaklasy”).

Funkcje wbudowane

Wszystkie nazwy wbudowane (funkcji, wyjątków itd.) istnieją w domniemanym zewnętrznym zasięgu, odpowiadającym modułowi `builtins` (w Pythonie 2.X ten moduł nosi nazwę `__builtin__`). Ponieważ w operacjach wyszukiwania nazw ten zasięg jest zawsze przeszukiwany jako ostatni, funkcje te są dostępne w programach bez importowania. Ich nazwy nie są jednak słowami zarezerwowanymi i mogą być ukrywane w wyniku przypisania do tej samej nazwy w zasięgach globalnym i lokalnym. Aby uzyskać dodatkowe informacje dotyczące dowolnego z wywołań zestawionych w tym podrozdziale, można uruchomić polecenie `help(funkcja)`.

W tym podrozdziale skoncentrowano się na Pythonie 3.X, ale podano szczegóły dotyczące funkcji wbudowanych wspólnych dla większości wersji Pythona. Informacje specyficzne dla Pythona 2.X można znaleźć w następnym podrozdziale („Funkcje wbudowane w Pythonie 2.X”).

`abs(N)`

Zwraca wartość bezwzględną liczby *N*.

`all(obiekt_iterowalny)`

Zwraca `True` tylko wtedy, gdy wszystkie elementy *objektu_ite*rowalnego mają wartość `true`.

`any(obiekt_iterowalny)`

Zwraca `True` tylko wtedy, gdy dowolny element *objektu_itero*walnego ma wartość `true`. Wskazówka: wyrażenia `filter(bool, I)` oraz `[x for x in I if x]` pobierają wszystkie elementy o wartości `true` w obiekcie iterowalnym *I*.

`ascii(obiekt)`

Działa podobnie jak funkcja `repr()` — zwraca łańcuch znaków zawierający drukowalną reprezentację obiektu. Znaki spoza kodu ASCII w wynikowym łańcuchu znaków są poprzedzane sekwencjami specjalnymi `\x`, `\u` albo `\U`. Wynik działania funkcji jest podobny do wyniku działania funkcji `repr()` w Pythonie 2.X.

`bin(N)`

Konwertuje liczbę całkowitą na tekstową postać liczby dwójkowej. Wynik jest prawidłowym wyrażeniem Pythona. Jeśli argument *N* nie jest obiektem `int` Pythona, musi on definiować metodę `__index__()`, która zwraca liczbę całkowitą. Wskazówka: patrz także `int(string, 2)` w celu konwertowania literałów binarnych `0bNNN` w kodzie oraz kod typu `b` w funkcji `str.format()`.

`bool([X])`

Zwraca wartość `Boolean` obiektu *X*, wykorzystując standardową procedurę sprawdzania wartości logicznych. Jeśli argument *X* ma wartość `false` lub zostanie pominięty, funkcja zwraca `False`, a w przeciwnym razie zwraca `True`. Istnieje także klasa `bool` będąca klasą podrzędną klasy `int`. Klasa `bool` nie może mieć dalszych klas podrzędnych. Jej jedynymi egzemplarzami są obiekty `False` i `True`.

`bytearray([arg [, kodowanie [, błędy]]])`

Zwraca nową tablicę bajtów. Typ `bytearray` to mutowalna sekwencja liczb typu `small integer` z zakresu `0...255`, które (jeśli to możliwe) wyświetlają się w postaci tekstu ASCII. W zasadzie jest to mutowalna odmiana typu `bytes` obsługującego większość działań na sekwencjach mutowalnych, a także większość metod typu łańcuchowego `str`. Argument *arg* może być: łańcuchem typu `str` z podanym *kodowaniem* (i opcjonalnie *błędami*), tak jak w przypadku funkcji `str()` (opisanej później na tej liście); liczbą całkowitą, oznaczającą rozmiar tablicy do zainicjowania wartościami `NULL`; obiektem iterowalnym, złożonym z wartości `small integer`, służącym do zainicjowania tablicy — na przykład łańcucha `bytes` lub innych danych `bytearray`; obiektem implementującym interfejs przeglądania pamięci (wcześniej znanym jako bufor), używanym do zainicjowania tablicy. W przypadku braku argumentu *arg* tworzona jest tablica zerowej długości. Więcej informacji można znaleźć w podrozdziale „Metody typów `bytes` i `bytearray`”.

`bytes([arg [, kodowanie [, błędy]]])`

Zwraca nowy obiekt `bytes` będący niemutowalną sekwencją liczb typu `integer` z zakresu `0...255`. Typ `bytes` jest niemutowalną wersją typu `bytearray`. Oba typy obsługują te same niezmienniające

metody tekstowe oraz takie same operacje na sekwencjach. Typ `bytes` jest powszechnie używany do reprezentowania łańcuchów bajtowych danych binarnych, złożonych z 8-bitowych bajtów. Argumenty konstruktora są interpretowane w taki sam sposób jak dla `bytearray()`. Obiekty `bytes` można również tworzyć za pomocą literału `b'ccc'`. Więcej informacji można znaleźć w podrödziale „Metody typów `bytes` i `bytearray`”.

`callable(obiekt)`

Zwraca `True`, jeśli *obiekt* jest wywoływalny; w przeciwnym razie zwraca `False`. Ta funkcja jest dostępna w Pythonie 2.X. W Pythonie 3.X usunięto ją z wydań 3.0 i 3.1, ale przywrócono w Pythonie 3.2. We wcześniejszych wersjach linii 3.X należy zamiast niej używać wywołania `hasattr(obiekt, '__call__')`.

`chr(I)`

Zwraca jednoznakowy łańcuch znaków, którego kod Unicode ma wartość *I*. Funkcja działa odwrotnie do funkcji `ord()` (tzn. `chr(97)` zwraca `'a'`, a `ord('a')` zwraca `97`).

`classmethod(funkcja)`

Zwraca metodę klasy odpowiadającą *funkcji*. Metoda klasy otrzymuje niejawnie klasę jako pierwszy argument, podobnie jak metoda egzemplarza otrzymuje egzemplarz. Przydatna do zarządzania danymi klasy. W Pythonie w wersji 2.4 i późniejszych można skorzystać z dekoratora `@classmethod` (patrz „Instrukcja `def`”).

`compile(łańcuchznaków, nazwapliku,`

`↳ rodzaj [, flagi [, zakaz_dziedziczenia]])`

Kompiluje *łańcuchznaków* na obiekt kodu. Argument *łańcuchznaków* to dowolna instrukcja Pythona; *nazwapliku* to ciąg używany w komunikatach o błędach (zwykle jest to nazwa pliku, z którego odczytano kod, lub `<łańcuchznaków>`, jeśli instrukcja została wpisana w trybie interaktywnym). *rodzaj* może mieć wartość `'exec'`, jeśli *łańcuchznaków* zawiera instrukcje, `'eval'`, jeśli jest to wyrażenie, lub `'single'`. W ostatnim przypadku instrukcja wyświetla wynik instrukcji wyrażeniowej przyjmującej wartość inną niż `None`. Zwrócony obiekt kodu może zostać uruchomiony za pomocą wywołań wbudowanych funkcji `exec()` lub `eval()`. Ostatnie dwa opcjonalne argumenty zarządzają tym, które z instrukcji planowanych do wprowadzenia w przyszłych wersjach (oznaczonych jako `future`) będą uwzględniane w kompilacji. Jeśli argumentów tych nie podano, kod zostanie skompilowany z ustawieniami występującymi w momencie wywołania funkcji `compile()` (więcej informacji na ten temat można znaleźć w podröczniku Pythona).

`complex([rzeczywista [, urojona]])`

Tworzy obiekt liczby zespolonej (do tego celu można również wykorzystać przyrostek `J` lub `j`: `rzeczywista+urojonaJ`). Domyślnie argument `urojona` ma wartość 0. W przypadku pominięcia obu argumentów funkcja zwraca `0j`.

`delattr(obiekt, nazwa)`

Usuwa atrybut `nazwa` (łańcuch znaków) z *obiekту*. Funkcja ta ma działanie podobne jak instrukcja `del obiekt.nazwa`, ale w przypadku funkcji `nazwa` jest łańcuchem znaków, a nie zmienną (np. wywołanie `delattr(a, 'b')` jest równoważne instrukcji `del a.b`).

`dict([odwzorowanie | obiekt_iterowalny | słowa_kluczowe])`

Zwraca nowy słownik zainicjowany za pomocą odwzorowania, sekwencji (lub innego obiektu iterowalnego) złożonej z par klucz-wartość albo zbioru argumentów kluczowych. W przypadku braku argumentów zwraca pusty słownik — nazwę klasy, na podstawie której można tworzyć klasy podrzędne.

`dir([obiekt])`

Funkcja wywołana bez argumentów zwraca listę nazw dostępnych w bieżącym zasięgu lokalnym (przestrzeni nazw). W przypadku podania argumentu w postaci dowolnego *obiekту* z atrybutami zwraca listę nazw atrybutów powiązanych z tym *obiektem*. Działa na modułach, klasach i egzemplarzach klas, a także na obiektach wbudowanych z atrybutami (listy, słowniki itp.). Wynik funkcji uwzględnia dziedziczone atrybuty i jest posortowany. W celu uzyskania prostej listy atrybutów pojedynczego obiektu można skorzystać z metody `__dict__`. Wywołanie to uruchamia niestandardową metodę `obiekt.__dir__()` (o ile ją zdefiniowano). Może ona dostarczać nazwy wyliczonych atrybutów w klasach dynamicznych lub w klasach proxy.

`divmod(X, Y)`

Zwraca krotkę $(X / Y, X \% Y)$.

`enumerate(obiekt_iterowalny, start=0)`

Zwraca obiekt wyliczenia `enumerate`. Argument `obiekt_iterowalny` powinien być sekwencją albo innym obiektem obsługującym protokół iteracji. Metoda `__next__()` iteratora zwróconego przez funkcję `enumerate()` zwraca krotkę zawierającą *licznik* (od wartości `start` lub domyślnie od zera) oraz odpowiadającą mu *wartość*, uzyskaną z iteracji po obiekcie `obiekt_iterowalny`. Wywołanie jest przydatne do wyznaczania poindeksowanych ciągów, w przypadku gdy w pętlach `for` są wymagane zarówno pozycje, jak i elementy: $(0, x[0])$, $(1, x[1])$, $(2, x[2])$ itd.

Funkcja jest dostępna w Pythonie w wersji 2.3 i w wersjach późniejszych. Więcej informacji na temat reguł stałych enumeracji w Pythonie 3.4 można znaleźć w podrozdziale „Moduł enum”.

`eval(wyrazenie [, globalny [, lokalny]])`

Oblicza wartość *wyrazenia*, które może być łańcuchem znaków zawierającym wyrażenie Pythona albo skompilowanym obiektem kodu. Jeśli nie zostaną przekazane argumenty oznaczające słowniki przestrzeni nazw *globalny* lub *lokalny*, to *wyrazenie* jest obliczane w zasięgach przestrzeni nazw wywołania `eval`. Jeśli zostanie przekazany tylko argument *globalny*, to argument *lokalny* domyślnie przyjmie jego wartość. Funkcja zwraca wartość *wyrazenia*. Informacje na temat instrukcji wywoływanych dynamicznie można znaleźć również w opisie funkcji `compile()` (wcześniej w tym podrozdziale) oraz funkcji `exec()`. Wskazówka: nie należy używać tej instrukcji do sprawdzania niezauważalnych ciągów kodu, ponieważ są one uruchamiane tak samo jak kod programu.

`exec(instrukcje [, globalny [, lokalny]])`

Uruchamia *instrukcje*. Argument ten może być łańcuchem znaków zawierającym instrukcje Pythona albo skompilowanym obiektem kodu. Jeśli argument *instrukcje* jest łańcuchem znaków, to jest on parsowany jako blok instrukcji Pythona, a następnie uruchamiany, jeśli nie ma w nim błędów składniowych. Jeśli jest to obiekt kodu, to jest on uruchamiany. Argumenty *globalny* i *lokalny* mają takie samo znaczenie jak w funkcjach `eval()` oraz `compile()` i mogą być używane do prekompilacji obiektów kodu. W Pythonie 2.X działanie to jest dostępne w formie instrukcji (patrz podrozdział „Instrukcje w Pythonie 2.X”). W historii Pythona operacja ta przekształcała się pomiędzy formami funkcji i instrukcji więcej niż jeden raz. Wskazówka: nie należy używać tej instrukcji do sprawdzania niezauważalnych ciągów kodu, ponieważ są one uruchamiane tak samo jak kod programu.

`filter(funkcja, obiekt_iterowalny)`

Zwraca te elementy argumentu *obiekt_iterowalny*, dla których *funkcja* zwraca `true`. Funkcja pobiera jeden parametr. Jeśli *funkcja* ma wartość `None`, funkcja `filter` zwraca wszystkie elementy w *obiekcie_iterowalnym*, które mają wartość `true`. Taki sam efekt można uzyskać, jeśli prześlemy do *funkcji* wbudowany typ `bool`.

W Pythonie 2.X to wywołanie zwraca *listę*. W Pythonie 3.X zwraca *obiekt_iterowalny* generujący wartości na żądanie, który można przejrzeć tylko raz (jeśli występuje potrzeba generowania wyników, należy opakować wywołanie funkcji `list()`).

`float([x])`

Konwertuje liczbę lub ciąg znaków *x* na liczbę zmiennoprzecinkową (lub wartość 0.0, jeśli nie podano argumentu). Przykłady użycia można także znaleźć w podrozdziale „Liczby”. Jest to nazwa klasy, na podstawie której można tworzyć klasy podrzędne.

`format(wartość [, specyfikacjaformatu])`

Konwertuje obiekt *wartość* na jego sformatowaną reprezentację, zgodną z tekstowym argumentem *specyfikacjaformatu*. Interpretacja argumentu *specyfikacjaformatu* zależy od typu argumentu *wartość*. Większość typów wbudowanych używa standardowej składni formatowania (opisano ją przy okazji omawiania metod formatowania tekstu, we wcześniejszej części książki — patrz opis argumentu *specyfikacjaformatu* w podrozdziale „Składnia metody formatującej”). Wywołanie `format(wartość, specyfikacjaformatu)` wywołuje metodę `wartość.__format__(specyfikacjaformatu)` i jest bazową operacją metody `str.format()`, np. odpowiednikiem wywołania `format(1.3333, '.2f')` jest `'{0:.2f}'.format(1.3333)`.

`frozenset([obiekt_iterowalny])`

Zwraca obiekt „zamrożonego” zbioru, którego elementy pochodzą z obiektu iterowalnego. Zamrożone zbiory są niemutowalne. Nie mają metod aktualizacji i mogą być zagnieżdżane w innych zbiorach.

`getattr(obiekt, nazwa [, domyślna])`

Zwraca wartość atrybutu *nazwa* (łańcucha znaków) z obiektu. Funkcja ta ma podobne działanie do wywołania `obiekt.nazwa`, ale w przypadku funkcji `getattr nazwa` jest łańcuchem znaków, a nie zmienną (np. wywołanie `getattr(a, 'b')` jest równoważne wywołaniu `a.b`). Jeśli atrybut o podanej nazwie nie istnieje, funkcja zwraca wartość *domyślna*, o ile ten argument podano. W przeciwnym razie następuje zgłoszenie wyjątku `AttributeError`.

`globals()`

Zwraca słownik zawierający zmienne globalne obiektu wywołującego (np. nazwy z modułu okalającego).

`hasattr(obiekt, nazwa)`

Zwraca `True`, jeśli *obiekt* posiada atrybut *nazwa* (łańcuch znaków); w przeciwnym razie zwraca `False`.

`hash(obiekt)`

Zwraca wartość skrótu dla obiektu *obiekt* (o ile taka wartość istnieje). Wartości skrótów to liczby całkowite używane do szybkiego porównywania kluczy słownika podczas wyszukiwania w słownikach. Wywołuje metodę `obiekt.__hash__()`.

`help([obiekt])`

Wywołuje wbudowany system pomocy (ta funkcja jest przeznaczona do wykorzystania w trybie interaktywnym). W przypadku braku argumentu interaktywna sesja pomocy uruchamia konsolę interpretera. Jeśli argument jest łańcuchem znaków, jest on interpretowany jako nazwa modułu, funkcji, klasy, metody, słowa kluczowego lub tematu dokumentacji, a funkcja wyświetla tekst pomocy. Jeśli argument jest dowolnym obiektem innego rodzaju, generowana jest pomoc dla tego obiektu (np. `help(list.pop)`).

`hex(N)`

Konwertuje liczbę całkowitą *N* na tekstową postać liczby szesnastkowej. Jeśli argument *N* nie jest obiektem `int` Pythona, musi on definiować metodę `__index__()`, która zwraca liczbę całkowitą (w Pythonie 2.X zamiast tej metody wywoływana jest metoda `__hex__()`).

`id(obiekt)`

Zwraca identyfikator *obiektu* — unikatową dla procesu wywołującego liczbę całkowitą identyfikującą obiekt spośród wszystkich obiektów (jest to adres obiektu w pamięci).

`__import__(nazwa, [...inne_argumenty...])`

W fazie wykonywania programu importuje i zwraca moduł na podstawie jego *nazwy* (np. `mod = __import__('mymod')`). To wywołanie jest, ogólnie rzecz biorąc, szybsze od konstruowania i uruchamiania ciągu instrukcji `import` za pomocą instrukcji `exec()`. Ta funkcja jest wywoływana wewnętrznie przez instrukcje `import` i `from`. Można ją przesłonić w celu personalizacji operacji importowania. Wszystkie argumenty z wyjątkiem pierwszego mają zaawansowane role (ang. *advanced roles*) (patrz podręcznik *Python Library Reference*). Warto również zapoznać się ze standardowym modulem bibliotecznym `imp` oraz z opisem wywołania `importlib`. ↪ `import_module()`, a także z podrozdziałem „Instrukcja `import`”.

`input([symbolzachęty])`

Wyświetla *symbolzachęty*, jeśli ten argument zostanie przekazany, a następnie czyta wiersz ze standardowego strumienia wejściowego *stdin* (`sys.stdin`) i zwraca go w formie łańcucha znaków. Obcina końcową sekwencję `\n` na końcu wiersza oraz zgłasza wyjątek `EOFError` w przypadku osiągnięcia końca strumienia *stdin*. Funkcja `input()` wykorzystuje mechanizm GNU `readline` na platformach, które go obsługują. W Pythonie 2.X ta funkcja ma nazwę `raw_input()`.

`int([liczba | łańcuch znaków [, podstawa]])`

Przekształca liczbę lub łańcuch znaków na liczbę całkowitą. Konwersja liczb zmiennoprzecinkowych na całkowite powoduje obcinanie w kierunku 0. Argument *podstawa* można przekazać tylko wtedy, gdy pierwszy argument jest łańcuchem znaków. Jego wartość domyślna wynosi 10. Jeśli argument *podstawa* zostanie przekazany jako 0, to jego wartość będzie wyznaczona na podstawie zawartości łańcucha znaków; w przeciwnym razie jako podstawę konwersji wykorzystuje się wartość przekazaną za pomocą argumentu *podstawa*. Argument *podstawa* może przyjmować wartości 0 oraz z zakresu 2...36. Łańcuch znaków określający dane do konwersji może być poprzedzony znakiem i otoczony znakami białych spacji. Jeśli do funkcji nie zostaną przekazane żadne parametry, to zwraca ona 0. Przykłady użycia można znaleźć w podrozdziale „Liczby”. Jest to nazwa klasy, na podstawie której można tworzyć klasy podrzędne.

`isinstance(obiekt, informacjeklasie)`

Zwraca `True`, jeśli *obiekt* jest egzemplarzem klasy *informacjeklasie* lub egzemplarzem jej dowolnej klasy podrzędnej. Argument *informacjeklasie* może również być krotką złożoną z klas i (lub) typów. W Pythonie 3.X wszystkie typy są klasami, zatem nie ma potrzeby korzystania ze specjalnego przypadku w odniesieniu do typów. W Pythonie 2.X drugi argument może również być obiektem typu. Dzięki temu funkcja ta może być przydatna jako alternatywne narzędzie do sprawdzania typów (`isinstance(X, Typ)` zamiast `type(X) is Typ`).

`issubclass(klasa1, klasa2)`

Zwraca `true`, jeśli *klasa1* jest podklasą klasy *klasa2*. Argument *klasa2* może być również podany w postaci krotki złożonej z klas.

`iter(obiekt [, wartownik])`

Zwraca obiekt iteratora, który można wykorzystać do krokowego przeglądania elementów *obiekту*. Obiekty iteratorów zwrócone przez funkcję mają metodę `__next__()`, która zwraca następny element bądź zgłasza wyjątek `StopIteration` w celu zakończenia przeglądania. Ten protokół, o ile *obiekt* go wykorzystuje, jest używany w Pythonie we wszystkich kontekstach iteracyjnych do przechodzenia do następnego elementu. Wbudowana funkcja `next(I)` także automatycznie wywołuje metodę `I.__next__()`. Jeśli funkcja zostanie wywołana jako jednoargumentowa, to zakłada się, że *obiekt* powinien dostarczyć własny iterator bądź powinna to być sekwencja. W wersji dwuargumentowej pierwszy argument

jest obiektem wywołalnym, który jest wywoływany tak długo, aż zwróci wartość *wartownik*. Wywołanie funkcji `iter()` może być przeciążane w klasach implementujących metodę `__iter__()`.

W Pythonie 2.X obiekty iteratorów mają metody o nazwie `next()` zamiast `__next__()`. Aby była zapewniona zgodność w przód i wstecz, wbudowana funkcja `next()` jest także dostępna w wersji 2.X (począwszy od wersji 2.6). Funkcja ta wywołuje metodę `I.next()` zamiast `I.__next__()`. W wersjach wcześniejszych niż 2.6 zamiast tego można ręcznie wywołać metodę `I.next()`. Zobacz też opis funkcji `next()` na tej liście oraz podrozdział „Protokół iteracji”.

`len(obiekt)`

Zwraca liczbę elementów (długość) kolekcji *obiekt*, która może być sekwencją, odwzorowaniem, zbiorem lub inną konstrukcją (np. kolekcją zdefiniowaną przez użytkownika).

`list([obiekt_iterowalny])`

Zwraca nową listę składającą się ze wszystkich elementów dowolnego obiektu iterowalnego. Jeśli *obiekt_iterowalny* jest już listą, funkcja zwraca jej kopię. Jeśli do funkcji nie zostaną przekazane żadne argumenty, to zwraca ona nową, pustą listę. Jest to nazwa klasy, na podstawie której można tworzyć klasy podrzędne.

`locals()`

Zwraca słownik zawierający zmienne lokalne obiektu wywołującego (każda zmienna lokalna jest zwracana jako jedna pozycja *klucz:wartość*).

`map(funkcja, obiekt_iterowalny [, obiekt_iterowalny]*)`

Stosuje *funkcję* do każdego elementu dowolnej sekwencji lub innego obiektu iterowalnego i zwraca kolejne wyniki. Na przykład wywołanie `map(abs, (1, -2))` zwraca 1 i 2. Jeśli do funkcji zostanie przekazany dodatkowy obiekt iterowalny, to *funkcja* musi pobierać tyle argumentów, ile obiektów iterowalnych przekazano. W każdym wywołaniu do funkcji przekazywane są argumenty — po jednym z każdego obiektu iterowalnego. Iteracja kończy się wraz z końcem krótszego obiektu iterowalnego.

W Pythonie 2.X to wywołanie zwraca *listę* złożoną z indywidualnych wyników wywołań. W Pythonie 3.X zwraca *obiekt_iterowalny* generujący wartości na żądanie, który można przejrzeć tylko raz (jeśli występuje potrzeba generowania wyników, należy opakować wywołanie funkcji `list()`).

Poza tym w Pythonie 2.X (ale nie w Pythonie 3.X) w przypadku gdy *funkcja* ma wartość *None*, wywołanie *map()* pobiera do listy wyników wszystkie elementy. Jeśli do funkcji zostanie przekazanych wiele obiektów iterowalnych, to wynik funkcji zawiera elementy tych obiektów iterowalnych połączone w krotki, a wszystkie obiekty iterowalne są uzupełniane wartościami *None* do długości najdłuższego z nich. Podobne narzędzie jest dostępne w Pythonie 3.X w standardowym module bibliotecznym *itertools*.

```
max(obiekt_iterowalny [, argument]* [, klucz=funkcja])
```

W przypadku przekazania do funkcji pojedynczego argumentu *obiekt_iterowalny* funkcja zwraca największy element niepustego obiektu (np. łańcuch znaków, krotkę bądź listę). Jeśli zostanie przekazany więcej niż jeden argument, funkcja zwraca największą wartość spośród wszystkich argumentów. Opcjonalny argument *kluczowy* oznacza jednoargumentową funkcję przekształcania wartości, podobną do tej, która jest wykorzystywana w wywołaniach *list.sort()* oraz *sorted()* (zobacz podrozdziały „Listy” i „Funkcje wbudowane”).

```
memoryview(obiekt)
```

Zwraca obiekt widoku pamięci utworzony na podstawie przekazanego argumentu *obiekt*. Obiekty widoków pamięci umożliwiają uzyskanie dostępu z kodu Pythona do wewnętrznych danych obiektów obsługujących protokół, bez potrzeby ich kopiowania. Pamięć może być interpretowana w postaci prostego ciągu bajtów lub bardziej złożonych struktur danych. Do obiektów wbudowanych obsługujących protokół widoków pamięci należą typy *bytes* i *bytearray*. Więcej informacji na ten temat można znaleźć w dokumentacji Pythona. Widoki pamięci w głównej mierze zastępują protokół bufora i wbudowaną funkcję *buffer()*, występujące w Pythonie 2.X, chociaż w Pythonie 2.7 wprowadzono funkcję *memoryview()* w celu zapewnienia zgodności z wersją 3.X.

```
min(obiekt_iterowalny [, argument]* [, klucz=funkcja])
```

W przypadku przekazania do funkcji pojedynczego argumentu *obiekt_iterowalny* funkcja zwraca najmniejszy element niepustego obiektu (np. łańcuch znaków, krotkę bądź listę). Jeśli zostanie przekazany więcej niż jeden argument, funkcja zwraca najmniejszą wartość spośród wszystkich argumentów. Argument *klucz* ma takie samo znaczenie jak w przypadku funkcji *max()*.

```
next(iterator [, domyślne])
```

Pobiera następny element z *iteratora* poprzez wywołanie jego metody *__next__()* (w Pythonie 3.X). W przypadku gdy wyczerpie

się *iterator*, zwracana jest wartość *domyślnie*, o ile zostanie przekazana. W przeciwnym razie zgłaszany jest wyjątek `StopIteration`. Funkcja jest również dostępna w Pythonie 2.6 i 2.7, ale wywołuje metodę `iterator.next()` zamiast `iterator.__next__()`. Poma-ga to zapewnić zgodność w przód wersji 2.X z wersją 3.X oraz zgodność wstecz wersji 3.X z wersją 2.X. W Pythonie 2.X w wersjach wcześniejszych niż 2.6 takie wywołanie nie występuje. Zamiast niego należy ręcznie wywołać metodę `iterator.next()`. Zobacz też opis funkcji `iter()` na tej liście oraz podrozdział „Proto-kół iteracji”.

`object()`

Zwraca nowy obiekt. Typ `object` jest klasą bazową dla wszystkich klas nowego stylu. W Pythonie 2.X należą do nich wszystkie klasy jawnie dziedziczące własności z klasy `object` oraz wszystkie klasy w Pythonie 3.X. Zawiera niewielki zbiór domyślnych metod (patrz `dir(object)`).

`oct(N)`

Konwertuje liczbę całkowitą *N* na tekstową postać liczby ósemkowej. Jeśli argument *N* nie jest obiektem `int` Pythona, musi on definiować metodę `__index__()`, która zwraca liczbę całkowitą (w Pythonie 2.X zamiast tej metody wywoływana jest metoda `__oct__()`).

`open(...)`

```
open(plik
    [, tryb='r'
    [, buforowanie=-1
    [, kodowanie=None           # tylko dla trybu tekstowego
    [, błędy=None               # tylko dla trybu tekstowego
    [, znak_nowego_wiersza=None # tylko dla trybu tekstowego
    [, zamknij_deskr_pliku=True, # tylko deskryptory
    [, program_otwierający=None ]]]]]) # Niestandardowy program
                                         # otwierający — od wersji 3.3+
```

Zwraca nowy obiekt `file` połączony z zewnętrznym plikiem o nazwie *plik* bądź zgłasza wyjątek `IOError` (albo podklasę klasy `OSError`, począwszy od wersji 3.3), w przypadku gdy operacja otwierania się nie powiedzie. W tym podrozdziale opisano metodę `open()` Pythona 3.X. Sposób użycia dla wersji Pythona 2.X opisano w podrozdziale „Funkcje wbudowane Pythona 2.X”.

Argument *plik* ma zazwyczaj postać łańcucha znaków lub ciągu bajtów z nazwą pliku do otwarcia (a także ścieżką do pliku, o ile plik nie znajduje się w bieżącym katalogu roboczym). Argument *plik* może być również podany w postaci liczbowego deskryptora.

Jeśli zostanie podany deskryptor, plik jest zamykany, w przypadku gdy zwrócony obiekt wejścia-wyjścia zostanie zamknięty, chyba że argument *zamknij_deskr_pliku* jest ustawiony na `False`. Wszystkie opcje można przekazywać w postaci argumentów kluczowych.

tryb jest opcjonalnym łańcuchem znaków określającym tryb otwarcia pliku. Jego domyślną wartością jest `'r'`, co oznacza otwarcie pliku do czytania w trybie tekstowym. Inne powszechnie używane wartości to `'w'`, oznaczająca zapis (z zastępowaniem zawartości pliku, jeśli on już istnieje), oraz `'a'` w przypadku zapisu na końcu pliku. Jeśli w trybie tekstowym nie zostanie określony argument *kodowanie*, to będzie zastosowane kodowanie zależne od platformy, a znaki zakończenia wiersza zostaną domyślnie przekształcone na sekwencję `'\n'`. W przypadku czytania i zapisywania „surowych” bajtów należy zastosować tryby binarne `'rb'`, `'wb'` lub `'ab'` i zrezygnować z określania argumentu *kodowanie*.

Dostępne tryby można ze sobą łączyć: `'r'` oznacza odczyt (domyślnie); `'w'` — zapis z usunięciem poprzedniej zawartości pliku; `'a'` służy do zapisywania na końcu pliku, jeśli plik istnieje; `'b'` włącza tryb binarny; `'t'` oznacza tryb tekstowy (domyślnie); `'+'` otwiera plik dyskowy do aktualizacji (odczytu bądź zapisu); `'U'` włącza tryb uniwersalnego znaku końca wiersza (argument został dołączony tylko w celu zapewnienia zgodności wstecz). Domyślny tryb `'r'` działa tak samo jak `'rt'` (otwarcie pliku w celu odczytania tekstu). Losowy dostęp do plików binarnych gwarantują tryby `'w+b'`, który otwiera plik i obcina go do długości zera bajtów, oraz `'r+b'`, który otwiera plik bez obcinania.

Python rozróżnia pliki otwarte w trybach binarnym i tekstowym nawet wtedy, gdy nie robi tego wykorzystywany system operacyjny:

- W operacjach *wejściowych* pliki otwarte w trybie binarnym (poprzez dołączenie `'b'` do argumentu *tryb*) zwracają zawartość w postaci obiektów `bytes` bez wykorzystywania dekodowania Unicode i przekształcania znaków zakończenia wiersza. W trybie tekstowym (domyślnie lub po dołączeniu `'t'` do argumentu *tryb*) zawartość pliku jest zwracana w postaci łańcuchów znaków po zdekodowaniu bajtów z wykorzystaniem kodowania jawnie przekazanego za pomocą argumentu *kodowanie* albo wartości domyślnej zależnej od platformy. Z kolei konwersja znaków zakończenia wiersza jest realizowana na podstawie wartości argumentu *znak_nowego_wiersza*.

- W operacjach *wyjściowych* tryb binarny oczekuje danych typu bytes bądź bytearray i zapisuje je do pliku w niezmienionej postaci. Pliki otwarte w trybie tekstowym oczekują obiektów str. Są one kodowane zgodnie z wartością argumentu *kodowanie*, a przed zapisem jest stosowana konwersja znaku zakończenia wiersza zgodnie z wartością argumentu *znak_nowego_wiersza*.

Argument *buforowanie* to opcjonalna liczba całkowita, używana do ustawienia strategii buforowania. Domyślnie jest włączone pełne buforowanie. Aby je wyłączyć, należy przekazać 0 (właśność dozwolona wyłącznie w trybie binarnym). Wartość 1 włącza buforowanie liniowe, natomiast wartość > 1 — pełne buforowanie. Operacje transferu danych przy włączonym buforowaniu nie zawsze są realizowane natychmiast (aby wymusić transfer, należy użyć metody *plik.flush()*).

Argument *kodowanie* oznacza nazwę kodowania używaną do dekodowania bądź kodowania zawartości plików tekstowych podczas przesyłania. Argument powinien być wykorzystywany tylko w trybie tekstowym. Domyślne kodowanie jest ustawieniem zależnym od platformy, ale istnieje możliwość przekazania dowolnego kodowania obsługiwanego w Pythonie. Listę obsługiwanych typów kodowania można znaleźć w opisie modułu *codecs*.

błędy to opcjonalny łańcuch znaków określający sposób obsługi błędów kodowania. Argument powinien być wykorzystywany tylko w trybie tekstowym. Przekazanie wartości 'strict' (wartość domyślna dla None) spowoduje, że w przypadku wystąpienia błędu kodowania zostanie zgłoszony wyjątek *ValueError*. Przekazanie wartości 'ignore' spowoduje ignorowanie błędów (choć ignorowanie błędów kodowania może prowadzić do utraty danych). Wartość 'replace' umożliwia wykorzystanie znacznika zastępczego dla nieprawidłowych danych. Listę dozwolonych wartości argumentu można znaleźć w opisie funkcji *codecs.register_error()* w dokumentacji Pythona. Informacje o powiązanych narzędziach można znaleźć w opisie funkcji *str()* na tej liście.

Argument *znak_nowego_wiersza* zarządza sposobem wykorzystania uniwersalnych sekwencji przejścia do nowego wiersza. Ma zastosowanie tylko w trybie tekstowym. Argument może mieć wartość None (domyślnie), a także ' ', '\n', '\r' i '\r\n'.

- Jeśli argument *znak_nowego_wiersza* w operacjach *wejściowych* ma wartość `None`, to włączony jest tryb uniwersalnych znaków przejścia do nowego wiersza — wiersze mogą się kończyć sekwencjami `'\n'`, `'\r'` lub `'\r\n'` i wszystkie one będą przekształcone na `'\n'` przed zwróceniem do obiektu wywołującego. Jeśli argument *znak_nowego_wiersza* ma wartość `''`, to tryb uniwersalnych znaków przejścia do nowego wiersza jest włączony, ale znaki przejścia do nowego wiersza są przekazywane do obiektu wywołującego bez przekształcania. Jeśli argument ma dowolną inną z dozwolonych wartości, to wiersze wejściowe są zakańczane podanym ciągiem znaków, a znaki przejścia do nowego wiersza są zwracane do obiektu wywołującego bez przekształcania.
- Jeśli w operacjach *wyjściowych* argument *znak_nowego_wiersza* ma wartość `None`, to zapisywane sekwencje `'\n'` są konwertowane na domyślny separator wierszy systemu (`os.linesep`). W przypadku gdy argument ma wartość `''`, znaki przejścia do nowego wiersza nie są konwertowane. Jeśli argument ma dowolną inną spośród dozwolonych wartości, to wszystkie zapisywane sekwencje `'\n'` są przekształcane na podany łańcuch znaków.

Jeśli argument *zamknij_deskr_pliku* ma wartość `False`, to po zamknięciu pliku związany deskryptor pliku będzie w dalszym ciągu otwarty. Własność ta nie działa, gdy nazwa pliku zostanie podana w postaci łańcucha znaków. W takim przypadku argument musi mieć wartość `True` (co jest wartością domyślną). Jeśli zostanie przekazany argument *program_otwierający* oznaczający obiekt wywoływalny w Pythonie 3.3 lub wersjach późniejszych, to deskryptor pliku uzyskuje się za pomocą wywołania *program_otwierający(plik, flagi)* z argumentami takimi jak dla wywołania `os.open()` (patrz „Moduł systemowy os”).

Więcej informacji na temat interfejsu obiektów zwracanych przez wywołanie `open()` można znaleźć w podrozdziale „Pliki”. Wskazówka: każdy obiekt, który obsługuje interfejs metod obiektu `File`, może być używany w kontekstach, które oczekują pliku (patrz *socketobj.makefile()*, metody *io.StringIO(str)* i *io.BytesIO(bytes)* Pythona 3.X oraz *StringIO.stringIO(str)* Pythona 2.X — wszystkie ze standardowej biblioteki Pythona).

UWAGA

Ponieważ tryb pliku w Pythonie 3.X implikuje zarówno opcje konfiguracyjne, jak i tekstowe typy danych, warto pomyśleć o funkcji `open()` w kategoriach *dwóch różnych odmian* — tekstowej i binarnej, zgodnie z ustawieniami łańcucha opisującego tryb. W celu obsługi dwóch typów plików programiści Pythona, zamiast dostarczać dwie odrębne funkcje `open()`, zdecydowali się na przeciążanie pojedynczej funkcji za pomocą argumentów specyficznych dla trybu oraz zróżnicowanych typów zawartości. Biblioteka klas `io` — dla której w Pythonie 3.X funkcja `open()` jest frontonem — dostosowuje typy obiektów plików do podanego trybu. Więcej informacji na temat modułu `io` można znaleźć w dokumentacji Pythona. Moduł `io` jest również dostępny w Pythonie 2.X, począwszy od wydania 2.6, jako alternatywa dla wbudowanego typu `file`, ale stanowi on standardowy interfejs plikowy dla funkcji `open()` Pythona 3.X.

`ord(c)`

Zwraca kod jednoznakowego łańcucha znaków `c`. W przypadku znaków ASCII jest to 7-bitowy kod ASCII znaku `c`. W przypadku ciągu Unicode jest to kod Unicode jednoznakowego łańcucha Unicode. Zobacz także opis funkcji `chr()` na tej liście, która jest odwrotnością funkcji `ord()`.

`pow(x, y [, z])`

Zwraca wartość `x` podniesioną do potęgi `y` [modulo `z`]. Funkcja działa podobnie jak operator wyrażeniowy `**`.

`print(...)`

```
print([obiekt [, obiekt]*]  
      [, sep=' ' [, end='\n']  
      [, file=sys.stdout] [, flush=False])
```

Drukuje do strumienia `file` opcjonalny *obiekt* lub obiekty rozdzielone separatorami `sep` i zakończone sekwencją `end` z opcjonalnym wymuszeniem opróżnienia bufora (`flush`) po drukowaniu. Ostatnie cztery argumenty, o ile występują, powinny być przekazane w postaci argumentów kluczowych o podanych wyżej wartościach domyślnych. Parametr `flush` jest dostępny od Pythona 3.3.

Wszystkie argumenty niekluczowe są przekształcane na łańcuchy znaków — podobnie jak w przypadku funkcji `str()` — i zapisywane do strumienia. Argumenty `sep` i `end` albo muszą być łańcuchami znaków, albo mieć wartość `None` (co oznacza użycie wartości domyślnych). W przypadku braku argumentu *obiekt* zapisywany jest argument `end`. Argument `file` musi być obiektem ze zdefiniowaną metodą `write(łańcuch_znaków)`, ale nie musi to być plik.

W przypadku braku argumentu `file` (lub jeśli ma on wartość `None`) wykorzystywana jest wartość `sys.stdout`.

W Pythonie 2.X działanie funkcji `print` jest dostępne w formie instrukcji. Patrz także „Instrukcja `print`”.

`property([fget[, fset[, fdel[, doc]]]])`

Zwraca atrybut właściwości klas nowego stylu (klas będących pochodnymi klasy `object`). Argument *fget* oznacza funkcję pobierającą wartość atrybutu, *fset* to funkcja ustawiająca wartość atrybutu, a *fdel* to funkcja usuwająca atrybut. Wywołanie można wykorzystać jako dekorator funkcji. Zwraca ono obiekt z metodami *getter*, *setter* i *deleter*, który także może zostać użyty w roli dekoratora (patrz „Instrukcja `def`”). Ta metoda jest zaimplementowana z deskryptorami (patrz „Deskryptory”).

`range([początek[, koniec [, krok]])`

Zwraca kolejne liczby całkowite z przedziału pomiędzy *początek* a *koniec*. Jeśli do funkcji zostanie przekazany jeden argument, to zwraca ona liczby całkowite od zera do *koniec*-1. W przypadku podania dwóch argumentów zwraca liczby całkowite od wartości *początek* do *koniec*-1. Jeśli zostaną przekazane trzy argumenty, funkcja zwraca liczby całkowite z przedziału od *początek* do *koniec*-1, dodając do każdego kolejnego wyniku wartość *krok*. Argumenty *początek* i *krok* mają domyślne wartości odpowiednio 0 i 1.

Argument *krok* może mieć wartość > 1 w celu pominięcia elementów. Wywołanie `range(0, 20, 2)` zwraca listę parzystych liczb całkowitych od 0 do 18. Może on również mieć wartość ujemną, co powoduje zliczanie w dół od wartości *początek* (`range(5, -5, -1)` to lista od 5 do -4). Funkcja jest często używana do generowania list przesunięć lub liczników w pętli `for` oraz innych iteracjach.

W Pythonie 2.X to wywołanie zwraca *listę*. W Pythonie 3.X zwraca *obiekt iterowalny* generujący wartości na żądanie, który można przeglądać wiele razy (jeśli występuje potrzeba generowania wyników, należy opakować wywołanie funkcji `list()`).

`repr(obiekt)`

Zwraca łańcuch znaków zawierający drukowalną i potencjalnie parsowalną (jako kod) reprezentację dowolnego obiektu. Łańcuch znaków, ogólnie rzecz biorąc, przyjmuje formę pozwalającą na parsowanie za pomocą funkcji `eval()` lub udostępnia więcej szczegółów w porównaniu z funkcją `str()` (na tej liście). Tylko w Pythonie 2.X funkcja jest równoważna wywołaniu ``obiekt`` (wyrażenie

w lewych apostrofach, które usunięto z Pythona 3.X). Więcej informacji można znaleźć w opisie metody `__repr__()` w podrozdziale „Metody przeciążające operatory”.

`reversed(sekwencja)`

Zwraca odwrócony obiekt iterowalny. Argument *sekwencja* powinien być obiektem posiadającym metodę `__reversed__()` lub obsługującym protokół sekwencji (metody `__len__()` i `__getitem__()` z argumentami całkowitymi rozpoczynającymi się od 0).

`round(X [, N])`

Zwraca zmiennoprzecinkową wartość *X* zaokrągloną do *N* cyfr po przecinku (kropce dziesiętnej). Argument *N* ma wartość domyślną zero. Może mieć wartość ujemną, co oznacza cyfry z lewej strony kropki dziesiętnej. Jeśli funkcja jest wywoływana z jednym argumentem, to zwracana wartość jest liczbą całkowitą, w przeciwnym razie jest wartością tego samego typu co *X*. W Pythonie 2.X wynik jest zawsze zmiennoprzecinkowy. W Pythonie 3.X funkcja wywołuje metodę *X*.`__round__()`.

`set([obiekt_iterowalny])`

Zwraca zbiór, którego elementy pochodzą z obiektu iterowalnego. Elementy muszą być niezmiennie. W celu reprezentowania zbioru zbiorów zagnieżdżone zbiory powinny być obiektami frozenset. Jeśli nie podano argumentu *obiekt_iterowalny*, funkcja zwraca nowy, pusty zbiór. Funkcja ta jest dostępna od wersji 2.4 Pythona. Więcej informacji można znaleźć w podrozdziale „Zbiory” oraz opisie literału {...} dostępnego w Pythonie 3.X i 2.7.

`setattr(obiekt, nazwa, wartość)`

Przypisuje *wartość* do atrybutu *nazwa* (łańcucha znaków) w *obiekcie*. Funkcja ta ma podobne działanie jak wyrażenie *obiekt.nazwa = wartość*, ale w przypadku funkcji `setattr nazwa` jest łańcuchem znaków, a nie zmienną (np. wywołanie `setattr(a, 'b', c)` jest równoważne wywołaniu `a.b = c`).

`slice([początek,] koniec [, krok])`

Zwraca obiekt wycinka reprezentujący zakres z atrybutami tylko do odczytu *początek*, *koniec* i *krok*. Każdy z nich może mieć wartość None. Argumenty mają takie samo znaczenie jak w przypadku funkcji `range()`. Obiektów wycinków można używać zamiast notacji *i:j:k* (np. wyrażenie *X[i:j]* jest równoważne wyrażeniu *X[slice(i, j)]*).

`sorted(obiekt_iterowalny, key=None, reverse=False)`

Zwraca nową posortowaną listę składającą się z elementów *obiekту_iterowalnego*. Opcjonalne argumenty kluczowe `key` i `reverse` mają takie samo znaczenie jak w przypadku metody `list.sort()` opisanej w podrozdziale „Listy”. Argument `key` jest jednoargumentową funkcją transformacji wartości. Funkcja działa na każdym obiekcie iterowalnym, zwracając nowy obiekt, zamiast modyfikować listę w miejscu. Z tego powodu jest przydatna w pętlach `for`, gdzie można jej używać do sortowania. Pozwala to uniknąć stosowania osobnych instrukcji wywołania funkcji sortującej ze względu na zwracane wyniki `None`. Funkcja jest dostępna w Pythonie 2.4 i w wersjach późniejszych.

W Pythonie 2.X to wywołanie ma sygnaturę `sorted(obiekt_iterowalny, cmp=None, key=None, reverse=False)`, gdzie opcjonalne argumenty `cmp`, `key` i `reverse` mają takie samo znaczenie jak argumenty opisanej wcześniej w tej książce metody `list.sort()` z Pythona 2.X.

`staticmethod(funkcja)`

Zwraca statyczną metodę odpowiadającą *funkcji*. Metoda statyczna nie otrzymuje jawnie pierwszego argumentu, dlatego jest przydatna do przetwarzania atrybutów klas niezależnych od egzemplarzy. W Pythonie w wersji 2.4 i późniejszych można skorzystać z dekoratora `@staticmethod` (patrz „Instrukcja def”). W Pythonie 3.X ta wbudowana funkcja nie jest wymagana dla prostych funkcji w klasach wywoływanych tylko za pośrednictwem obiektów klasy (a nie za pośrednictwem egzemplarzy obiektów).

`str([obiekt [, kodowanie [, błędy]])`

To wywołanie (będące równocześnie nazwą typu, na którego podstawie można tworzyć klasy podrzędne) w Pythonie 3.X działa w jednym z dwóch trybów określonych przez wzorzec wywołania:

- **Drukowanie łańcucha znaków:** jeśli zostanie przekazany sam argument *obiekt*, to funkcja zwraca przyjazną dla użytkownika jego drukowalną reprezentację. W przypadku łańcuchów znaków jest to sam łańcuch znaków. Różnica w stosunku do funkcji `repr(x)` polega na tym, że funkcja `str(x)` nie próbuje zawsze zwracać łańcucha znaków, który można przekazać do funkcji `eval()`. Celem funkcji `str()` jest jedynie zwrócenie drukowalnego łańcucha znaków. Jeśli do funkcji nie zostaną

przekazane żadne argumenty, to zwraca ona pusty łańcuch znaków. Więcej informacji można znaleźć w opisie metody `__str__()` w podrozdziale „Metody przeciążające operatory”.

- *Dekodowanie Unicode*: jeśli zostaną przekazane argumenty *kodowanie* i (lub) *błędy*, to spowoduje to zdekodowanie obiektu (ciągu bajtów lub bufora znaków) z wykorzystaniem kodeka właściwego dla argumentu *kodowanie*. Parametr *kodowanie* jest łańcuchem znaków określającym nazwę kodowania. Jeśli kodowanie nie jest znane, funkcja zgłasza wyjątek `LookupError`. Obsługa błędów jest realizowana zgodnie z wartością argumentu *błędy*. Jeśli ma on wartość `'strict'` (domyślnie), to w przypadku błędów kodowania funkcja zgłasza wyjątek `ValueError`. Z kolei wartość `'ignore'` powoduje, że błędy są ignorowane. Wartość `'replace'` sprawia, że znaki wejściowe, które nie mogą być zdekodowane, są zastępowane uniwersalnym znakiem zastępczym Unicode `U+FFFD`. Warto się również zapoznać z opisem modułu `codecs` oraz podobnie działającą metodą `bytes.decode()` (wywołanie `b'a\xe4'.decode('latin-1')` jest równoważne wywołaniu `str(b'a\xe4', 'latin-1')`).

W Pythonie 2.X wywołanie tej funkcji ma prostszą sygnaturę `str([obiekt])`. Funkcja zwraca łańcuch znaków zawierający drukowalną reprezentację argumentu *obiekt* (co jest równoważne pierwszemu trybowi użycia z Pythona 3.X opisanemu powyżej). Dekodowanie Unicode zaimplementowano w Pythonie 2.X za pomocą metod przetwarzania tekstu lub wywołania `unicode()`. W gruncie rzeczy jest ono takie samo jak wywołanie `str()` z Pythona 3.X (patrz następny podrozdział).

`sum([obiekt_iterowalny [, początek])`

Sumuje argument *początek* oraz elementy obiektu iterowalnego od lewej do prawej i zwraca wynik. Argument *początek* ma domyślną wartość 0. Elementami obiektu iterowalnego powinny być liczby (nie mogą to być łańcuchy znaków). Wskazówka: aby scalić elementy obiektu iterowalnego składającego się z łańcuchów znaków, należy użyć metody `''.join([obiekt_iterowalny])`.

`super([typ [, obiekt]])`

Zwraca klasę nadrzędną *typu*. W przypadku pominięcia drugiego argumentu zwrócony obiekt nadrzędny będzie niezwiązany. Jeśli drugi argument jest obiektem, to funkcja `isinstance([obiekt], typ)` musi zwracać `true`, a jeśli jest typem, to funkcja `issubclass([obiekt],`

typ) musi zwracać *true*. To wywołanie działa dla wszystkich klas w Pythonie 3.X, ale tylko dla klas nowego stylu w Pythonie 2.X. W tym drugim przypadku argument *typ* nie jest opcjonalny.

W Pythonie 3.X wywołanie funkcji `super()` bez argumentów w metodzie klasy ma działanie identyczne z wywołaniem `super(klasa_zawierajaca, argument_self_metody)`. Wywołanie to w sposób jawny lub niejawny tworzy związany obiekt proxy łączący egzemplarz *self* z lokalizacją klasy wywołującej w sekwencji MRO klasy *self*. Ten obiekt proxy można później wykorzystać do odwoływania się do atrybutów klasy nadrzędnej oraz wywoływania jej metod. Więcej informacji na temat porządkowania MRO można znaleźć w podrozdziale „Klasy nowego stylu — MRO”.

Ponieważ funkcja `super()` zawsze wybiera *następną* klasę w sekwencji MRO — pierwszą klasę za klasą wywołującą zawierającą żądany atrybut, niezależnie od tego, czy jest to rzeczywista klasa nadrzędna, czy nie — można jej używać do routowania wywołań metod. W drzewie klas z *pojedynczym dziedziczeniem* wywołanie to można wykorzystać do ogólnego odwoływania się do klas nadrzędnych bez jawnego ich nazywania. W drzewach z *wielokrotnym dziedziczeniem* to wywołanie może być użyte do zaimplementowania kooperatywnej dystrybucji wywołań metod, które propagują wywołania za pośrednictwem drzewa.

Drugi tryb wykorzystania — kooperatywna dystrybucja wywołań metod — może być przydatna w diamentach, ponieważ łańcuch wywołań metod odwiedza każdą klasę nadrzędną tylko raz. Funkcja `super()` może jednak powodować wywoływanie klas nadrzędnych w sposób różny od spodziewanego lub oczekiwanego. Technika dystrybucji metod z wykorzystaniem funkcji `super()`, ogólnie rzecz biorąc, wymaga spełnienia trzech warunków:

- *Kotwice* — metoda wywoływana przez funkcję `super()` musi istnieć, a to wymaga dodatkowego kodu, jeśli nie istnieje kotwica łańcucha wywołań.
- *Argumenty* — metoda wywoływana przez funkcję `super()` musi mieć taką samą sygnaturę w całym drzewie klasy, co może pogarszać elastyczność, szczególnie w przypadku metod poziomu implementacji, takich jak konstruktory.
- *Zastosowanie* — każde wystąpienie metody wywoływanej przez funkcję `super()` oprócz ostatniej samo musi wywoływać funkcję

`super()`, co może utrudniać używanie istniejącego kodu, zmianę porządku wywołań, przesłanianie metod oraz kodowanie samodzielnych klas.

Ze względu na te ograniczenia wywoływanie metod klas nadrzędnych poprzez jawne wskazywanie nazw klas nadrzędnych zamiast używania funkcji `super()` może w niektórych przypadkach być prostsze, bardziej przewidywalne lub bardziej oczekiwane. Dla klasy nadrzędnej `S` jawna i tradycyjna forma `S.method(self)` jest równoważna niejawnej formie `super().method()`. Więcej informacji na temat specjalnego przypadku wyszukiwania atrybutów z wykorzystaniem funkcji `super()` można znaleźć w podrozdziale „Algorytm dziedziczenia nowego stylu”. Zamiast uruchamiać pełne dziedziczenie, obiekty wynikowe skanują zależną od kontekstu ogonową część sekwencji MRO drzewa klasy, wybierając pierwszy pasujący deskryptor lub wartość.

`tuple([obiekt_iterowalny])`

Zwraca nową krotkę zawierającą te same elementy co przekazany `obiekt_iterowalny`. Jeśli `obiekt_iterowalny` jest już krotką, to jest on zwracany bezpośrednio (nie jest zwracana jego kopia). To wystarczy, ponieważ krotki są niemutowalne. W przypadku braku argumentów funkcja zwraca nową, pustą krotkę. Jest to również nazwa klasy, na podstawie której można tworzyć klasy podrzędne.

`type(obiekt | (nazwa, klasy_bazowe, słownik))`

To wywołanie (będące równocześnie nazwą typu, na którego podstawie można tworzyć klasy podrzędne) jest używane w dwóch różnych trybach określonych przez wzorec wywołania:

- *Z jednym argumentem* — zwraca obiekt `type` reprezentujący `obiekt`. Jest to przydatne do sprawdzania typów w instrukcjach `if` (np. `type(x)==type([])`). Warto się również zapoznać z modulem `types` ze standardowej biblioteki, w której znajdują się definicje predefiniowanych typów niebędących nazwami wbudowanymi, a także z opisem funkcji `isinstance()` we wcześniejszej części tego podrozdziału. W klasach nowego stylu wywołanie `type(obiekt)` ma, ogólnie rzecz biorąc, działanie identyczne z wywołaniem `obiekt.__class__`. W Pythonie 2.X moduł `types` obejmuje również synonimy dla większości typów wbudowanych.
- *Z trzema argumentami* — funkcja odgrywa rolę konstruktora zwracającego nowy obiekt typu. Jest to dynamiczna forma

instrukcji `class`. Tekstowy argument *nazwa* jest nazwą klasy, która staje się atrybutem `__name__`; krotka *klasy_bazowe* zawiera klasy bazowe i staje się atrybutem `__bases__`, natomiast *słownik* określa przestrzeń nazw zawierającą definicje treści klas i staje się atrybutem `__dict__`. Na przykład poniższe instrukcje są równoważne:

```
class X(object): a = 1
X = type('X', (object,), dict(a=1))
```

Takiego odwzorowania powszechnie używa się do konstruowania *metaklas*. W takich przypadkach takie wywołania `type()` są generowane automatycznie i zazwyczaj wywołują metodę `__new__()` i (lub) `__init__()` metaklasy z argumentami tworzenia klasy dla podklas wybranego typu.

Patrz także podrozdziały „Metaklasy”, „Dekoratory klas z Pythona 3.X, 2.6 i 2.7” oraz opis metody `__new__()` w podrozdziale „Metody przeciążające operatory”.

`vars([obiekt])`

W przypadku braku argumentów zwraca słownik zawierający nazwy pochodzące z bieżącego zasięgu lokalnego. W razie podania argumentu w postaci modułu, klasy lub egzemplarza klasy funkcja zwraca słownik odpowiadający przestrzeni nazw atrybutu obiektu (tzn. efekt wywołania jego metody `__dict__`). Wynik działania funkcji nie powinien być modyfikowany. Wskazówka: funkcja przydatna do odwoływania się do zmiennych w wyrażeniach formatowania łańcuchów znaków.

`zip([obiekt_iterowalny [, obiekt_iterowalny]*])`

Zwraca ciąg krotek, gdzie *i*-ta krotka zawiera *i*-te elementy z każdego z argumentów *obiekt_iterowalny*. Na przykład `zip('ab', 'cd')` zwraca krotki `('a', 'c')` oraz `('b', 'd')`. Do funkcji trzeba przekazać co najmniej jeden obiekt iterowalny. W przeciwnym przypadku wynik jest pusty. Ciągi wyników są obcinane do długości najkrótszego obiektu iterowalnego przekazanego jako argument. W przypadku pojedynczego argumentu *obiekt_iterowalny* funkcja zwraca ciąg jednoelementowych krotek. Funkcję można również wykorzystać do rozpakowywania spakowanych krotek: `X, Y = zip(*zip(T1, T2))`.

W Pythonie 2.X to wywołanie zwraca *listę*, a w Pythonie 3.X — *obiekt iterowalny* generujący wartości na żądanie, który można przejrzeć tylko raz (jeśli istnieje potrzeba generowania wyników, należy opakować wywołanie funkcji `list()`). W Pythonie 2.X

(ale nie w Pythonie 3.X) w przypadku przekazania wielu obiektów iterowalnych o tej samej długości funkcja `zip()` działa podobnie jak funkcja `map()`, do której przekazano pierwszy argument `None`.

Funkcje wbudowane w Pythonie 2.X

W poprzednim podrozdziale zaprezentowano różnice semantyczne pomiędzy funkcjami dostępnymi *zarówno* w wersji 3.X, jak i 2.X Pythona. Poniżej opisano różnice w treści występujące pomiędzy dwoma liniami wersji Pythona.

Funkcje wbudowane Pythona 3.X nieobsługiwane w Pythonie 2.X

W Pythonie 2.X nie występują następujące funkcje wbudowane Pythona 3.X:

`ascii()`

Działa podobnie jak funkcja `repr()` z Pythona 2.X.

`exec()`

W Pythonie 2.X bardzo podobne semantycznie działanie jest dostępne w formie instrukcji.

`memoryview()`

Zostało udostępnione w Pythonie 2.7 dla zapewnienia zgodności z wersją 3.X.

`print()`

Dostępna w module `__builtin__` Pythona 2.X, ale nie może być bezpośrednio używana bez importowania `__future__`, ponieważ w Pythonie 2.X `print` ma postać instrukcji i jest słowem zarezerwowanym (patrz „Instrukcja `print`”).

Funkcje wbudowane Pythona 2.X nieobsługiwane w Pythonie 3.X

W Pythonie 2.X dostępne są wymienione poniżej funkcje wbudowane. Niektóre z nich są dostępne w Pythonie 3.X w innej formie:

`apply(funkcja, pargs, [, kargs])`

Wywołuje dowolny wywołujący obiekt *funkcja* (funkcję, metodę, klasę itp.) i przekazuje argumenty pozycyjne w krotce *pargs* oraz argumenty kluczowe w słowniku *kargs*. Zwraca wynik wywołania argumentu *funkcja*.

W Pythonie 3.X tę funkcję usunięto. Zamiast niej należy wykorzystać składnię wywołania z rozpakowywaniem argumentów: *funkcja(*pargs, **kargs)*. Ta forma jest również preferowana w Pythonie 2.X ze względu na to, że jest ogólniejsza i bardziej symetryczna z wyrażeniami z gwiazdkami stosowanymi w definicjach funkcji (patrz „Instrukcja wyrażeniowa”).

`basestring()`

Klasa bazowa dla łańcuchów znaków — zwykłych oraz Unicode (przydatna do wykonywania testów `isinstance()`).

W Pythonie 3.X wszystkie dane tekstowe są reprezentowane za pomocą jednego typu `str` (zarówno 8 bitów, jak i bogatsze, Unicode).

`buffer(obiekt [, przesunięcie [, rozmiar]])`

Zwraca nowy obiekt bufora dla przekazanego argumentu *obiekt* (patrz *Python Library Reference*).

W Pythonie 3.X tę funkcję usunięto. Podobną funkcjonalność zapewnia nowa funkcja wbudowana `memoryview()`. Jest ona również dostępna w wersji 2.7 w celu zapewnienia zgodności w przód.

`cmp(X, Y)`

Zwraca ujemną liczbę całkowitą, zero bądź dodatnią liczbę całkowitą, co oznacza odpowiednio $X < Y$, $X == Y$ lub $X > Y$.

W Pythonie 3.X tę funkcję usunięto, ale można ją symulować za pomocą wyrażenia: $(X > Y) - (X < Y)$. Jednak najbardziej powszechne przypadki użycia funkcji `cmp()` (funkcje porównujące w operacjach sortowania oraz metody `__cmp__()` klas) także zostały usunięte z Pythona 3.X.

`coerce(X, Y)`

Zwraca krotkę zawierającą dwa numeryczne argumenty X i Y przekonwertowane na wspólny typ.

Funkcję tę usunięto z Pythona 3.X (funkcja była stosowana głównie w klasycznych klasach Pythona 2.X).

`execfile(nazwapliku [, globalny [, lokalny]])`

Funkcja działa podobnie jak `eval()`, ale uruchamia kod umieszczony w pliku, którego tekstową nazwę przekazano za pomocą argumentu *nazwapliku* (zamiast wyrażenia). W odróżnieniu od operacji importowania nie powoduje to utworzenia nowego obiektu modułu dla pliku. Funkcja zwraca `None`. Przestrzenie nazw kodu w pliku *nazwapliku* działają tak samo jak dla funkcji `eval()`.

W Pythonie 3.X tę funkcję można zasymulować za pomocą wywołania: `exec(open(nazwapliku).read())`.

`file(nazwapliku [, tryb[, rozmiarbufora]])`

Alias dla wbudowanej funkcji `open()` oraz nazwa klasy bazowej dla wbudowanego typu `file`.

W Pythonie 3.X nazwę `file` usunięto. Do tworzenia obiektów `file` należy użyć funkcji `open()`, a działania na plikach można wykonywać za pomocą funkcji z modułu `io` (moduł `io` jest wykorzystywany przez funkcję `open()` w Pythonie 3.X, natomiast w wersji 2.X, począwszy od wersji 2.6, jest to opcjonalne).

`input([symbolzachęty])` (postać pierwotna z wersji 2.X)

Wyświetla argument *symbolzachęty*, o ile go przekazano. Następnie czyta wiersz ze standardowego strumienia wejściowego *stdin* (`sys.stdin`), uruchamia jako kod Pythona i zwraca wynik działania. Działa tak jak wywołanie `eval(raw_input(symbolzachęty))`. Wskazówka: nie należy używać tej instrukcji do sprawdzania niezauważanych ciągów kodu, ponieważ są one uruchamiane tak samo jak kod programu.

W Pythonie 3.X ze względu na zmianę nazwy funkcji `raw_input()` na `input()` pierwotna funkcja Pythona 2.X `input()` nie jest już dostępna, ale można ją zasymulować za pomocą wywołania: `eval(input((symbolzachęty)))`.

`intern(łańcuchznaków)`

Wprowadza łańcuchznaków do tablicy „łańcuchów internowanych” i zwraca ten łańcuch. Łańcuchy internowane są „nieśmiertelne”. Wykorzystuje się je jako mechanizm optymalizacji wydajności (można je porównywać za pomocą szybkiego operatora tożsamości `is` zamiast operatora równości `==`).

W Pythonie 3.X tę funkcję przeniesiono do modułu `sys.intern()`. Aby ją wykorzystać, należy zaimportować moduł `sys`. Więcej informacji można znaleźć w podrozdziale „Moduł `sys`”.

`long(X [, podstawa])`

Konwertuje liczbę lub ciąg znaków *X* na długą liczbę całkowitą. Argument *podstawa* można przekazać tylko wtedy, gdy *X* jest łańcuchem znaków. Jeśli argument ten ma wartość 0, to podstawa jest wyznaczana z kontekstu łańcucha znaków, w przeciwnym razie wartość argumentu jest używana jako podstawa konwersji. Jest to nazwa klasy, na podstawie której można tworzyć klasy podrzędne.

W Pythonie 3.X typ liczb całkowitych `int` obsługuje liczby `long` dowolnej precyzji, a zatem typ `int` obejmuje typ `long` z Pythona 2.X. W Pythonie 3.X należy używać funkcji `int()`.

`raw_input([symbolzachęty])`

Odpowiednik funkcji `input()` (opisanej wcześniej) z Pythona 3.X w Pythonie 2.X. Wyświetla *symbolzachęty*, czyta wiersz i zwraca wynik, ale nie wyznacza wartości następnego wiersza wejściowego.

W Pythonie 3.X należy używać wbudowanej funkcji `input()`.

`reduce(funkcja, obiekt_iterowalny [, start])`

Stosuje dwuargumentową funkcję *funkcja* do kolejnych elementów obiektu iterowalnego, aby zredukować kolekcję do pojedynczej wartości. Jeśli występuje argument *start*, jest on wstawiany na początek argumentu *obiekt_iterowalny*.

W Pythonie 3.X ta funkcja jest w dalszym ciągu dostępna jako `functools.reduce()`. Aby ją wykorzystać, należy zaimportować moduł `functools`.

`reload(moduł)`

Powoduje ponowne załadowanie, parsowanie i uruchomienie wcześniej zaimportowanego modułu w bieżącej przestrzeni nazw. Ponowne uruchomienie sprawi, że poprzednie wartości atrybutów modułu zostaną zastąpione w miejscu. Argument *moduł* musi się odwoływać do istniejącego obiektu modułu. Nie może to być nowa nazwa ani łańcuch znaków. Funkcja przydaje się w trybie interaktywnym, kiedy chcemy ponownie załadować moduł po jego zmodyfikowaniu bez konieczności restartowania Pythona. Funkcja zwraca obiekt *moduł*. Warto się także zapoznać z zawartością tabeli `sys.modules`, w której są przechowywane zaimportowane moduły (można je stamtąd usunąć w celu wymuszenia ponownego importu).

W Pythonie 3.X ta funkcja jest w dalszym ciągu dostępna jako `imp.reload()`. Aby ją wykorzystać, należy zaimportować moduł `imp`.

`unichr(I)`

Zwraca jednoznakowy łańcuch znaków Unicode zawierający znak, którego kod Unicode ma wartość *I* (np. `unichr(97)` zwraca ciąg `u'a'`). Funkcja ma odwrotne działanie do funkcji `ord()` stosowanej dla łańcuchów znaków Unicode oraz wersji Unicode funkcji `chr()`. Argument musi być liczbą z zakresu `0...65535` włącz-

nie. W przypadku podania argumentu spoza tego zakresu Python zgłasza wyjątek `ValueError`.

W Pythonie 3.X standardowe łańcuchy znaków reprezentują tekst Unicode — zamiast funkcji `unichr()` należy stosować funkcję `chr()`, np. `ord('\xe4')` zwraca 228, natomiast zarówno `chr(228)`, jak i `chr(0xe4)` zwracają `'ä'`.

```
unicode([obiekt [, kodowanie [, błędy]])
```

Funkcja działa podobnie do funkcji `str()` z Pythona 3.X (więcej informacji można znaleźć w opisie funkcji `str()` w poprzednim podrozdziale). Jeśli zostanie przekazany tylko *jeden* argument, to funkcja zwraca przyjazną dla użytkownika reprezentację obiektu, ale w postaci łańcucha Unicode Pythona 2.X. Jeśli zostanie przekazany *więcej niż jeden* argument, to funkcja realizuje dekodowanie Unicode łańcucha *obiekt* z wykorzystaniem kodeka odpowiadającego argumentowi *kodowanie*. Obsługa błędów jest wykonywana zgodnie z wartością argumentu *błędy*. Działaniem domyślnym jest zdekodowanie ciągu w trybie ścisłym, co oznacza, że błędy kodowania zgłaszają wyjątek `ValueError`.

Listę plików obsługujących kodowanie można znaleźć w opisie modułu `codecs` w dokumentacji *Python Library Reference*. W Pythonie 2.X niektóre obiekty dostarczają metodę `__unicode__()`, która zwraca łańcuch Unicode zgodny z wynikiem wywołania `unicode(X)`.

W Pythonie 3.X nie istnieje osobny typ dla łańcuchów znaków Unicode — wszystkie dane tekstowe są reprezentowane za pomocą jednego typu `str` (zarówno 8-bitowe, jak i bogatsze, Unicode), natomiast dane binarne złożone z 8-bitowych bajtów reprezentuje typ `bytes`. Do obsługi tekstu Unicode należy używać zwykłych łańcuchów znaków `str`. Do dekodowania „surowych” bajtów na postać Unicode zgodną z podanym kodowaniem należy używać funkcji `bytes.decode()` bądź `str()`, natomiast do przetwarzania plików tekstowych Unicode powinno się wykorzystywać standardowe obiekty `file`.

```
xrange([początek,] koniec [, krok])
```

Działa podobnie do funkcji `range()`, ale nie zapamiętuje całej listy naraz (zamiast tego generuje jednorazowo po jednej liczbie całkowitej). Taka własność przydaje się w pętlach `for`, kiedy do przejścia jest duży zakres, a jest mało pamięci. Funkcja pozwala na oszczędność miejsca w pamięci, ale ogólnie rzecz biorąc, nie przyspiesza działania.

W Pythonie 3.X zmodyfikowano wcześniejszą funkcję `range()` w taki sposób, by zwracała obiekt iterowalny, zamiast tworzyć wynikową listę w pamięci. Funkcja `range()` Pythona 3.X obejmuje swoim zakresem funkcjonalnym usuniętą funkcję `xrange()` z Pythona 2.X.

Poza tym funkcja otwierania plików `open()` zmieniła się w Pythonie 3.X na tyle mocno, że uzasadnione jest indywidualne omówienie wariantu funkcji dla Pythona 2.X (w Pythonie 2.X wywołanie `codecs.open()` ma wiele własności funkcji `open()` Pythona 3.X — w tym obsługę translacji kodowania Unicode podczas przesyłania):

```
open(nazwapliku [, tryb[, rozmiarbufora]])
```

Zwraca nowy obiekt `file`, połączony z zewnętrznym plikiem o nazwie *nazwapliku* (łańcuch znaków), bądź zgłasza wyjątek `IOError`, kiedy otwarcie pliku się nie powiedzie. Jeśli nazwa pliku nie zawiera prefiksu ścieżki do katalogu, to jest ona powiązana z bieżącym katalogiem roboczym. Pierwsze dwa argumenty, ogólnie rzecz biorąc, mają takie samo znaczenie jak argumenty funkcji `fopen()` języka C, a plik jest obsługiwany przez system `stdio`. W przypadku użycia funkcji `open()` dane pliku zawsze są reprezentowane w skrypcie w postaci zwykłego łańcucha znaków `str`, złożonego z bajtów zapisanych w pliku (wywołanie `codecs.open()` interpretuje zawartość pliku jako tekst Unicode reprezentowany przez obiekty `unicode`).

Jeśli argument *tryb* zostanie pominięty, to przyjmowana jest domyślna wartość `'r'`. Dozwolonymi wartościami tego argumentu są `'r'` dla operacji wejściowych i `'w'` dla operacji wyjściowych (z usunięciem poprzedniej zawartości pliku). Z kolei wartość `'a'` oznacza dołączanie danych na końcu pliku. Dla plików binarnych dostępne są również tryby `'rb'`, `'wb'` i `'ab'` (pozwalają one na wyłączenie konwersji znaków zakończenia wiersza na sekwencję `\n`). W większości systemów do trybów aktualizacyjnych w operacjach wejścia-wyjścia można również dołączać operator `+` (np. `'r+'` dla operacji odczytu-zapisu oraz `'w+'` w celu odczytu i zapisu, ale z początkową inicjalizacją do postaci pustego pliku).

Domyślna wartość argumentu *rozmiarbufora* zależy od implementacji. Może mieć wartość 0 dla danych niebuforowanych, 1 dla danych buforowanych w trybie wierszowym, wartość ujemną dla buforowania domyślnego w systemie bądź też określony, specyficzny rozmiar. Operacje transferu danych przy włączonym buforowaniu nie zawsze są realizowane natychmiast (aby wymu-

sić transfer, należy użyć metody `flush()`). Patrz także opis modułu `io` ze standardowej biblioteki Pythona — jest to alternatywa dla typu `file` w Pythonie 2.X oraz standardowy interfejs plikowy dla operacji `open()` w Pythonie 3.X.

Wbudowane wyjątki

W tym podrozdziale opisano wyjątki, które Python może zgłaszać podczas działania programów. Podrozdział omawia stan wbudowanych wyjątków w Pythonie 3.3, w którym wprowadzono nowe klasy dla błędów związanych z systemem. Te nowe klasy obejmują wcześniejsze klasy ogólne z informacjami o stanie, ale uwzględniają informacje wspólne dla większości wersji Pythona. Informacje dotyczące różnic pomiędzy poszczególnymi wersjami można znaleźć nieco niżej, w podrozdziałach dotyczących Pythona 3.2 i 2.X.

Od wydania Pythona w wersji 1.5 wszystkie wbudowane wyjątki są *obiektami klas* (wcześniej były łańcuchami znaków). Wbudowane wyjątki są dostarczone w przestrzeni nazw zasięgu wbudowanego (patrz „Przestrzeń nazw i reguły zasięgu”). Z wieloma wbudowanymi wyjątkami są powiązane informacje o stanie, które dostarczają szczegółów na ich temat. Wyjątki definiowane przez użytkownika są, ogólnie rzecz biorąc, klasami pochodnymi zbioru wyjątków wbudowanych (patrz „Instrukcja `raise`”).

Klasy bazowe (kategorie)

Wymienione poniżej wyjątki są wykorzystywane tylko jako klasy bazowe dla innych wyjątków.

`BaseException`

Główna klasa bazowa dla wszystkich wyjątków wbudowanych. Z tej klasy nie powinny bezpośrednio dziedziczyć klasy definiowane przez użytkowników. Do tego celu służy klasa `Exception`. Funkcja `str()` wywołana dla egzemplarza tej klasy zwraca reprezentację argumentów konstruktora przekazanych podczas tworzenia egzemplarza (w przypadku braku takich argumentów przekazywany jest pusty ciąg znaków). Wspomniane argumenty egzemplarza konstruktora są dostępne w postaci krotki w atrybucie `args` egzemplarza. Klasy pochodne dziedziczą ten protokół.

Exception

Główna klasa bazowa dla wszystkich wyjątków wbudowanych oraz wszystkich wyjątków, które nie powodują zakończenia działania systemu. Jest to bezpośrednia klasa pochodna klasy `BaseException`.

Z tej klasy powinny dziedziczyć wszystkie wyjątki zdefiniowane przez użytkowników. Wspomniane dziedziczenie jest wymagane dla wyjątków użytkownika w Pythonie 3.X. W Pythonie 2.6 i 2.7 jest ono wymagane dla klas nowego stylu, ale dopuszczalne jest również definiowanie samodzielnych klas wyjątków.

Instrukcja `try`, która przechwytuje ten wyjątek, obsługuje wszystkie wyjątki poza zdarzeniami wyjścia z systemu, ponieważ `Exception` jest klasą bazową dla wszystkich wyjątków oprócz `SystemExit`, `KeyboardInterrupt` i `GeneratorExit` (wymienione trzy klasy dziedziczą bezpośrednio po klasie `BaseException`).

ArithmeticError

Kategoria wyjątków błędów arytmetycznych — klasa bazowa dla wyjątków `OverflowError`, `ZeroDivisionError` i `FloatingPointError` oraz klasa pochodna klasy `Exception`.

BufferError

Wywoływany w przypadku braku możliwości wykonania operacji związanych z buforami. Klasa pochodna klasy `Exception`.

LookupError

Błędy sekwencji oraz indeksu przy mapowaniu — klasa bazowa dla wyjątków `IndexError` i `KeyError`. Wyjątek jest również zgłaszany dla niektórych błędów wyszukiwania `Unicode`. Klasa pochodna klasy `Exception`.

OSError (nowość w Pythonie 3.3)

Wywoływany, gdy funkcja systemowa spowoduje błąd związany z systemem, taki jak niepowodzenie operacji wejścia-wyjścia albo operacji plikowej. Począwszy od Pythona 3.3, ten wyjątek jest główną klasą bazową dla nowego zbioru opisowych wyjątków związanych z systemem. Wyjątki te zostały wymienione w podrozdziale „Szczegółowe wyjątki `OSError`”. Nowe wyjątki obejmują wcześniejsze wyjątki ogólne z informacjami o stanie, które są używane w wersji 3.2 i wersjach wcześniejszych, opisane w podrozdziale „Wbudowane wyjątki w Pythonie 3.2”.

W Pythonie 3.3 `OSError` jest klasą pochodną klasy `Exception` i zawiera wspólne atrybuty informacyjne dostarczające informacji

o błędach systemu: `errno` (kod liczbowy), `strerror` (tekstowy komunikat), `winerror` (w systemie Windows) oraz `filename` (w przypadku wyjątków dotyczących ścieżek do plików). W wersji 3.3 klasa ta obejmuje dawne klasy `EnvironmentError`, `IOError`, `WindowsError`, `VMSError`, `socket.error`, `select.error` i `mmap.error` oraz jest synonimem dla klasy `os.error`. Dodatkowe informacje na temat atrybutów można znaleźć w podrozdziale „Moduł systemowy `os`” w dalszej części tej książki.

Wyjątki szczegółowe

Wymienione poniżej klasy zawierają bardziej szczegółowe wyjątki, które są faktycznie zgłaszane. Poza tym `NameError`, `RuntimeError`, `SyntaxError`, `ValueError` i `Warning` są wyjątkami szczegółowymi, a jednocześnie klasami bazowymi dla innych wyjątków wbudowanych.

`AssertionError`

Zgłaszany, gdy test instrukcji `assert` ma wartość `false`.

`AttributeError`

Zgłaszany w przypadku niepowodzenia odwołania lub przypisania wartości do atrybutu.

`EOFError`

Zgłaszany w przypadku osiągnięcia końca pliku podczas wywoływania funkcji `input()` (lub `raw_input()` w Pythonie 2.X). Metody czytania danych z pliku w przypadku osiągnięcia końca pliku nie zgłaszają wyjątku, tylko pusty obiekt.

`FloatingPointError`

Zgłaszany w przypadku niepowodzenia operacji zmiennoprzecinkowej.

`GeneratorExit`

Zgłaszany podczas wywoływania metody `close()` generatora. Ten wyjątek dziedziczy bezpośrednio po klasie `BaseException` zamiast `Exception`, ponieważ to nie jest błąd.

`ImportError`

Zgłaszany w przypadku niepowodzenia wyszukiwania modułu lub atrybutu za pomocą instrukcji `import` lub `from`. Od Pythona 3.3 egzemplarze zawierają atrybuty `name` i `path`, identyfikujące moduł, który wywołał błąd. Są one przekazywane jako argumenty kluczowe do konstruktora.

IndentationError

Zgłaszany w przypadku niepoprawnych wcięć występujących w kodzie źródłowym. Wyjątek ten jest pochodną klasy `SyntaxError`.

IndexError

Zgłaszany w przypadku odwołania do indeksu sekwencji spoza zakresu (przy pobieraniu danych bądź przypisywaniu). Indeksy wycinków są „bezgłośnie” przekształcane na wartość z dozwolonego zakresu. W przypadku odwołania do indeksu, który nie jest liczbą całkowitą, zgłaszany jest wyjątek `TypeError`.

KeyError

Zgłaszany w przypadku odwołania do nieistniejącego klucza mapowania (przy pobieraniu). Operacje przypisania do nieistniejącego klucza powodują jego utworzenie.

KeyboardInterrupt

Zgłaszany w momencie wciśnięcia przez użytkownika klawiszy przerwania (standardowo `Ctrl+C` lub `Delete`). W czasie wykonywania programu regularnie sprawdzane jest występowanie przerw. Ten wyjątek dziedziczy bezpośrednio po klasie `BaseException`. Ma to na celu zapobieżenie przypadkowemu przechwyceniu wyjątku przez kod przechwytyjący wyjątek `Exception`, co spowodowałoby zakończenie działania interpretera.

MemoryError

Zgłaszany w momencie wystąpienia naprawialnego błędu wyczerpywania się pamięci. Jeśli przyczyną tego stanu był niekontrolowany program, to ten wyjątek powoduje wyświetlenie śladu stosu.

NameError

Zgłaszany w przypadku niepowodzenia operacji wyszukiwania lokalnej bądź globalnej niekwalifikowanej nazwy.

NotImplementedError

Zgłaszany w przypadku niepowodzenia definiowania oczekiwanych protokołów. Wyjątek ten mogą zgłaszać metody klas abstrakcyjnych, w przypadku gdy wymagają zdefiniowania metody. Jest on pochodną klasy `RuntimeError` (nie należy go mylić z wyjątkiem `NotImplemented` — specjalnym wbudowanym obiektem wyjątku, zwracanym przez niektóre metody przeciążające operatory, w przypadku kiedy typy operandów są nieobsługiwane — patrz podrozdział „Metody przeciążające operatory”).

OverflowError

Zgłaszany podczas wystąpienia przepełnienia w trakcie operacji arytmetycznych. Wyjątek nie może wystąpić podczas operacji na liczbach całkowitych, ponieważ zapewniają one dowolną precyzję. Ze względu na ograniczenia podstawowego języka C, na którym bazuje Python, w większości operacji zmiennoprzecinkowych przepełnienia również nie są sprawdzane.

ReferenceError

Zgłaszany w związku ze *słabymi odwołaniami* — narzędziami do utrzymywania referencji do obiektów, które nie blokują ich rekultywacji (np. pamięci podręczne). Patrz moduł `weakref` ze standardowej biblioteki Pythona.

RuntimeError

Rzadko używany wyjątek do przechwytywania wszystkich wyjątków.

StopIteration

Zgłaszany na końcu progresji wartości w obiektach iteratorów. Wyjątek ten zgłasza wbudowana funkcja `next(I)` oraz metoda `I.__next__()` (w Pythonie 2.X `I.next()`).

Od wersji 3.3 Pythona egzemplarze zawierają atrybut `value`, który albo odzwierciedla jawnie pozycyjny argument konstruktora, albo jest automatycznie ustawiany na zwracaną wartość podaną w kończącej iterację instrukcji `return` funkcji generatora. Atrybut ten ma domyślnie wartość `None`. Jest on dostępny również w standardowej krotce `args` wyjątku. Nie jest używany przez automatyczne iteracje. Ponieważ przed wersją 3.3 funkcje generatorów *nie* mogły zwracać wartości (w przypadku takiej próby generowały błąd syntaktyczny), korzystanie z tego rozszerzenia nie jest kompatybilne z wcześniejszymi wersjami 2.X i 3.X. Patrz także „Instrukcja `yield`”.

SyntaxError

Zgłaszany w przypadku napotkania przez parsery błędu składni. Może wystąpić podczas operacji importowania, wywołań funkcji `eval()` i `exec()`, a także w trakcie odczytywania kodu z pliku skryptu najwyższego poziomu bądź standardowego wejścia. Egzemplarze tej klasy mają atrybuty `filename`, `lineno`, `offset` i `text`, które pozwalają na uzyskanie szczegółowych informacji o wyjątku. Wywołanie metody `str()` dla egzemplarza wyjątku zgłasza sam komunikat.

SystemError

Zgłaszany w momencie wystąpienia błędów wewnętrznych interpretera, które nie są wystarczająco poważne, aby było uzasadnione jego zamknięcie (takie wyjątki należy rejestrować).

SystemExit

Zgłaszany podczas wywołania metody `sys.exit(N)`. Jeśli wyjątek ten nie zostanie obsłużony, to interpreter Pythona kończy działanie bez wyświetlania śladu stosu. Jeśli przekazana wartość `N` jest liczbą całkowitą, to oznacza ona status wyjścia systemu (przekazywany do funkcji `exit` języka C). Jeśli ma ona wartość `None`, status wyjścia jest interpretowany jako 0 (sukces). W przypadku wartości innego typu wyświetlana jest wartość obiektu, a status wyjścia wynosi 1 (niepowodzenie). Ten wyjątek dziedziczy bezpośrednio po klasie `BaseException`. Ma to na celu zapobieżenie przypadkowemu przechwyceniu wyjątku przez kod przechwytyjący wyjątek `Exception`, co spowodowałoby zakończenie działania interpretera. Patrz także opis modułu `sys.exit()` w punkcie „Moduł `sys`”.

Wyjątek ten zgłasza wywołanie metody `sys.exit()`, dzięki czemu są uruchamiane handlersy końcowe (klauszule `finally` instrukcji `try`), a debugger może uruchomić skrypt bez utraty kontroli. Wtedy w razie potrzeby funkcja `os._exit()` natychmiast kończy działanie (np. w procesie dziecka po wywołaniu funkcji `fork()`). Specyfikację funkcji `exit` można znaleźć w dokumentacji standardowego modułu bibliotecznego `atexit`.

TabError

Zgłaszany w przypadku znalezienia w kodzie źródłowym niepoprawnej mieszanki spacji i tabulacji. Wyjątek ten jest pochodną klasy `IndentationError`.

TypeError

Zgłaszany w przypadku zastosowania działania bądź funkcji do obiektu nieodpowiedniego typu.

UnboundLocalError

Zgłaszany w momencie odwołań do lokalnych nazw, do których jeszcze nie przypisano wartości. Wyjątek ten jest pochodną klasy `NameError`.

UnicodeError

Zgłaszany w przypadku błędów kodowania lub dekodowania łańcuchów znaków `Unicode`. Jest to klasa bazowa (kategoria) oraz klasa pochodna klasy `ValueError`. Wskazówka: niektóre narzędzia do obsługi `Unicode` mogą zgłaszać wyjątek `LookupError`.

UnicodeEncodeError

UnicodeDecodeError

UnicodeTranslateError

Zgłaszane w przypadku wystąpienia błędów związanych z przetwarzaniem tekstów Unicode. Są to klasy pochodne klasy `UnicodeError`.

ValueError

Zgłaszany, w przypadku gdy wbudowane działanie bądź funkcja otrzyma argument, który ma właściwy typ, ale nieodpowiednią wartość, a sytuacja nie jest opisana przez bardziej szczegółowy wyjątek, na przykład `IndexError`.

ZeroDivisionError

Zgłaszany w momencie operacji dzielenia lub modulo przez 0.

Szczegółowe wyjątki OSError

W Pythonie 3.3 i wersjach późniejszych wymienione poniżej klasy będące pochodnymi klasy `OSError` identyfikują błędy systemowe i odpowiadają kodom błędów systemowych dostępnych w klasie `EnvironmentError` we wcześniejszych wersjach Pythona (patrz „Wbudowane wyjątki w Pythonie 3.2”). Patrz także opis klasy `OSError` w podrozdziale „Klasy bazowe (kategorie)”. Można tam znaleźć informacje dotyczące atrybutów wspólnych dla wymienionych poniżej podklas:

BlockingIOError

Zgłaszany, w przypadku gdy operacja zablokowałaby zbiór obiektów w operacji nieblokującej. Ma dodatkowy atrybut `characters_written`, który oznacza liczbę znaków zapisanych do strumienia przed zablokowaniem.

ChildProcessError

Wywoływany w przypadku niepowodzenia operacji na procesie potomnym.

ConnectionError

Klasa bazowa dla wyjątków powiązanych z połączeniami: `BrokenPipeError`, `ConnectionAbortedError`, `ConnectionRefusedError` i `ConnectionResetError`.

BrokenPipeError

Zgłaszany podczas próby zapisu do potoku, w sytuacji kiedy druga strona została zamknięta, albo w trakcie próby pisania do gniazda, które zamknięto do zapisu.

`ConnectionAbortedError`

Zgłaszany w przypadku przerwania próby nawiązania połączenia przez partnera.

`ConnectionRefusedError`

Zgłaszany, w przypadku gdy partner odmówi próby nawiązania połączenia.

`ConnectionResetError`

Zgłaszany w przypadku zresetowania połączenia przez partnera.

`FileExistsError`

Zgłaszany podczas próby utworzenia pliku lub katalogu, który już istnieje.

`FileNotFoundError`

Zgłaszany podczas żądania dostępu do pliku lub katalogu, które nie istnieją.

`InterruptedError`

Zgłaszany, w przypadku gdy wywołanie systemowe zostanie przerwane przez sygnał z zewnątrz.

`IsADirectoryError`

Zgłaszany podczas żądania wykonania operacji plikowej, na przykład `os.remove()`, na katalogu.

`NotADirectoryError`

Zgłaszany podczas żądania wykonania operacji katalogowej, na przykład `os.listdir()`, na obiekcie niebędącym katalogiem.

`PermissionError`

Zgłaszany podczas wykonywania operacji z odpowiednimi uprawnieniami dostępu (np. uprawnieniami do systemu plików).

`ProcessLookupError`

Zgłaszany, w przypadku gdy proces nie istnieje.

`TimeoutError`

Zgłaszany, w przypadku gdy upłynie limit czasu wykonania funkcji na poziomie systemu.

Wyjątki kategorii ostrzeżeń

Wymienione poniżej wyjątki są wykorzystywane w kategorii ostrzeżeń:

`Warning`

Klasa bazowa dla wszystkich wyjątków kategorii ostrzeżeń. Jest bezpośrednią pochodną klasy `Exception`.

UserWarning

Ostrzeżenia generowane przez kod użytkownika.

DeprecationWarning

Ostrzeżenia dotyczące niezalecanych własności.

PendingDeprecationWarning

Ostrzeżenia związane z własnościami, które będą wycofywane w przyszłości.

SyntaxWarning

Ostrzeżenia dotyczące problematycznej składni.

RuntimeWarning

Ostrzeżenia dotyczące problematycznego działania kodu.

FutureWarning

Ostrzeżenia związane z konstrukcjami, które w przyszłości zmienią się pod względem semantycznym.

ImportWarning

Ostrzeżenia dotyczące prawdopodobnych błędów importowania modułów.

UnicodeWarning

Ostrzeżenia związane z danymi tekstowymi Unicode.

BytesWarning

Ostrzeżenia związane z obiektami bytes i buffer (memoryview).

ResourceWarning

Wprowadzone w Pythonie 3.2. Klasa bazowa dla ostrzeżeń związanych z wykorzystaniem zasobów.

Framework ostrzeżeń

Ostrzeżenia są wydawane, w przypadku gdy przyszłe zmiany w języku mogą doprowadzić do niestabilności działania kodu w kolejnych wydaniach Pythona oraz w innych kontekstach. Ostrzeżenia można skonfigurować w taki sposób, aby wyświetlały komunikaty, zgłaszały wyjątki lub były ignorowane. Framework ostrzeżeń można wykorzystać w celu wydawania ostrzeżeń poprzez wywoływanie funkcji `warnings.warn()`:

```
warnings.warn("konstrukcja przestarzała", DeprecationWarning)
```

Ponadto można dodawać filtry w celu zablokowania niektórych ostrzeżeń. Istnieje także możliwość stosowania wzorców wyrażeń regularnych w odniesieniu do nazw lub nazw modułów. Pozwala to

na wyłączanie ostrzeżeń na różnym poziomie ogólności. Można na przykład wyłączyć ostrzeżenia dotyczące wykorzystania niezalecanego modułu `regex` za pomocą następującego wywołania:

```
import warnings
warnings.filterwarnings(action = 'ignore',
                        message='.*regex module*',
                        category=DeprecationWarning,
                        module = '__main__')
```

Wywołanie to powoduje dodanie filtru, który wpływa tylko na ostrzeżenia klasy `DeprecationWarning` zgłaszane w module `__main__`. Filtr ten powoduje zastosowanie wyrażenia regularnego w celu dopasowania jedynie tych komunikatów, które wymieniają niezalecany moduł `regex`. Takie ostrzeżenia są ignorowane. Ostrzeżenia mogą być również wyświetlane tylko raz, wyświetlane za każdym razem, kiedy wykonuje się kod, który je zgłasza, albo przekształcane na wyjątki powodujące zatrzymanie działania programu (o ile wyjątki te nie zostaną obsłużone). Więcej informacji na ten temat można znaleźć w dokumentacji modułu `warnings` w Pythonie w wersji 2.1 i późniejszych. Warto się również zapoznać z opisem argumentu wiersza poleceń `-W`. Opisano go w podrozdziale „Opcje wiersza poleceń Pythona”.

Wbudowane wyjątki w Pythonie 3.2

W Pythonie 3.2 i wersjach wcześniejszych dostępne są dodatkowe wyjątki wymienione poniżej. Od Pythona 3.3 scalono je jako wyjątki pochodne `OSError`. Wyjątki wyszczególnione poniżej zachowano w wersji 3.3 w celu zapewnienia zgodności, ale być może zostaną usunięte w przyszłych wydaniach:

`EnvironmentError`

Kategoria wyjątków, które występują na zewnątrz Pythona; klasa bazowa dla wyjątków `IOError` i `OSError` oraz klasa pochodna klasy `Exception`. Zgłoszony egzemplarz zawiera atrybuty informacyjne `errno` i `strerror` (oraz ewentualnie `filename` w przypadku wyjątków związanych ze ścieżkami do plików). Atrybuty te są również dostępne za pośrednictwem atrybutu `args` — dostarczają informacji o kodach błędów systemowych oraz szczegóły związane z komunikatami o błędach.

`IOError`

Zgłaszany w przypadku niepowodzenia operacji wejścia-wyjścia albo operacji związanej z plikami. Pochodna klasy `EnvironmentError`. Zawiera informacje o stanie opisane wcześniej na tej liście.

`OSError` (wersja z Pythona 3.2)

Zgłaszany w przypadku błędów związanych z modulem `os` (jego wyjątek `os.error`). Jest to pochodna klasy `EnvironmentError`. Zawiera informacje o stanie opisane wcześniej na tej liście.

`VMSError`

Zgłaszany w przypadku błędów specyficznych dla systemu VMS. Klasa pochodna klasy `OSError`.

`WindowsError`

Zgłaszany w przypadku błędów specyficznych dla systemu Windows. Klasa pochodna klasy `OSError`.

Wbudowane wyjątki w Pythonie 2.X

Zbiór dostępnych wyjątków, a także kształt hierarchii klas wyjątków w Pythonie 2.X różni się nieco od wyjątków z wersji 3.X, które zostały opisane powyżej. Na przykład w Pythonie 2.X:

- `Exception` jest klasą wyjątków najwyższego poziomu (nie jest nią `BaseException`, która nie występuje w Pythonie 2.X).
- `StandardError` to dodatkowa klasa pochodna klasy `Exception`, która jest główną klasą znajdującą się w hierarchii ponad wszystkimi wbudowanymi wyjątkami oprócz `SystemExit`.

Więcej informacji na temat konkretnej wersji można znaleźć w dokumentacji Pythona 2.X.

Wbudowane atrybuty

Niektóre obiekty obsługują atrybuty specjalne predefiniowane przez Pythona. Poniżej przedstawiono częściową listę, ponieważ wiele typów zawiera własne, unikatowe atrybuty. Więcej informacji na ich temat można znaleźć w dokumentacji *Python Library Reference*¹⁵:

¹⁵ Począwszy od Pythona 2.1, dowolne zdefiniowane przez użytkownika atrybuty można dodawać również do obiektów `function`. W tym celu wystarczy przypisać do nich wartości. Patrz „Wartości domyślne i atrybuty”. Python 2.X obsługuje również atrybuty specjalne `I.__methods__` oraz `I.__members__` — listy nazw metod i składowych danych dla niektórych typów wbudowanych. W Pythonie 3.X atrybuty te usunięto. Zamiast nich należy zastosować wbudowaną funkcję `dir()`.

- `X.__dict__`
Słownik wykorzystywany do przechowywania zapisywalnych atrybutów obiektu `X`.
- `I.__class__`
Obiekt klasy, na podstawie której wygenerowano egzemplarz `I`. W Pythonie w wersji 2.2 i późniejszych dotyczy to także typów obiektowych. Większość obiektów ma atrybut `__class__` (np. `[].__class__ == list == type([])`).
- `C.__bases__`
Krotka zawierająca nazwy klas bazowych klasy `C`, wymienionych w nagłówku instrukcji klasy `C`.
- `C.__mro__`
Wyliczona ścieżka MRO w drzewie klasy `C` nowego stylu (patrz „Klasy nowego stylu — MRO”).
- `X.__name__`
Nazwa obiektu `X` w postaci tekstowej. W przypadku klas jest to nazwa używana w nagłówkach instrukcji, a w przypadku modułów — nazwa wykorzystywana w operacjach importu. Dla modułu na najwyższym poziomie programu (np. głównego skryptu uruchamiającego program) jest to nazwa `"__main__"`.

Standardowe moduły biblioteczne

Standardowe moduły biblioteczne są zawsze dostępne, ale trzeba je zaimportować, by można było ich używać w modułach klienckich. Aby uzyskać do nich dostęp, należy skorzystać z jednego z poniższych formatów:

- `import moduł` i pobieranie nazw atrybutów (`moduł.nazwa`);
- `from moduł import nazwa` i używanie nazw modułu bez ich kwalifikowania (`nazwa`);
- `from moduł import *` i stosowanie nazw modułu bez ich kwalifikowania (`nazwa`).

Aby na przykład stosować nazwę `argv` z modułu `sys`, należy użyć instrukcji `import sys` i posługiwać się nazwą `sys.argv` lub instrukcji `from sys import argv` i korzystać z nazwy `argv`. Pierwsza, pełna forma — `moduł.nazwa` — jest używana tylko w nagłówkach zawartości list w celu zapewnienia kontekstu na wielostronicowych listingach. W opisach często jest wykorzystywana tylko *nazwa*.

Istnieje wiele standardowych modułów bibliotecznych. Podlegają one zmianom w co najmniej takim samym tempie jak sam język. W związku z tym lista modułów opisanych w poniższych podrozdziałach *nie jest wyczerpująca*. Opisano w nich jedynie najczęściej stosowane nazwy w powszechnie używanych modułach. Pełniejszy opis standardowych modułów bibliotecznych można znaleźć w dokumentacji *Python Library Reference*.

We wszystkich podrozdziałach z dokumentacją modułów:

- wymienione nazwy eksportów, za którymi występują nawiasy okrągłe, są *funkcjami* do wywołania; inne nazwy są prostymi atrybutami (tzn. nazwami zmiennych w modułach);
- opisano moduły *Pythona 3.X*, ale ogólnie rzecz biorąc, opis ma zastosowanie zarówno do wersji 3.X, jak i 2.X, z wyjątkiem tych miejsc, gdzie zaznaczono inaczej. Informacje na temat różnic pomiędzy wersjami można znaleźć w dokumentacji Pythona.

Moduł sys

Moduł *sys* zawiera *narzędzia związane z interpreterem* — konstrukcje dotyczące interpretera albo jego procesu zarówno w Pythonie 3.X, jak i 2.X. Oprócz tego daje dostęp do pewnych komponentów środowiskowych, takich jak wiersz polecenia, standardowe strumienie itp. Więcej informacji na temat narzędzi związanych z procesami można znaleźć w opisie modułu *os* w punkcie „Moduł systemowy *os*”.

`sys.argv`

Lista łańcuchów znaków użytych w wierszu polecenia: [*nazwa_skryptu*, *argumenty*...]. Podobnie jak w przypadku tablicy `argv` w języku C, element `argv[0]` oznacza nazwę pliku skryptu (ewentualnie z pełną ścieżką); łańcuch `'-c'` dla opcji wiersza polecenia `-c`; nazwę ścieżki modułu dla opcji `-m`; ciąg `'-'` dla opcji `-`; a pusty łańcuch znaków, jeśli nie przekazano nazwy skryptu lub opcji polecenia. Patrz także „Specyfikacja programu w wierszu polecenia”.

`sys.byteorder`

Oznacza natywny porządek bajtów (np. `'big'` w przypadku platform `big-endian`, `'little'` dla platform `little-endian`).

`sys.builtin_module_names`

Krotka składająca się z tekstowych nazw modułów języka C wkompirowanych w interpreter Pythona.

`sys.copyright`

Łańcuch znaków zawierający notkę o prawach autorskich do Pythona.

`sys.displayhook(wartość)`

Wywoływana przez Pythona w celu wyświetlenia wyników w sesjach interaktywnych. Aby dostosować wyjście do indywidualnych potrzeb, należy przypisać `sys.displayhook` do funkcji jednoargumentowej.

`sys.dont_write_bytecode`

Jeśli ten atrybut zostanie ustawiony na `true`, Python nie będzie zapisywał plików `.pyc` lub `.pyo` podczas importowania modułów źródłowych (patrz też opis opcji wiersza polecenia `-B` w podrozdziale „Opcje wiersza poleceń Pythona”).

`sys.excepthook(typ, wartość, ślad)`

Wywoływana przez Pythona w celu wyświetlenia szczegółowych informacji o nieobsłużonych wyjątkach na standardowym urządzeniu błędu. Aby dostosować sposób wyświetlania wyjątków do indywidualnych potrzeb, należy przypisać `sys.excepthook` do funkcji trójargumentowej.

`sys.exc_info()`

Zwraca krotkę złożoną z trzech wartości opisujących aktualnie obsługiwany wyjątek: (*typ*, *wartość*, *ślad*), gdzie *typ* oznacza klasę wyjątku, *wartość* jest egzemplarzem klasy zgłaszanego wyjątku, natomiast *ślad* jest obiektem gwarantującym dostęp do stosu wywołań w postaci, jaką miał w momencie wystąpienia wyjątku. Wartość specyficzna dla bieżącego wątku. W Pythonie 1.5 i wersjach późniejszych obejmuje funkcje `exc_type`, `exc_value` oraz `exc_traceback` (wszystkie trzy usunięto w Pythonie 3.X). Informacje dotyczące posługiwania się obiektami śladu można znaleźć w opisie modułu `traceback` w dokumentacji *Python Library Reference* oraz w podrozdziale „Instrukcja try”.

`sys.exec_prefix`

Atrybut ten należy przypisać do łańcucha znaków zawierającego informacje na temat prefiksu katalogu, w którym zainstalowano pliki Pythona charakterystyczne dla platformy. Domyślną wartością jest `/usr/local` lub argument podany w czasie kompilacji. Atrybut można wykorzystać do lokalizowania współdzielonych modułów bibliotecznych (z katalogu `<exec_prefix>/lib/python<version>/lib-dynload`) oraz plików konfiguracyjnych.

`sys.executable`

Łańcuch znaków zawierający pełną ścieżkę dostępu do programu interpretera Pythona wykonującego program wywołujący ten atrybut.

`sys.exit([N])`

Kończy działanie procesu Pythona, zwracając status *N* (domyślnie 0) poprzez zgłoszenie wbudowanego wyjątku `SystemExit` (można go przechwycić w instrukcji `try` i zignorować). Więcej informacji na temat szczegółów użycia można znaleźć w opisie wyjątku `SystemExit` (w podrozdziale „Wbudowane wyjątki”). W opisie funkcji `os.exit()` (w podrozdziale „Moduł systemowy `os`”) można się zapoznać z podobnym narzędziem, które powoduje natychmiastowe zakończenie działania bez przetwarzania wyjątków (przydatne do obsługi procesów potomnych po wywołaniu `os.fork()`). Warto się również zapoznać z opisem modułu `atexit` ze standardowej biblioteki Pythona, w której można znaleźć ogólną specyfikację funkcji `exit`.

`sys.flags`

Wartości opcji wiersza poleceń Pythona — po jednym atrybucie na opcję (zobacz dokumentacja Pythona).

`sys.float_info`

Wyświetla szczegółowe informacje dotyczące implementacji operacji zmiennoprzecinkowych w Pythonie za pośrednictwem zbioru atrybutów (patrz dokumentacja Pythona).

`sys.getcheckinterval()`

W Pythonie 3.1 i wersjach wcześniejszych zwraca „interwał sprawdzania” interpretera (zobacz `setcheckinterval()` dalej na tej liście). W Pythonie 3.2 i wersjach późniejszych funkcję tę zastąpiła funkcja `getswitchinterval()`.

`sys.getdefaultencoding()`

Zwraca nazwę bieżącego domyślnego kodowania tekstów, używanego w implementacji Unicode.

`sys.getfilesystemencoding()`

Zwraca nazwę kodowania używanego do konwersji nazw plików Unicode na systemowe nazwy plików lub `None`, w przypadku gdy użyto domyślnego kodowania w systemie.

`sys._getframe([głębokość])`

Zwraca obiekt ramki ze stosu wywołań Pythona (patrz *Python Library Reference*).

`sys.getrefcount(obiekt)`

Zwraca bieżącą wartość licznika odwołań (ang. *reference count*) *obiektu* — +1 dla argumentu wywołania.

`sys.getrecursionlimit()`

Zwraca maksymalny limit głębokości stosu wywołań Pythona. Zobacz też `setrecursionlimit()` dalej na tej liście.

`sys.getsizeof(obiekt [, domyślnie])`

Zwraca rozmiar *obiektu* w bajtach. Argument *obiekt* może być obiektem dowolnego typu. Wszystkie wbudowane obiekty zwracają prawidłowe wyniki, natomiast zewnętrzne rozszerzenia są zależne od implementacji. Argument *domyślnie* określa wartość, która zostanie zwrócona, w przypadku gdy typ obiektowy nie implementuje interfejsu pobierania rozmiaru.

`sys.getswitchinterval()`

W Pythonie 3.2 i nowszych wersjach zwraca ustawienie „interwału sprawdzania” bieżącego wątku (zobacz `setswitchinterval()` dalej na tej liście). W Pythonie 3.1 i wersjach wcześniejszych należy używać funkcji `getcheckinterval()`.

`sys.getwindowsversion()`

Zwraca obiekt opisujący bieżącą wersję systemu Windows (patrz dokumentacja Pythona).

`sys.hexversion`

Numer wersji Pythona zakodowany w postaci pojedynczej liczby całkowitej (najlepiej przeglądać go za pomocą wbudowanej funkcji `hex()`). Numer zwiększa się wraz z każdym nowym wydaniem.

`sys.implementation`

Wprowadzony w Pythonie 3.3 obiekt udostępniający informacje na temat implementacji działającego interpretera Pythona (nazwa, wersja itp.). Patrz dokumentacja Pythona.

`sys.int_info`

Wyświetla szczegółowe informacje dotyczące implementacji typu `integer` w Pythonie za pośrednictwem zbioru atrybutów (patrz dokumentacja Pythona).

`sys.intern(łańcuchznaków)`

Wprowadza *łańcuchznaków* do tablicy „łańcuchów internowanych” i zwraca ten łańcuch — albo sam łańcuch znaków, albo jego kopię. Internowanie łańcuchów znaków zapewnia niewielką poprawę wydajności operacji wyszukiwania w słownikach. Jeśli zarówno klucze słownika, jak i klucz wyszukiwania są interno-

wane, to porównywanie kluczy (po poddaniu ich działaniu funkcji mieszającej) można zrealizować poprzez porównanie wskaźników zamiast łańcuchów znaków. Zazwyczaj nazwy używane w programach Pythona są internowane automatycznie, a słowniki wykorzystywane do przechowywania atrybutów modułów, klas i instancji zawierają internowane klucze.

`sys.last_type`, `sys.last_value`, `sys.last_traceback`

Obiekty typu, wartości i śladu dla ostatniego nieprzechwyconego wyjątku (w większości przydatne do debugowania po awarii).

`sys.maxsize`

Liczba całkowita określająca maksymalną wartość, jaką może przyjąć zmienna typu `Py_ssize_t`. Zwykle jest to wartość $2^{31} - 1$ na platformie 32-bitowej oraz $2^{63} - 1$ na platformie 64-bitowej.

`sys.maxunicode`

Liczba całkowita oznaczająca największy obsługiwany kod znaku Unicode. W Pythonie 3.3 i wersjach późniejszych jest to zawsze wartość 1114111 (szesnastkowo `0x10FFFF`) ze względu na elastyczny system przechowywania łańcuchów znaków, które mają zmienny rozmiar. Przed wersją 3.3 wartość ta zależy od opcji konfiguracji, która określa, czy znaki Unicode są przechowywane jako UCS-2, czy jako UCS-4, i może wynosić `0xFFFF` lub `0x10FFFF`.

`sys.modules`

Słownik modułów, które są już załadowane. Każdemu modułowi odpowiada jedna para *nazwa:obiekt*. Wskazówka: słownik może się zmienić, co ma wpływ na przyszłe operacje importowania (na przykład instrukcja `del sys.modules['nazwa']` powoduje ponowne załadowanie modułu przy następnym imporcie).

`sys.path`

Lista łańcuchów znaków określających ścieżkę wyszukiwania podczas importowania modułów. Jest inicjalizowana na podstawie zmiennej powłoki `PYTHONPATH`, plików ścieżek *.pth* oraz dowolnych wartości domyślnych, zależnych od implementacji. Wskazówka: ten atrybut oraz związana z nim lista mogą się zmienić, co ma wpływ na przyszłe operacje importowania (np. instrukcja `sys.path.append('C:\\dir')` dynamicznie dodaje katalog do ścieżki wyszukiwania modułów).

Pierwsza pozycja — `path[0]` — oznacza katalog zawierający skrypt użyty do wywołania interpretera Pythona. Jeśli katalog skryptu jest niedostępny (np. jeśli interpreter zostanie wywołany w trybie

interaktywnym lub jeśli skrypt jest czytany ze standardowego wejścia), to `path[0]` oznacza pusty łańcuch znaków, co powoduje, że Python najpierw poszukuje modułów w bieżącym katalogu roboczym. Katalog skryptu jest wstawiany przed elementami pochodzącymi ze zmiennej środowiskowej `PYTHONPATH`. Patrz także „Instrukcja `import`”.

`sys.platform`

Łańcuch znaków określający system, w którym działa Python, np.: 'win32', 'darwin', 'linux2', 'cygwin', 'os2', 'freebsd8', 'sunos5', 'PalmOS3' itp. Przydatny do wykonywania testów w kodzie zależnym od platformy.

Wartość 'win32' oznacza wszystkie współczesne odmiany systemu Windows. Jednak dla przypadku ogólnego wartość tę można sprawdzać za pomocą instrukcji `sys.platform[:3]=='win'` lub `sys.platform.startswith('win')`. Począwszy od Pythona 3.3, wszystkie platformy linuksowe zwracają ciąg 'linux', ale w skryptach, podobnie jak dla systemu Windows, wartość tę można sprawdzać za pomocą instrukcji `str.startswith('linux')`, ponieważ wcześniej były stosowane stałe 'linux2' lub 'linux3'.

`sys.prefix`

Atrybut ten należy przypisać do łańcucha znaków zawierającego informacje na temat prefiksu katalogu, w którym zainstalowano pliki Pythona charakterystyczne dla platformy. Domyślną wartością jest `/usr/local` lub argument podany w czasie kompilacji. Moduły biblioteczne Pythona są instalowane w katalogu `<prefiks>/lib/python<wersja>`; niezależne od platformy pliki nagłówkowe są przechowywane w katalogu `<prefiks>/include/python<wersja>`.

`sys.ps1`

Łańcuch znaków określający podstawowy symbol zachęty w trybie interaktywnym. Jeśli do tego atrybutu nie zostanie przypisana specyficzna wartość, to domyślnie jest używany symbol `>>>`.

`sys.ps2`

Łańcuch znaków określający dodatkowy symbol zachęty, używany do kontynuacji złożonych instrukcji w trybie interaktywnym. Jeśli do tego atrybutu nie zostanie przypisana specyficzna wartość, to domyślnie jest używany symbol `...`.

`sys.setcheckinterval(reps)`

W Pythonie 3.2 i wersjach późniejszych zastąpione funkcją `setswit` `chinterval()` (opisaną na tej liście). W wersji 3.2 i wersjach nowszych funkcja ta jest nadal obecna, ale nie ma znaczenia, ponieważ

implementacja przełączania wątków i zadań asynchronicznych została przepisana.

W Pythonie 3.1 i wersjach wcześniejszych za pomocą tego wywołania można ustawić częstotliwość *reps*, z jaką interpreter wykonuje okresowe zadania (np. przełączanie wątków, uruchamianie handlerów sygnałów). Wartość ta jest mierzona w liczbie wirtualnych instrukcji maszynowych (domyślnie 100). Ogólnie rzecz biorąc, instrukcja Pythona jest przekształcana na wiele instrukcji maszyny wirtualnej. Niższe wartości argumentu zwiększają responsywność wątków, ale jednocześnie zwiększają koszty związane z ich przełączaniem.

`sys.setdefaultencoding(nazwa_kodowania)`

Metoda została usunięta w Pythonie 3.2. Wywołanie służące do ustawiania bieżącego, domyślnego kodowania tekstów używanego w implementacji Unicode na *nazwa_kodowania*. Przeznaczone do wykorzystania przez moduł *site*. Metoda ta jest dostępna tylko podczas uruchamiania.

`sys.setprofile(funkcja)`

Wywołanie służy do ustawiania funkcji profilowania systemu na *funkcja* — „hak” systemu profilowania (nie jest uruchamiany dla każdego wiersza). Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference*.

`sys.setrecursionlimit(głębokość)`

Wywołanie ustawia maksymalną głębokość stosu wywołań Pythona na wartość *głębokość*. Ograniczenie to zapobiega przepełnieniu stosu C i awarii Pythona w wyniku nieskończonej rekurencji. Domyślnie głębokość wynosi 1000 w systemie Windows, ale wartość ta może być też inna. Dla funkcji głęboko rekurencyjnych mogą być wymagane wyższe wartości.

`sys.setswitchinterval(interwał)`

W Pythonie 3.2 i wersjach późniejszych ustawia interwał przełączania wątków interpretera na wartość *interwał* wyrażoną w sekundach. Jest to wartość zmiennoprzecinkowa (np. 0.005 oznacza 5 milisekund), która określa idealną długość odcinków czasu przydzielonych do uruchomionych równolegle wątków Pythona. Rzeczywista wartość może być wyższa, zwłaszcza jeśli stosuje się długo działające funkcje wewnętrzne lub metody, a wybór wątków zaplanowanych na końcu przedziału jest dokonywany przez system operacyjny (interpreter Pythona nie posiada własnego mechanizmu szeregowania).

W Pythonie 3.1 i wersjach wcześniejszych zamiast tego wywołania należy używać funkcji `setcheckinterval()` (opisanej na tej liście).

`sys.settrace(funkcja)`

Wywołanie służy do ustawiania funkcji śladu systemu na *funkcja*. Jest to lokalizacja programu lub używany przez debuggery „hak” do wywołania zwrotnego dla zmian lokalizacji bądź stanu programu. Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference*.

`sys.stdin`

Standardowy strumień wejściowy — predefiniowany otwarty obiekt pliku *stdin*. Można go przypisać do dowolnego obiektu z implementacją metod `read` w celu ustawienia strumienia wejściowego z poziomu skryptu (np. `sys.stdin=MyObj()`). Strumień jest używany w operacjach wejścia interpretera, w tym przez funkcję wbudowaną `input()` (oraz `raw_input()` w Pythonie 2.X).

`sys.stdout`

Standardowy strumień wyjściowy — predefiniowany otwarty obiekt pliku *stdout*. Można go przypisać do dowolnego obiektu z implementacją metod `write` w celu ustawienia strumienia wyjściowego z poziomu skryptu (np. `sys.stdout=open('log', 'a')`). Używany do wyświetlania niektórych komunikatów, wykorzystywany przez wbudowaną funkcję `print()` (i instrukcję `print` w Pythonie 2.X). Jeśli jest taka potrzeba, to można użyć zmiennej środowiskowej `PYTHONIOENCODING` w celu przesłonięcia kodowania zależnego od platformy (patrz „Zmienne środowiskowe”) oraz opcji `-u` dla strumieni niebuforowanych (patrz „Opcje wiersza poleceń Pythona”).

`sys.stderr`

Standardowy obiekt pliku — domyślnie podłączony do standardowego strumienia błędów *stderr*. Wewnątrz skryptu można go przypisać do dowolnego obiektu z implementacją metod `write` (np. `sys.stderr=opakowanegniazdo`). Używany do wyświetlania komunikatów (błędów) interpretera.

`sys.__stdin__, sys.__stdout__, sys.__stderr__`

Początkowe wartości `stdin`, `stderr` i `stdout` w momencie uruchamiania programu (mogą służyć do przywrócenia domyślnych wartości; zwykle podczas przypisywania wartości np. do `sys.stdout` należy zapisać starą wartość i odtworzyć ją wewnątrz klauzuli `finally`). Uwaga: mogą mieć wartość `None` dla aplikacji GUI w systemie Windows bez konsoli.

`sys.thread_info`

Wyświetla szczegółowe informacje dotyczące implementacji wątków w Pythonie za pośrednictwem zbioru atrybutów. Nowość wprowadzona w Pythonie 3.3 (patrz dokumentacja Pythona).

`sys.tracebacklimit`

Maksymalna liczba poziomów śladu wyświetlanego dla nieobsłużonych wyjątków. Domyślnie ma wartość 1000, o ile nie zostanie przypisana.

`sys.sys.version`

Łańcuch znaków zawierający numer wersji interpretera Pythona.

`sys.version_info`

Krotka zawierająca pięć komponentów identyfikacji wersji: numer główny, numer pomocniczy, numer mikro, poziom wydania oraz poprawkę. W przypadku Pythona 3.0.1 ma wartość (3, 0, 1, 'final', 0). W najnowszych wydaniach jest to krotka identyfikowana przez nazwę. Dostęp do jej elementów można uzyskać poprzez elementy krotki lub przez nazwy atrybutów. Dla Pythona 3.3.0 wywołanie `sys.version_info` powoduje wyświetlenie wartości `sys.version_info(major=3, minor=3, micro=0, releaselevel='final', serial=0)`. Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference*.

`sys.winver`

Numer wersji używany do tworzenia kluczy rejestru na platformach windowsowych (dostępny tylko w systemie Windows; patrz podręcznik *Python Library Reference*).

Moduł string

Moduł `string` definiuje stałe i zmienne do przetwarzania *obiektów tekstowych*. Patrz także podrozdział „Ciągi znaków” — można tam znaleźć opis narzędzi `Template` i `Formatter` z modułu `string`, służących do zastępowania szablonów w łańcuchach znaków i formatowania.

Funkcje i klasy modułu

Począwszy od Pythona 2.0, większość funkcji z modułu `string` jest także dostępna w postaci *metod* obiektów `string`. Wywołania metod są preferowane i wydajniejsze. Są one preferowane w wersji 2.X, natomiast w Pythonie 3.X jest to jedyna dostępna opcja. Więcej informacji na ten temat oraz listę wszystkich dostępnych metod obiektów `string`,

których tutaj nie powtórzono, można znaleźć w podrozdziale „Ciągi znaków”. Poniżej zestawiono wyłącznie elementy unikatowe dla modułu `string`.

`string.capitalize(s, sep=None)`

Dzieli argument na słowa z wykorzystaniem metody `s.split()`, zamienia pierwszą literę w każdym słowie na wielką za pomocą metody `s.capitalize()` oraz scala słowa pisane wielką literą za pomocą metody `s.join()`. W przypadku braku opcjonalnego argumentu `sep` lub jeśli ma on wartość `None`, ciągi białych spacji są zastępowane pojedynczą spacją, a wiodące i końcowe znaki białych spacji są usuwane. W przeciwnym razie do dzielenia i scalania słów używany jest argument `sep`.

`string.maketrans(z, na)`

Zwraca tabelę translacji, którą można przekazać do metody `bytes.translate()`. Tabela ta odwzorowuje każdy znak z argumentu `z` na znak z tej samej pozycji w argumentcie `na`; argumenty `z` oraz `na` muszą być tej samej długości.

`string.Formatter`

Klasa pozwalająca na tworzenie własnych obiektów formatujących wykorzystujących te same mechanizmy co metoda `str.format()`, która została opisana w sekcji „Metoda formatująca” w podrozdziale „Ciągi znaków”.

`string.Template`

Klasa udostępniająca mechanizm zastępowania szablonów, opisany w podrozdziale „Zastępowanie szablonów w łańcuchach znaków”.

Stałe

`string.ascii_letters`

Łańcuch znaków `ascii_lowercase + ascii_uppercase`.

`string.ascii_lowercase`

Łańcuch znaków `'abcdefghijklmnopqrstuvwxyz'`; niezależny od ustawień językowych.

`string.ascii_uppercase`

Łańcuch znaków `'ABCDEFGHIJKLMNOPQRSTUVWXYZ'`; niezależny od ustawień językowych.

`string.digits`

Łańcuch znaków `'0123456789'`.

`string.hexdigits`
Łańcuch znaków '0123456789abcdefABCDEF'.

`string.octdigits`
Łańcuch znaków '01234567'.

`string.printable`
Kombinacja stałych `digits`, `ascii_letters`, `punctuation` i `whitespace`.

`string.punctuation`
Łańcuch znaków uznawanych za przestankowe w określonych ustawieniach regionalnych.

`string.whitespace`
Łańcuch znaków zawierający spację, znak wysuwu wiersza, wysuwu strony oraz tabulację pionową: ' \t\n\r\v\f'.

Moduł systemowy os

Moduł `os` jest podstawowym interfejsem usług *systemu operacyjnego* (OS) zarówno w Pythonie 3.X, jak i 2.X. Zapewnia ogólną obsługę systemów operacyjnych, a także standardowy, niezależny od platformy interfejs usług systemu operacyjnego. Moduł `os` zawiera narzędzia do obsługi środowiska, procesów, plików, poleceń powłoki i wiele innych. Zawiera również zagnieżdżony moduł podrzędny `os.path`, który dostarcza przenośny interfejs do narzędzi przetwarzania katalogów.

Skrypty korzystające z modułów `os` i `os.path` do programowania systemowego są, ogólnie rzecz biorąc, *przenośne* na większości platform Pythona. Niektóre eksporty modułu `os` nie są jednak dostępne na wszystkich platformach (np. funkcja `fork()` jest dostępna na platformach Unix i Cygwin, ale nie ma jej w standardowej, windowsowej wersji Pythona). Ze względu na to, że przenośność tych wywołań może się z czasem zmieniać, warto sięgnąć do podręcznika *Python Library Reference* w celu uzyskania szczegółowych informacji, właściwych dla wybranej platformy.

Poniżej opisano często wykorzystywane narzędzia z tego modułu. *Jest to lista częściowa* — w dokumentacji standardowej biblioteki Pythona można znaleźć szczegółowe informacje dotyczące narzędzi dostępnych w tym module. Na niektórych platformach jest ich ponad 200. Można tam również znaleźć informacje dotyczące różnic pomiędzy platformami i wersjami, które tutaj pominięto. W poszczególnych podpunktach zestawiono duże obszary funkcjonalne modułu `os`:

- *Narzędzia administracyjne* — eksporty dotyczące modułu.
- *Stałe wykorzystywane do zapewnienia przenośności* — stałe do przeszukiwania ścieżek i katalogów.
- *Polecenia powłoki* — uruchamianie poleceń i skryptów.
- *Narzędzia do obsługi środowiska* — środowisko uruchomieniowe wraz z jego kontekstem.
- *Narzędzia do obsługi deskryptorów plików* — przetwarzanie plików na podstawie przekazanych deskryptorów.
- *Narzędzia do obsługi nazw ścieżek* — przetwarzanie plików na podstawie nazw ścieżek.
- *Zarządzanie procesami* — tworzenie procesów i zarządzanie nimi.
- *Moduł os.path* — usługi dotyczące nazw ścieżek.

Warto się również zapoznać z opisem *powiązanych modułów systemowych* w standardowej bibliotece Pythona. Jeśli nie zaznaczono inaczej, ich opisu należy szukać w dokumentacji Pythona: `sys` — narzędzia obsługi procesu interpretera (patrz „Moduł `sys`”); `subprocess` — zarządzanie procesami (patrz „Moduł `subprocess`”); `threading` i `queue` — narzędzia do obsługi wielowątkowości (patrz „Moduły obsługi wątków”); `socket` — obsługa sieci i IPC (patrz „Moduły i narzędzia do obsługi internetu”); `glob` — rozszerzenia nazw plików (np. `glob('*.*py')`); `tempfile` — obsługa plików tymczasowych; `signal` — obsługa sygnałów; `multiprocessing` — API dla procesów, podobne do tego, które jest dostępne dla wątków; `getopt`, `optparse` oraz, w wersji 3.2 i późniejszej, `argparse` — przetwarzanie wiersza polecenia.

Narzędzia administracyjne

Poniżej zestawiono różne eksporty dotyczące modułu.

`os.error`

Alias wbudowanego wyjątku `OSError` — patrz „Wbudowane wyjątki”. Zgłaszany w momencie wystąpienia błędów modułu `os`. Wyjątek ma dwa atrybuty: `errno` — liczbowy kod błędu zgodny z POSIX (np. wartość zmiennej `errno` języka C), oraz `strerror` — odpowiadający mu komunikat o błędzie dostarczany przez system operacyjny i formatowany zgodnie z właściwą funkcją C (np. dostarczany przez funkcję `perror()` i formatowany zgodnie z funkcją `os.strerror()`). W przypadku wyjątków związanych z nazwą

ścieżki do pliku (np. `chdir()`, `unlink()`) egzemplarz wyjątku zawiera też atrybut `filename` — przekazaną nazwę pliku. Informacje dotyczące nazw kodów błędów zdefiniowanych dla określonego systemu operacyjnego można znaleźć w opisie modułu `errno` w dokumentacji *Python Library Reference*.

`os.name`

Nazwa modułów specyficznych dla systemu operacyjnego. Ich nazwy są kopiowane na najwyższy poziom modułu `os` (np. `posix`, `nt`, `mac`, `os2`, `ce` lub `java`). Patrz także opis wyjątku `sys.platform` w podrozdziale „Moduł `sys`”.

`os.path`

Zagnieżdżony moduł zawierający przenośne narzędzia do przetwarzania nazw ścieżek. Na przykład `os.path.split()` to niezależne od platformy narzędzie do przetwarzania nazw katalogów, które wewnętrznie wykorzystuje odpowiednie wywołanie specyficzne dla platformy.

Stałe wykorzystywane do zapewnienia przenośności

Poniżej opisano *narzędzia przenośności plików* dotyczące katalogów, ścieżek wyszukiwania, znaków wysuwu wiersza i innych. Są one automatycznie ustawiane na wartość odpowiednią dla platformy, w której działa skrypt. Są użyteczne zarówno podczas parsowania, jak i konstruowania łańcuchów zależnych od platformy. Patrz też „Moduł `os.path`”.

`os.curdir`

Łańcuch znaków reprezentujący bieżący katalog (np. `.` dla systemu Windows i standardu POSIX, `:` dla systemu Mac OS).

`os.pardir`

Łańcuch znaków reprezentujący katalog nadrzędny (np. `..` dla standardu POSIX, `::` dla systemu Mac OS).

`os.sep`

Łańcuch znaków do oddzielania katalogów (np. `/` dla systemu Unix, `\` dla Windowsa lub `:` dla Mac OS).

`os.altsep`

Alternatywny ciąg separatora lub `None` (np. `/` dla systemu Windows).

`os.extsep`

Znak oddzielający bazową nazwę pliku od rozszerzenia (np. `.`).

`os.pathsep`

Znak używany do oddzielania komponentów ścieżki wyszukiwania, używanych na przykład w zmiennych środowiskowych `PATH` i `PYTHONPATH` (np. `;` dla systemu Windows, `:` dla systemu Unix).

`os.defpath`

Domyślna ścieżka wyszukiwania używana w wywołaniach `os.exec*p*`, jeśli powłoka nie zawiera ustawienia `PATH`.

`os.linesep`

Łańcuch znaków używany do zakańczania wierszy na bieżącej platformie (np. `\n` dla standardu POSIX, `\r` dla systemu Mac OS oraz `\r\n` dla systemu Windows). Atrybutu tego nie należy używać podczas zapisywania wierszy w plikach trybu tekstowego — zamiast niego należałoby użyć mechanizmu automatycznej translacji sekwencji `'\n'` (patrz `open()` w podrozdziale „Funkcje wbudowane”).

`os.devnull`

Ścieżka plikowa do urządzenia „null” (dla tekstu, który ma być pominięty). Ma wartość `'/dev/null'` dla standardu POSIX oraz `'nul'` w przypadku systemu Windows (dostępna również w module pomocniczym `os.path`).

Polecenia powłoki

Poniższe funkcje uruchamiają *programy lub skrypty* we wskazanym systemie operacyjnym. W Pythonie 2.X w tym module są dostępne wywołania `os.popen2/3/4`. W Pythonie 3.X zastąpiono je wywołaniem `subprocess.Popen` — narzędziem, które — ogólnie rzecz biorąc — oferuje większą kontrolę nad poleceniami potomnymi (patrz „Moduł `subprocess`”). Wskazówka: nie należy używać tej instrukcji do sprawdzania niezauważalnych ciągów kodu, ponieważ są one uruchamiane tak samo jak kod programu.

`os.system(cmd)`

Uruchamia ciąg polecenia `cmd` jako podrzędny proces powłoki. Zwraca status wyjścia utworzonego procesu. W odróżnieniu od wywołania `popen()` nie łączy się ze standardowymi strumieniami polecenia `cmd` za pomocą potoków. Wskazówka: aby uruchomić polecenie w tle w systemie Unix, należy dodać znak `&` na końcu polecenia `cmd` (np. `os.system('python main.py &')`); aby łatwo uruchamiać programy w systemie Windows, należy skorzystać z polecenia start systemu DOS (np. `os.system('start file.html')`).

`os.startfile ścieżkadopliku)`

Uruchamia plik razem z aplikacją, która jest z nim powiązana. Wywołanie działa analogicznie do dwukrotnego kliknięcia nazwy pliku w Eksploratorze Windows lub do przekazania nazwy pliku jako argumentu do polecenia start systemu Windows (np. `os.system('start ścieżka')`). Plik jest otwierany w aplikacji, z którą powiązano rozszerzenie pliku. Wywołanie nie czeka na zakończenie działania aplikacji i, ogólnie rzecz biorąc, nie wyświetla okna konsoli systemu Windows (znanego również jako *Wiersz polecenia*). Jest dostępne wyłącznie w systemie Windows. Wprowadzono je w wersji 2.0.

`os.popen(cmd, mode='r', buffering=None)`

Otwiera potok do lub z polecenia powłoki *cmd* w celu przesłania bądź przechwycenia danych. Zwraca obiekt otwartego pliku, który można wykorzystać do czytania ze standardowego strumienia wyjściowego polecenia *cmd* — `stdout` (tryb 'r'), lub pisania do standardowego strumienia wejściowego polecenia *cmd* — `stdin` (tryb 'w'). Na przykład `dirlist = os.popen('ls -l *.py').read()` czyta wynik uniksowego polecenia `ls`.

Argumentem *cmd* może być dowolny ciąg polecenia możliwy do wpisania w konsoli systemowej lub powłoce. Argument *mode* może mieć wartość 'r' bądź 'w', a jego domyślną wartością jest 'r'. Argument *buffering* ma takie samo znaczenie jak we wbudowanej funkcji `open()`. Polecenie *cmd* działa samodzielnie. Jego status wyjścia jest zwracany przez metodę `close()` obiektu pliku. Jeśli status wyjścia wynosi 0 (brak błędów), zwracana jest wartość `None`. Do czytania wyjścia wiersz po wierszu (i ewentualnie przeplatania operacji w pełniejszy sposób) należy użyć wywołania `readline()` lub iteracji.

W Pythonie 2.X występują również odmiany wywołania: `popen2()`, `popen3()` i `popen4()`, pozwalające na łączenie się z innymi strumieniami uruchomionego polecenia (np. `popen2()` zwraca krotkę (*strumień_stdin_procesu_dziecka*, *strumień_stdout_procesu_dziecka*)). W Pythonie 3.X te wywołania usunięto. Zamiast nich należy używać funkcji `subprocess.Popen()`. Moduł `subprocess` w wersji 2.4 i w wersjach późniejszych pozwala skryptom na uruchamianie nowych procesów, łączenie się z ich potokami wejścia, wyjścia i błędów oraz uzyskiwanie kodów wynikowych (patrz „Moduł `subprocess`”).

`os.spawn*(argumenty...)`

Rodzina funkcji służących do uruchamiania programów i poleceń. Więcej informacji na ten temat można znaleźć w podrozdziale „Zarządzanie procesami” w dalszej części książki lub w podręczniku *Python Library Reference*. Moduł `subprocess` zapewnia alternatywę dla tych wywołań (zobacz „Moduł `subprocess`”).

Narzędzia do obsługi środowiska

Poniższe atrybuty eksportują *kontekst uruchomieniowy*: środowisko powłoki, bieżący katalog itp.

`os.environ`

Obiekt przypominający słownik zawierający zmienne środowiskowe. Odwołanie `os.environ['USER']` zwraca wartość zmiennej `USER` powłoki (odpowiednik odwołania `$USER` w systemie Unix oraz `%USER%` w systemie Windows). Słownik jest inicjowany w momencie uruchomienia programu. Zmiany wprowadzone w `os.environ` w wyniku przypisania wartości do kluczy są eksportowane na zewnątrz Pythona z wykorzystaniem wywołania `putenv()` języka C. Są one dziedziczone przez procesy uruchomione później w dowolny sposób, a także przez powiązany kod C. Więcej informacji na temat interfejsu `bytes` w Pythonie w wersji 3.2 i późniejszych można znaleźć w dokumentacji Pythona, w opisie wywołania `os.environb`.

`os.putenv(nazwazmienniej, wartość)`

Ustawia zmienną środowiskową o nazwie *nazwazmienniej* na tekstową *wartość*. Wpływa na procesy potomne uruchomione za pomocą wywołań `system()`, `popen()`, `spawnv()`, `fork()` oraz `execv()`. Instrukcje przypisania do kluczy słownika `os.environ` automatycznie wywołują funkcję `os.putenv()`, ale wywołania `os.putenv()` nie aktualizują atrybutu `os.environ`, dlatego preferowane jest wywołanie `os.environ`.

`os.getenv(nazwazmienniej, default=None)`

Zwraca wartość zmiennej środowiskowej *nazwazmienniej*, o ile ona istnieje. W przeciwnym razie zwraca wartość domyślną. W bieżącej wersji wywołanie to po prostu indeksuje ładowany domyślnie słownik środowiska za pomocą wywołania `os.environ.get(nazwa ↪ zmienniej, wartośćdomyślna).nazwazmienniej, wartośćdomyślna`, a wyniki są łańcuchami str. Informacje dotyczące reguł kodowania

Unicode zamieszczono w dokumentacji Pythona. Informacje dotyczące odpowiednika dla typu `bytes` dla wersji Pythona 3.2 można znaleźć w opisie wywołania `os.getenvb()`.

`os.getcwd()`

Zwraca nazwę bieżącego katalogu roboczego w postaci tekstowej.

`os.chdir(ścieżka)`

Zmienia bieżący katalog roboczy procesu na wartość przekazaną za pomocą argumentu *ścieżka* — ciągu oznaczającego nazwę katalogu. Kolejne operacje plikowe są wykonywane względem nowego katalogu roboczego. Wskazówka: wywołanie nie aktualizuje zmiennej `sys.path` wykorzystywanej do importów modułów, ale jego pierwsze użycie może być generycznym desygnatorem bieżącego katalogu roboczego.

`os.strerror(kod)`

Zwraca komunikat o błędzie odpowiadający kodowi błędu podanemu za pomocą argumentu *kod*.

`os.times()`

Zwraca krotkę składającą się z pięciu elementów z informacjami dotyczącymi czasu procesora procesu wywołującego. Elementy krotki są liczbami zmiennoprzecinkowymi oznaczającymi wykorzystany czas procesora w sekundach: (*czas-użytkownika*, *czas-systemu*, *czas-użytkownika-procesu-dziecka*, *czas-systemu-procesu-dziecka*, *czas-rzeczywisty*). Patrz też „Moduł `time`”.

`os.umask(maska)`

Ustawia liczbowy atrybut `umask` na wartość *maska* i zwraca poprzednią wartość.

`os.uname()`

Zwraca krotkę danych tekstowych opisujących system operacyjny: (*nazwasystemu*, *nazwawężła*, *wydanie*, *wersja*, *komputer*).

Narzędzia do obsługi deskryptorów plików

Wymienione poniżej funkcje przetwarzają *pliki na podstawie przekazanych deskryptorów* (*dp* jest liczbą całkowitą oznaczającą deskryptor pliku). Pliki bazujące na deskryptorach w module `os` służą do wykonywania niskopoziomowych zadań plikowych. Nie są to *obiekty plikowe* `stdio` zwracane przez wbudowaną funkcję `open()`. Do większości zadań przetwarzania plików należy używać obiektów plików, a nie deskryptorów. Więcej informacji na ten temat można znaleźć w opisie funkcji `open()` w podrozdziale „Funkcje wbudowane”. Jeśli jest taka potrzeba,

można skorzystać z wywołania `os.fdopen()` oraz metody obiektu pliku `fileno()` w celu dokonania konwersji pomiędzy dwoma formami reprezentacji plików. Wbudowana funkcja `open()` w Pythonie 3.X akceptuje deskryptory plików.

UWAGA

Określenie *deskryptory plików* w znaczeniu używanym w tym rozdziale różni się od „deskryptorów klas” (opis tych drugich można znaleźć w podrozdziale „Narzędzia do obsługi deskryptorów plików”). Należy również pamiętać, że w Pythonie 3.4 deskryptory plików *mają nie być domyślnie dziedziczone* w podprocesach. W tej wersji mają być wprowadzone nowe wywołania modułu `os`: `get_inheritable(dp)` i `set_inheritable(dp, Boolean)`, które mają zarządzać tym domyślnym ustawieniem.

`os.close(dp)`

Zamyka plik o deskrypcorze *dp* (argument *dp* nie jest obiektem *file*).

`os.dup(dp)`

Zwraca duplikat deskryptora pliku *dp*.

`os.dup2(dp, dp2)`

Kopiuje plik o deskrypcorze *dp* do pliku o deskrypcorze *dp2* (wcześniej zamyka plik *dp2*, jeśli jest on otwarty).

`os.fdopen(dp, *argumenty, **kwargs)`

Zwraca wbudowany *obiekt file* (`stdio`) powiązany z deskryptorem pliku *dp* (liczbą całkowitą). Alias dla wbudowanej funkcji `open()`. Akceptuje te same argumenty co funkcja `open()`, z tą różnicą, że pierwszym argumentem funkcji `fdopen()` zawsze musi być całkowitoliczbowy deskryptor pliku (patrz opis funkcji `open()` w podrozdziale „Funkcje wbudowane”). Konwersję pomiędzy plikami opisywanymi za pomocą deskryptorów a obiektami *file* zwykle przeprowadza się automatycznie za pomocą wbudowanej funkcji `open()`. Wskazówka: aby przekształcić obiekt *file* na deskryptor, należy użyć metody *obiektfile.fileno()*.

`os.fstat(dp)`

Zwraca status pliku o deskrypcorze *dp* (podobnie jak `stat()`).

`os.ftruncate(dp, rozmiar)`

Obcina plik odpowiadający deskryptorowi pliku *dp*, tak aby jego rozmiar wynosił co najwyżej *rozmiar* bajtów.

`os.isatty(dp)`

Zwraca `True`, w przypadku gdy plik o deskrytorze `dp` jest otwarty i połączony z urządzeniem typu `tty` (interaktywnym). W przeciwnym razie zwraca `False` (wywołanie może zwracać wartości 1 lub 0 w starszych wersjach Pythona).

`os.lseek(dp, pozycja, jak)`

Ustawia bieżącą pozycję pliku o deskrytorze `dp` na wartość `pozycja` (w celu realizacji losowego dostępu do pliku). Argument `jak` może mieć wartość 0 w celu ustawienia pozycji względem początku pliku, 1 — w celu ustawienia jej względem bieżącej pozycji, albo 2 — w celu ustawienia pozycji względem końca.

`os.open(nazwapliku, flagi [, mode=0o777], [dir_fd=None])`

Otwiera plik o deskrytorze `dp` i zwraca deskryptor (liczbę całkowitą, a nie obiekt `file` modułu `stdio`). Funkcja przeznaczona wyłącznie do wykonywania niskopoziomowych zadań przetwarzania plików. Nie działa tak samo jak wbudowana funkcja `open()`, która jest preferowana w przypadku większości operacji przetwarzania plików (patrz „Funkcje wbudowane”).

Argument `nazwapliku` oznacza łańcuch znaków opisujący nazwę ścieżki (może być względna). Argument `flagi` to maska bitowa — w celu połączenia zdefiniowanych w module `os` stałych, które odpowiadają flagom neutralnym dla platformy oraz specyficznym dla platformy (patrz tabela 18.), należy użyć operatora `|`. Domyślną wartością argumentu `mode` jest `0o777` (ósemkowo). Dla bieżącej wartości `umask` najpierw jest stosowana maska. Argument `dir_fd` został wprowadzony w Pythonie 3.3. Obsługuje ścieżki względne w stosunku do deskryptorów plikowych katalogów (patrz dokumentacja Pythona). Wskazówka: wywołanie `os.open()` może być użyte razem z flagą `os.O_EXCL` w celu zablokowania plików przed możliwością jednoczesnych aktualizacji lub przeprowadzania innych operacji związanych z synchronizacją procesów.

`os.pipe()`

Tworzy anonimowy potok. Patrz „Zarządzanie procesami”.

`os.read(dp, n)`

Czyta co najwyżej `n` bajtów z pliku o deskrytorze `dp` i zwraca te bajty w postaci łańcucha znaków.

`os.write(dp, str)`

Zapisuje wszystkie bajty łańcucha `str` do pliku o deskrytorze `dp`.

Tabela 18. Przykładowe flagi funkcji `os.open` (można je łączyć za pomocą funkcji `OR`)

<code>O_APPEND</code>	<code>O_EXCL</code>	<code>O_RDONLY</code>	<code>O_TRUNC</code>
<code>O_BINARY</code>	<code>O_NDELAY</code>	<code>O_RDWR</code>	<code>O_WRONLY</code>
<code>O_CREAT</code>	<code>O_NOCTTY</code>	<code>O_RSYNC</code>	
<code>O_DSYNC</code>	<code>O_NONBLOCK</code>	<code>O_SYNC</code>	

Narzędzia do obsługi nazw ścieżek

Wymienione poniżej funkcje *przetwarzają pliki* na podstawie ich nazw ścieżek (*ścieżka* to tekstowa postać ścieżki do pliku). Patrz też „Moduł `os.path`”. W Pythonie 2.X ten moduł zawiera również narzędzia do przetwarzania plików tymczasowych. W Pythonie 3.X przeniesiono je do modułu `tempfile`. Począwszy od Pythona w wersji 3.3, do niektórych spośród tych narzędzi dodano nowy, opcjonalny argument `dir_fd`, którego tu nie pokazano. Argument ten obsługuje ścieżki względem deskryptorów plikowych katalogów. Szczegółowe informacje można znaleźć w dokumentacji Pythona.

`os.chdir(ścieżka)`

`os.getcwd()`

Bieżący katalog roboczy. Patrz punkt „Narzędzia do obsługi środowiska”.

`os.chmod(ścieżka, tryb)`

Zmienia tryb pliku reprezentowanego przez argument *ścieżka* na numeryczny *tryb*.

`os.chown(ścieżka, uid, gid)`

Zmienia identyfikator właściciela (grupy) pliku reprezentowanego przez argument *ścieżka* na *uid/gid*.

`os.link(ścieżka-źródłowa, ścieżka-docelowa)`

Tworzy twarde dowiązanie o nazwie *ścieżka-docelowa* do pliku *ścieżka-źródłowa*.

`os.listdir(ścieżka)`

Zwraca listę nazw wszystkich plików zapisanych w katalogu *ścieżka*. Jest to szybka i przenośna alternatywa dla modułu `glob`. ➤ `glob(wzorzec)`, pozwalająca na uruchamianie poleceń wyświetlania zawartości katalogu z poziomu powłoki za pomocą mechanizmów dostępnych w module `os.popen()`. Patrz też moduł `glob` w dokumentacji Pythona. Można tam znaleźć opis rozwinięcia nazw plików. Warto również zajrzeć do opisu wywołania `os.walk()`

w dalszej części tego podrozdziału w celu zapoznania się ze sposobami przechodzenia przez pełne drzewo katalogów.

W Pythonie 3.X to wywołanie zwraca wartość bytes zamiast str w celu stłumienia dekodowania Unicode nazwy pliku zgodnie z domyślnymi ustawieniami platformy (to zachowanie ma również zastosowanie do wywołań `glob.glob()` oraz `os.walk()`). W Pythonie 3.2 i wersjach późniejszych argument *ścieżka*, jeśli zostanie pominięty, przyjmuje domyślną wartość `.` — co oznacza bieżący katalog roboczy.

`os.lstat(ścieżka)`

Tak jak `stat()`, ale nie uwzględnia dowiązań symbolicznych.

`os.mkfifo(ścieżka [, mode=0o666])`

Tworzy kolejkę FIFO (potok z nazwą) identyfikowaną przez ciąg *ścieżka* oraz o uprawnieniach dostępu określonych przez liczbowy argument *tryb* (ale jej nie otwiera). Domyślną wartością trybu jest 0o666 (ósemkowo). Do argumentu *tryb* najpierw jest stosowana maska (bieżąca wartość `umask`). W Pythonie 3.3 to wywołanie obsługuje dodatkowy argument kluczowy `dir_fd`.

Kolejki FIFO są potokami występującymi w systemie plików, które można otwierać i przetwarzać tak samo jak zwykłe pliki. Pozwalają one jednak na synchronizowany dostęp między niezależnie uruchomionymi klientami i serwerami za pomocą wspólnej nazwy. Kolejki FIFO istnieją do momentu, aż zostaną usunięte. Wywołanie to obecnie nie jest dostępne na platformach uniksowych (w tym na platformie Cygwin w systemie Windows). Jest jednak dostępne w standardowej, windowsowej wersji Pythona. Podobne cele często można osiągnąć za pomocą gniazd (patrz moduł `socket` w podrozdziale „Moduły i narzędzia do obsługi internetu” oraz w dokumentacji Pythona).

`os.mkdir(ścieżka [, tryb])`

Tworzy katalog o nazwie *ścieżka* oraz o podanym *trybie*. Domyślną wartością trybu jest 0o777 (ósemkowo).

`os.makedirs(ścieżka [, tryb])`

Funkcja do rekurencyjnego tworzenia katalogów. Działa podobnie jak `mkdir()`, ale tworzy wszystkie katalogi pośrednich poziomów wymagane do utworzenia katalogu-liścia. Zgłasza wyjątek, w przypadku gdy katalog-liść już istnieje lub jeśli nie można go utworzyć. Domyślną wartością argumentu *tryb* jest 0o777 (ósemkowo). W Pythonie 3.2 i wersjach późniejszych to wywołanie ma

dodatkowy opcjonalny argument `exists_ok`; więcej informacji można znaleźć w dokumentacji Pythona.

`os.readlink(ścieżka)`

Zwraca ścieżkę, do której odwołuje się dowiązanie symboliczne *ścieżka*.

`os.remove(ścieżka)`

`os.unlink(ścieżka)`

Usuwa plik o nazwie *ścieżka*. Funkcja `remove()` działa identycznie jak funkcja `unlink()`. Informacje dotyczące usuwania katalogów można znaleźć w opisie funkcji `rmdir()` i `removedirs()` w dalszej części tej listy.

`os.removedirs(ścieżka)`

Funkcja do rekurencyjnego usuwania katalogów. Działa podobnie jak `rmdir()`, ale w przypadku pomyślnego usunięcia katalogu-liścia kasowane są kolejne katalogi odpowiadające segmentom ścieżki — od prawej do lewej. Usuwanie trwa tak długo, aż zostanie wykasowana cała ścieżka albo wystąpi błąd. Zgłasza wyjątek, w przypadku gdy katalog-liść nie może być usunięty.

`os.rename(ścieżka-źródłowa, ścieżka-docelowa)`

Zmienia nazwę (przenosi plik) *ścieżka-źródłowa* na *ścieżka-docelowa*. Patrz też metoda `os.replace()`, począwszy od Pythona 3.3, w dokumentacji Pythona.

`os.renames(stara-ścieżka, nowa-ścieżka)`

Funkcja do rekurencyjnej zmiany nazwy katalogów lub plików. Działa podobnie jak funkcja `rename()`, ale najpierw próbuje utworzyć wszystkie katalogi pośrednie wymagane do stworzenia nowego katalogu. Po zmianie nazwy usuwane są kolejne katalogi odpowiadające segmentom ścieżki starej nazwy. Do tego celu wykorzystywana jest funkcja `removedirs()`.

`os.rmdir(ścieżka)`

Usuwa katalog o nazwie *ścieżka*.

`os.stat(ścieżka)`

Uruchamia wywołanie systemowe `stat` dla argumentu *ścieżka*. Zwraca krotkę złożoną z liczb całkowitych, opisujących niskopoziomowe informacje o plikach (elementy tej krotki są definiowane i przetwarzane przez narzędzia z modułu `stat`).

`os.symlink(ścieżka-źródłowa, ścieżka-docelowa)`

Tworzy dowiązanie symboliczne o nazwie *ścieżka-docelowa* do pliku *ścieżka-źródłowa*.

```
os.utime(ścieżka, (czas-dostępu, czas-modyfikacji))
```

Ustawia czasy dostępu i modyfikacji dla pliku *ścieżka*.

```
os.access(ścieżka, tryb)
```

Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference* lub na stronach podręcznika *man* systemu Unix.

```
os.walk(...)
    os.walk(korzeń
        [, topdown=True
        [, onerror=None]
        [, followlinks=False]]])
```

Generuje nazwy plików w drzewie katalogów poprzez przecho-
dzenie drzewa w kierunku góra – dół bądź dół – góra. Dla każ-
dego katalogu w drzewie o korzeniu w katalogu *korzeń* (włącznie
z nim samym) zwraca krotkę składającą się z trzech elementów:
(*ścieżka-do-katalogu*, *nazwy-katalogów*, *nazwy-plików*). Znaczenie
poszczególnych argumentów jest następujące:

- *ścieżka-do-katalogu* jest łańcuchem znaków opisującym ścieżkę do katalogów;
- *nazwy-katalogów* to lista nazw podkatalogów katalogu *ścieżka-do-katalogu* (z wyłączeniem katalogów *.* i *..*);
- *nazwy-plików* zawiera listę nazw plików w obrębie katalogu *ścieżka-do-katalogu*.

Należy zwrócić uwagę, że nazwy na listach nie zawierają kom-
ponentów ścieżek. Aby uzyskać pełną ścieżkę (rozpoczynającą
się od katalogu *korzeń*) do pliku lub katalogu w obrębie ścieżki
ścieżka-do-katalogu, należy skorzystać z wywołania `os.path.join`
↪(*ścieżka-do-katalogu*, *nazwa*).

Jeśli opcjonalny argument `topdown` ma wartość `true` lub jest nie-
określony, to trójka dla katalogu jest generowana *przed* stworze-
niem trójek dla jego podkatalogów (katalogi są tworzone w kie-
runku góra – dół). Jeśli argument `topdown` ma wartość `false`, to
trójka dla katalogu jest generowana *po* stworzeniu trójek dla jego
podkatalogów (katalogi są tworzone w kierunku dół – góra).
Opcjonalny argument `onerror` powinien oznaczać funkcję, która
będzie wywoływana z jednym argumentem — egzemplarzem
`os.error`. Domyślnie funkcja nie uwzględnia dowiązań symbo-
licznych prowadzących do katalogów. Aby odwiedzać katalogi
wskazywane przez dowiązania symboliczne (w systemach, które
je obsługują), należy ustawić opcję `followlinks` na `True`.

Kiedy argument `topdown` ma wartość `true`, lista *nazwy-katalogów* może być modyfikowana w miejscu w celu sterowania wyszukiwaniem. Wtedy wywołanie `os.walk()` będzie przeglądało rekurencyjnie tylko te podkatalogi, których nazwy są wymienione na liście *nazwy-katalogów*. Jest to przydatne do oczyszczenia wyszukiwania, narzucenia określonej kolejności przeglądania itp.

W Pythonie 2.X jest dostępne wywołanie `os.path.walk()` o podobnej funkcjonalności, przy czym zamiast generatora wykorzystywane jest wywołanie zwrotne handlerów zdarzeń. W Pythonie 3.X wywołanie `os.path.walk()` usunięto ze względu na jego nadmiarowość. Zamiast niego należy używać wywołania `os.walk()`. Patrz też moduł `glob` w dokumentacji Pythona. Można tam znaleźć opis rozwijania nazw plików, na przykład `glob.glob(r'***.py')`.

Zarządzanie procesami

Poniższe funkcje są wykorzystywane do tworzenia procesów oraz do *zarządzania procesami i programami*. O innych sposobach uruchamiania programów i plików można również przeczytać w podrozdziale „Polecenia powłoki”. Wskazówka: nie należy używać tej instrukcji do sprawdzania niezauważalnych ciągów kodu, ponieważ są one uruchamiane tak samo jak kod programu.

`os.abort()`

Wysyła sygnał SIGABRT do bieżącego procesu. W systemie Unix domyślnym działaniem jest utworzenie zrzutu pamięci. W systemie Windows proces natychmiast zwraca kod wyjścia o wartości 3.

`os.execl(ścieżka, arg0, arg1, ...)`

Odpowiednik funkcji `execv(ścieżka, (arg0, arg1, ...))`.

`os.execl(ścieżka, arg0, arg1, ..., środowisko)`

Odpowiednik funkcji `execve(ścieżka, (arg0, arg1, ...), środowisko)`.

`os.execlp(ścieżka, arg0, arg1, ...)`

Odpowiednik funkcji `execvp(ścieżka, (arg0, arg1, ...))`.

`os.execve(ścieżka, argumenty, środowisko)`

Działa podobnie jak funkcja `execv()`, ale słownik *środowisko* zastępuje zmienne środowiska powłoki. Argument *środowisko* musi odwzorowywać łańcuchy znaków na inne łańcuchy znaków.

`os.execvp(ścieżka, argumenty)`

Działa podobnie jak funkcja `execv(ścieżka, argumenty)`, ale powiela działania powłoki w zakresie wyszukiwania plików wykonywalnych na liście katalogów. Lista katalogów jest odczytywana za pomocą wywołania `os.environ['PATH']`.

`os.execvpe(ścieżka, argumenty, środowisko)`

Połączenie funkcji `execve()` i `execvp()`. Lista katalogów jest odczytywana za pomocą wywołania `os.environ['PATH']`.

`os.execv(ścieżka, argumenty)`

Uruchamia plik wykonywalny, identyfikowany przez parametr *ścieżka* z argumentami *argumenty*, zastępując w tym procesie bieżący program (interpreter Pythona). Parametr *argumenty* może być krotką albo listą łańcuchów znaków. Pierwszym jego elementem, zgodnie z konwencją, jest nazwa pliku wykonywalnego (`argv[0]`). Ta funkcja nigdy nie zwraca sterowania, o ile nie wystąpi błąd podczas uruchamiania nowego programu.

`os._exit(n)`

Natychmiast kończy działanie procesu, zwracając status *n* bez wykonywania operacji typowych dla kończenia programu. Zwykle używana wyłącznie w procesach potomnych po wykonaniu polecenia `fork`. Standardowym sposobem zakończenia procesu jest wywołanie `sys.exit(n)`.

`os.fork()`

Inicjuje proces potomny (uruchamianą równolegle wirtualną kopię procesu wywołującego). Zwraca 0 w procesie potomnym oraz nowy identyfikator procesu potomnego w procesie rodzica. Funkcja nie jest dostępna w standardowej windowsowej wersji Pythona, ale występuje w wersji Pythona działającej w Windowsie w środowisku Cygwin. Ogólnie rzecz biorąc, wywołania `popen()`, `system()`, `spawnv()` oraz funkcje z modułu `subprocess` są bardziej przenośne).

`os.getpid()`

`os.getppid()`

Zwraca identyfikator bieżącego procesu (wywołującego). Funkcja `getppid()` zwraca identyfikator procesu rodzica.

`os.getuid()`

`os.geteuid()`

Zwraca identyfikator użytkownika procesu. Funkcja `geteuid()` zwraca efektywny identyfikator użytkownika.

`os.kill(pid, sig)`

Wysyła sygnał *sig* do procesu o identyfikatorze *pid*, potencjalnie zabijając go (w przypadku niektórych typów sygnałów). Informacje na temat rejestrowania handlerów sygnałów oraz stałych opisujących sygnały można znaleźć w opisie modułu `signal` ze standardowej biblioteki Pythona.

`os.mkfifo(ścieżka [, tryb])`

Patrz podrozdział „Narzędzia do obsługi nazw ścieżek” (pliki używane do synchronizacji procesów).

`os.nice(inkrementacja)`

Dodaje parametr *inkrementacja* do atrybutu opisującego „uprzejmość” procesu (ang. *niceness*) — tzn. obniża priorytet, z jakim proces uzyskuje dostęp do procesora.

`os.pipe()`

Zwraca krotkę zawierającą deskryptory plików (*deskryptor_odczyt*, *deskryptor_zapis*) do czytania i zapisywania nowego, anonimowego (tzn. bez przypisanej nazwy) potoku. Funkcja używana do komunikacji pomiędzy procesami.

`os.plock(op)`

Blokuje segmenty programu w pamięci. Argument *op* (zdefiniowany w pliku `<sys./lock.h>`) określa segmenty, które są blokowane.

`os.spawnv(tryb, ścieżka, argumenty)`

Uruchamia program identyfikowany przez parametr *ścieżka* z argumentami *argumenty* w nowym procesie. Parametr *argumenty* może być listą lub krotką. Argument *tryb* jest stałą operacyjną złożoną z następujących nazw: `P_WAIT`, `P_NOWAIT`, `P_NOWAITO`, `P_OVERLAY` i `P_DETACH`. W systemie Windows funkcja ma przybliżone działanie do kombinacji funkcji `fork()+execv()` (funkcja `fork()` nie jest jeszcze dostępna w standardowej wersji Pythona dla systemu Windows, ale funkcje `popen()` i `system()` są). Alternatywne mechanizmy gwarantujące większe możliwości są dostępne w module `subprocess` (patrz podrozdział „Moduł subprocess”).

`os.spawnve(tryb, ścieżka, argumenty, środowisko)`

Działa podobnie jak funkcja `spawnv()`, ale przekazuje zawartość odwzorowania *środowisko*, w którym są zapisane zmienne środowiska powłoki uruchamianego programu.

`os.wait()`

Oczekuje na zakończenie procesu potomnego. Zwraca krotkę zawierającą identyfikator procesu potomnego oraz status wyjścia.

`os.waitpid(idp, opcje)`

Oczekuje na zakończenie procesu potomnego o identyfikatorze *idp*. Argument *opcje* ma wartość 0 w przypadku standardowego użytkowania lub `os.WNOHANG` w celu uniknięcia zawieszenia w sytuacji, gdy nie jest dostępny status procesu potomnego. Jeśli argument *idp* ma wartość 0, to żądanie jest stosowane do dowolnego procesu potomnego w grupie procesu bieżącego. Zobacz też funkcje sprawdzania statusu wyjścia procesów, udokumentowane w podręczniku *Python Library Reference* (np. funkcja `WEXITSTATUS` ↪ *(status)* pobierająca status wyjścia).

Moduł `os.path`

Moduł `os.path` dostarcza dodatkowe usługi związane z przetwarzaniem ścieżek dostępu do plików i katalogów. Mechanizmy te ułatwiają przenoszenie programów pomiędzy platformami. Jest to moduł zagnieżdżony — nazwy, które w nim występują, są zagnieżdżone w module `os` w obrębie modułu podrzędnego `os.path` (np. dostęp do funkcji `exists` można uzyskać poprzez zaimportowanie modułu `os` i użycie funkcji `os.path.exists`).

Większość funkcji występujących w tym module pobiera argument *ścieżka* — tekstową postać ścieżki do katalogu lub pliku (np. `'C:\\dir1\\spam.txt'`). Ścieżki dostępu do katalogów, ogólnie rzecz biorąc, są kodowane zgodnie z konwencjami obowiązującymi dla platformy, a w przypadku braku prefiksu do katalogu są wyznaczone względem bieżącego katalogu roboczego. Wskazówka: na większości platform w roli separatora katalogów można stosować ukośnik (/). W Pythonie 2.X moduł `os.path` zawiera funkcję `os.path.walk()`, którą w Pythonie 3.X zastąpiono funkcją `os.walk()` (patrz „Narzędzia do obsługi nazw ścieżek”).

`os.path.abspath(ścieżka)`

Zwraca znormalizowaną, bezwzględną wersję argumentu *ścieżka*. W większości platform funkcja ta jest odpowiednikiem funkcji `normpath(join(os.getcwd(), ścieżka))`.

`os.path.basename(ścieżka)`

Odpowiednik drugiej części pary zwracanej przez funkcję `split` ↪ *(ścieżka)*.

`os.path.commonprefix(lista)`

Zwraca najdłuższy prefiks ścieżki (znak po znaku), będący prefiksem wszystkich ścieżek na liście określonej przez argument *lista*.

- `os.path.dirname(ścieżka)`
Odpowiednik pierwszej części pary zwracanej przez funkcję `split` ↪(*ścieżka*).
- `os.path.exists(ścieżka)`
Zwraca `true`, jeśli łańcuch znaków *ścieżka* zawiera nazwę istniejącego pliku.
- `os.path.expanduser(ścieżka)`
Rozwija znak `~` w argumencie *ścieżka* na nazwę użytkownika i zwraca przetworzony łańcuch znaków.
- `os.path.expandvars(ścieżka)`
Rozwija zmienne środowiskowe `$` w argumencie *ścieżka* i zwraca przetworzony łańcuch znaków.
- `os.path.getatime(ścieżka)`
Zwraca czas ostatniego dostępu do pliku identyfikowanego przez argument *ścieżka* (liczba sekund, które upłynęły od 1 stycznia 1970 r. — tzw. epoki Unix; ang. *Unix epoch*).
- `os.path.getmtime(ścieżka)`
Zwraca czas ostatniej modyfikacji pliku identyfikowanego przez argument *ścieżka* (liczba sekund, które upłynęły od 1 stycznia 1970 r.).
- `os.path.getsize(ścieżka)`
Zwraca rozmiar pliku identyfikowanego przez argument *ścieżka* (w bajtach).
- `os.path.isabs(ścieżka)`
Zwraca `true`, jeśli łańcuch znaków *ścieżka* oznacza bezwzględną ścieżkę do pliku lub katalogu.
- `os.path.isfile(ścieżka)`
Zwraca `true`, jeśli łańcuch znaków *ścieżka* oznacza ścieżkę do zwykłego pliku (nie katalogu).
- `os.path.isdir(ścieżka)`
Zwraca `true`, jeśli łańcuch znaków *ścieżka* oznacza ścieżkę do katalogu.
- `os.path.islink(ścieżka)`
Zwraca `true`, jeśli łańcuch znaków *ścieżka* oznacza dowiązanie symboliczne.
- `os.path.ismount(ścieżka)`
Zwraca `true`, jeśli łańcuch znaków *ścieżka* oznacza punkt montowania.

`os.path.join(ścieżka1 [, ścieżka2 [, ...]])`

Inteligentnie scala jeden lub więcej komponentów ścieżki (uwzględniając specyficzne dla platformy konwencje separatorów).

`os.path.normcase(ścieżka)`

Normalizuje wielkość liter w nazwie ścieżki. W systemie Unix funkcja nie zmienia nazwy ścieżki. W systemach plików, w których małe litery nie są odróżniane od wielkich, zastępuje wszystkie litery małymi. W systemie Windows dodatkowo zastępuje separator / znakiem \.

`os.path.normpath(ścieżka)`

Normalizuje nazwę ścieżki. Usuwa redundantne separatory i odwołania do wyższego poziomu. W systemie Windows konwertuje znak / na \.

`os.path.realpath(ścieżka)`

Zwraca kanoniczną postać ścieżki określonej nazwy pliku. Eliminuje dowiązania symboliczne występujące w ścieżce.

`os.path.samefile(ścieżka1, ścieżka2)`

Zwraca true, jeśli oba argumenty odwołują się do tego samego pliku bądź katalogu.

`os.path.sameopenfile(plik1,plik2)`

Zwraca true, jeśli oba argumenty (obiekty file) odwołują się do tego samego pliku.

`os.path.samestat(stat1, stat2)`

Zwraca true, jeśli obie krotki *stat* odwołują się do tego samego pliku.

`os.path.split(ścieżka)`

Rozdziela argument *ścieżka* na krotkę (*głowa*, *ogon*), gdzie *ogon* oznacza ostatni komponent ścieżki, a *głowa* to wszystkie komponenty aż do komponentu *ogon*. Identyczną krotką jest (`dirname(ścieżka)`, `basename(ścieżka)`).

`os.path.splitdrive(ścieżka)`

Rozdziela argument *ścieżka* na parę ('napęd:', *ogon*) (w systemie Windows).

`os.path.splitext(ścieżka)`

Rozdziela argument *ścieżka* na krotkę (*nazwa_główna*, *rozszerzenie*), gdzie ostatni komponent elementu *nazwa_główna* nie zawiera kropki, a element *rozszerzenie* jest pusty bądź rozpoczyna się od kropki.

`os.path.walk(ścieżka, gość, dane)`

Alternatywa funkcji `os.walk()` dostępna tylko w Pythonie 2.X. Zamiast generatora katalogu bazuje na funkcji wywołania zwrotnego *gość* ze stanem *dane*. Usunięta z Pythona 3.X. W tej wersji należy używać wywołania `os.walk()`, a nie `os.path.walk()`.

Moduł dopasowywania wzorców re

Moduł `re` jest standardowym interfejsem *dopasowywania wyrażeń regularnych* zarówno w Pythonie 3.X, jak i 2.X. Wzorce wyrażeń regularnych (ang. *regular expressions* — *re*) oraz dopasowywane teksty mają postać łańcuchów znaków. Ten moduł musi być importowany.

Funkcje modułu

Interfejs najwyższego poziomu modułu zawiera narzędzia do natychmiastowego dopasowywania lub prekompilacji wzorców. Tworzy obiekty wzorca (*obiektyw*) i obiekt dopasowania (*obiekt d*), opisane w dalszej części.

`re.compile(wzorzec [, flagi])`

Kompiluje łańcuch znaków wyrażenia regularnego *wzorzec* na obiekt wyrażenia regularnego (*obiektyw*) w celu późniejszego dopasowywania. Argument *flagi* (można je łączyć za pomocą bitowego operatora `|`) zawiera poniższe elementy (dostępne na najwyższym poziomie modułu `re`):

`re.A` lub `re.ASCII`, lub `(?a)`

Powoduje, że w wyrażeniach `\w`, `\W`, `\b`, `\B`, `\s` oraz `\S` jest realizowane dopasowywanie tylko kodów ASCII zamiast pełnego dopasowywania Unicode. Ma to znaczenie tylko dla łańcuchów Unicode. W przypadku łańcuchów bajtowych jest ignorowane. Warto zwrócić uwagę, że dla zapewnienia zgodności wstecz w dalszym ciągu jest dostępna flaga `re.U` (podobnie jak jej synonim `re.UNICODE` oraz wbudowany odpowiednik `?u`), ale w Pythonie 3.X jest ona redundantna, ponieważ dopasowywanie łańcuchów znaków w tym przypadku jest domyślnie wykonywane w trybie Unicode (a dopasowywanie Unicode dla danych bajtowych nie jest dozwolone).

`re.I` lub `re.IGNORECASE`, lub `(?i)`

Dopasowywanie bez rozróżniania wielkich i małych liter.

re.L *lub* re.LOCALE, *lub* (?L)

Powoduje, że wzorce `\w`, `\W`, `\b`, `\B`, `\s`, `\S`, `\d` oraz `\D` zależą od bieżących ustawień języka (w Pythonie 3.X domyślnym trybem jest Unicode).

re.M *lub* re.MULTILINE, *lub* (?m)

Dopasowywanie każdego wiersza osobno zamiast całego łańcucha znaków.

re.S *lub* re.DOTALL, *lub* (?s)

Dopasowywanie *wszystkich* znaków, włącznie ze znakami nowego wiersza.

re.U *lub* re.UNICODE, *lub* (?u)

Powoduje, że wzorce `\w`, `\W`, `\b`, `\B`, `\s`, `\S`, `\d` oraz `\D` zależą od właściwości znaków Unicode (nowość w wersji 2.0; nadmiarowe w Pythonie 3.X).

re.X *lub* re.VERBOSE, *lub* (?x)

Ignoruje we wzorcu białe spacje (znaki spoza zestawów znaków).

re.match(worzec, łańcuch_znaków [, flagi])

Jeśli zero (bądź więcej) znaków na początku argumentu *łańcuch_znaków* jest zgodne ze wzorcem *worzec*, to funkcja zwraca egzemplarz właściwego obiektu dopasowania. W przypadku braku pasującego obiektu zwraca None.

re.search(worzec, łańcuch_znaków [, flagi])

Skanuje argument *łańcuch_znaków* w poszukiwaniu pasującego wzorca *worzec*. Zwraca egzemplarz pasującego obiektu *obiekt_d* lub None w przypadku braku dopasowania. Argument *flagi* ma takie samo znaczenie jak w przypadku funkcji `compile()`.

re.split(worzec, łańcuch_znaków [, maxsplit=0])

Dzieli argument *łańcuch_znaków* na wystąpienia wzorca *wzo* ➔ *rzec*. Jeśli we wzorcu zostaną użyte nawiasy `()`, to funkcja zwróci także wystąpienia wzorców *worzec* lub podwzorców.

re.sub(worzec, ciąg_zastępczy, łańcuch_znaków [, count=0])

Zwraca łańcuch znaków uzyskany w wyniku zastąpienia (pierwszych *count*) wystąpień wzorca *worzec* (łańcucha znaków lub obiektu wyrażenia regularnego) w argumencie *łańcuch_znaków* przez *ciąg_zastępczy*. Argument *ciąg_zastępczy* może być łańcuchem znaków lub funkcją wywoływaną

z argumentem w postaci pojedynczego egzemplarza obiektu dopasowania (*obiekt*). Funkcja ta musi zwrócić zastępczy łańcuch znaków. Argument *ciąg_zastępczy* może także zawierać sekwencje specjalne \1, \2 itp. w celu wykorzystania podciągów znaków pasujących do grup o wskazanych numerach lub \0 w przypadku uwzględnienia wszystkich.

`re.subn(wzorzec, ciąg_zastępczy, łańcuch_znaków [, count=0])`

Funkcja działa tak samo jak *sub*, ale zwraca krotkę (*nowy_łańcuch_znaków*, *liczba_wykonanych_zastąpień*).

`re.findall(wzorzec, łańcuch_znaków [, flagi])`

Zwraca listę łańcuchów znaków składającą się z wszystkich nienakładających się na siebie dopasowań wzorca *wzorzec* wewnątrz łańcucha *łańcuch_znaków*. Jeśli we wzorcu występuje jedna bądź więcej grup, funkcja zwraca listę grup.

`re.finditer(wzorzec, łańcuch_znaków [, flagi])`

Zwraca obiekt iterowalny po wszystkich nienakładających się na siebie dopasowaniach wzorca wyrażenia regularnego *wzorzec* wewnątrz łańcucha *łańcuch_znaków* (obiektów dopasowania).

`re.escape(łańcuch_znaków)`

Zwraca ciąg *łańcuch_znaków*, w którym wszystkie niealfanumeryczne znaki zostały poprzedzone znakiem lewego ukośnika. Dzięki temu zwrócony ciąg może być skompilowany jako literal łańcuchowy.

Obiekty wyrażeń regularnych

Obiekty wyrażeń regularnych (*obiektyw*) są zwracane przez funkcję `re.compile()`. Mają one wymienione poniżej atrybuty. Niektóre z nich tworzą obiekty dopasowania (*obiekt*):

obiektyw.flags

Argument *flagi* użyty w momencie kompilacji obiektu wyrażenia regularnego.

obiektyw.groupindex

Słownik składający się z elementów {nazwa-grupy: numer-grupy} we wzorcu.

obiektyw.pattern

Łańcuch znaków wzorca, na podstawie którego skompilowano obiekt wyrażenia regularnego.

```
obiektyw.match(łańcuch_znaków [, pozycja [, pozycja_końcowa]])
obiektyw.search(łańcuch_znaków [, pozycja [, pozycja_końcowa]])
obiektyw.split(łańcuch_znaków [, maxsplit=0])
obiektyw.sub(ciąg_zastępczy, łańcuch_znaków [, count=0])
obiektyw.subn(ciąg_zastępczy, łańcuch_znaków [, count=0])
obiektyw.findall(łańcuch_znaków [, pozycja [, pozycja_końcowa]])
obiektyw.finditer(łańcuch_znaków [, pozycja [, pozycja_końcowa]])
```

Działanie takie samo jak opisanych wcześniej funkcji modułu `re`, ale bez jawnego argumentu `wzorzec`. Argumenty `pozycja` i `pozycja_końcowa` zawierają indeksy początkowy i końcowy dopasowania. Pierwsze dwie funkcje mogą tworzyć obiekty dopasowania (*obiektd*).

Obiekty dopasowania

Obiekty dopasowania (*obiektd*) są zwracane w efekcie pomyślnego wykonania operacji `match()` i `search()`. Poniżej wymieniono ich atrybuty (informacje o innych atrybutach, których tu nie wymieniono, można znaleźć w podręczniku *Python Library Reference*).

obiektd.pos, *obiektd*.endpos

Wartości argumentów `pos` i `endpos` przekazanych do funkcji `search` lub `match`.

obiektd.re

Obiekt wyrażenia regularnego, którego metoda `match` lub `search` utworzyła dany obiekt dopasowania (patrz łańcuch `pattern`).

obiektd.string

Argument `string` przekazany do funkcji `match` lub `search`.

obiektd.group([*g* [, *g*]*)

Zwraca podciągi znaków dopasowane za pomocą grup wzorca wymienionych w nawiasach. Akceptuje zero lub więcej grup identyfikowanych za pomocą identyfikatorów *g* w postaci *liczb* lub *nazw*. Są one opisane odpowiednio przez wzorce (R) oraz (?P<*nazwa*>R). W przypadku gdy do funkcji zostanie przekazany jeden argument, wynikiem jej działania jest podciąg znaków pasujący do grupy, której numer przekazano. Jeśli przekazano wiele argumentów, wynik jest krotką zawierającą po jednym podciągu znaków na argument. W przypadku braku argumentów funkcja zwraca cały pasujący podciąg znaków. Jeśli numer dowolnej z grup wynosi zero, zwrócona wartość jest całym pasującym łańcuchem znaków. W przeciwnym razie funkcja zwraca łańcuch znaków pasujący do odpowiedniego numeru grupy we wzorcu. Grupy we wzorcu są numerowane od 1...N — od lewej do prawej.

`obietd.groups()`

Zwraca krotkę złożoną ze wszystkich grup dopasowania. Grupy, które nie partycypują w dopasowaniu, mają wartość `None`.

`obietd.groupdict()`

Zwraca słownik zawierający wszystkie nazwane podgrupy dopasowania. Kluczami w słowniku są nazwy podgrup.

`obietd.start([g]), obietd.end([g])`

Indeksy początku i końca podciągu znaków zgodnego z argumentem *g* (lub cały dopasowany łańcuch znaków, w przypadku gdy nie podano argumentu *g*). Dla obiektu dopasowania *M* zachodzi `M.string[M.start(g):M.end(g)] == M.group(g)`.

`obietd.span([g])`

Zwraca krotkę (`obietd.start(g)`, `obietd.end(g)`).

`obietd.expand(szablon)`

Zwraca łańcuch znaków uzyskany w wyniku zastąpienia sekwencji specjalnych w łańcuchu znaków *szablon*, w sposób podobny do metody `sub`. Sekwencje specjalne w postaci `\n` są konwertowane na właściwe znaki, natomiast odwołania liczbowe (`\1`, `\2`) oraz odwołania do nazw (`\g<1>`, `\g<nazwa>`) są zastępowane przez odpowiednie grupy.

Składnia wzorców

Łańcuchy znaków wzorców określa się za pomocą formuł konkatenacji (patrz tabela 19.), a także przy użyciu znakowych sekwencji specjalnych (patrz tabela 20.). Mogą także występować sekwencje specjalne Pythona (np. `\t` dla tabulacji). Łańcuchy znaków wzorców są dopasowywane do łańcuchów znaków tekstu. Wynikiem dopasowania jest wartość `Boolean`, a także pogrupowane podciągi znaków, pasujące do podwzorców umieszczonych w nawiasach:

```
>>> import re
>>> obiektw = re.compile('witaj,[ \t]*(.*)')
>>> obietd = obiektw.match('witaj, świecie!')
>>> obietd.group(1)
'swiecie!'
```

W tabeli 19. *C* oznacza dowolny znak. *R* to dowolna formuła wyrażenia regularnego w lewej kolumnie tabeli, natomiast *m* i *n* to liczby całkowite. Każda formuła zazwyczaj wykorzystuje jak największą część dopasowywanego łańcucha znaków. Wyjątkiem są tzw. formuły „niezachłanne” (ang. *nongreedy*), które wykorzystują jak najmniejszą część

Tabela 19. Składnia wzorców wyrażeń regularnych

Formuła	Opis
.	Odpowiada dowolnemu znakowi (jeśli użyto flagi DOTALL, to także ze znakami nowego wiersza).
^	Odpowiada początkowi łańcucha znaków (w trybie MULTILINE dotyczy wszystkich wierszy).
\$	Odpowiada końcowi łańcucha znaków (w trybie MULTILINE dotyczy wszystkich wierszy).
C	Każdy znak niebędący znakiem specjalnym pasuje do samego siebie.
R*	Zero lub więcej wystąpień poprzedzającego wyrażenia regularnego R (tyle, ile jest możliwe).
R+	Jedno lub więcej wystąpień poprzedzającego wyrażenia regularnego R (tyle, ile jest możliwe).
R?	Zero lub więcej wystąpień poprzedzającego wyrażenia regularnego R.
R{m}	Odpowiada dokładnie m powtórzeniom poprzedzającego wyrażenia regularnego R.
R{m,n}	Odpowiada powtórzeniom poprzedzającego wyrażenia regularnego R — od numeru m do n.
R*?, R+?, R??, R{m,n}?	To samo co *, + i ?, ale dopasowują jak najmniejszą liczbę znaków (wystąpień) — tzw. wyrażenia <i>niezachłanne</i> .
[...]	Definiuje zbiór znaków, np. [a-zA-Z] oznacza wszystkie litery (patrz też tabela 20.).
[^...]	Definiuje uzupełnienie zbioru znaków — pasuje do znaków spoza zbioru.
\	Unieszkodliwia znaki specjalne (np. *?+ ()) i wprowadza sekwencje specjalne (patrz tabela 20.). Ze względu na obowiązujące w Pythonie reguły formułę tę należy zapisywać jako \\ lub r'\'
\\	Odpowiada literałowi \. Ze względu na obowiązujące w Pythonie reguły formułę tę należy zapisywać jako \\ lub r'\'
\numer	Odpowiada zawartości grupy o tym samym numerze, np. r'(.+)\1' odpowiada łańcuchowi '42 42'.
R R	Alternatywa — odpowiada lewemu bądź prawemu wyrażeniu regularnemu R.
RR	Konkatenacja — odpowiada obu wyrażeniom regularnym R.
(R)	Odpowiada dowolnemu wyrażeniu regularnemu wewnątrz nawiasów () oraz rozgranicza grupę (zachowuje dopasowane podciągi).
(?:R)	To samo co (R), ale nie rozgranicza grupy.

Tabela 19. Składnia wzorców wyrażeń regularnych — ciąg dalszy

Formuła	Opis
(?=R)	Asercja w przód — pasuje do ciągu, jeśli pasuje on do wyrażenia <i>R</i> , ale nie konsumuje żadnej części łańcucha (np. 'X(=?Y)' pasuje do X, jeśli za X występuje Y).
(?!R)	Negatywna asercja w przód — pasuje do ciągu, jeśli nie jest on zgodny ze wzorcem <i>R</i> . Negacja wyrażenia (=?R).
(?P<nazwa>R)	Odpowiada dowolnemu wyrażeniu regularnemu wewnątrz nawiasów () oraz rozgranicza nazwaną grupę (np. r'(?P<i d>[a-zA-Z_]\w*)' definiuje grupę o nazwie i d).
(?P=nazwa)	Pasuje do dowolnego tekstu dopasowanego przez wcześniejszą grupę <i>nazwa</i> .
(?#...)	Komentarz. Zawartość ignorowana.
(?litera)	Argument <i>litera</i> jest jedną z wartości a, i, L, m, s, x lub u. Ustawia flagę (re.A, re.I, re.L itd.) dla całego wyrażenia.
(?<=R)	Pozytywna asercja wstecz — ciąg jest zgodny ze wzorcem, jeśli poprzedza go wyrażenie <i>R</i> o stałej szerokości.
(?<!R)	Negatywna asercja wstecz — ciąg jest zgodny ze wzorcem, jeśli nie poprzedza go wyrażenie <i>R</i> o stałej szerokości.
(?(id/ nazwa)wzorczetak wzorzecnie)	Próba dopasowania do wzorca <i>wzorczetak</i> , jeśli istnieje grupa o podanym identyfikatorze i d lub <i>nazwie</i> . W przeciwnym razie dopasowuje do opcjonalnego wzorca <i>wzorzecnie</i> .

dopasowywanego łańcucha, o ile cały wzorec w dalszym ciągu pasuje do docelowego łańcucha znaków.

Zestawione w tabeli 20. wzorce \b, \B, \d, \D, \s, \S, \w oraz \W zachowują się różnie w zależności od ustawionych flag. W Pythonie 3.X domyślnie działają w trybie Unicode, o ile nie zostanie użyta flaga ASCII (?a). Wskazówka: aby lewe ukośniki w sekwencjach specjalnych zestawionych w tabeli 20. były interpretowane literalnie, należy użyć surowych łańcuchów znaków (r'\n').

Moduły utrwalania obiektów

Na system *utrwalania obiektów* standardowej biblioteki (pamięć masową obiektów Pythona) składają się trzy moduły:

dbm (anydbm w Pythonie 2.X)

Pliki bazujące na kluczach — wyłącznie tekstowe.

pickle (oraz cPickle w Pythonie 2.X)

Serializuje obiekty w pamięci na strumieniu plikowe (ze strumieni plikowych).

Tabela 20. Sekwencje specjalne używane we wzorcach wyrażeni regularnych

Sekwencja	Opis
<code>\numer</code>	Odpowiada zawartości grupy o numerze <i>numer</i> (licząc od 1)
<code>\A</code>	Pasuje tylko na początku łańcucha znaków
<code>\b</code>	Pusty łańcuch znaków na granicy słów
<code>\B</code>	Pusty łańcuch znaków poza granicą słów
<code>\d</code>	Dowolna cyfra dziesiętna (podobnie jak <code>[0–9]</code>)
<code>\D</code>	Dowolny znak oprócz cyfry dziesiętnej (podobnie jak <code>[^0–9]</code>)
<code>\s</code>	Dowolny znak białej spacji (podobnie jak <code>[\t\n\r\f\v]</code>)
<code>\S</code>	Dowolny znak różny od białej spacji (podobnie jak <code>[^\t\n\r\f\v]</code>)
<code>\w</code>	Dowolny znak alfanumeryczny
<code>\W</code>	Dowolny znak niebędący znakiem alfanumerycznym
<code>\Z</code>	Pasuje tylko na końcu łańcucha znaków

`shelve`

Przechowuje obiekty w plikach bazujących na kluczach — zapisuje i odczytuje obiekty do (z) plików dbm w formacie `pickle`.

Moduł `shelve` implementuje trwały magazyn obiektów. Wykorzystuje moduł `pickle` do konwersji (serializacji) obiektów Pythona przechowywanych w pamięci na liniowe łańcuchy. Z kolei moduł `dbm` służy do zapisywania tych liniowych łańcuchów w plikach, do których dostęp można uzyskać za pomocą kluczy. Wszystkie trzy moduły mogą być wykorzystywane bezpośrednio.

UWAGA

W Pythonie 2.X moduł `dbm` nosi nazwę `anydbm`, natomiast moduł `cPickle` jest zoptymalizowaną wersją modułu `pickle`, którą można bezpośrednio zaimportować. Moduł ten, o ile jest dostępny, jest automatycznie wykorzystywany przez moduł `shelve`. W Pythonie 3.X moduł `cPickle` przemianowano na `_pickle`. Moduł ten, o ile jest dostępny, jest automatycznie wykorzystywany przez moduł `pickle`. Nie trzeba go importować bezpośrednio, gdyż jest on importowany niejawnie razem z modulem `shelve`.

Należy także zwrócić uwagę, że w Pythonie 3.X interfejs Berkeley DB dla modułu `dbm` (znany także jako `bsddb`) nie jest dostarczany razem z Pythonem, ale jest zewnętrznym rozszerzeniem *open source*, które należy instalować osobno (informacje na ten temat można znaleźć w podręczniku *Python 3.X Library Reference*).

Moduły dbm i shelve

dbm to system plików dających *dostęp do informacji za pośrednictwem kluczy* — łańcuchy znaków są zapisywane w plikach, a dostęp do nich można uzyskać za pośrednictwem tekstowych kluczy. Moduł dbm dostarcza implementację plików z informacjami dostępnymi za pośrednictwem kluczy w środowisku interpretera Pythona. Definiuje API typu słownikowego dla skryptów.

Utrwalony obiekt shelve jest plikiem obiektu z *dostępem za pośrednictwem klucza* — jest wykorzystywany tak jak prosty plik dbm, z tą różnicą, że moduł dbm należy zastąpić modulem shelve, a zapamiętane *wartości* mogą być obiektami Pythona niemal dowolnego typu (choć *klucze* są nadal łańcuchami znaków). Pod wieloma względami pliki dbm i shelve działają jak *słowniki*, które przed użyciem należy otworzyć, a po wprowadzeniu zmian zamknąć. Działają wszystkie operacje odwzorowań oraz niektóre metody obsługi słowników. Pliki są automatycznie odwzorowywane na zewnętrzne pliki.

Otwieranie plików

W przypadku modułu shelve należy zaimportować bibliotekę, a następnie wywołać jej metodę open(). Aby utworzyć nowy bądź otworzyć istniejący plik shelve, należy skorzystać z następujących instrukcji:

```
import shelve
plik = shelve.open(nazwapliku
    [, flag='c'
    [, protocol=None
    [, writeback=False]]])
```

Aby utworzyć nowy bądź otworzyć istniejący plik dbm w przypadku modułu dbm, należy zaimportować bibliotekę, a następnie wywołać jej metodę open(). Powoduje to pobranie pomocniczej biblioteki obsługi plików dbm: dbm.bsd, dbm.gnu, dbm.ndbm lub dbm.dumb (ostatnia jest zawsze dostępną opcją domyślną):

```
import dbm
plik = dbm.open(nazwapliku
    [, flag='r'
    [, mode]])
```

Argument *nazwapliku* zarówno dla modułu shelve, jak i dbm jest łańcuchem znaków opisującym względną bądź bezwzględną ścieżkę do zewnętrznego pliku, w którym zapisano dane.

Argument `flag` ma takie samo znaczenie dla modułu `shelve` jak dla `dbm` (moduł `shelve` przekazuje go do `dbm`). Może mieć wartość `'r'` — w celu otwarcia istniejącej bazy danych w trybie tylko do odczytu, `'w'` — w celu otwarcia istniejącej bazy danych do odczytu i zapisu. Tryb `'c'` powoduje utworzenie bazy danych, o ile nie istnieje (dla modułu `shelve` jest to wartość domyślna), z kolei tryb `'n'` zawsze tworzy nową, pustą bazę danych. Moduł `dbm.dumb` (używany domyślnie w Pythonie w wersji 3.X, w przypadku gdy nie zainstalowano innej biblioteki) ignoruje argument `flag` — baza danych jest zawsze otwierana do aktualizacji, a jeśli nie istnieje, to jest tworzona.

W przypadku modułu `dbm` opcjonalny argument `mode` oznacza uniksowy tryb pliku. Jest on używany tylko wtedy, kiedy trzeba utworzyć bazę danych. Jego domyślna wartość wynosi `0o666` (ósemkowo).

W przypadku modułu `shelve` argument `protocol` jest przekazywany z modułu `shelve` do `pickle`. Argument ten dostarcza numer protokołu `pickle` (opisanego w podrozdziale „Moduł `pickle`”) używanego do zapisywania utrwalanych obiektów. Jego domyślna wartość wynosi `0` w Pythonie 2.X oraz `3` w Pythonie 3.X.

Domyślnie zmiany w obiektach pobranych z pamięci zewnętrznej nie są automatycznie zapisywane na dysku. Jeśli opcjonalny parametr `writeback` zostanie ustawiony na `True`, wszystkie odczytywane pozycje zostaną zbuforowane w pamięci i zapisane na dysku w momencie zamykania. Dzięki temu aktualizacja mutowalnych pozycji w pamięci trwałej staje się łatwiejsza, ale bufor zużywa pamięć, a operacja zamykania jest wolniejsza, ponieważ wszystkie obiekty, z których korzystano, są zapisywane na dysku. Aby ręcznie zaktualizować obiekty `shelve`, należy ponownie przypisać obiekty do ich kluczy.

Operacje na plikach

Po utworzeniu pliki `shelve` i `dbm` mają niemal identyczny interfejs, podobny do słowników. Opisano go poniżej:

```
plik[klucz] = wartość
```

Zapisywanie — tworzy lub modyfikuje pozycję odpowiadającą *kluczowi*. W przypadku modułu `dbm` *wartość* jest łańcuchem znaków. W przypadku modułu `shelve` — dowolnym obiektem.

```
wartość = plik[klucz]
```

Pobieranie — ładuje *wartość* odpowiadającą *kluczowi*. W przypadku modułu `shelve` rekonstruuje obiekt w pamięci.

```
liczba = len(plik)
```

Rozmiar — zwraca liczbę zapisanych pozycji.

```
indeks = plik.keys()
```

Skorowidz — pobiera obiekt iterowalny złożony z zapisanych kluczy (listę w Pythonie 2.X).

```
for klucz in plik: ...
```

Iteracje — iterator kluczy (można go wykorzystać w dowolnym kontekście iteracyjnym).

```
znaleziono = klucz in plik # W Pythonie 2.X także has_key()
```

Zapytanie — True, jeśli istnieje pozycja odpowiadająca *kluczowi*.

```
del plik[klucz]
```

Usuwanie — usuwa pozycję odpowiadającą *kluczowi*.

```
plik.close()
```

Ręczne zamknięcie pliku. Wymagane w celu zapisania aktualizacji na dysku dla niektórych interfejsów dbm.

Moduł pickle

Moduł *pickle* jest narzędziem do *serializacji obiektów* — konwertuje niemal dowolne obiekty Pythona występujące w pamięci na serializowane strumienie bajtowe. Pozwala również na konwersję w drugą stronę. Te strumienie bajtowe mogą być skierowane do dowolnego obiektu plikowego, który ma oczekiwane metody odczytu i zapisu. Są one wykorzystywane przez moduł *shelve* do wewnętrznej reprezentacji danych. Operacja odczytu (ang. *unpickling*) odtwarza obiekt w oryginalnej postaci, z tą samą wartością, ale z nową tożsamością (adresem).

Patrz też wcześniejsza uwaga na temat zoptymalizowanych modułów *cPickle* Pythona 2.X oraz *_pickle* Pythona 3.X. Warto się również zapoznać z metodą *makefile* obiektów gniazd pozwalających na przesyłanie serializowanych obiektów w sieciach bez ręcznego wywoływania gniazd (patrz dokumentacja Pythona oraz podrozdział „Moduły i narzędzia do obsługi internetu”).

Interfejsy utrwalania

Moduł *pickle* obsługuje następujące wywołania:

```
P = pickle.Pickler(obiektpliku [, protocol=None])
```

Tworzy nowy obiekt utrwalający (ang. *pickler*) służący do zapisywania obiektów w pliku wyjściowym.

```
P.dump(obiekt)
```

Zapisuje obiekt do pliku (strumienia) obiektu utrwalającego.

```
pickle.dump(obiekt, obiektpliku [, protocol=None])
```

Połączenie poprzednich dwóch funkcji — utrwalą obiekt w pliku.

```
łańcuch_znaków = pickle.dumps(obiekt [, protocol=None])
```

Zwraca utrwaloną reprezentację obiektu w postaci łańcucha znaków (w Pythonie 3.X łańcucha bajtów).

Interfejsy odtwarzania

Moduł `pickle` obsługuje następujące wywołania:

```
U = pickle.Unpickler(obiektpliku,  
                    encoding="ASCII", errors="strict")
```

Tworzy obiekt odtwarzający służący do załadowania obiektów z pliku wejściowego.

```
obiekt = U.load()
```

Odczytuje obiekt z pliku (strumienia) obiektu odtwarzającego.

```
obiekt = pickle.load(obiektpliku,  
                    encoding="ASCII", errors="strict")
```

Połączenie poprzednich dwóch funkcji — odtwarza obiekt z pliku.

```
obiekt = pickle.loads(łańcuch_znaków,  
                    encoding="ASCII", errors="strict")
```

Odczytuje obiekt z łańcucha znaków (w Pythonie 3.X z ciągu bajtów).

Uwagi na temat sposobu użycia modułu `pickle`

- W Pythonie 3.X pliki używane do przechowywania utrwalonych obiektów powinny zawsze być otwarte w trybie binarnym dla wszystkich protokołów, ponieważ obiekt utrwalający generuje ciągi typu `bytes`, a pliki trybu tekstowego nie obsługują zapisywania danych typu `bytes` (w wersji 3.X pliki trybu tekstowego kodują i dekodują tekst Unicode).
- W Pythonie 2.X pliki używane do przechowywania utrwalonych obiektów muszą być otwarte w trybie binarnym dla wszystkich protokołów utrwalania ≥ 1 ze względu na konieczność wyłączenia translacji znaków końca wiersza w utrwalanych danych

binarnych. Protokół 0 bazuje na kodzie ASCII, zatem pliki utrwalane za pomocą tego protokołu mogą być otwierane w trybie tekstowym lub binarnym, pod warunkiem że tryb jest wybierany w sposób spójny.

- Argument *obiektpliku* to otwarty obiekt plikowy lub dowolny obiekt implementujący atrybuty obiektu pliku wywoływane przez interfejs. Obiekt utrwalający (*Pickler*) wywołuje metodę pliku `write()` i przekazuje do niej argument tekstowy. Obiekt odtwarzający (*Unpickler*) wywołuje metodę pliku `read()` i przekazuje do niej liczbę bajtów do przeczytania oraz metodę `readline()` bez argumentów.
- Za pomocą opcjonalnego argumentu *protokół* wybierany jest format utrwalanych danych. Jest on wykorzystywany zarówno w konstruktorze obiektu *Pickler*, jak i w funkcjach pomocniczych modułów `dump` i `dumps`. Argument ten przyjmuje wartości z zakresu `0...3`, przy czym wyższe numery protokołów są, ogólnie rzecz biorąc, wydajniejsze, ale mogą być nieobsługiwane przez obiekty odtwarzające we wcześniejszych wydaniach Pythona. Domyślny numer protokołu w Pythonie 3.X to 3. Dane utrwalone za pomocą tego protokołu nie mogą być odtworzone w Pythonie 2.X. Domyślny numer protokołu w Pythonie 2.X to 0. Protokół ten jest mniej wydajny, ale zapewnia większy stopień przenośności. Podanie wartości `-1` protokołu powoduje automatyczne użycie najwyższego obsługiwanego protokołu. Podczas *odtworzenia* Python ustala protokół na podstawie zawartości utrwalonych danych.
- Opcjonalne argumenty kluczowe obiektu *Unpickler* — `encoding` i `errors` — są dostępne wyłącznie w Pythonie 3.X. Są one używane do dekodowania 8-bitowych egzemplarzy obiektów tekstowych, utrwalonych przez Pythona 2.X. Ich domyślne wartości to odpowiednio `'ASCII'` i `'strict'`. Opis podobnych narzędzi można znaleźć przy okazji omawiania funkcji `str()` w podrozdziale „Funkcje wbudowane”.
- *Pickler* i *Unpickler* to eksportowane klasy, które można dostosować do indywidualnych potrzeb poprzez tworzenie klas podrzędnych. O dostępnych metodach można przeczytać w podręczniku *Python Library Reference*.

UWAGA

W czasie pisania tej książki trwała dyskusja nad propozycją stworzenia zoptymalizowanego protokołu utrwalania numer 4. Jego planowane wdrożenie miało nastąpić w Pythonie 3.4, ale nadal ma on status prototypu, dlatego nie wiadomo, kiedy ostatecznie się pojawi. Jeśli zostanie dołączony, może się stać nową opcją domyślną wersji 3.X. Może być niekompatybilny wstecz i nierozpoznawany we wszystkich wcześniejszych wersjach Pythona. *Aktualizacja z ostatniej chwili*: ta zmiana pojawi się oficjalnie w pierwszym wydaniu beta wersji 3.4. Szczegółowe informacje można znaleźć w dokumencie „What’s new”.

Moduł GUI tkinter i narzędzia

tkinter (w Pythonie 2.X Tkinter, a w Pythonie 3.X pakiet modułów) to przenośna biblioteka, która dostarcza mechanizmy konstrukcji *graficznego interfejsu użytkownika* (ang. *Graphical User Interface*, GUI). Jest dostarczana wraz z Pythonem jako standardowy moduł biblioteczny. tkinter udostępnia obiektowy interfejs do biblioteki *open source* Tk i implementuje w środowiskach Windows, X-Windows i Mac OS natywne wrażenie oraz sposób działania interfejsów użytkownika aplikacji w Pythonie. Jest przenośny, łatwy w użytku, dobrze udokumentowany, powszechnie wykorzystywany, dojrzały i dobrze wspierany. Istnieją także inne przenośne narzędzia zewnętrznych producentów do tworzenia interfejsów GUI w Pythonie, na przykład *wxPython* i *PyQT*. Charakteryzują się one obszerniejszym zbiorem widgetów, ale ogólnie rzecz biorąc, wymagają bardziej skomplikowanego kodowania.

Przykład użycia modułu tkinter

W skryptach korzystających z modułu tkinter *widgets* są klasami (np. Button, Frame), które użytkownik może przystosować do własnych potrzeb, *opcje* to argumenty kluczowe (np. text="wciśnij"), natomiast *kompozycja* odnosi się do osadzania obiektów, a nie do nazw ścieżek (np. Label(top,...)):

```
from tkinter import *          # widgety, stałe

def msg():                     # handler wywołania zwrotnego
    print('witaj, stdout...')

top = Frame()                  # utworzenie kontenera
top.pack()
Label(top, text="Witaj, świecie").pack(side=TOP)
```

```
widget = Button(top, text="wciśnij", command=msg)
widget.pack(side=BOTTOM)
top.mainloop()
```

Podstawowe widgety modułu tkinter

W tabeli 21. zestawiono podstawowe klasy widgetów dostępne w module tkinter. Są one rzeczywistymi klasami Pythona, na podstawie których można tworzyć klasy podrzędne i które można osadzać w innych obiektach. Aby stworzyć urządzenie ekranu, należy utworzyć egzemplarz odpowiedniej klasy, skonfigurować ją i skorzystać z jednej z metod interfejsu menedżera geometrii (ang. *geometry manager*), np. `Button(text='witaj').pack()`. Oprócz klas wymienionych w tabeli 21. moduł tkinter dostarcza wiele nazw predefiniowanych (znanych także jako stałe), które są używane do konfigurowania widgetów (np. `RIGHT`, `BOTH`, `YES`). Są one automatycznie ładowane z modułu `ttkinter.constants` (w Pythonie 2.X z modułu `Tkconstants`).

Wywołania okien dialogowych

Moduł `ttkinter.messagebox` (`tkMessageBox` w Pythonie 2.X)

```
showinfo(title=None, message=None, **opcje)
showwarning(title=None, message=None, **opcje)
showerror(title=None, message=None, **opcje)
askquestion(title=None, message=None, **opcje)
askokcancel(title=None, message=None, **opcje)
askyesno(title=None, message=None, **opcje)
askretrycancel(title=None, message=None, **opcje)
```

Moduł `ttkinter.simpledialog` (`tkSimpleDialog` w Pythonie 2.X)

```
askinteger(tytuł, pytanie, **kw)
askfloat(tytuł, pytanie, **kw)
askstring(tytuł, pytanie, **kw)
```

Moduł `ttkinter.colorchooser` (`tkColorChooser` w Pythonie 2.X)

```
askcolor(color=None, **opcje)
```

Moduł `ttkinter.filedialog` (`tkFileDialog` w Pythonie 2.X)

```
class Open
class SaveAs
class Directory
askopenfilename(**opcje)
```

Tabela 21. Podstawowe klasy widgetów modułu tkinter

Klasa widgetu	Opis
Label	Prosty obszar komunikatów
Button	Prosty widget przycisku oznaczony etykietą
Frame	Kontener służący do dołączania i aranżowania obiektów innych widgetów
Toplevel, Tk	Okna najwyższego poziomu zarządzane przez menedżer okien
Message	Wielowierszowe pole do wyświetlania tekstu (etykieta)
Entry	Proste, jednowierszowe pole do wprowadzania tekstu
Checkbutton	Dwustanowy widget przycisku. Używany do wielokrotnego wyboru
Radiobutton	Dwustanowy widget przycisku. Używany do jednokrotnego wyboru
Scale	Widget suwaka ze skalowanymi pozycjami
PhotoImage	Obiekt obrazu pozwalający na umieszczanie kolorowych fotografii na innych widgetach
BitmapImage	Obiekt obrazu pozwalający na umieszczanie bitmap na innych widgetach
Menu	Opcje powiązane z obiektem Menubutton lub oknem najwyższego poziomu
Menubutton	Przycisk otwierający menu złożone z opcji do wyboru (oraz podmenu)
Scrollbar	Pasek do przewijania innych widgetów (np. Listbox, Canvas, Text)
Listbox	Lista nazw do wyboru
Text	Wielowierszowy widget do przeglądania (edycji) tekstu. Zapewnia obsługę czcionek itp.
Canvas	Obszar rysowania grafiki: linie, okręgi, zdjęcia, tekst itp.
OptionMenu	<i>Obiekt złożony</i> : rozwijana lista wyboru
PanedWindow	Interfejs okna z wieloma panelami
LabelFrame	Widget ramki oznaczonej etykietą
Spinbox	Widget do wielokrotnego wyboru
ScrolledText	Nazwa dostępna w Pythonie 2.X (w Pythonie 3.X dostępna w module <code>tktinter.scrolledtext</code>). <i>Obiekt złożony</i> : tekst z dołączonym paskiem przewijania
Dialog	Nazwa dostępna w Pythonie 2.X (w Pythonie 3.X dostępna w module <code>tktinter.dialog</code>). <i>Przestarzałe</i> : narzędzie do tworzenia okien dialogowych (poniżej opisano nowocześniejsze wywołania okien dialogowych)

```

asksaveasfilename(**opcje)
askopenfile(mode="r", **opcje)
asksaveasfile(mode="w", **opcje)
askdirectory(**opcje)

```

Opcje dostępne w wywołaniach popularnych okien dialogowych to `defaultextension` (dodawane do nazwy pliku, jeśli nie podano go jawnie), `filetypes` (sekwencja krotek (*etykieta*, *wzorzec*)), `initialdir` (początkowy

katalog pamiętany przez klasy), initialfile (początkowy plik), parent (okno, w którym ma być umieszczone okno dialogowe) oraz title (tytuł okna dialogowego).

Dodatkowe klasy i narzędzia modułu tkinter

W tabeli 22. zestawiono niektóre powszechnie używane interfejsy modułu tkinter, a także narzędzia wykraczające poza podstawowe klasy widgetów oraz standardowe okna dialogowe. Większość z nich to standardowe narzędzia biblioteki. Wyjątkiem są niektóre w ostatnim rzędzie (np. Pillow). Więcej informacji można znaleźć w internecie.

Tabela 22. Dodatkowe narzędzia dostępne w module tkinter

Kategoria	Dostępne narzędzia
Klasy zmiennych dziedziczących po klasie Variable	StringVar, IntVar, DoubleVar, BooleanVar (w module tkinter)
Metody zarządzania geometrią	Metody pack(), grid(), place() obiektów widgetów oraz opcje konfiguracji w module
Planowane wywołania zwrotne	Metody after(), wait() i update(); wywołania zwrotne plikowych operacji wejścia-wyjścia
Inne narzędzia modułu tkinter	Dostęp do schowka, metody niskopoziomowego przetwarzania zdarzeń, opcje konfiguracji widgetów, obsługa modalnych okien dialogowych
Rozszerzenia modułu tkinter	PMW: dodatkowe widgety; PIL (znane także jako Pillow): przetwarzanie obrazów, drzewa, obsługa czcionek, operacje typu „przeciągnij i upuść”, widgety tix, widgety ttk z obsługą motywów itp.

Porównanie biblioteki Tcl/Tk z modułem tkinter Pythona

W tabeli 23. porównano interfejs API modułu tkinter Pythona z bazową biblioteką Tk napisaną w języku Tcl. Ogólnie rzecz biorąc, ciągi poleceń języka Tcl odwzorowano na obiekty Pythona. W szczególności interfejs GUI Tk modułu tkinter różni się od tego, który jest dostępny w języku Tcl, w następujący sposób:

Tworzenie

Widgety tworzy się jako obiekty egzemplarzy klas poprzez wywołanie klasy widget.

Tabela 23. Porównanie biblioteki Tk z modulem tkinter

Działanie	Tcl/Tk	Python/tkinter
Tworzenie	<code>frame .panel</code>	<code>panel = Frame()</code>
Obiekty nadrzędne	<code>button .panel.quit</code>	<code>quit = Button(panel)</code>
Opcje	<code>button .panel.go - fg black</code>	<code>go = Button(panel, fg='black')</code>
Konfiguracja	<code>.panel.go config - bg red</code>	<code>go.config(bg='red') go['bg'] = 'red'</code>
Działania	<code>.popup invoke</code>	<code>popup.invoke()</code>
Pakowanie	<code>pack .panel -side left -fill x</code>	<code>panel.pack(side=LEFT, fill=X)</code>

Obiekty nadrzędne (rodzice)

Obiekty nadrzędne to wcześniej utworzone obiekty, które są przekazywane do konstruktorów klas widgetów.

Opcje widgetów

Opcje są konstruktorami, kluczowymi argumentami konfiguracyjnymi albo zaindeksowanymi kluczami.

Działania

Działania na widgetach (akcje) zaimplementowano jako metody obiektów klasy widget modułu tkinter.

Wywołania zwrotne

Handlery wywołań zwrotnych to dowolne obiekty wywoławalne: funkcje, metody, obiekty `lambda`, klasy zawierające metodę `__call__` itp.

Rozszerzenia

Widgety rozszerza się, wykorzystując mechanizmy dziedziczenia klas Pythona.

Kompozycja

Interfejsy konstruuje się poprzez dołączanie obiektów, a nie poprzez konkatenację nazw.

Powiązane zmienne

Zmienne powiązane z widgetami są obiektami klasy tkinter z metodami.

Moduły i narzędzia do obsługi internetu

W tym podrozdziale podsumowano obsługę *mechanizmów internetowych* w Pythonie 3.X i 2.X. Przedstawiono tu krótki przegląd niektórych powszechnie stosowanych modułów w zbiorze modułów internetowych należących do standardowej biblioteki Pythona. *To jest tylko reprezentatywna próbka*. Pełniejszą listę można znaleźć w podręczniku *Python Library Reference*.

socket

Niskopoziomowa obsługa komunikacji sieciowej (TCP/IP, UDP itp.). Interfejsy do wysyłania i odbierania danych za pośrednictwem gniazd w rodzaju BSD — wywołanie `socket.socket()` tworzy obiekt z metodami obsługi gniazd (np. `obiekt.bind()`). Większość modułów obsługi protokołów i serwerów wewnętrznie korzysta z tego modułu.

socketserver (w Pythonie 2.X SocketServer)

Framework do obsługi wątków (rozwidlania) serwerów sieciowych.

xdrlib

Kodowanie danych binarnych (patrz też moduły `socket` wcześniej na tej liście).

select

Interfejsy dla funkcji `select()` systemów Unix i Windows. Oczekują na aktywność jednego ze zbioru plików bądź gniazd. Powszechnie używane do zwielokrotniania strumieni lub implementowania limitów czasu. W systemie Windows działają tylko z gniazdami (nie z plikami).

cgi

Obsługa skryptów CGI działających po stronie serwera: metoda `cgi.FieldStorage()` parsuje strumień wejściowy, metoda `cgi.escape()` stosuje konwencje sekwencji specjalnych HTML w odniesieniu do strumieni wyjściowych. W celu parsowania i dostępu do informacji dotyczących formularzy po wykonaniu skryptu CGI wywoływana jest metoda `formularz=cgi.FieldStorage()`. `formularz` jest obiektem słownikowym z jedną pozycją na pole (np. `formularz["nazwa"].value` to wartość tekstowa pola formularza *nazwa*).

urllib.request (w Pythonie 2.X urllib, urllib2)

Pobiera strony WWW oraz wynik działania skryptów serwera spod ich adresów internetowych (adresów URL). Wywołanie `urllib.request.urlopen(url)` zwraca obiekt `file` z metodami `read`.

Dostępne jest również wywołanie `urllib.request.urlretrieve` ↗(*zdalny*, *lokalny*). Obsługuje adresy URL HTTP, HTTPS, FTP i plików lokalnych.

`urllib.parse` (w Pythonie 2.X `urlparse`)

Parsuje łańcuch znaków URL i rozdziela go na komponenty. Zawiera także narzędzia do „unieszkodliwiania” znaków specjalnych w tekście URL. Wywołanie `urllib.parse.quote_plus(str)` „unieszkodliwia” znaki specjalne w tekście wstawianym do strumieni wyjściowych HTML.

`ftplib`

Moduł obsługi FTP (ang. *file transfer protocol*). Dostarcza interfejsy do operacji transferu plików internetowych z poziomu programów w Pythonie. Po wywołaniu `ftp=ftplib.FTP('nazwaserwisuFTP')` obiekt *ftp* zawiera metody do logowania, zmiany katalogów, pobierania (zapisywania) plików i listingów itp. Obsługuje transfer plików binarnych i tekstowych. Działa tylko na komputerach z Pythonem oraz połączeniem z internetem.

`poplib`, `imaplib`, `smtplib`

Moduły obsługi protokołów POP, IMAP (pobieranie poczty) i SMTP (wysyłanie poczty).

Pakiet `email`

Parsuje i konstruuje wiadomości e-mail z nagłówkami i załącznikami. Zawiera także obsługę MIME zarówno dla treści, jak i nagłówków.

`http.client` (w Pythonie 2.X `httplib`), `nntplib`, `telnetlib`

Moduły obsługi klientów HTTP (WWW), NNTP (news) oraz Telnet.

`http.server` (w Pythonie 2.X `CGIHTTPServer` i `SimpleHTTPServer`)

Implementacje żądań serwera HTTP.

Pakiet `xml`, pakiet `html` (w Pythonie 2.X `htmllib`)

Parsuje dokumenty XML i HTML oraz treść stron WWW. Pakiet `xml` obsługuje modele parsowania DOM, SAX oraz `ElementTree`. Obsługuje parsowanie `Expat`.

Pakiet `xmlrpc` (w Pythonie 2.X `xmlrpclib`)

Protokół wywoływania zdalnych metod XML-RPC.

`uu`, `binhex`, `base64`, `binascii`, `quopri`

Koduje i dekoduje dane binarne (lub inne), transmitowane w postaci tekstu.

Niektóre z wymienionych modułów, uporządkowane według typu protokołu, zestawiono w tabeli 24. Nazwy dla Pythona 2.X różniące się od tych, które obowiązują w Pythonie 3.X, można znaleźć na liście powyżej.

Tabela 24. Wybrane moduły obsługi internetu w Pythonie 3.X, uporządkowane według protokołu

Protokół	Przeznaczenie	Numer portu	Moduł Pythona
HTTP	Strony WWW	80	http.client, urllib.request, xmlrpc.*
NNTP	Wiadomości Usenet	119	nntplib
Domyślny protokół danych FTP	Transfer plików	20	ftplib, urllib.request
Sterowanie FTP	Transfer plików	21	ftplib, urllib.request
SMTP	Wysyłanie wiadomości e-mail	25	smtplib
POP3	Pobieranie poczty e-mail	110	poplib
IMAP4	Pobieranie poczty e-mail	143	imaplib
Telnet	Interfejs wiersza poleceń	23	telnetlib

Inne standardowe moduły biblioteczne

W tym podrozdziale opisano szereg przydatnych, dodatkowych standardowych modułów bibliotecznych instalowanych wraz z Pythonem. O ile nie zaznaczono inaczej, narzędzia opisane w tym podrozdziale mają zastosowanie zarówno do Pythona 3.X, jak i 2.X. Szczegółowe informacje na ich temat wraz z opisem wbudowanych narzędzi można znaleźć w podręczniku *Python Library Reference*, a także w serwisach internetowych PyPI (opisano je w podrozdziale „Różne wskazówki”). Można też skorzystać z wybranej wyszukiwarki internetowej, by poszukać informacji na temat zewnętrznych modułów i związanych z nimi narzędzi.

Moduł math

Moduł `math` eksportuje *narzędzia matematyczne* zgodne ze standardem języka C do wykorzystania w Pythonie. Eksporty tego modułu w Pythonie 3.3 zestawiono w tabeli 25. Siedem ostatnich dodatków do Pythona 3.2 i 3.3 oznaczono pogrubioną czcionką. Moduł z Pythona 2.7 jest identyczny, ale nie ma w nim funkcji `log2` ani `isfinite`. Eksportowane metody modułu `math` mogą być nieco inne w różnych wydaniach. Wszystkie pozycje wymienione w tabeli są wywoływalnymi funkcjami (końcowe nawiasy zostały pominięte). Wyjątkiem są wywołania `pi` oraz `e`.

Tabela 25. Eksporty modułu `math` w Pythonie 3.3

<code>acos</code>	<code>acosh</code>	<code>asin</code>	<code>asinh</code>	<code>atan</code>
<code>atan2</code>	<code>atanh</code>	<code>ceil</code>	<code>copysign</code>	<code>cos</code>
<code>cosh</code>	<code>degrees</code>	<code>e</code>	<code>erf</code>	<code>erfc</code>
<code>exp</code>	<code>expm1</code>	<code>fabs</code>	<code>factorial</code>	<code>floor</code>
<code>fmod</code>	<code>frexp</code>	<code>fsum</code>	<code>gamma</code>	<code>hypot</code>
<code>isfinite</code>	<code>isinf</code>	<code>isnan</code>	<code>ldexp</code>	<code>lgamma</code>
<code>log</code>	<code>log10</code>	<code>log1p</code>	<code>log2</code>	<code>modf</code>
<code>pi</code>	<code>pow</code>	<code>radians</code>	<code>sin</code>	<code>sinh</code>
<code>sqrt</code>	<code>tan</code>	<code>tanh</code>	<code>trunc</code>	

Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference*. Można również zaimportować moduł `math` i uruchomić polecenie `help(math.nazwa)`. W ten sposób można się zapoznać z nazwami argumentów i uwagami dotyczącymi funkcji zamieszczonych w poniższej tabeli. Wywołanie `dir(math)` powoduje wyświetlenie zawartości modułu. Warto się również zapoznać z modulem `cmath` z biblioteki Pythona, zawierającym narzędzia do obsługi liczb zespolonych, oraz systemem *NumPy*, dostarczającym narzędzi do zaawansowanych obliczeń numerycznych.

Moduł time

W tym podrozdziale opisano narzędzia związane z obsługą *godzin i dat*: przetwarzanie godzin, formatowanie i przerwy. Poniżej zamieszczono częściową listę eksportów modułu `time`. Więcej informacji na ten temat można znaleźć w podrozdziałach „Moduł `datetime`”, „Moduł `timeit`”, a także w podręczniku *Python Library Reference*.

`time.clock()`

Zwraca czas procesora lub czas rzeczywisty, jaki upłynął od początku procesu bądź od pierwszego wywołania funkcji `clock()`. Dokładność i semantyka zależą od platformy (patrz dokumentacja Pythona). Zwraca liczbę sekund wyrażoną w postaci liczby zmiennoprzecinkowej. Przydaje się do mierzenia szybkości działania oraz czasu wykonywania się różnych fragmentów kodu.

`time.time()`

Zwraca liczbę zmiennoprzecinkową reprezentującą czas UTC w sekundach od początku epoki. W systemie Unix „epoka” oznacza datę 1 stycznia 1970 r. Funkcja `time()` na niektórych platformach może być dokładniejsza niż `clock()` (zobacz dokumentację Pythona).

`time.ctime(sekundy)`

Konwertuje czas, jaki upłynął od epoki Unix (wyrażony w sekundach), na tekstową reprezentację lokalnego czasu (np. `ctime` ↪ `(time())`). Argument jest opcjonalny. W przypadku jego pominięcia domyślną wartością jest czas bieżący.

`time.sleep(sekundy)`

Zawiesza wykonanie procesu (wątku wywołującego) na liczbę sekund wyrażoną argumentem *sekundy*. Argument *sekundy* może być liczbą zmiennoprzecinkową. W ten sposób można reprezentować ułamki sekund.

Kolejne dwie metody są dostępne wyłącznie w *Pythonie 3.3* i wersjach późniejszych. Mają one na celu przenośne dostarczanie danych czasowych (ale nie mogą być bezpośrednio porównywane z wywołaniami w starszych wersjach Pythona). W obu przypadkach punkt odniesienia zwróconej wartości jest nieokreślony, dlatego wynik jest zwracany wyłącznie w postaci różnicy pomiędzy wynikami kolejnych wywołań:

`time.perf_counter()`

Zwraca wartość licznika wydajności wyrażoną w ułamkach sekund. Jest ona zdefiniowana jako zegar z najwyższą dostępną rozdzielczością i służy do pomiaru krótkich odcinków czasu. Uwzględnia czas, który upłynął w momentach uśpienia, i obowiązuje w całym systemie. Wynik może być traktowany jako tzw. czas ścienny (ang. *wall time*). Jeśli metoda jest dostępna, jest standardowo stosowana w module `timeit`.

```
time.process_time()
```

Zwraca wartość wyrażoną w ułamkach sekund oznaczającą sumę czasu procesora (w trybach systemowym i użytkownika) bieżącego procesu. Nie uwzględnia czasu, który upłynął w momencie uśpienia, i z definicji obowiązuje na poziomie procesu.

Moduł `timeit`

W tym podrozdziale opisano narzędzia do przenośnego pomiaru czasu wykonania łańcuchów kodu lub wywołań funkcji. Szczegółowe informacje na ten temat można znaleźć w dokumentacji Pythona.

Interfejs wiersza poleceń:

```
py[thon] -m timeit [-n liczba] [-r powtórzenia]  
               [-s konfiguracja]* [-t] [-c] [-p] [-h] [instrukcja]*
```

Argument *liczba* wskazuje, ile razy zostanie uruchomiona instrukcja (wartość domyślna to wyliczona potęga liczby 10); *powtórzenia* oznaczają liczbę powtórzeń (domyślnie 3); *konfiguracja* (zero lub więcej — każda wartość z opcją `-s`) to kod, który ma być uruchomiony pomiędzy mierzonymi instrukcjami; *instrukcja* (zero lub więcej) oznacza mierzony kod; `-h` wyświetla pomoc, natomiast opcje `-t`, `-c` i `-p` określają używane timery — `time.time()`, `time.clock()` lub — począwszy od Pythona 3.3 — `time.process_time()` (wartością domyślną w wersji 3.3 i wersjach późniejszych jest `time.perf_counter()`). Wyświetlane wyniki pokazują najlepszy czas pomiędzy wykonanymi uruchomieniami w liczbie oznaczonej argumentem *powtórzenia*. Funkcja pomaga neutralizować przejściowe wahania obciążenia systemu.

Interfejs API:

```
timeit.Timer(stmt='pass', setup='pass', timer=domyślny)
```

Używane przez opisane poniżej funkcje pomocnicze. Zarówno argument `stmt`, jak i `setup` oznaczają łańcuchy kodu (do oddzielenia instrukcji można wykorzystać znaki `;` lub `\n` oraz spacje i `\t` dla wcięcia) bądź wywoływalne funkcje bez argumentów. Domyślna wartość argumentu `timer` zależy od platformy i wersji.

```
timeit.repeat(stmt='pass', setup='pass',  
              timer=domyślny, repeat=3, number=1000000)
```

Tworzy egzemplarz obiektu `Timer` z wykorzystaniem podanego kodu `stmt` i `setup` oraz funkcji `timer` i uruchamia jego metodę `repeat` z licznikiem `repeat` oraz liczbą uruchomień określoną argumentem `number`. Zwraca listę wyników czasu: pobiera wartość `min()` dla najlepszego czasu `repeat`.

```
timeit.timeit(stmt='pass', setup='pass',
              timer=domyślny, number=1000000)
```

Tworzy egzemplarz obiektu `Timer` z wykorzystaniem podanego kodu `stmt` i `setup` oraz funkcji `timer` i uruchamia jego metodę `timeit` z liczbą uruchomień określoną argumentem `number`. Uruchamia kod `setup` jeden raz. Zwraca czas uruchomienia instrukcji `stmt` `number` razy.

Moduł `datetime`

Poniżej omówiono narzędzia do *przetwarzania dat*: odejmowania od siebie dat, dodawania dni do dat itp. Więcej informacji na ten temat oraz podobnych narzędzi można znaleźć w podrozdziale „Moduł `time`”, a także w podręczniku *Python Library Reference*.

```
>>> from datetime import date, timedelta
>>> date(2013, 11, 15) - date(2013, 10, 29) # Pomiędzy
datetime.timedelta(17)

>>> date(2013, 11, 15) + timedelta(60)      # Przyszłość
datetime.date(2014, 1, 14)
>>> date(2013, 11, 15) - timedelta(410)     # Przeszłość
datetime.date(2012, 10, 1)
```

Moduł `random`

Poniżej opisano różne wywołania związane z obsługą *losowości*: liczby losowe, tasowanie i wybieranie. Szczegółowe informacje na ten temat można znaleźć w dokumentacji Pythona.

```
>>> import random
>>> random.random()      # Losowa liczba zmiennoprzecinkowa z zakresu [0, 1)
0.7082048489415967
>>> random.randint(1, 10) # Losowa liczba całkowita z zakresu [x, y]
8
>>> L = [1, 2, 3, 4]
>>> random.shuffle(L)    # Tasowanie L w miejscu
>>> L
[2, 1, 4, 3]
>>> random.choice(L)     # Wybór losowego elementu
4
```

Moduł `json`

Poniżej omówiono narzędzia do tłumaczenia struktur słowników i list Pythona na format tekstu JSON i z niego. Jest to przenośny *format reprezentacji danych* używany w takich systemach jak MongoDB (pod nazwą BSON) oraz SL4A w systemie Android (jako JSON-RPC). Warto się

również zapoznać z natywnymi mechanizmami serializacji w Pythonie opisanymi w podrozdziale „Moduł pickle”; obsługą XML omówioną w podrozdziale „Moduły i narzędzia do obsługi internetu” oraz innymi pojęciami związanymi z obsługą baz danych w podrozdziale „API baz danych Python SQL”.

```
>>> R = {'job': ['dev', 1.5], 'emp': {'who': 'Bogdan'}}

>>> import json
>>> json.dump(R, open('savejson.txt', 'w'))
>>> open('savejson.txt').read()
'{"emp": {"who": "Bogdan"}, "job": ["dev", 1.5]}
>>> json.load(open('savejson.txt'))
{'emp': {'who': 'Bogdan'}, 'job': ['dev', 1.5]}

>>> R = dict(title='PyRef5E', pub='orm', year=2014)
>>> J = json.dumps(R, indent=4)
>>> P = json.loads(J)
>>> P
{'rok': 2014, 'tytuł': 'PyRef5E', 'wyd': 'orm'}
>>> print(J)
{
    "rok": 2014,
    "tytuł": "PyRef5E",
    "wyd": "orm"
}
```

Moduł subprocess

Poniżej omówiono narzędzia do uruchamiania *wierszy poleceń*, korzystania ze związanych z nimi trzech strumieni, pobierania kodów wyjścia oraz definiowania uruchomień powłoki alternatywnie do niektórych narzędzi modułu *os*, takich jak *os.popen()* i *os.spawnv()*. Więcej informacji na ten temat można znaleźć w podrozdziale „Moduł systemowy *os*” oraz w dokumentacji Pythona. Wskazówka: nie należy używać tych narzędzi do uruchamiania niezaufanych ciągów poleceń powłoki, ponieważ mogą one inicjować dowolne komendy, na które pozwala proces Pythona. W kodzie zamieszczonym poniżej skrypt *m.py* wyświetla zawartość zmiennej *sys.argv* wiersza polecenia:

```
>>> from subprocess import call, Popen, PIPE
>>> call('python m.py -x', shell=True)
['m.py', '-x']
0
>>> pipe = Popen('python m.py -x', stdout=PIPE)
>>> pipe.communicate()
(b"['m.py', '-x']\r\n", None)
>>> pipe.returncode
0
>>> pipe = Popen('python m.py -x', stdout=PIPE)
```

```
>>> pipe.stdout.read()
b"['m.py', '-x']\r\n"
>>> pipe.wait()
0
```

Moduł enum

Dostępny od Pythona 3.4. Zapewnia standardową obsługę *typów wyliczeniowych* — zbiorów nazw symbolicznych (nazywanych również *elementami*) powiązanych z unikatowymi, stałymi wartościami. Wywołań z tego modułu nie należy mylić z wywołaniem `enumerate()` wykorzystywanym do sekwencyjnego wyliczania wyników iteratora (patrz „Funkcje wbudowane”):

```
>>> from enum import Enum
>>> class PyBooks(Enum):
    Learning5E = 2013
    Programming4E = 2011
    PocketRef5E = 2014

>>> print(PyBooks.PocketRef5E)
PyBooks.PocketRef5E
>>> PyBooks.PocketRef5E.name,
    PyBooks.PocketRef5E.value
('PocketRef5E', 2014)

>>> type(PyBooks.PocketRef5E)
<enum 'PyBooks'>
>>> isinstance(PyBooks.PocketRef5E, PyBooks)
True
>>> for book in PyBooks: print(book)
...
PyBooks.Learning5E
PyBooks.Programming4E
PyBooks.PocketRef5E

>>> bks = Enum('Books', 'LP5E PP4E PR5E')
>>> list(bks)
[<Books.LP5E: 1>, <Books.PP4E: 2>, <Books.PR5E: 3>]
```

Moduł struct

Moduł `struct` dostarcza interfejs do parsowania i konstruowania *spakowanych binarnych danych* jako łańcuchów znaków. Mają one przypominać konstrukcje `struct` języka C. Moduł jest powszechnie używany w połączeniu z binarnymi trybami otwarcia pliku `rb` i `wb` lub innymi źródłami danych binarnych. Informacje na temat typu danych *format* oraz kodów *endian* można znaleźć w podręczniku *Python Library Reference*.

```
łańcuch = struct.pack(format, v1, v2, ...)
```

Zwraca *łańcuch* (w Pythonie 3.X bytes oraz str w Pythonie 2.X) zawierający wartości *v1*, *v2* itp., spakowane zgodnie z podanym łańcuchem *format*. Argumenty muszą dokładnie pasować do wartości wymaganych przez kody typu formatu. Ciąg formatu może określać format endian wyniku. Służy do tego jego pierwszy znak. Mogą w nim być również zapisane liczniki powtórzeń dla indywidualnych kodów typu.

```
krotka = struct.unpack(format, łańcuch_znaków)
```

Rozpakowuje *łańcuch_znaków* (w Pythonie 3.X bytes oraz str w Pythonie 2.X) do postaci krotki obiektów Pythona zgodnie z podanym łańcuchem formatu.

```
struct.calcsize(format)
```

Zwraca rozmiar struktury (a tym samym łańcucha znaków) odpowiadający określonemu formatowi.

Poniżej pokazano przykład, który demonstruje, w jaki sposób w Pythonie 3.X można pakować i rozpakowywać dane, posługując się funkcją `struct` (w Pythonie 2.X wykorzystywane są zwykle łańcuchy str zamiast łańcuchów bytes). Python 3.X, począwszy od wersji 3.2, wymaga typu bytes dla wartości s zamiast str. Wartość '4si' w poniższym kodzie oznacza to samo co wyrażenie `char[4]+int` języka C):

```
>>> import struct
>>> data = struct.pack('4si', b'spam', 123)
>>> data
b'spam{\x00\x00\x00'
>>> x, y = struct.unpack('4si', data)
>>> x, y
(b'spam', 123)

>>> open('data', 'wb').write(
    struct.pack('>if', 1, 2.0))
8
>>> open('data', 'rb').read()
b'\x00\x00\x00\x01@\x00\x00\x00'
>>> struct.unpack('>if', open('data', 'rb').read())
(1, 2.0)
```

Moduły obsługi wątków

Wątki to „lekkie” procesy, które współdzielą pamięć globalną (tzn. zasięgi leksykalne oraz wewnętrzne struktury interpretera). Wszystkie one *uruchamiają funkcje równolegle (współbieżnie)* w obrębie tego samego procesu. Moduły obsługi wątków Pythona umożliwiają przenośną pracę pomiędzy platformami. Są one przystosowane do uruchamiania

nieblokujących zadań w kontekstach związanych z operacjami wejścia-wyjścia oraz interfejsem użytkownika.

Patrz także opis funkcji `setcheckinterval()` i `setswitchinterval()` w podrzdziale „Moduł `sys`”. Warto się również zapoznać ze standardowym modulem bibliotecznym `multiprocessing`, który implementuje API bazujące na wątkach do obsługi przenośnych procesów.

`_thread` (w Pythonie 2.X nosi nazwę `thread`)

Podstawowy, niskopoziomowy interfejs obsługi wątków w Pythonie. Udostępnia narzędzia do uruchamiania, zatrzymywania i synchronizacji funkcji działających równolegle. Aby w nowym wątku uruchomić funkcję *funkcja* z argumentami pozycyjnymi pochodzącymi z krotki *krotka_argumentów* oraz argumentami kluczowymi ze słownika *krotka_argumentów*, należy skorzystać z wywołania: `_thread.start_new_thread(funkcja, argumenty [, krotka_argumentów])`. Funkcja `start_new_thread` jest synonimem funkcji `start_new` (w Pythonie 3.X jest ona uważana za przestarzałą). Do synchronizacji wątków wykorzystuje się blokady: *blokada*=`thread.allocate_lock()`; *blokada.acquire()*; *obiekty_aktualizacji*; *blokada.release()*.

`threading`

Moduł `threading` bazuje na module `thread`. Dostarcza klasy obsługi wątków, które można przystosować do własnych potrzeb. Są to klasy: `Thread`, `Condition`, `Semaphore`, `Lock`, `Timer`, wątki obsługi demonów, złączenia wątków (oczekiwanie) itp. Aby przeciążyć metodę uruchamiania akcji, należy utworzyć klasę podrzędną klasy `Thread`. Moduł daje większe możliwości niż `_thread`, ale jednocześnie wymaga więcej kodu w prostszych przypadkach użycia.

`queue` (w Pythonie 2.X nosi nazwę `Queue`)

Obsługa kolejki FIFO złożonej z implementacji obiektów dla wielu producentów i wielu konsumentów. Jest szczególnie przydatna w przypadku aplikacji z obsługą wątków. Blokuje operacje `get()` i `put()` w celu synchronizacji dostępu do danych w kolejce. Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference*.

API baz danych Python SQL

Interfejs API dostępu do *relacyjnych baz danych* SQL w Pythonie zapewnia przenośność skryptów pomiędzy pakietami obsługi baz danych SQL różnych producentów. Dla każdego producenta należy zainstalować

specyficzny moduł rozszerzeń, ale skrypty można pisać zgodnie z przenośnym interfejsem API dostępu do baz danych. Po migracji do pakietu baz danych innego producenta standardowe skrypty bazodanowe SQL powinny działać bez zmian.

Należy zwrócić uwagę, że większość modułów rozszerzeń baz danych nie należy do standardowej biblioteki Pythona. Są to *komponenty zewnętrzne*, które trzeba pobrać i zainstalować osobno. Wyjątek: wbudowany pakiet obsługi relacyjnych baz danych SQLite jest dołączony do Pythona w postaci *standardowego modułu bibliotecznego* `sqlite3`. Zawiera on funkcje utrwalania danych oraz tworzenia prototypów.

Z prostszymi alternatywami mechanizmów utrwalania danych można się zapoznać w podrozdziale „Moduły utrwalania obiektów”. Warto również się zaznajomić z dodatkowymi popularnymi narzędziami baz danych zewnętrznych producentów. Należą do nich system przechowywania dokumentów *MongoDB* bazujący na JSON; obiektowe bazy danych, takie jak *ZODB* itp. Dostępne są również systemy mapowania obiektowo-relacyjnego, na przykład *SQLAlchemy* i *SQLObject*, a także interfejsy API bazujące na chmurze, na przykład magazyn danych *App Engine*.

Przykład użycia interfejsu API

W poniższym fragmencie kodu wykorzystano standardowy moduł biblioteczny SQLite. Dla zaoszczędzenia miejsca pominięto niektóre zwracane wartości. Sposób użycia dla innych korporacyjnych systemów baz danych, takich jak MySQL, PostgreSQL i Oracle, jest podobny, ale wymaga zastosowania innych parametrów połączenia oraz instalacji modułów rozszerzeń. W niektórych przypadkach obsługiwane są specyficzne dla producenta (a tym samym nieprzenośne) rozszerzenia SQL.

```
>>> from sqlite3 import connect
>>> conn = connect(r'C:\code\temp.db')
>>> curs = conn.cursor()

>>> curs.execute('create table emp (who, job, pay)')
>>> prefix = 'insert into emp values '
>>> curs.execute(prefix + "('Bogdan', 'inf', 100)")
>>> curs.execute(prefix + "('Zofia', 'inf', 120)")

>>> curs.execute("select * from emp where pay > 100")
>>> for (who, job, pay) in curs.fetchall():
...     print(who, job, pay)
...
Zofia inf 120
```

```
>>> result = curs.execute("select who, pay from emp")
>>> result.fetchall()
[('Bogdan', 100), ('Zofia', 120)]
>>> query = "select * from emp where job = ?"
>>> curs.execute(query, ('inf',)).fetchall()
[('Bogdan', 'inf', 100), ('Zofia', 'inf', 120)]
```

Interfejs modułu

W kolejnych podrozdziałach zamieszczono *częściową* listę eksportów modułu. Szczegółowe informacje, które tu pominięto, można znaleźć w pełnej specyfikacji interfejsu API pod adresem <http://www.python.org>. Poniżej omówiono narzędzia dostępne na najwyższym poziomie interfejsu modułu (*dbmod*):

dbmod.connect(parametry...)

Konstruktor dla obiektów połączenia (*połączenie*). Reprezentuje połączenie do bazy danych. Parametry są specyficzne dla producenta bazy danych.

dbmod.paramstyle

Łańcuch znaków informujący o typie formatowania markera parametrów (np. w stylu qmark = ?).

dbmod.Warning

Wyjątek zgłaszany w przypadku ważnych ostrzeżeń (dotyczących na przykład usuwania danych).

dbmod.Error

Wyjątek będący klasą bazową dla wszystkich pozostałych wyjątków.

Obiekty połączeń

Obiekty połączeń (*połączenie*) odpowiadają na następujące metody:

połączenie.close()

Natychmiastowe zamknięcie połączenia (w odróżnieniu od zamknięcia w momencie wywołania metody `__del__`).

połączenie.commit()

Zatwierdzenie zaległych transakcji i zapisanie ich do bazy danych.

połączenie.rollback()

Cofnięcie bazy danych do stanu z początku zaległej transakcji. Zamknięcie połączenia bez zatwierdzenia zmian powoduje niejawne cofnięcie bazy danych do stanu sprzed nawiązania połączenia.

`połączenie.cursor()`

Zwraca nowy obiekt kursora (*kursor*) do przesyłania ciągów znaków poleceń SQL w połączeniu.

Obiekty kursora

Obiekty kursora (*kursor*) reprezentują kursory bazy danych i służą do zarządzania kontekstem operacji pobierania danych:

`kursor.description`

Ciąg siedmioelementowych sekwencji. Każda z nich zawiera informacje opisujące jedną kolumnę wyników: (*nazwa*, *kod_typu*, *rozmiar_wyświetlany*, *rozmiar_wewnętrzny*, *precyzja*, *skala*, czy ↪ *dozwolone_wartości_null*).

`kursor.rowcount`

Określa liczbę wierszy zwróconych przez ostatnią instrukcję `execute*` (w przypadku instrukcji DQL takich jak `select`) lub liczbę wierszy, których instrukcja dotyczyła (w przypadku takich instrukcji DML jak `update` lub `insert`).

`kursor.callproc(nazwaproc [, parametry])`

Wywołuje z bazy danych składowaną procedurę o podanej nazwie. Sekwencja parametrów musi zawierać po jednej pozycji dla każdego argumentu, jakiego oczekuje procedura. Wynik jest zwracany w postaci zmodyfikowanej kopii parametrów wejściowych.

`kursor.close()`

Natychmiastowe zamknięcie kursora (w odróżnieniu od zamknięcia w momencie wywołania metody `__del__`).

`kursor.execute(operacja [,parametry])`

Przygotowuje i uruchamia operację bazodanową (zapytanie bądź polecenie). Wstawienie wielu wierszy w pojedynczej operacji wymaga podania parametrów w postaci listy krotek (choć preferowaną funkcją do takiego celu jest `executemany()`).

`kursor.executemany(operacja, sekwencja_parametrów)`

Przygotowuje operację bazodanową (zapytanie lub polecenie) i uruchamia je dla wszystkich sekwencji parametrów lub odwzorowań w sekwencji *sekwencja_parametrów*. Działanie funkcji jest podobne do wielu wywołań funkcji `execute()`.

`kursor.fetchone()`

Pobiera następny wiersz z zestawu wyników zapytania. Zwraca pojedynczą sekwencję lub `None`, jeśli kolejne dane nie są dostępne.

Przydatne w przypadku dużych zbiorów danych lub przy małych szybkościach dostarczania rekordów.

`kursor.fetchmany([size=kursor.arraysize])`

Pobiera następny zbiór wierszy z zestawu wyników zapytania. Zwraca pojedynczą sekwencję sekwencji (np. listę krotek). W przypadku braku kolejnych wierszy zwracana jest pusta sekwencja.

`kursor.fetchall()`

Pobiera wszystkie (pozostałe) wiersze z zestawu wyników zapytania. Zwraca je jako sekwencję sekwencji (np. listę krotek).

Obiekty typów i konstruktory

`Date(rok, miesiąc, dzień)`

Tworzy obiekt, który przechowuje datę.

`Time(godzina, minuta, sekunda)`

Tworzy obiekt, który przechowuje godzinę.

`None`

Wartości SQL NULL w parametrach wejściowych i wynikach są reprezentowane przez wartość Pythona `None`.

Idiomy Pythona i dodatkowe wskazówki

W tym podrozdziale zestawiono powszechnie stosowane triki kodowania w Pythonie oraz ogólne wskazówki na temat posługiwania się tym językiem — poza tymi, które opisano na kartach tej książki. Więcej informacji na zasygnalizowane tu tematy można znaleźć w podręcznikach *Python Library Reference* oraz *Python Language Reference*, dostępnych pod adresem <http://www.python.org/doc/>.

Wskazówki dotyczące rdzenia języka

- `S[:]` tworzy kopię najwyższego poziomu (tzw. płytką kopię) dowolnej sekwencji. Aby utworzyć pełną kopię, należy skorzystać z wywołania `copy.deepcopy(X)`. Instrukcje `list(L)` oraz `D.copy()` służą do kopiowania list i słowników (od wersji 3.3 służy do tego także wywołanie `L.copy()`).
- Instrukcja `L[:0]=obiekt_iterowalny` wstawia elementy na początek listy `L` w miejscu.

- Instrukcje `L[len(L):]=obiekt_iterowalny`, `L.extend(obiekt_iterowalny)` oraz `L+=obiekt_iterowalny` wstawiają wiele elementów na koniec listy `L` w miejscu.
- Metody `L.append(X)` i `X=L.pop()` można wykorzystać do zaimplementowania działań w miejscu na stosie, przy czym koniec listy jest wierzchołkiem stosu.
- Do przeglądania słowników w pętli można użyć instrukcji `klucz in D.keys()`. W wersji 2.2 oraz w wersjach późniejszych można po prostu skorzystać z instrukcji `klucz in D`. W Pythonie 3.X te dwie formy instrukcji są równoważne, ponieważ metoda `keys()` zwraca obiekt iterowalny.
- W celu przetwarzania w pętli kluczy słowników w wersji 2.4 i późniejszych należy użyć instrukcji `klucz in sorted(D)`. Forma `K=D.keys(); K.sort(); for klucz in K:` działa także w Pythonie 2.X, ale nie w 3.X, ponieważ wyniki metody `keys()` są obiektami widoków, a nie listami.
- Instrukcja `X=A or B or None` przypisuje do `X` pierwszy obiekt o wartości `true` spośród obiektów `A` i `B`. Jeśli oba mają wartość `false` (tzn. mają wartość 0 lub pustą), przypisuje `None`.
- Instrukcja `X,Y = Y,X` zamienia wartości `X` i `Y` bez konieczności jawnego przypisywania `X` do zmiennej tymczasowej.
- Instrukcja `red, green, blue = range(3)` przypisuje ciąg liczb całkowitych w postaci prostego typu wyliczeniowego złożonego z nazw. W roli typu wyliczeniowego można również wykorzystać atrybuty klas i słowniki. W Pythonie 3.4 i wersjach późniejszych można skorzystać z bardziej jawnej i funkcjonalnie bogatszej formy typów wyliczeniowych w standardowym module bibliotecznym `enum`.
- Aby zapewnić wykonanie dowolnego końcowego bloku kodu, należy skorzystać z konstrukcji `try/finally`. Jest to szczególnie użyteczne podczas korzystania z blokad (założenie blokady przed wykonaniem instrukcji `try`, zwolnienie w bloku `finally`).
- Aby zapewnić wykonanie końcowego bloku kodu specyficznego dla obiektu, należy skorzystać z konstrukcji `with/as`. Konstrukcję tę wykorzystuje się w odniesieniu do obiektów obsługujących protokół menedżera kontekstu (np. automatyczne zamknięcie pliku, automatyczne zwolnienie blokady wątku).

- Aby interaktywnie przeglądać wszystkie wyniki obiektów iterowalnych w Pythonie 3.X, należy opakować je w wywołanie `list()`. Obejmuje to takie obiekty jak `range()`, `map()`, `zip()`, `filter()`, `dict.keys()` i wiele innych.

Wskazówki dotyczące środowiska

- W celu dodania kodu samotestującego lub wywołania funkcji `main` należy skorzystać z instrukcji `if __name__ == '__main__':` na końcu modułu. Dotyczy to wyłącznie wykonywania kodu, a nie sytuacji, gdy jest on importowany jako komponent biblioteczny.
- Aby załadować zawartość pliku w pojedynczym wyrażeniu, należy użyć instrukcji `dane=open(nazwa_pliku).read()`. Poza implementacją CPython wymuszenie natychmiastowego odzyskiwania zasobów systemowych (np. wewnątrz pętli) może wymagać jawnego wywołania funkcji `close`.
- W celu przetwarzania plików tekstowych wiersz po wierszu w wersji 2.2 Pythona i w wersjach późniejszych należy skorzystać z instrukcji `for wiersz in plik` (w starszych wersjach należy skorzystać z instrukcji `for wiersz in plik.readlines()`).
- Do pobierania argumentów wiersza poleceń służy instrukcja `sys.argv`.
- Aby pobrać ustawienia środowiska powłoki, należy użyć odwołania `os.environ`.
- Standardowe strumienie to: `sys.stdin`, `sys.stdout` oraz `sys.stderr`.
- Aby uzyskać listę plików pasujących do podanego wzorca, należy użyć wywołania: `glob.glob(wzorzec)`.
- Aby uzyskać listę plików i podkatalogów z określonej ścieżki (np. `"."`), należy skorzystać z wywołania `os.listdir(ścieżka)`.
- Aby przeglądać całe drzewo katalogów w Pythonie 3.X i 2.X, należy użyć funkcji `os.walk()` (w Pythonie 2.X jest również dostępna funkcja `os.path.walk()`).
- Aby uruchomić polecenia powłoki z poziomu skryptów Pythona, można skorzystać z wywołań `os.system(wiersz_polecenia)`, `wyjście=os.popen('wiersz_polecenia', 'r').read()`. Instrukcja w drugiej postaci czyta standardowe wyjście uruchomionego programu. Można ją także wykorzystać do czytania wyjścia wiersz po wierszu.

- W Pythonie 3.X i 2.X za pośrednictwem modułu `subprocess` są dostępne także inne strumienie uruchomionego polecenia. W Pythonie 2.X są również dostępne wywołania `os.popen2/3/4()`. Na platformach uniksowych podobny efekt dają wywołania `os.fork()/os.exec*()`.
- Aby skrypt Pythona stał się wykonywalny na platformach uniksowych, należy dodać na jego początku wiersz `#!/usr/bin/env python` lub `#!/usr/local/bin/python` i nadać mu uprawnienia do uruchamiania za pomocą polecenia `chmod`.
- W systemie Windows pliki uruchamiają się bezpośrednio po kliknięciu ze względu na ustawienia w rejestrze. Od wersji 3.3 mechanizm launchera w Windowsie rozpoznaje również sekwencję typową dla Uniksa - `#!/`, patrz "Python Launcher dla systemu Windows".
- Funkcje `print()` i `input()` (w Pythonie 2.X znane jako `print` i `raw_input()`) wykorzystują strumienie `sys.stdout` oraz `sys.stdin` — w celu wewnętrznego przekierowania wejścia-wyjścia należy przypisać je do obiektów plikowych. W Pythonie 3.X można też skorzystać z instrukcji `print(..., file=F)` (lub z formy `print >> F, ...` w Pythonie 2.X).
- W przypadku niepowodzenia podczas wyświetlania tekstu Unicode spoza kodu ASCII, na przykład nazw plików i innej zawartości, należy ustawić zmienną środowiskową `PYTHONIOENCODING` na wartość `utf8` (lub inną).

Wskazówki dotyczące użytkowania

- Aby włączyć eksperymentalne własności języka, należy skorzystać z instrukcji `from __future__ import nazwawłasności` (włączenie tych własności może spowodować problemy z działaniem istniejącego kodu).
- Intuicyjne postrzeganie wydajności programów w Pythonie jest zwykle błędne — przed podjęciem prób optymalizacji bądź migracją do języka C zawsze należy dokonać odpowiednich pomiarów. Do tego celu można użyć modułów `profile`, `time` i `timeit` (a także `cProfile`).
- Warto się również zapoznać z modułami `unittest` (znanym także jako `PyUnit`) oraz `doctest`. Są w nich dostępne narzędzia do automatycznego testowania, dostarczane razem ze standardową

biblioteką Pythona. Moduł `unittest` to framework klas. Mechanizmy modułu `doctest` pozwalają na skanowanie ciągów dokumentacyjnych w poszukiwaniu testów i wyników.

- Funkcja `dir([obiekt])` przydaje się do inspekcji przestrzeni nazw atrybutów. Wywołanie `print(obiekt.__doc__)` często daje dostęp do dokumentacji.
- Funkcja `help([obiekt])` daje dostęp do interaktywnej pomocy dla modułów, funkcji, typów itp. Wywołanie `help(str)` zwraca tekst pomocy dla typu `str`; wywołanie `help('moduł')` wyświetla pomoc na temat modułów nawet wtedy, gdy nie zostały one jeszcze zaimportowane, natomiast wywołanie `help('temat')` zwraca pomoc dotyczącą słów kluczowych i innych tematów pomocy (aby uzyskać listę tematów pomocy, należy użyć wywołania `help(↪('tematy'))`).
- W dokumentacji modułu `pydoc` można znaleźć informacje na temat wydobywania i wyświetlania ciągów dokumentacyjnych powiązanych z modułami, funkcjami, klasami i metodami. Począwszy od wersji 3.2, polecenie `python -m pydoc -b` powoduje uruchomienie interfejsu dla dokumentacji PyDoc w przeglądarce (dla trybu klienta GUI można zastąpić opcję `-b` opcją `-g`).
- Więcej informacji na temat wyłączania ostrzeżeń o niezalecanych właściwościach języka można znaleźć w podrozdziale „Framework ostrzeżeń” oraz w opisie flagi wiersza poleceń `-W`, w podrozdziale „Opcje wiersza poleceń Pythona”.
- Z możliwościami dystrybucji programów w Pythonie można się zapoznać w dokumentacji programów *Distutils* i *eggs*. Zagadnienia dystrybucji programów opisano również w kolejnych punktach.
- Programy *PyInstaller*, *py2exe*, *cx_freeze*, *py2app* i inne umożliwiają pakowanie programów Pythona w formie niezależnych plików wykonywalnych (plików *.exe* w systemie Windows).
- *NumPy*, *SciPy*, *Sage* oraz powiązane z nimi pakiety to rozszerzenia przekształcające Pythona w naukowe narzędzie programistyczne zawierające między innymi obiekty wektorów, biblioteki matematyczne itp. W Pythonie 3.4 zapowiadane jest wprowadzenie nowego podstawowego modułu bibliotecznego *statistics*.
- Pakiet *ZODB* zawiera pełną obsługę OODB, która pozwala na utrwalanie natywnych obiektów Pythona według klucza. Pakiety *SQLObject* i *SQLAlchemy* zawierają mechanizmy odwzorowań

obiektoowo-relacyjnych. Dzięki nim można używać klas w połączeniu z relacyjnymi tabelami. System *MongoDB* dostarcza bazującą na JSON opcję bazodanową NoSQL.

- *Django*, *App Engine*, *Web2py*, *Zope*, *Pylons*, *TurboGears* to przykłady frameworków do tworzenia aplikacji webowych w Pythonie.
- Program *SWIG* (między innymi) to narzędzie generujące kod umożliwiający używanie bibliotek języków C i C++ w obrębie skryptów Pythona.
- *IDLE* to środowisko programistyczne z interfejsem GUI dostarczane z Pythonem. Jest wyposażone w edytory tekstu z mechanizmami kolorowania składni, przeglądarki obiektów, narzędzia debugowania itp. Inne opcje zintegrowanych środowisk programistycznych to między innymi *PythonWin*, *Komodo*, *Eclipse* i *NetBeans*.
- W pomocy programu *Emacs* można znaleźć wskazówki dotyczące edycji (uruchamiania) kodu w środowisku edytora tekstowego Emacs. Większość innych edytorów także obsługuje Pythona (zawierają między innymi mechanizmy automatycznych wcięć, kolorowania itp.). Należą do nich edytory *VIM* oraz *IDLE*. Patrz zakładka *editors* w witrynie *www.python.org*.
- Przenoszenie kodu do Pythona 3.X — w celu generowania ostrzeżeń o niezgodności należy w Pythonie 2.X skorzystać z opcji wiersza polecenia `-3`. Skrypt *2to3* pozwala na automatyczną konwersję większości kodu z wersji 2.X na kod działający w Pythonie 3.X. Warto się również zapoznać z systemem *six*, który dostarcza warstwę zapewniającą kompatybilność 2.X/3.X. Program *3to2* umożliwia konwersję kodu 3.X na kod działający w środowisku interpreterów 2.X. Podobnie program *pies* umożliwia kompatybilność pomiędzy różnymi liniami Pythona.

Różne wskazówki

- Interesujące witryny internetowe, na które warto zaglądać:

<http://www.python.org>

Macierzysta strona Pythona.

<http://oreilly.com>

Macierzysta strona wydawnictwa O'Reilly.

<http://www.python.org/pypi>

Dodatkowe zewnętrzne narzędzia Pythona.

<http://www.rmi.net/~lutz>

Strona internetowa autora tej książki.

- Filozofia Pythona: `import this`.
- W przykładach kodu Pythona należy używać zmiennych `spam` i `eggs` zamiast `foo` i `bar`.
- Zawsze patrz na jasną stronę życia.

A

ASCII, *Patrz:* typ tekstowy ASCII
asercja, 104

B

backport, *Patrz:* poprawka
bajt, 22, 23
baza danych relacyjna, 232
biblioteka
 Tk, 217, 220
 tkinter, 217
błąd
 interpretera, 168
 sekwencji, 164
 składni, 167, 171
 systemowy, 164, 169
bufor, 158

C

cel
 operatora %, 33, 34
 zagnieżdżanie, 35
 zastępowanie, 33, 35, 36
ciąg
 pusty, 30
 tekstowy, 39, 46
 znaków, 22, 23, 30, 138, 159
 ' \t\n\r\v\f', 185
 '01234567', 185
 '0123456789', 184
 '0123456789abcdefABCDEF', 185
 'abcdefghijklmnopqrstuvwxyz',
 184

'ABCDEFGHIJKLMNOPQRSTUVWXYZ
UVWXYZ', 184
formatowanie, 32, 33, 34
internowanie, 178
przestankowych, 185

D

debugger, 182
decimal, *Patrz:* liczba dziesiętna
dekodowanie Unicode, 153
dekorator, 88, 98, 110, 150
 @staticmethod, 152
depth first left right, *Patrz:* DFLR
deskryptor, 131
 klas, 192
 plików, 192
 pliku, 191, 193, 200
destruktor, 120
DFLR, 114
diament, 113, 115
diamond pattern, *Patrz:* diament
dictionary comprehension, *Patrz:*
 słownik składany
domknięcie, 109, 110
dowiązanie symboliczne, 202
dziedziczenie, 111, 114, 163, 192
 nowego stylu, 115, 116, 117
 wielokrotne, 113

E

egzemplarz
 atrybut, 117
 tworzenie, 119
element widok, 57

F

- fraction, *Patrz*: ułamek
- funkcja, *Patrz też*: instrukcja, moduł
 - `__import__`, 141
 - abs, 135
 - adnotacja, 86
 - all, 135
 - anonimowa, 18
 - any, 135
 - apply, 157
 - argument, 85
 - domyślny, 87
 - kluczowy, 86
 - ascii, 136, 157
 - basestring, 158
 - bin, 136
 - bool, 121, 136
 - buffer, 158
 - bytearray, 136
 - bytes, 121, 136
 - callable, 137
 - chr, 137
 - classmethod, 89, 98, 137
 - cmp, 158
 - coerce, 158
 - compile, 106, 137, 204, 206
 - complex, 138
 - deklaracja opisująca, *Patrz*:
 - dekorator
 - delattr, 138
 - dict, 57, 138
 - dir, 125, 138
 - divmod, 138
 - enumerate, 138
 - eval, 106, 139, 150, 167
 - exec, 106, 139, 157, 167
 - execfile, 106, 158
 - execv, 198, 199
 - execve, 198
 - execvp, 198
 - fabryka, 88
 - file, 159
 - filter, 54, 139
 - float, 140
 - format, 121, 140
 - frozenset, 66, 140
 - generators, 18, 89, 90
 - getattr, 140
 - getswitchinterval, 177, 178
 - globals, 140
 - hasattr, 140
 - hash, 140
 - help, 72, 141
 - hex, 134, 141
 - id, 141
 - imp.reload, 93, 160
 - importlib.import_module, 93
 - input, 141, 159, 160, 165
 - int, 142
 - intern, 159
 - isinstance, 142
 - issubclass, 142
 - iter, 54, 142
 - len, 125, 143
 - list, 143
 - locals, 143
 - lokalna, 85
 - long, 159
 - map, 53, 54, 143
 - max, 144
 - memoryview, 144, 157
 - metoda klasy, 75, 98
 - min, 144
 - namiaszka, 84
 - next, 54, 55, 126, 142, 144, 167
 - normpath, 201
 - object, 145
 - oct, 145
 - okalaająca, 108, 109
 - open, 61, 145, 149, 159, 162, 192
 - operator.index, 130
 - ord, 149, 160
 - os.popen, 189
 - os._exit, 199
 - os.abort, 198
 - os.access, 197
 - os.altsep, 187
 - os.chdir, 191, 194
 - os.chmod, 194
 - os.chown, 194
 - os.close, 192
 - os.defpath, 188
 - os.devnull, 188

- os.dup, 192
- os.environ, 190
- os.execl, 198
- os.execle, 198
- os.execlp, 198
- os.execv, 199
- os.execve, 198
- os.execvp, 199
- os.execvpe, 199
- os.extsep, 187
- os.fdopen, 192
- os.fork, 199
- os.fstat, 192
- os.ftruncate, 192
- os.getcwd, 191, 194
- os.getenv, 190
- os.geteuid, 199
- os.getpid, 199
- os.getppid, 199
- os.getuid, 199
- os.isatty, 193
- os.kill, 200
- os.linesep, 188
- os.link, 194
- os.listdir, 194
- os.lseek, 193
- os.lstat, 195
- os.makedirs, 195
- os.mkdir, 195
- os.mkfifo, 195, 200
- os.nice, 200
- os.open, 193, 194
- os.path.abspath, 201
- os.path.basename, 201
- os.path.commonprefix, 201
- os.path.dirname, 202
- os.path.exists, 202
- os.path.expanduser, 202
- os.path.expandvars, 202
- os.path.getatime, 202
- os.path.getmtime, 202
- os.path.getsize, 202
- os.path.isabs, 202
- os.path.isdir, 202
- os.path.isfile, 202
- os.path.islink, 202
- os.path.ismount, 202
- os.path.join, 203
- os.path.normcase, 203
- os.path.normpath, 203
- os.path.realpath, 203
- os.path.samefile, 203
- os.path.sameopenfile, 203
- os.path.samestat, 203
- os.path.split, 203
- os.path.splitdrive, 203
- os.pathsep, 188
- os.pipe, 193, 200
- os.plock, 200
- os.putenv, 190
- os.read, 193
- os.readlink, 196
- os.remove, 196
- os.removedirs, 196
- os.rename, 196
- os.renames, 196
- os.rmdir, 196
- os.spawnv, 200
- os.spawnve, 200
- os.startfile, 189
- os.stat, 196
- os.strerror, 191
- os.symlink, 196
- os.system, 188
- os.times, 191
- os.umask, 191
- os.uname, 191
- os.unlink, 196
- os.utime, 197
- os.wait, 200
- os.waitpid, 201
- os.walk, 197
- os.write, 193
- pow, 149
- print, 149, 150, 157
- property, 89, 113, 150
- range, 150, 161
- raw_input, 160, 165
- re.compile, 106, 137, 204, 206
- reduce, 160
- reload, 93, 160
- repr, 120, 136, 150, 157
- reversed, 151
- round, 151

funkcja, *Patrz też*: instrukcja, moduł
 równoległa, 231
 set, 66, 151
 setattr, 151
 setcheckinterval, 232
 setswitchinterval, 232
 slice, 151
 sorted, 152
 staticmethod, 89, 98, 152
 str, 121, 150, 152, 153, 161
 string.capwords, 184
 string.Formatter, 184
 string.maketrans, 184
 string.Template, 184
 sum, 153
 super, 8, 115, 153, 154
 sys.__stderr__, 182
 sys.__stdin__, 182
 sys.__stdout__, 182
 sys._getframe, 177
 sys.argv, 175
 sys.builtin_module_names, 175
 sys.byteorder, 175
 sys.copyright, 176
 sys.displayhook, 176
 sys.dont_write_bytecode, 176
 sys.exc_info, 176
 sys.excepthook, 176
 sys.exec_prefix, 176
 sys.executable, 177
 sys.exit, 177
 sys.flags, 177
 sys.float_info, 177
 sys.getcheckinterval, 177
 sys.getdefaultencoding, 177
 sys.getfilesystemencoding, 177
 sys.getrecursionlimit, 178
 sys.getrefcount, 178
 sys.getsizeof, 178
 sys.getswitchinterval, 177, 178
 sys.getwindowsversion, 178
 sys.hexversion, 178
 sys.implementation, 178
 sys.int_info, 178
 sys.intern, 178
 sys.last_traceback, 179
 sys.last_type, 179
 sys.last_value, 179
 sys.maxsize, 179
 sys.maxunicode, 179
 sys.modules, 179
 sys.path, 179
 sys.platform, 180
 sys.prefix, 180
 sys.ps1, 180
 sys.ps2, 180
 sys.setcheckinterval, 180
 sys.setdefaultencoding, 181
 sys.setprofile, 181
 sys.setrecursionlimit, 181
 sys.setswitchinterval, 181
 sys.settrace, 182
 sys.stderr, 182
 sys.stdin, 182
 sys.stdout, 182
 sys.sys.version, 183
 sys.thread_info, 183
 sys.tracebacklimit, 183
 sys.version_info, 183
 sys.winver, 183
 śladu systemu, 182
 tuple, 155
 tworzenie, 85
 type, 68, 113, 155
 unichr, 160
 unicode, 161
 vars, 156
 warnings.warn, 171
 wbudowana, 108
 nazwa, 74
 współbieżna, 231
 xrange, 161
 zagnieżdżona, 91, 109
 zip, 156

G

garbage collector, *Patrz*: odśmianie
 geometry manager, *Patrz*: menedżer
 geometrii
 gniazdo, 113
 GUI, 15, 217

H

hash, *Patrz:* skróót

I

indeksowanie, 19, 25

instrukcja, 71, 75, *Patrz też:* funkcja

assert, 104, 165

blok, 71

break, 84, 100

class, 74, 97, 98, 111

continue, 84, 100

def, 85, 87, 106, 111

del, 84, 138

exec, 106, 141

for, 53, 54, 83, 84, 90, 152

from, 95, 96, 97, 141, 165

global, 91

if, 82

import, 92, 93, 95, 141, 165

pakiet, 93

nonlocal, 91, 106, 109

pass, 84

print, 80, 81, 82, 106

przypisania, 75, 98, 107, 111

krotek, 76

sekwencji, 76, 77, 78

wielocelowego, 76

raise, 84, 100, 102, 103, 106

return, 89, 91, 100

składnia, 70

try, 84, 99, 100, 101, 103, 104, 106,
109, 164, 168

klauzula, 100, 101

while, 83, 84

with, 104, 105, 106, 131

wyrażeniowa, 79

yield, 20, 89, 90

interfejs

API, 232, 233

Berkeley DB, 211

dopasowywania wyrażeń

regularnych, 204

menedżera geometrii, 218

modułu, 234

obiektów zwracanych, 148

PyDoc, 72

usług systemu operacyjnego, 185

internet, 222

interpreter, 175, 178

interaktywny, 74

ścieżka dostępu, 177

wątek, 181

wersja, 183

inwersja, *Patrz:* operacja bitowa NOT

iteracja, 55, 138

kontekst, 54

protokół, 54, 55, 89, 126

widoku, 57

iterator, 54, 83, 89

K

klasa, 69, 97, 98, 110

atrybut, 98, 111, 117

prywatny, 112

bazowa, *Patrz:* klasa nadrzędna

egzemplarz, 102, 111

Exception, 103

gniazdo, *Patrz:* gniazdo

klasyczna, 113, 114

metoda, 111

nadrzędna, 97

nazwa, 74

nowego stylu, 103, 111, 113, 114,
116, 117, 150

Pickler, 216

String, 39

Unpickler, 216

wyjątków, *Patrz:* wyjątek klasa

klasa-deskryptor, 130

klasa-właściciel, 130

klauzula

as, 92, 96, 101, 104

else, 100

except, 100, 101, 102, 109

finally, 100, 102, 104, 168

from, 90, 102

klucz, 21, 23, 57, 126, 144, 212

krotki, 57

mapowania, 166

rejestr, 183

skrót, 121

klucz
słownika, 178
widok, 57
wyszukiwania, 178
klucz-wartość, 57
kod
bajtowy, 92
blok
wejściowy, 104
zagnieżdżony, 104
zamykający, 104
wielokrotne wykorzystanie, 98
źródłowy, 92
wcięcia, 166
kolejka FIFO, 195
kolekcja, 125
długość, 143
komentarz, 72
komunikat, 104
konstruktor, 120
krotka, 19, 22, 23, 33, 60, 61, 134, 155
klucz, 57
kodowanie, 33
przypisanie, 76
zagnieżdżona, 33
kursor, 235

L

launcher, 15, 17
liczba, 26
całkowita, 142, 159
dziesiętna, 26, 28
ułamkowa, *Patrz:* ułamek
zespolona, 26, 27, 138
zmiennoprzecinkowa, 27, 140, 142, 151
lista, 22, 23, 50, 156, 228
literał, 50
obiekt
wstawianie, 52
wyszukiwanie, 51
pozycja, 50
składana, 52, 53, 54, 55, 56, 110
z zagnieżdżonymi pętlami for, 53
sortowanie, 51
wyrażeń, 52

literał, 18, 26
b'ccc', 49
bytes, 29
'ccc', 46
listy, *Patrz:* lista literał
łańcuchowy, 31
słownikowy, *Patrz:* słownik literał
tworzenie, 30
u'ccc', 47, 49
znakowy, 31

Ł

łańcuch
wyjątków, *Patrz:* wyjątek łańcuch
znaków, *Patrz:* ciąg znaków

M

mapa, 110, 125
indeksowanie, 126
menedżer
geometrii, 218
kontekstu, 104, 131
protokół, 105
zagnieżdżony, 105
metaklasa, 99, 156
Method Resolution Order, *Patrz:* MRO
metoda, 21
__abs__, 24, 130
__add__, 119, 127
__and__, 128
__bool__, 121, 125, 132, 133
__bytes__, 121, 132
__call__, 99, 122, 137, 221
__cmp__, 22, 123, 124, 133, 158
__coerce__, 134
__complex__, 25, 130
__contains__, 22, 125
__del__, 120, 234, 235
__delattr__, 123
__delete__, 113, 131
__delitem__, 23, 24, 41, 127, 133
__delslice__, 23, 134
__dict__, 123
__dir__, 125, 132
__div__, 127, 134

<code>__divmod__</code>	128	<code>__mod__</code>	128
<code>__enter__</code>	105, 131	<code>__mul__</code>	127
<code>__eq__</code>	121, 123, 124, 133	<code>__ne__</code>	123, 124
<code>__exit__</code>	105, 106, 132	<code>__neg__</code>	24, 130
<code>__float__</code>	25, 130	<code>__new__</code>	99, 119, 120, 156
<code>__floordiv__</code>	128	<code>__next__</code>	54, 90, 126, 132, 142, 144
<code>__format__</code>	121	<code>__nonzero__</code>	121, 125, 132, 133
<code>__ge__</code>	123, 124, 133	<code>__oct__</code>	130, 133, 134, 145
<code>__get__</code>	113, 116, 117, 131	<code>__or__</code>	128
<code>__getattr__</code>	112, 113, 114, 116, 118, 122, 123, 125, 132	<code>__pos__</code>	24, 130
<code>__getattribute__</code>	113, 116	<code>__pow__</code>	128
<code>__getitem__</code>	22, 23, 54, 118, 119, 125, 126, 127, 133, 151	<code>__radd__</code>	128, 129
<code>__getslice__</code>	23, 133	<code>__rand__</code>	128
<code>__gt__</code>	123	<code>__rcmp__</code>	133
<code>__hash__</code>	57, 121, 140	<code>__rdivmod__</code>	128
<code>__hex__</code>	130, 133, 134, 141	<code>__repr__</code>	120, 121
<code>__iadd__</code>	41, 77, 129	<code>__reversed__</code>	127, 151
<code>__iand__</code>	129	<code>__rfloordiv__</code>	128
<code>__ifloordiv__</code>	129	<code>__rlshift__</code>	128
<code>__ilshift__</code>	129	<code>__rmod__</code>	41, 128
<code>__imod__</code>	129	<code>__rmul__</code>	128
<code>__import__</code>	93	<code>__ror__</code>	128
<code>__imul__</code>	41, 129	<code>__round__</code>	130, 132
<code>__index__</code>	130, 133, 134, 136, 141, 145	<code>__rpow__</code>	128
<code>__init__</code>	90, 93, 94, 99, 103, 119, 120, 156	<code>__rrshift__</code>	128
<code>__instancecheck__</code>	125, 132	<code>__rshift__</code>	128
<code>__int__</code>	24, 130	<code>__rsub__</code>	128
<code>__invert__</code>	24, 130	<code>__rtruediv__</code>	128, 134
<code>__ior__</code>	129	<code>__rxor__</code>	128
<code>__ipow__</code>	129	<code>__set__</code>	113, 117, 131
<code>__irshift__</code>	129	<code>__setattr__</code>	112, 114, 117, 118, 122, 123
<code>__isub__</code>	129	<code>__setitem__</code>	23, 41, 126, 133
<code>__iter__</code>	22, 23, 54, 55, 83, 90, 125, 126, 143	<code>__setslice__</code>	23, 134
<code>__itruediv__</code>	129, 134	<code>__slots__</code>	113, 124, 125, 132
<code>__ixor__</code>	129	<code>__str__</code>	120, 121, 135, 153
<code>__le__</code>	123	<code>__sub__</code>	127
<code>__len__</code>	23, 24, 121, 125, 126, 127, 132, 133, 151	<code>__subclasscheck__</code>	125, 132
<code>__long__</code>	24, 134	<code>__truediv__</code>	127, 132
<code>__lshift__</code>	128	<code>__unicode__</code>	135, 161
<code>__lt__</code>	22, 123, 124, 133	<code>__X__</code>	98
<code>__metaclass__</code>	99, 135	<code>__xor__</code>	128
		<code>bytes.translate</code>	184
		<code>close</code>	90, 165
		<code>D.clear</code>	58
		<code>D.copy</code>	58
		<code>D.get</code>	59

- metoda
 - D.has_key, 59
 - D.items, 58
 - D.iteritems, 59
 - D.keys, 58
 - D.pop, 59
 - D.popitem, 59
 - D.setdefault, 59
 - D.update, 59
 - D.values, 58
 - D.viewitems, 59
 - delim.join, 46
 - destruktor, 120
 - dict.fromkeys, 59
 - działania dwuargumentowego, 128, 129
 - działania na zbiorach, 67
 - file, 61
 - file.close, 64
 - file.closed, 65
 - file.fileno, 64
 - file.flush, 64
 - file.isatty, 64
 - file.mode, 65
 - file.name, 65
 - file.seek, 64
 - file.tell, 64
 - file.truncate, 64
 - for line in infile, 63
 - formatująca, 34, 38
 - składnia, 36
 - generators, 90
 - has_key, 56, 57
 - I.__class__, 113
 - I.__next__, 54, 55, 83, 142, 143, 167
 - I.next, 54, 55, 143
 - infile, 62
 - infile.read, 62
 - infile.readline, 63
 - infile.readlines, 63
 - io.BytesIO, 62, 148
 - io.StringIO, 62, 148
 - isinstance, 125
 - issubclass, 125
 - items, 56, 57
 - iter, 83
 - iterators, 56
 - iteritems, 56
 - iterkeys, 56
 - interval, 56
 - keys, 56, 57
 - klasy, 137
 - abstrakcyjnej, 166
 - String, 39
 - konstruktor, 120
 - L.append, 51
 - L.clear, 52
 - L.copy, 52
 - L.count, 52
 - L.extend, 51
 - L.index, 51
 - L.insert, 52
 - L.pop, 52
 - L.remove, 52
 - L.reverse, 51
 - L.sort, 51, 52, 57
 - obiekt gniazda.makefile, 62
 - obiektu string, 183
 - open, 61, 62, 63, 66, 212
 - outfile, 63
 - outfile.write, 63
 - outfile.writelines, 64
 - prawostronna, 128
 - przeciążająca operator, 98
 - przeciążania klas, 118
 - przeciążania operatorów, 68, 118, 132
 - przypisania z aktualizacją, 129
 - reversed, 51
 - s.capitalize, 184
 - S.capitalize, 44
 - S.casefold, 44
 - S.center, 45
 - S.count, 43
 - S.endswith, 43
 - S.expandtabs, 44
 - S.find, 42
 - S.format, 44
 - S.index, 42
 - s.join, 184
 - S.join, 43
 - S.ljust, 44
 - S.lower, 44
 - S.lstrip, 44
 - S.replace, 43

- S.find, 42
- S.rindex, 42
- S.rjust, 45
- S.rstrip, 44
- s.split, 184
- S.split, 43
- S.splitlines, 43
- S.startswith, 43
- S.strip, 44
- S.swapcase, 44
- S.title, 45
- S.translate, 45
- S.upper, 44
- S.zfill, 45
- send, 90
- separatora, 46
- statyczna, 152
- str.format, 38, 41, 121, 140, 184
- StringIO.stringIO, 148
- sys.exit, 168
- T.count, 61
- T.index, 61
- throw, 90
- values, 56, 57
- viewitems, 57
- viewkeys, 57
- viewvalues, 57
- write, 81
- X.__iter__, 54
- X.__round__, 151
- xreadlines, 66
- moduł, 94, *Patrz też:* funkcja
 - __builtin__, 135, 157
 - anydbm, 211
 - atrybut prywatny, 112
 - biblioteczny, 174, 175
 - builtins, 135
 - datetime, 228
 - dbm, 61, 210, 211, 212, 213
 - docelowy, 92
 - dopasowywania wzorców
 - tekstowych re, 41
 - enum, 230
 - glob, 186
 - imp, 160
 - importowanie, 92, 95, 96
 - interfejs, 234
 - internetowy, 222
 - json, 228
 - kwalifikacja, 92
 - math, 225
 - multiprocessing, 186
 - nazwa, 75
 - obiekt, *Patrz:* obiekt modułu
 - obsługi wątków, 231
 - os, 62, 185
 - os.path, 185, 201
 - pickle, 61, 210, 211, 214, 215
 - queue, 186, 232
 - re, 204
 - shelve, 61, 211, 212
 - signal, 186
 - socket, 186
 - string, 183, 184
 - struct, 230
 - subprocess, 186, 229
 - sys, 159, 175, 186
 - tempfile, 186, 194
 - threading, 186, 232
 - time, 225
 - timeit, 227
 - tkinter, 217, 218, 219, 220
 - utrwalania obiektów, 210
 - weakref, 167
- MRO, 8, 114, 115, 154

N

- nazwa
 - kwalifikowana, 107
 - niekwalifikowana, 107, 108
 - zasięg, 108

O

- obiekt, 145
 - bufora, *Patrz:* bufor
 - dopasowania, 207
 - egzemplarza, 98
 - file, 192
 - generatora, 55, 89
 - integer, 130
 - iteratora, 142

- ul>
- obiekt
 - iterowalny, 54, 66, 67, 83, 89, 143, 151, 156, 160
 - klasy, 97, 111
 - kursora, *Patrz*: kursor
 - modułu, 92
 - NotImplemented, 124, 127
 - pliku, 61
 - ramki, 177
 - referencja, 75
 - rozmiar, 178
 - serializacja, 214
 - shelve, 212
 - składany, 109
 - str, 29, 39
 - string, 39, 183
 - utrwalanie, 210
 - widoku, 57
 - widoku pamięci, 144
 - wyrażeń regularnych, 206
- object-oriented programming, *Patrz*: programowanie obiektowe
- odśmiecianie, 18, 120
- odzworowanie, 21, 23, 24
- OOP, *Patrz*: programowanie obiektowe
- operacja
 - bitowa
 - AND, 18
 - NOT, 19
 - OR, 18
 - XOR, 18
 - katalogowa, 170
 - logiczna, 21, 22
 - AND, 18
 - negacja, 18
 - OR, 18
 - matematyczna, 225
 - plikowa, 170
 - porównania, 22
 - wejściowa, 146
 - wycinania, 126
 - wyjściowa, 147
 - zmiennoprzecinkowa, 165, 177
- operator, 19, 127
 - %, 33, 34
 - dwuargumentowy, 128
 - porównania, 20, 123
 - równości, 18
 - trójargumentowy, 18
 - wyraźeniowy, 18, 19
- ostrzeżenie, 170, 171
 - blokowanie, 171
- P**
- pakiet, 93, 94, 96
 - pamięć
 - globalna, 231
 - wyczerpywanie się, 166
 - parsowanie, 230
 - plik, 61
 - .pth, 92
 - .pyc, 93
 - __init__.py, 93, 94
 - buforowanie, 66
 - dbm, 213
 - deskryptor, 148, 191, 193, 200
 - JSON, 228
 - koniec, 165
 - modyfikacja, 202
 - otwieranie, 145, 162, 212
 - tryb, 146
 - py.exe, 15
 - pyw.exe, 15
 - rozmiar, 202
 - shelve, 213
 - ścieżka, 194, 195, 196, 197, 201
 - tryb otwarcia, 65
 - tworzenie, 212
 - wejściowy, 62
 - wykonywalny, 199
 - zamykanie, 192
 - polecenie
 - help, 40
 - powłoki, 185, 188
 - python -m pydoc -b, 72
 - poprawka, 57
 - print, 121
 - proces
 - potomny, 199, 200, 201
 - tworzenie, 198, 199
 - uprzejmość, 200

- programowanie
 - funkcyjne, 110
 - obiektywne, 110
- protokół
 - iteracji, *Patrz:* iteracja protokołów
 - menedżera kontekstu, *Patrz:* menedżer kontekstu protokołów
 - sekwencji, *Patrz:* sekwencja protokołów
- przepełnienie, 167
- przerwanie, 166
- przeźrenie nazw, 91, 94, 107, 111, 138, 160
 - obiektu, 107
- punkt montowania, 202
- Python
 - idiomy, 236
 - implementacja, 7
 - launcher, *Patrz:* launcher
 - rdzeń języka, 236
 - środowisko, 238
 - użytkowanie, 239, 240
 - wersja, 15, 178, 183
- słowo zarezerwowane, 73, 74, 118
- SQL, 232
- SQLite, 233
- stała
 - ascii_letters, 184
 - ascii_lowercase, 184
 - ascii_uppercase, 184
 - digits, 184
 - hexdigits, 185
 - octdigits, 185
 - printable, 185
 - punctuation, 185
 - tekstowa, 31
 - whitespace, 185
- stos, 168, 177, 178, 181
- strumień, 229
 - standardowy, 182
- symbol zachęty, 180
- synonim, 92, 96
- system, 180
 - operacyjny, 185
 - plików, 212

S

- sekwencja, 21, 22, 23, 25, 125
 - działanie, 32
 - indeksowanie, 126
 - modyfikowalna, *Patrz:* sekwencja mutowalna
 - mutowalna, 23, 29, 46, 50
 - niemutowalna, 29, 46, 60
 - protokół, 126
 - przypisanie, 77
- serializacja, 214
- set comprehension, *Patrz:* zbiór składany
- skrót, 121
- slicing, *Patrz:* rozcinanie
- slot, 124, *Patrz:* gniazdo
- słownik, 23, 24, 56, 138, 140, 156, 213, 228
 - literał, 57
 - modułów, 179
 - poprawka, *Patrz:* poprawka
 - porównywanie, 57
 - składany, 56, 57

Ś

- ścieżka, 201
 - nazwa, 194, 195, 196, 197

T

- tabela translacji, 184
- tablica bajtowa, 22, 23
- target, *Patrz:* cel
- test
 - diagnostyczny, 104
 - is*, 45
 - zawartości, 45
- tryb
 - otwarcia plików, *Patrz:* plik tryb
 - otwarcia
 - tekstowy Unicode, 63
- typ, 113
 - Boolean, 21, 69
 - Ellipsis, 68
 - konwersja, 69
 - liczbowy, 24, 25, 26, 27
 - logiczny, 68

typ

- łańcuchowy, 29, 32
 - bytearray, 41, 46, 48, 49
 - bytes, 41, 46, 48
 - str, 29, 32, 41, 46
- model dynamiczny, 17
- None, 68
- NotImplemented, 68
- numeryczny, 21
- obiektowy, 17
- odpowiadający jednostkom programu, 69
- set, 28
- tekstowy
 - ASCII, 48, 136, 149
 - Unicode, 41, 46, 47, 48, 49, 135, 149, 153, 160, 161, 179
- wbudowany, 127
- wyliczeniowy, 230

U

- ułamek, 26, 28
- Unicode, *Patrz:* typ tekstowy Unicode
- uprawnienia dostępu, 170

W

- wartość widok, 57
- wątek, 183, 231
- widget, 217, 218, 219, 220
- widok
 - elementu, 57
 - klucza, *Patrz:* klucz widok
 - wartości, 57
- wiersz poleceń, 175, 177, 186, 229
 - flaga -O, 104
 - format, 9, 12
 - opcja, 9, 12, 14
 - specyfikacja programu, 11
- wycinanie, 126
- wycinek, 151, 166
 - prosty, 25
 - przypisanie, 26
 - rozszerzony, 25

wyjątek, 103

- ArithmeticError, 164
- AssertionError, 104, 165
- AttributeError, 165
- BaseException, 163
- BlockingIOError, 169
- BrokenPipeError, 169
- BufferError, 164
- BytesWarning, 171
- ChildProcessError, 169
- ConnectionAbortedError, 169, 170
- ConnectionError, 169
- ConnectionRefusedError, 169, 170
- ConnectionResetError, 170
- DeprecationWarning, 171
- EnvironmentError, 172
- EOFError, 165
- Exception, 164, 173
- FileExistsError, 170
- FileNotFoundError, 170
- FloatingPointError, 164, 165
- FutureWarning, 171
- GeneratorExit, 90, 164, 165
- ImportError, 165
- ImportWarning, 171
- IndentationError, 166
- IndexError, 54, 166
- InterruptedError, 170
- IOError, 145, 162, 172
- IsADirectoryError, 170
- KeyboardInterrupt, 164, 166
- KeyError, 166
- klasa, 103
 - bazowa, 163
- LookupError, 164
- łańcuch, 102
- MemoryError, 166
- NameError, 165, 166
- nazwa, 74
- nieobsłużony, 183
- nieprzechwycony, 179
- NotImplemented, 166
- NotImplementedError, 166
- OSError, 164, 172, 173
- OverflowError, 164, 167
- PendingDeprecationWarning, 171

- PermissionError, 170
- ProcessLookupError, 170
- przechwytywanie, 100
- ReferenceError, 167
- ResourceWarning, 171
- RuntimeError, 165, 167
- RuntimeWarning, 171
- StopIteration, 54, 91, 126, 142, 145, 167
- SyntaxError, 165, 167
- SyntaxWarning, 171
- SystemError, 168
- SystemExit, 164, 168
- TabError, 168
- TimeoutError, 170
- TypeError, 168
- UnboundLocalError, 168
- UnicodeDecodeError, 169
- UnicodeEncodeError, 169
- UnicodeError, 168
- UnicodeTranslateError, 169
- UnicodeWarning, 171
- UserWarning, 171
- ValueError, 165, 169
- VMSError, 173
- Warning, 170
- wbudowany, 163
- WindowsError, 173
- ZeroDivisionError, 164, 169
- zgłaszanie, 100, 102
- wyrażenie
 - formatujące łańcuch znaków, 33, 34, 38
 - generatorowe, 54, 55
 - in, 57
 - jako instrukcja, 79
 - lambda, 86, 87, 110
 - listowe, *Patrz:* lista składana
 - regularne, 204, 206, 207, 208, 211
 - składnia, 209, 210
 - rozcinające, 20
 - yield, 89
- wywołanie zwrotne, 182
- wzorzec
 - diamentu, 113, 115
 - wyrażeń regularnych, 171, 204

Z

- zakres, 166
- zasięg
 - funkcji wbudowanych, 108
 - globalny, 108
 - leksykalny, 107
 - lokalny, 108
 - nazw niekwalifikowanych, 108
 - zagnieżdżony statycznie, 109
- zbiór, 66
 - działania, 67
 - składany, 56
 - zamrożony, 140
- zmienna, 17, 72
 - globalna, 91
 - lokalna, 107, 143
 - nazwa, 17, 72, 73, 74
 - operacyjna, 13
 - pętli wyrażeń generatorowych, 55
 - PYTHONCASEOK, 14
 - PYTHONDEBUG, 15
 - PYTHONDONTWRITEBYTECODE, 15
 - PYTHONFAULTHANDLER, 14
 - PYTHONHASHSEED, 14
 - PYTHONHOME, 14
 - PYTHONINSPECT, 15
 - PYTHONIOENCODING, 14
 - PYTHONNOUSERSITE, 15
 - PYTHONOPTIMIZE, 15
 - PYTHONPATH, 13, 92, 179
 - PYTHONSTARTUP, 14
 - PYTHONUNBUFFERED, 15
 - PYTHONVERBOSE, 15
 - PYTHONWARNINGS, 15
 - sys.path, 92
 - środowiska powłoki, 198
 - środowiskowa, 13
- znak, 29
 - ", 71
 - #, 72
 - \$, 73
 - %, 33, 34, 36
 - %=, 129
 - &=, 129
 - *, 33, 78, 86

znak	=, 129
**, 78	~, 202
**=, 129	+=, 129
*=, 129	<<=, 129
„, 36	=, 129
..., 68	>>=, 129
/, 201	apostrof, 30
//, 129	cudzysłów, 30
/=, 129	desygatora, 30
;, 72	nawias klamrowy, 34
?, 73	oddzielający
@, 88	komponenty ścieżki
[.], 52	wyszukiwania, 188
^=, 129	nazwę pliku od rozszerzenia,
_, 74	187
__, 74, 118	przestankowy, 185
{}, 56, <i>Patrz</i> : znak nawias klamrowy	spacja biała, 42, 72

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 

KOMPLEKSOWO SZKOLIMY NOWOCZESNY BIZNES



IT



BIZNES



PROJEKTY



PROCESY

**NASZE SZKOLENIA SĄ PROWADZONE
ZGODNIE Z METODĄ**

BLENDED LEARNING

modelem kształcenia, który łączy tradycyjne szkolenie
z dostępem do nowoczesnych narzędzi - videokursów,
e-booków i audiobooków

T: 609 850 372 E: SZKOLENIA@HELION.PL

WWW.HELIONSZKOLENIA.PL