Python Tutorial

Wydanie 3.13.0

Guido van Rossum and the Python development team

października 17, 2024

Python Software Foundation
Email: docs@python.org

Spis treści

Python jest łatwym do nauki, wszechstronnym językiem programowania. Ma wydajne wysoko-poziomowe struktury danych i proste ale efektywne podejście do programowania zorientowanego obiektowo. Elegancka składnia Pythona i dynamiczne typowanie, wraz z jego naturą interpretowania, czyni go idealnym językiem do skryptów i szybkiego rozwijania aplikacji w wielu obszarach na większości platform.

Interpreter Pythona i obszerna biblioteka standardowa są swobodnie dostępne w formie źródeł oraz binarnej dla wszystkich głównych platform na stronie internetowej Pythona, https://www.python.org/, i mogą być dowolnie rozpowszechniane. Ta sama strona zawiera również dystrybucje i odniesienia do wielu wolnych zewnętrznych modułów, programów i narzędzi Pythona oraz dodatkowej dokumentacji.

Interpreter Pythona można łatwo rozszerzyć nowymi funkcjami i typami danych zaimplementowanymi w C lub C++ (lub innych językach wywoływalnych z C). Python jest również odpowiedni jako język rozszerzeń dla konfigurowalnych aplikacji.

Ten tutorial wprowadza nieformalnie czytelnika w podstawowe koncepcje i cechy języka i systemu Python. Pomocnym jest mieć interpreter Pythona pod ręką dla praktycznych doświadczeń, ale wszystkie przykłady są samowystarczalne, więc tutorial może być również czytany off-line.

Opisy standardowych obiektów i modułów znajdziesz w library-index. reference-index daje bardziej formalną definicję języka. Aby pisać rozszerzenia w C lub C++, przeczytaj extending-index i c-api-index. Jest również kilka książek omawiających wyczerpująco Pythona.

Ten tutorial nie próbuje wyczerpująco omówić każdej funkcji Pythona ani nawet każdej często używanej funkcji. Zamiast tego wprowadza wiele funkcjonalności najbardziej wartych zauważenia i da ci dobre zrozumienie smaku i stylu języka. Po przeczytaniu go będziesz w stanie czytać i pisać moduły i programy Pythona oraz będziesz gotowy uczyć się więcej na temat różnych modułów i bibliotek Pythona, opisanych w library-index.

Warto również przejrzeć *Słownik*.

ROZDZIAŁ 1

Na zaostrzenie apetytu

Jeśli dużo pracujesz na komputerach, znajdziesz w końcu jakieś zadanie, które chciałbyś zautomatyzować. Na przykład możesz chcieć wykonać znajdź-i-zamień w wielu plikach tekstowych lub zmienić nazwę i przearanżować w skomplikowany sposób zbiór plików fotografii. Być może chciałbyś napisać własną małą bazę danych, lub wyspecjalizowaną aplikację GUI lub prostą grę.

Jeśli jesteś zawodowym deweloperem oprogramowania, mógłbyś chcieć pracować z kilkoma bibliotekami C/C++/Java, ale uważałbyś zwykły cykl napisz/skompiluj/przetestuj/zrekompiluj za zbyt wolny. Może piszesz zestaw testów dla takiej biblioteki i pisanie kodu testowego jest dla ciebie nudnym zajęciem. Lub może napisałeś program, który mógłby użyć języka rozszerzeń i nie chcesz projektować i implementować całego nowego języka dla swojej aplikacji.

Python jest językiem dla ciebie.

Mógłbyś napisać skrypt w uniksowym shellu lub windowsowy program wsadowy dla niektórych z tych zadań, lecz skrypty shellowe są najlepsze w przenoszeniu plików i zmieniania danych tekstowych, nie nadają się najlepiej dla aplikacji z graficznym interfejsem użytkownika lub gier. Mógłbyś napisać program w C/C++/Javie, ale może zająć wiele czasu pracy deweloperskiej, aby dostać jedynie wstępny szkic programu. Python jest łatwiejszy w użyciu, dostępny na Windows, macOS i uniksowe systemy operacyjne i pomoże ci wykonać zdanie szybciej.

Python jest prosty w użyciu, ale jest prawdziwym językiem programowania, oferującym dużo więcej struktury i wsparcia dla dużych programów niż skrypty shell i pliki wsadowe mogą zaoferować. Z drugiej strony Python oferuje również dużo więcej sprawdzeń błędów niż C i będąc *językiem bardzo-wysokiego-poziomu*, ma wbudowane wysoko-poziomowe typy danych, takie jak elastyczne tablice i słowniki. Z powodu tych bardziej ogólnych typów danych Python jest odpowiedni dla dużo większej domeny problemów niż Awk lub nawet Perl, mimo to wiele rzeczy w Pythonie jest co najmniej tak prostych jak w tych językach.

Python pozwala ci podzielić twój program na moduły, które mogą zostać wykorzystane w innych programach Pythona. Ma dużą kolekcję standardowych modułów, które możesz użyć jako podstawę dla swoich programów — lub jako przykładów do rozpoczęcia nauki programowania w Pythonie. Niektóre z tych modułów dostarczają rzeczy jak plikowe wejście/wyjście, wywołania systemowe, gniazda oraz nawet interfejsy do narzędzi graficznych interfejsów użytkownika takich jak Tk.

Python jest językiem interpretowanym, co może oszczędzić ci znaczy czas podczas pracy nad programem, ponieważ nie jest potrzebna kompilacja i linkowanie. Interpreter może być używany interaktywnie, co ułatwia eksperymentowanie z funkcjami języka, pisanie programów „do wyrzucenia" lub testowanie funkcji podczas rozwijania programu metodą bottom-up. Jest również poręcznym biurkowym kalkulatorem.

Python pozwala programom być pisanym kompaktowo i czytelnie. Programy pisane w Pythonie są typowo dużo krótsze niż ich odpowiedniki w C, C++ lub Javie, z kilku powodów:

- wysoko-poziomowe typy danych pozwalają wyrazić złożone operacje w jednym wyrażeniu;
- grupowanie wyrażeń odbywa się za pomocą wcięć zamiast otwierających i zamykających nawiasów;
- deklaracje zmiennych lub argumentów nie są potrzebne.

Python jest *rozszerzalny*: jeśli wiesz, jak programować w C, prosto możesz dodać nową wbudowaną funkcję lub moduł do interpretera, zarówno aby wykonywać krytyczne operacje z maksymalną szybkością, lub linkować programy Pythona do bibliotek, które mogą być dostępne tylko w formie binarnej (takie jak specyficzne dla dostawcy biblioteki graficzne). Kiedy już jesteś naprawdę nakręcony, możesz połączyć interpreter Pythona z aplikacją napisaną w języku C i użyć go jako rozszerzenia lub języka poleceń dla tej aplikacji.

Swoją drogą, nazwa języka pochodzi od programu BBC „Latający Cyrk Monty Pythona" i nie ma nic wspólnego z gadami. Tworzenie odniesień do skeczy Monty Pythona w dokumentacji jest nie tylko dozwolne, wręcz do tego zachęcamy!

Teraz, gdy wszyscy jesteście podekscytowani Pythonem, będziecie chcieli zbadać go bardziej szczegółowo. Ponieważ najlepszym sposobem na nauczenie się języka jest korzystanie z niego, tutorial zaprasza do zabawy interpreterem Pythona podczas czytania.

W następnym rozdziale wyjaśniana jest mechanika używania interpretera. Są to dość przyziemne informacje, ale niezbędne do wypróbowania przykładów pokazanych później.

Reszta tutoriala wprowadza różne cechy języka i systemu Pythona poprzez przykłady, zaczynając od prostych wyrażeń, instrukcji i typów danych, poprzez funkcje i moduły, a na końcu dotykając zaawansowanych pojęć, takich jak wyjątki i klasy zdefiniowane przez użytkownika.

ROZDZIAŁ 2

Używanie interpretera Pythona

2.1 Wywoływanie Interpretera

Interpreter Pythona jest zazwyczaj zainstalowany jako `/usr/local/bin/python3.13` na maszynach, na których jest dostępny; wstawienie `/usr/local/bin` w ścieżkę wyszukiwania uniksowego shella umożliwia jego uruchomienie przez wpisanie komendy:

```
python3.13
```

do shella.[1] Jako że wybór katalogu, w którym znajdzie się interpreter, jest opcją instalacji, możliwe są inne lokalizacje; sprawdź ze swoim lokalnym pythonowym guru lub administratorem systemu. (Na przykład `/usr/local/python` jest popularną alternatywną lokalizacją.)

Na maszynach Windows, gdzie instalowałeś Pythona z Microsoft Store, będzie dostępna komenda `python3.13`. Jeśli masz zainstalowany launcher py.exe, możesz użyć komendy `py`. Inne sposoby uruchomienia Pythona znajdziesz w setting-envvars.

Wpisanie znaku końca pliku (`Control-D` w Uniksie, `Control-Z` w Windowsie) w główną konsolę powoduje zakończenie interpretera z kodem wyjścia zero. Jeśli to nie zadziała, możesz opuścić interpreter wpisując następującą komendę: `quit()`.

Funkcje edycji linii interpretera obejmują interaktywną edycję, zastępowanie historii i uzupełnianie kodu w systemach wspierających bibliotekę GNU Readline. Prawdopodobnie najszybszym sposobem sprawdzenia, czy posiadasz rozszerzone właściwości linii poleceń, jest naciśnięcie `Control-P` za pierwszym znakiem zachęty Pythona, który zobaczysz po jego uruchomieniu. Jeżeli zabrzęczy, masz edycję linii poleceń; zobacz wprowadzenie do klawiszy w dodatku *Interaktywna edycja danych wejściowych oraz podstawianie z historii*. Jeśli nic się nie zdarzy lub pojawi się `^P`, to edycja linii poleceń nie jest dostępna; będziesz mógł tylko używać klawisza backspace, aby usuwać znaki z bieżącego wiersza.

Interpreter działa podobnie do uniksowej powłoki: kiedy jest wywołany ze standardowym wejściem połączonym z urządzeniem tty, czyta i wykonuje komendy interaktywnie. Gdy zostanie wywołany z argumentem w postaci nazwy pliku lub z plikiem jako standardowym wejściem, czyta i wykonuje *skrypt* z tego pliku.

Drugim sposobem na uruchomienie interpretera jest `python -c komenda [arg] ...`, co wykonuje polecenie (polecenia) zawarte w *komendzie*, analogicznie do opcji `-c` powłoki. Jako że polecenia Pythona często zawierają spacje lub inne znaki, które są interpretowane przez powłokę, zazwyczaj najlepiej jest umieścić *komendę* w całości w cudzysłowie.

[1] Na Uniksie, interpreter Pythona 3.x nie jest domyślnie zainstalowany z plikiem wykonywalnym o nazwie `python`, aby nie konfliktował on z jednocześnie zainstalowanym plikiem wykonywalnym Pythona 2.x.

Niektóre moduły Pythona są też przydatne jako skrypty. Mogą być one wywołane przy użyciu `python -m moduł [arg] ...`, co wykonuje plik źródłowy dla *modułu* tak jakbyś wpisał jego pełną nazwę w linii komend.

Kiedy używa się pliku skryptu, czasami przydatne jest móc uruchomić skrypt i następnie wejść w interaktywny. Można to zrobić przekazując `-i` przed skryptem.

Wszystkie opcje linii komend są opisane w using-on-general.

2.1.1 Przekazywanie argumentów

Nazwa skryptu i dodatkowe argumenty, gdy są znane interpreterowi, są zamieniane w listę ciągów znaków i przypisywane zmiennej `argv` w module `sys`. Możesz dostać się do tej listy wykonując `import sys`. Długość listy jest przynajmniej równa jeden; gdy nie podano nazwy skryptu i żadnych argumentów wywołania, `sys.argv[0]`, jest pustym ciągiem. Gdy nazwa skryptu przekazana jest w postaci `'-'` (co oznacza standardowe wejście), `sys.argv[0]` przyjmuje wartość `'-'`. Gdy zostanie użyte `-c` *komenda*, `sys.argv[0]`, przyjmuje wartość `'-c'`. Gdy zostanie użyte `-m` *moduł*, `sys.argv[0]` przyjmie wartość pełnej nazwy znalezionego modułu. Opcje znalezione za `-c` *komenda* lub `-m` *moduł* nie są konsumowane przez przetwarzanie opcji interpretera Pythona, lecz pozostawiane w `sys.argv` do obsłużenia przez komendę lub moduł.

2.1.2 Tryb interaktywny

Jeżeli instrukcje są wczytywane z urządzenia tty, mówi się wtedy, że interpreter jest w *trybie interaktywnym*. Interpreter zachęca wtedy do podania kolejnej instrukcji wyświetlając tzw. znak zachęty, zwykle w postaci trzech znaków większości (>>>). Gdy wymaga kontynuacji instrukcji, w następnej linii wyświetla *drugi znak zachęty*, domyślnie trzy kropki (...). Interpreter wyświetla wiadomość powitania zawierającą jego wersję i notatkę o prawach autorskich przed wyświetleniem pierwszego znaku zachęty:

```
$ python3.13
Python 3.13 (default, April 4 2023, 09:25:04)
[GCC 10.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Linie kontynuacji są potrzebne przy wejściu w wielowierszową konstrukcję. Jako przykład, spójrzmy na to wyrażenie `if`:

```
>>> the_world_is_flat = True
>>> if the_world_is_flat:
...     print("Uważaj, żeby nie spaść!")
...
Uważaj, żeby nie spaść!
```

Więcej na temat trybu interaktywnego znajdziesz w *Tryb interaktywny*.

2.2 Interpreter i jego środowisko

2.2.1 Strona kodowa kodu źródłowego

Domyślnie pliki źródłowe Pythona są traktowane jako zakodowane w UTF-8. W tym kodowaniu znaki większości języków na świecie mogą być użyte jednocześnie w literałach ciągów znaków, identyfikatorach i komentarzach – jednak biblioteka standardowa używa jedynie znaków ASCII dla identyfikatorów, tej konwencji powinien przestrzegać każdy przenośny kod. Aby wyświetlić odpowiednio wszystkie te znaki, twój edytor musi rozpoznawać, że plik jest UTF-8 i musi używać fontu, który wspiera wszystkie znaki w tym pliku.

Aby zadeklarować kodowanie inne niż domyślne, powinno się dodać specjalną linię komentarza jako *pierwszą* linię pliku. Składnia jest następująca:

```
# -*- coding: encoding -*-
```

gdzie *encoding* jest jednym z poprawnych `codecs` wspieranych przez Pythona.

Na przykład aby zadeklarować używanie kodowania Windows-1252, pierwszą linią twojego kodu źródłowego powinno być:

```
# -*- coding: cp1252 -*-
```

Jedynym wyjątkiem dla reguły *pierwszej linii* jest, kiedy kod źródłowy zaczyna się *uniksową linią „shebang”*. W tym przypadku deklaracja kodowania powinna być dodana jako druga linia pliku. Na przykład:

```
#!/usr/bin/env python3
# -*- coding: cp1252 -*-
```

ROZDZIAŁ 3

Nieformalne wprowadzenie do Pythona

W nadchodzących przykładach wejście i wyjście są rozróżniane przez obecność lub nieobecność promptów (»> i …): aby powtórzyć przykład, musisz wpisać wszystko po prompcie, kiedy prompt jest widoczny; linie, które nie zaczynają się promptem są wyjściem z interpretera. Zwróć uwagę, że prompt drugiego rzędu z pustą linią w przykładzie oznacza, że musisz wpisać pustą linię; używa się tego do zakończenia wielo-liniowej komendy.

Wiele przykładów w tej instrukcji, także tych wpisywanych w konsoli interaktywnej, zawiera komentarze. Komentarze w Pythonie zaczynają się znakiem hash `#` i ciągną się do końca fizycznej linii. Komentarz może pojawić się na początku linii, po wiodących spacjach lub kodzie, lecz nie może być zawarty w literale ciągu znaków. Znak hash w ciągu znaków jest po prostu znakiem hash. Jako że komentarze mają wyjaśniać kod i nie są interpretowane przez Pythona, można je ominąć przy wpisywaniu przykładów.

Trochę przykładów:

```
# to jest pierwszy komentarz
spam = 1  # a to jest drugi komentarz
          # … i teraz trzeci!
text = "# To nie jest komentarz, bo jest w cudzysłowie."
```

3.1 Używanie Pythona jako kalkulatora

Wypróbujmy parę prostych poleceń Pythona. Uruchom interpreter i poczekaj na pojawienie się pierwszego znaku zachęty >>>. (Nie powinno to zająć dużo czasu.)

3.1.1 Liczby

Interpreter działa jak prosty kalkulator: można wpisać do niego wyrażenie, a on wypisze jego wartość. Składnia wyrażenia jest prosta: operatory `+`, `-`, `*` i `/` można użyć do arytmetyki; nawiasy (`()`) można użyć do grupowania. Na przykład:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> 8 / 5  # dzielenie zawsze zwraca liczbę zmiennoprzecinkową
1.6
```

Liczby całkowite (np. `2`, `4`, `20`) są typu `int`, te z częścią ułamkową (np. `5.0`, `1.6`) są typu `float`. Więcej o typach numerycznych dowiemy się więcej później w tym tutorialu.

Dzielenie (`/`) zawsze zwraca liczbę zmiennoprzecinkową. Aby zrobić *dzielenie całkowite* i uzyskać wynik całkowity możesz użyć operatora `//`; aby obliczyć resztę możesz użyć `%`:

```
>>> 17 / 3  # klasyczne dzielenie zwraca liczbę zmiennoprzecinkową
5.666666666666667
>>>
>>> 17 // 3  # dzielenie całkowite pomija część ułamkową
5
>>> 17 % 3  # operator % zwraca resztę z dzielenia
2
>>> 5 * 3 + 2  # iloraz całkowity * dzielnik + reszta
17
```

W Pythonie możesz użyć operatora `**`, do obliczania potęgowania[1]:

```
>>> 5 ** 2  # 5 do kwadratu
25
>>> 2 ** 7  # 2 do potęgi 7.
128
```

Znak równości (=) jest używany do przypisania wartości do zmiennej. Przypisanie do zmiennej nie jest wypisywane przez interpreter:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

Jeśli zmienna nie jest „zdefiniowana” (nie ma przypisanej wartości), próba jej użycia spowoduje błąd:

```
>>> n  # próba dostępu do niezdefiniowanej zmiennej
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python implementuje w pełni arytmetykę zmiennoprzecinkową; operatory z operandami typów mieszanych przekształcają operandy całkowite w zmiennoprzecinkowe:

```
>>> 4 * 3.75 - 1
14.0
```

W trybie interaktywnym ostatnie wyświetlone wyrażenie jest przypisywane do zmiennej _. Dzięki temu kiedy używasz Pythona jako biurkowego kalkulatora, jest nieco prościej kontynuować obliczenia, na przykład:

```
>>> tax = 12.5 / 100
>>> price = 100.50
>>> price * tax
12.5625
>>> price + _
113.0625
```

(ciąg dalszy na następnej stronie)

[1] Jako że `**` ma wyższą precedencję niż `-`, `-3**2` zostanie zinterpretowane jako `-(3**2)` i zwróci `-9`. Aby tego uniknąć i otrzymać `9`, możesz użyć `(-3)**2`.

(kontynuacja poprzedniej strony)

```
>>> round(_, 2)
113.06
```

Ta zmienna powinna być traktowana przez użytkownika jako tylko-do-odczytu. Nie przypisuj wprost do niej wartości — stworzyłbyś niezależną zmienną lokalną o tej samej nazwie maskując wbudowaną zmienną z jej magicznym zachowaniem.

Oprócz `int` i `float`, Python wspiera inne typy liczb, takie jak `Decimal` i `Fraction`. Python ma też wbudowane wsparcie dla liczb zespolonych i używa sufiksów `j` lub `J` do wskazania części urojonej (np. `3+5j`).

3.1.2 Tekst

Python umożliwia manipulację tekstem (reprezentowanym przez typ `str`, tzw. „string" lub „łańcuchem znaków") oraz liczbami. Obejmuje to znaki „`!`", słowa „`rabbit`", nazwy „`Paris`", zdania „`Got your back.`", itp. „`Yay! :)`". Mogą być one umieszczone w pojedynczych cudzysłowach (`'...'`) lub podwójnych cudzysłowach (`"..."`) z takim samym wynikiem[2].

```
>>> 'spam eggs'  # single quotes
'spam eggs'
>>> "Paris rabbit got your back :)! Yay!"  # double quotes
'Paris rabbit got your back :)! Yay!'
>>> '1975'  # digits and numerals enclosed in quotes are also strings
'1975'
```

Aby umieścić znak cudzysłowu, musimy go oznaczyć znakiem „ucieczki", poprzedzając znakiem `\`. Alternatywnie, możemy użyć innego rodzaju znaków do oznaczenia cudzysłowu:

```
>>> 'Oni powiedzieli \'tak\'.'  # użyj \' jako znaku ucieczki da pojedynczego␣
↪cudzysłowu…
"Oni powiedzieli 'tak'."
>>> "Oni powiedzieli 'tak'."  # …lub użyj podwójnego cudzysłowu
"Oni powiedzieli 'tak'."
>>> 'Oni powiedzieli "tak".'
'Oni powiedzieli "tak".'
>>> "Oni powiedzieli \"tak\"."
'Oni powiedzieli "tak".'
>>> 'Oni powiedzieli "tamci powiedzieli \'tak\'".'
'Oni powiedzieli "tamci powiedzieli \'tak\'".'
```

W interaktywnej powłoce Pythona definicja stringa i wyjściowy string może wyglądać inaczej. Funkcja `print()` wytwarza czytelne wyjście, poprzez pominięcie otaczających cudzysłowów i wypisując znaki ucieczki oraz specjalne znaki:

```
>>> s = 'Pierwsza linia.\nDruga linia.'  # \n oznacza nową linię
>>> s  # bez print(), znaki specjalne są zawarte w ciągu znaków
'Pierwsza linia.\nDruga linia.'
>>> print(s)  # z print(), znaki specjalne są interpretowane, więc \n tworzy nową␣
↪linię
Pierwsza linia.
Druga linia.
```

Jeśli nie chcesz, aby znaki poprzedzone `\` były interpretowane jako znaki specjalne, możesz użyć *surowych ciągów znaków* dodając `r` przed pierwszym cudzysłowem:

[2] W przeciwieństwie do innych języków, znaki specjalne takie jak `\n` mają to samo znaczenie zarówno z pojedynczym (`'…'`) jak i podwójnym (`"…"`) cudzysłowem. Jedyną różnicą między nimi jest to, że wewnątrz pojedynczego cudzysłowu nie musisz używać znaku ucieczki dla `"` (lecz musisz użyć znaku ucieczki `\'`) i vice versa.

```
>>> print('C:\jakas\nazwa')  # tutaj \n oznacza nową linię!
C:\jakas
azwa
>>> print(r'C:\jakas\nazwa')  # zwróć uwagę na r przed cudzysłowem
C:\jakas\nazwa
```

Istnieje jeden subtelny aspekt surowych ciągów znaków: surowy ciąg znaków nie może kończyć się nieparzystą liczbą znaków \; zobacz wpis FAQ, aby uzyskać więcej informacji i sposobów obejścia tego problemu.

String literals can span multiple lines. One way is using triple-quotes: """...""" or '''...'''. End of lines are automatically included in the string, but it's possible to prevent this by adding a \ at the end of the line. In the following example, the initial newline is not included:

```
>>> print("""\
... Usage: thingy [OPTIONS]
...      -h                        Display this usage message
...      -H hostname               Hostname to connect to
... """)
Usage: thingy [OPTIONS]
     -h                        Display this usage message
     -H hostname               Hostname to connect to

>>>
```

Ciągi mogą być łączone operatorem + i powtarzane przy użyciu *:

```
>>> # 3 razy 'un', a następnie 'ium'
>>> 3 * 'un' + 'ium'
'ununium'
```

Dwa lub więcej *literały ciągu znaków* (czyli te zawarte w cudzysłowach) obok siebie są automatycznie łączone.

```
>>> 'Py' 'thon'
'Python'
```

To zachowanie jest szczególnie przydatne, gdy chcesz dzielić długie ciągi:

```
>>> text = ('Umieść kilka ciągów znaków w nawiasach '
...         'aby je połączyć.')
>>> text
'Umieść kilka ciągów znaków w nawiasach aby je połączyć'.
```

Jednak działa tylko dla literałów, nie dla zmiennych lub wyrażeń:

```
>>> prefix = 'Py'
>>> prefiks 'thon'  # nie można łączyć zmiennej i literału ciagu znaków
  File "<stdin>", line 1
    prefix 'thon'
           ^^^^^^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
  File "<stdin>", line 1
    ('un' * 3) 'ium'
               ^^^^^
SyntaxError: invalid syntax
```

Jeśli chcesz połączyć zmienne lub zmienną i literał, użyj +:

```
>>> prefix + 'thon'
'Python'
```

Ciągi znaków mogą być *indeksowane*. Pierwszy znak ma indeks 0. Nie ma osobnego typu znakowego; znak jest po prostu ciągiem znaków o długości jeden:

```
>>> word = 'Python'
>>> word[0]  # znak na pozycji 0
'P'
>>> word[5]  # znak na pozycji 5
'n'
```

Indeksy mogą być też liczbami ujemnymi, aby zacząć odliczać od prawej:

```
>>> word[-1]  # ostatni znak
'n'
>>> word[-2]  # przedostatni znak
'o'
>>> word[-6]
'P'
```

Zwróć uwagę, że jako -0 to to samo co 0, ujemne indeksy zaczynają się od -1.

Dodatkowo do indeksowania, obsługiwany jest również *podział*. Podczas gdy indeksowanie służy do uzyskiwania pojedynczych znaków, *podział* pozwala na uzyskanie podciągu znaków:

```
>>> word[0:2]  # znaki od pozycji 0 (włącznie) do 2 (wyłącznie)
'Py'
>>> word[2:5]  # znaki od pozycji 2 (włącznie) do 5 (wyłącznie)
'tho'
```

Indeksy podzielonych fragmentów mają przydatne wartości domyślne; pominięty pierwszy indeks domyślnie jest zerem, pominięty drugi indeks domyślnie ma wartość długości dzielonego stringa.

```
>>> word[:2]  # znak od początku do pozycji 2 (wyłącznie)
'Py'
>>> word[4:]  # znak od pozycji 4 (włącznie) do końca
'on'
>>> word[-2:]  # znak od przedostatniego (włącznie) do końca
'on'
```

Zwróć uwagę, że początkowy indeks wchodzi w skład podciągu, a końcowy nie. W ten sposób `s[:i] + s[i:]` jest zawsze równe `s`:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Jednym ze sposobów na zapamiętanie, jak działa podział, to myślenie o indeksach wskazujących *pomiędzy* znakami, z lewą krawędzią pierwszego znaku numerowaną 0. Wtedy prawa krawędź ostatniego znaku ciągu o długości *n* ma indeks *n*, na przykład:

```
 +---+---+---+---+---+---+
 | P | y | t | h | o | n |
 +---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

W pierwszym wierszu liczb są pozycje indeksów od 0 do 6 w ciągu. W drugim wierszu odpowiadające im indeksy ujemne. Fragment od *i* do *j* składa się ze wszystkich znaków pomiędzy krawędziami oznaczonymi kolejno *i* i *j*.

Dla nieujemnych indeksów długość wydzielonego fragmentu to różnica indeksów, jeśli oba mieszczą się w zakresie. Na przykład długość `word[1:3]` to 2.

Próba użycia za dużego indeksu skończy się błędem:

```
>>> word[42]  # word ma tylko 6 znaków
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Jednak indeksy podzielonych fragmentów poza zakresem są obsłużone bezpiecznie przy podziale:

```
>>> word[4:42]
'on'
>>> word[42:]
''
```

Ciągi znaków Pythona nie mogą być zmieniane — są *niemutowalne*. W związku z tym przypisywanie wartości do indeksowanej pozycji w ciągu spowoduje błąd:

```
>>> word[0] = 'J'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Jeśli potrzebujesz innego ciągu znaków, powinieneś(-nnaś) stworzyć nowy:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

Wbudowana funkcja `len()` zwraca długość ciągu:

```
>>> s = 'superkalifradalistodekspialitycznie'
>>> len(s)
35
```

Zobacz także

textseq
Ciągi znaków są przykładami *typów sekwencyjnych* i obsługują wspólne operacje wspierane przez takie typy.

string-methods
Ciągi znaków wspierają dużą liczbę metod do podstawowych przekształceń i wyszukiwania.

f-strings
Literały ciągów znaków z osadzonymi wyrażeniami.

formatstrings
Informacje o formatowaniu ciągów znaków przy użyciu `str.format()`.

old-string-formatting
Stare operacje formatowania, wywoływane gdy ciągi znaków są lewymi operandami operatora `%` są opisane tutaj bardziej szczegółowo.

3.1.3 Listy

Python ma kilka *złożonych* typów danych, używanych do grupowania różnych wartości. Najbardziej wszechstronnym jest *lista*, która może zostać zapisana jako lista wartości (elementów) rozdzielonych przecinkami ujęta w nawiasy kwadratowe. Listy mogą zawierać elementy różnych typów, ale zazwyczaj wszystkie elementy mają ten sam typ.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Tak jak ciągi znaków (i wszystkie inne wbudowane typy *sekwencyjne*), do elementów list można odwoływać się przez indeksy oraz można z nich „wydzielać”:

```
>>> squares[0]  # indeksowanie zwraca element
1
>>> squares[-1]
25
>>> squares[-3:]  # slicing zwraca nową listę
[9, 16, 25]
```

Listy wspierają też operacje takie jak łączenie:

```
>>> squares + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

W przeciwieństwie do ciągów znaków, które są *niemutowalne*, listy są typem *mutowalnym*, w szczególności można zmieniać ich treść:

```
>>> cubes = [1, 8, 27, 65, 125]  # coś tu jest nie tak
>>> 4 ** 3  # sześcian 4 to 64, a nie 65!
64
>>> cubes[3] = 64  # zastąp nieprawidłową wartość
>>> cubes
[1, 8, 27, 64, 125]
```

Można również dodawać nowe elementy na końcu listy, przez użycie *metody* (dowiemy się więcej o metodach później) `list.append()`:

```
>>> cubes.append(216)  # dodaj sześcian 6
>>> cubes.append(7 ** 3)  # i sześcian 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Proste przypisanie w Pythonie nigdy nie kopiuje danych. Kiedy przypisujesz listę do zmiennej, zmienna odnosi się do *istniejącej listy*. Wszystkie zmiany dokonane na liście przez jedną zmienną będą widoczne przez wszystkie inne zmienne, które się do niej odwołują.:

```
>>> rgb = ["czerwony", "zielony", "niebieski"]
>>> rgba = rgb
>>> id(rgb) == id(rgba)  # odwołują się do tego samego obiektu
True
>>> rgba.append("alfa")
>>> rgb
["czerwony", "zielony", "niebieski", "alfa"]
```

Wszystkie operacje wykrawania zwracają nową listę zawierającą żądane elementy. Następujący slice więc zwraca płytką kopię listy:

```
>>> correct_rgba = rgba[:]
>>> correct_rgba[-1] = "alfa"
>>> correct_rgba
["czerwony", "zielony", "niebieski", "alfa"]
>>> rgba
["czerwony", "zielony", "niebieski", "alfa"]
```

Możliwe jest również przypisywanie do slice'ów. Może to zmienić rozmiar listy lub zupełnie ją wyczyścić:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> litery
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # zastąp niektóre wartości
>>> letters[2:5] = ['C', 'D', 'E']
>>> litery
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # teraz je usuń
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # wyczyść listę zastępując wszystkie elementy pustą listą
>>> letters[:] = []
>>> letters[:]
[]
```

Wbudowana funkcja `len()` ma również zastosowanie do list:

```
>>> letters = ['a', 'b', 'c', 'd']
>>> len(letters)
4
```

Można zagnieżdżać listy (tworzyć listy zawierające inne listy), na przykład:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2 Pierwsze kroki do programowania

Oczywiście możemy używać Pythona do zadań bardziej skomplikowanych niż dodawanie dwóch do dwóch. Na przykład możemy napisać początkowy podciąg ciągu Fibonacciego następująco:

```
>>> # ciąg Fibonacciego:
>>> # suma dwóch elementów określa następny
>>> a, b = 0, 1
>>> while a < 10:
...     print(a)
...     a, b = b, a+b
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
...
0
1
1
2
3
5
8
```

Ten przykład wprowadza kilka nowych funkcji.

- Pierwsza linia zawiera *wielokrotne przypisanie*: zmienne `a` i `b` jednocześnie dostają nowe wartości 0 i 1. W ostatniej linii jest ponownie wykorzystane, demonstrując, że wyrażenia po prawej stronie są ewaluowane wcześniej, zanim którekolwiek z przypisań ma miejsce. Wyrażenia po prawej stronie są ewaluowane od lewej do prawej.
- Pętla `while` wykonuje się dopóki warunek (tutaj: `a < 10`) pozostaje prawdziwy. W Pythonie, tak jak w C, każda niezerowa liczba całkowita jest prawdziwa; zero jest fałszywe. Warunek może być również ciągiem znaków lub listą, tak naprawdę jakąkolwiek sekwencją; cokolwiek o niezerowej długości jest prawdziwe, puste sekwencje są fałszywe. Warunek użyty w przykładzie jest prostym porównaniem. Standardowe operatory porównań pisane są tak samo jak w C: < (mniejsze niż), > (większe niż), == (równe), <= (mniejsze lub równe), >= (większe lub równe) i `!=` (różne).
- *Ciało* pętli jest *wcięte*: indentacja (wcięcia) jest sposobem na grupowanie instrukcji. W trybie interaktywnym trzeba wprowadzić znak(i) spacji lub tabulacji, aby wciąć wiersz. W praktyce będziesz przygotowywać bardziej skomplikowane dane wejściowe dla Pythona za pomocą edytora tekstu; wszystkie przyzwoite edytory tekstu mają funkcję automatycznych wcięć. W chwili, gdy wprowadza się jakąś instrukcję złożoną w czasie sesji interpretera Pythona, trzeba zakończyć ją pustym wierszem (bowiem interpreter nie wie, czy ostatni wprowadzony wiersz jest ostatnim z tej instrukcji). Ważne jest, aby każdy wiersz należący do tej samej grupy instrukcji, był wcięty o taką samą liczbę spacji lub znaków tabulacji.
- Funkcja `print()` wypisuje wartość argumentu(-ów), które jej podano. Różnica pomiędzy tą instrukcją, a zwykłym zapisem wyrażenia, które chce się wypisać (tak jak robiliśmy to w przykładzie z kalkulatorem) występuje w sposobie obsługi wielu wyrażeń i napisów. Łańcuchy znaków wypisywane są bez cudzysłowów, a pomiędzy nimi zapisywane są spacje, tak aby można było ładnie sformatować pojawiający się napis, na przykład:

  ```
  >>> i = 256*256
  >>> print('Wartość i wynosi', i)
  Wartość i wynosi 65536
  ```

  Keyword argument *end* można wykorzystać, aby uniknąć znaku nowej linii po wypisaniu lub aby zakończyć wypisanie innym ciągiem znaków:

  ```
  >>> a, b = 0, 1
  >>> while a < 1000:
  ...     print(a, end=',')
  ...     a, b = b, a+b
  ...
  0,1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
  ```

ROZDZIAŁ 4

Więcej narzędzi kontroli przepływu

Oprócz właśnie przedstawionego wyrażenia `while`, Python używa kilku innych, które napotkamy w tym rozdziale.

4.1 Instrukcje `if`

Prawdopodobnie najbardziej znanym typem instrukcji jest instrukcja `if`. Na przykład:

```
>>> x = int(input("Wprowadź liczbę całkowitą: "))
Wprowadź liczbę całkowitą: 42
>>> if x < 0:
...     x = 0
...     print('Wartość ujemna zmieniona na zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Jeden')
... else:
...     print('Więcej niż jeden')
...
Więcej niż jeden
```

Części `elif` może być zero lub więcej i część `else` jest opcjonalna. Keyword „`elif`" jest skrótem od „else if" i jest przydatny by uniknąć nadmiarowych wcięć. Sekwencja `if` … `elif` … `elif` … jest zamiennikiem instrukcji `switch` lub `case` z innych języków.

Jeśli porównujesz tę samą wartość z wieloma stałymi lub sprawdzasz poszczególne typy lub atrybuty, może ci się przydać instrukcja `match`. Więcej szczegółów znajdziesz w *Instrukcje match*.

4.2 Instrukcje `for`

Instrukcja `for` różni się troszeczkę w Pythonie od tego, co używasz w C lub Pascalu. Nie prowadzi się iteracji od liczby do liczby (jak w Pascalu) lub daje się użytkownikowi możliwość definiowania kroku iteracji i warunki zakończenia iteracji (jak w C). Instrukcja `for` w Pythonie powoduje iterację po elementach jakiejkolwiek sekwencji (listy lub łańcucha znaków), w takim porządku, w jakim są one umieszczone w danej sekwencji. Na przykład (gra słów niezamierzona):

```
>>> # Zmierz kilka napisów:
>>> words = ['kot', 'okienko', 'defenestracja']
>>> for w in words:
...     print(w, len(w))
...
kot 3
okienko 7
defenestracja 13
```

Kod, który zmienia kolekcję podczas iterowania po niej, może być trudny. Zamiast tego, zazwyczaj prostsze jest przejść pętlą po kopii kolekcji lub stworzyć nową kolekcję:

```
# Utwórz próbną kolekcję
users = {'Hans': 'aktywny', 'Éléonore': 'nieaktywny', '□□□': 'aktywny'}

# Strategia:  Iteracja po kopii
for user, status in users.copy().items():
    if status == 'nieaktywny':
        del users[user]

# Strategia:  Utwórz nową kolekcję
active_users = {}
for user, status in users.items():
    if status == 'aktywny':
        active_users[user] = status
```

4.3 Funkcja `range()`

Jeśli potrzebujesz iterować po sekwencji liczb, przydatna jest wbudowana funkcja `range()`. Generuje ciągi arytmetyczne:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Podany punkt końcowy nigdy nie jest częścią generowanej sekwencji; `range(10)` generuje 10 wartości, poprawne indeksy dla elementów sekwencji o długości 10. Możliwe jest zacząć zakres od innej liczby lub podać inne zwiększenie (nawet ujemne; czasem jest to nazywane „krokiem”):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]

>>> list(range(0, 10, 3))
[0, 3, 6, 9]

>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

By przeiterować po indeksach sekwencji możesz połączyć `range()` i `len()` w następujący sposób:

```
>>> a = ['Wyszły', 'w', 'pole', 'kurki', 'trzy']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Wyszły
1 w
2 pole
3 kurki
4 trzy
```

Jednak w większości takich przypadków wygodnie jest użyć funkcji `enumerate()`, patrz *Techniki pętli*.

Dzieje się dziwna rzecz jeśli po prostu wydrukujesz zakres:

```
>>> range(10)
range(0, 10)
```

Pod wieloma względami obiekt zwracany przez `range()` zachowuje się, jakby był listą, ale w rzeczywistości nią nie jest. Jest obiektem, który zwraca kolejne elementy żądanej sekwencji w trakcie twojego iterowania po nim, lecz naprawdę nie tworzy listy, tak więc oszczędza miejsce w pamięci komputera.

Mówimy, że taki obiekt to *iterable*, to znaczy odpowiedni jako cel dla funkcji i konstrukcji, które spodziewają się czegoś, z czego można pobierać kolejne elementy aż do wyczerpania zapasu. Widzieliśmy, że instrukcja `for` jest takim konstruktem, podczas gdy przykładem funkcji, która spodziewa się obiektu iterable jest `sum()`:

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Później napotkamy więcej funkcji, które zwracają iterable i biorą iterable jako argumenty. W rozdziale *Struktury danych*, omówimy bardziej szczegółowo `list()`.

4.4 Instrukcje `break` oraz `continue`

Instrukcja `break` wyłamuje się z najbardziej wewnętrznej pętli `for` lub `while`:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(f"{n} równa się {x} * {n//x}")
...             break
...
4 równa się 2 * 2
6 równa się 2 * 3
8 równa się 2 * 4
9 równa się 3 * 3
```

Instrukcja `continue` kontynuuje następną iterację pętli:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print(f"Znaleziono liczbę parzystą {num}")
...         continue
...     print(f"Znaleziono liczbę nieparzystą {num}")
...
Znaleziono liczbę parzystą 2
Znaleziono liczbę nieparzystą 3
Znaleziono liczbę parzystą 4
Znaleziono liczbę nieparzystą 5
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Znaleziono liczbę parzystą 6
Znaleziono liczbę nieparzystą 7
Znaleziono liczbę parzystą 8
Znaleziono liczbę nieparzystą 9
```

4.5 Klauzule `else` na pętlach

W pętli `for` lub `while` instrukcja `break` może być sparowana z klauzulą `else`. Jeśli pętla zakończy się bez wykonania `break`, wykonana zostanie klauzula `else`.

W pętli `for`, klauzula `else` jest wykonywana po tym, jak pętla zakończy swoją ostatnią iterację, czyli jeśli nie nastąpiło przerwanie.

W pętli `while`, klauzula wykonuje się ostatni raz po tym, jak warunek pętli staje się fałszywy.

In either kind of loop, the `else` clause is **not** executed if the loop was terminated by a `break`. Of course, other ways of ending the loop early, such as a `return` or a raised exception, will also skip execution of the `else` clause.

Jest to zilustrowane w poniższej pętli `for`, która szuka liczb pierwszych:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'równa się', x, '*', n//x)
...             break
...     else:
...         # pętla przeszła całość nie znajdując dzielnika
...         print(n, 'jest liczbą pierwszą')
...
2 jest liczbą pierwszą
3 jest liczbą pierwszą
4 równa się 2 * 2
5 jest liczbą pierwszą
6 równa się 2 * 3
7 jest liczbą pierwszą
8 równa się 2 * 4
9 równa się 3 * 3
```

(Tak, to jest poprawny kod. Przyjrzyjcie się uważnie: klauzula `else` należy do pętli `for`, **nie** do instrukcji `if`.)

Jednym ze sposobów myślenia o klauzuli else jest wyobrażenie sobie jej w połączeniu z `if` wewnątrz pętli. Podczas wykonywania pętli będzie ona wykonywać sekwencję if/if/if/else. Warunek `if` znajduje się wewnątrz pętli, napotykany wiele razy. Jeśli warunek zostanie kiedykolwiek spełniony, nastąpi `break`. Jeśli warunek nigdy się nie spełni, zostanie wykonana klauzula `else` spoza pętli.

W przypadku użycia z pętlą, klauzula `else` ma więcej wspólnego z klauzulą `else` instrukcji `try` niż z instrukcją `if`: klauzula `else` instrukcji `try` działa, gdy nie wystąpi wyjątek, a klauzula `else` pętli działa, gdy nie wystąpi `break`. Więcej informacji na temat instrukcji `try` i wyjątków można znaleźć w rozdziale *Obsługa wyjątków*.

4.6 Instrukcje `pass`

Instrukcja `pass` nie robi nic. Można jej użyć, gdy składnia wymaga instrukcji a program nie wymaga działania. Na przykład:

```
>>> while True:
...     pass  # Aktywne oczekiwanie na przerwanie z klawiatury (Ctrl+C)
...
```

Często jej się używa do tworzenia minimalnych klas:

```
>>> class MyEmptyClass:
...     pass
...
```

Innym miejscem, w którym można użyć `pass` to placeholder dla funkcji lub ciała warunku, kiedy pracujesz nad nowym kodem. Pozwoli ci to myśleć na bardziej abstrakcyjnym poziomie. `pass` jest „po cichu” ignorowane:

```
>>> def initlog(*args):
...     pass   # Pamiętaj, aby to zaimplementować!
...
```

4.7 Instrukcje `match`

Instrukcja `match` bierze wyrażenie i porównuje jego wartość do kolejnych wzorców podanych jako jeden lub więcej blok case. Jest to powierzchownie podobne do instrukcji switch w C, Javie lub JavaScripcie (i wielu innych językach), ale bardziej jest podobne do pattern matchingu w takich językach jak Rust lub Haskell. Jedynie pierwszy pasujący wzorzec jest zostaje wykonany i może on również wydobywać komponenty (elementy sekwencji lub atrybuty obiektu) z wartości do zmiennych.

Najprostsza forma porównuje wartość podmiotu z jednym lub wieloma literałami:

```
def http_error(status):
    match status:
        case 400:
            return "Niewłaściwe żądanie"
        case 404:
            return "Nie znaleziono"
        case 418:
            return "Jestem czajniczkiem"
        case _:
            return "Coś jest nie tak z internetem"
```

Zwróć uwagę na ostatni blok: „nazwa zmiennej” _ zachowuje się jak *dzika karta* i zawsze znajduje dopasowanie. Jeśli żaden z case'ów nie będzie dopasowany, żadne z rozgałęzień nie będzie wykonane.

Możesz łączyć kilka literałów w jeden wzorzec używając `|` („or”):

```
case 401 | 403 | 404:
    return "Niedozwolone"
```

Wzorce mogą wyglądać jak przypisania rozpakowujące i można ich używać do powiązywania zmiennych:

```
# punkt to dwukrotka (x, y)
match point:
    case (0, 0):
        print("Początek")
    case (0, y):
        print(f"Y={y}")
    case (x, 0):
        print(f"X={x}")
    case (x, y):
        print(f"X={x}, Y={y}")
    case _:
        raise ValueError("Nie punkt")
```

Przyjrzyj się temu uważnie! Pierwszy wzorzec ma dwa literały i można o nim myśleć jako o rozszerzeniu dosłownego wzorca pokazanego wyżej. Ale następne dwa wzorce składają się z literału i zmiennej. Ta zmienna *wiąże* wartość

z podmiotu (`point`). Czwarty wzorzec wyłapuje dwie wartości, co czyni go konceptualnie bliższym do przypisania z „rozpakowaniem” `(x, y) = point`.

Jeśli używasz klas, aby nadać strukturę swoim danym, możesz użyć nazwę klasy oraz listę argumentów przypominającą konstruktor, ale z możliwością wyłapania atrybutów w zmienne:

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

def where_is(point):
    match point:
        case Point(x=0, y=0):
            print("Początek")
        case Point(x=0, y=y):
            print(f"Y={y}")
        case Point(x=x, y=0):
            print(f"X={x}")
        case Point():
            print("Gdzieś indziej")
        case _:
            print("Nie punkt")
```

Możesz użyć parametrów pozycyjnych z jakimiś wbudowanymi klasami, które posiadają kolejność dla atrybutów (na przykład dataclasses). Możesz też określić wybrane pozycje dla atrybutów we wzorcach ustawiając specjalny atrybut `__match_args__` w twoich klasach. Jeśli jest on ustawiony na („x”, „y”), wszystkie następujące wzorce są równoważne (i wszystkie powiązują atrybut `y` ze zmienną `var`):

```
Point(1, var)
Point(1, y=var)
Point(x=1, y=var)
Point(y=var, x=1)
```

Rekomendowanym sposobem na czytanie wzorców jest patrzenie na nie jako na rozszerzoną formę tego, co wstawił(a)byś po lewej stronie przypisania, aby zrozumieć które zmienne będą ustawione na co. Tylko samodzielne nazwy (jak `var` powyżej) zyskują przypisanie w instrukcji match. Nazwy z kropkami (jak `foo.bar`), nazwy atrybutów (`x=` i `y=` powyżej) lub nazwy klas (rozpoznawane przez „(…)” przy nich jak `Point` powyżej) nie mają nigdy przypisań.

Wzory mogą być zagnieżdżane w dowolny sposób. Na przykład, jeśli mamy krótką listę punktów, z dodanym `__match_args__`, możemy dopasować ją w następujący sposób:

```
class Point:
    __match_args__ = ('x', 'y')
    def __init__(self, x, y):
        self.x = x
        self.y = y

match points:
    case []:
        print("Brak punktów")
    case [Point(0, 0)]:
        print("Początek")
    case [Point(x, y)]:
        print(f"Pojedynczy punkt {x}, {y}")
    case [Point(0, y1), Point(0, y2)]:
        print(f"Dwa na osi Y na {y1}, {y2}")
    case _:
        print("Coś innego")
```

Możemy dodać klauzulę `if` do wzorca, nazywaną „guardem”. Jeśli guard jest fałszywy, `match` próbuje dopasować następny blok case. Zwróć uwagę, że przechwycenie wartości dzieje się zanim wyliczona jest wartość guarda:

```
match point:
    case Point(x, y) if x == y:
        print(f"Y=X at {x}")
    case Point(x, y):
        print(f"Nie na przekątnej")
```

Kilka innych ważnych funkcjonalności tej instrukcji:

- Podobnie do przypisań z „rozpakowywaniem”, wzorce z krotkami i listami mają to samo znaczenie i dopasowują się do dowolnych sekwencji. Ważnym wyjątkiem jest to, że nie dopasowują się do iteratorów i ciągów znaków.
- Wzorce sekwencji wspierają rozszerzone rozpakowywanie: `[x, y, *rest]` i `(x, y, *rest)` działają podobnie do przypisań z rozpakowywaniem. Nazwa po `*` może być również _, aby `(x, y, *_)` dopasowywała się do sekwencji o conajmniej dwóch elementach bez powiązywania ze zmienną pozostałych elementów.
- Wzorce mapowania: `{"bandwidth": b, "latency": l}` przechwytuje wartości `"bandwidth"` i `"latency"` ze słownika. W przeciwieństwie do wzorców sekwencji, nadmiarowe klucze są ignorowane. Rozpakowywanie typu `**rest` jest również wspierane. (Ale `**_` byłoby nadmierne, więc jest niedozwolone.)
- Można przechwytywać podwzorce używając słowa kluczowego `as`:

  ```
  case (Point(x1, y1), Point(x2, y2) as p2): ...
  ```

  przechwyci drugi element wejścia jako `p2` (jeśli wejście jest sekwencją dwóch punktów)
- Większość literałów jest porównywanych przez równość, lecz singletony `True`, `False` i `None` są porównywane przez identyczność.
- Wzorce mogą używać nazwanych stałych. Trzeba je podawać po kropce, aby uniknąć interpretacji przechwycenia jako zwykłej zmiennej:

  ```
  from enum import Enum
  class Color(Enum):
      RED = 'red'
      GREEN = 'green'
      BLUE = 'blue'

  color = Color(input("Wprowadź swój wybór spośród 'red', 'blue' lub 'green': "))

  match color:
      case Color.RED:
          print("Widzę czerwone!")
      case Color.GREEN:
          print("Zielono mi")
      case Color.BLUE:
          print("Niebo jest niebieskie")
  ```

Bardziej szczegółowe wyjaśnienie i dodatkowe przykłady możesz znaleźć w **PEP 636**, który jest napisany w formie tutoriala.

4.8 Definiowanie funkcji

Możemy stworzyć funkcję, która wypisuje ciąg Fibonacciego do wskazanej granicy:

```
>>> def fib(n):    # write Fibonacci series less than n
...     """Print a Fibonacci series less than n."""
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Now call the function we just defined:
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Słowo kluczowe `def` oznacza *definicję* funkcji. Po nim musi następować nazwa funkcji oraz lista formalnych parametrów otoczona nawiasami. Instrukcje, które stanowią ciało funkcji zaczynają się w następnej linii i muszą być wcięte.

Opcjonalnie, pierwszy wiersz ciała funkcji może być gołym napisem (literałem): jest to tzw. napis dokumentujący lub inaczej *docstring*. (Więcej o docstringach znajdziesz w sekcji *Napisy dokumentujące (docstringi)*.) Istnieją pewne narzędzia, które używają docstringów do automatycznego tworzenia drukowanej lub dostępnej online dokumentacji albo pozwalają użytkownikowi na interaktywne przeglądanie kodu. Dobrym zwyczajem jest pisane napisów dokumentacyjnych w czasie pisania programu: spróbuj się do tego przyzwyczaić.

Wykonanie funkcji powoduje stworzenie nowej tablicy symboli lokalnych używanych w tej funkcji. Mówiąc precyzyjniej: wszystkie przypisania do zmiennych lokalnych funkcji powodują umieszczenie tych wartości w lokalnej tablicy symboli. Odniesienia do zmiennych najpierw szukają swych wartości w lokalnej tablicy symboli, potem w lokalnych tablicach symboli funkcji otaczających, potem w globalnej, a dopiero na końcu w tablicy nazw wbudowanych w interpreter. Tak więc, zmiennym globalnym ani zmiennym w otaczających funkcjach nie można wprost przypisać wartości w ciele funkcji (chyba, że zostaną wymienione w niej za pomocą instrukcji `global` lub dla zmiennych w otaczających funkcjach, wymienionych w instrukcji `nonlocal`), aczkolwiek mogą w niej być używane (czytane).

Parametry (argumenty) wywołania funkcji wprowadzane są do lokalnej tablicy symboli w momencie wywołania funkcji. Tak więc, argumenty przekazywane są jej przez wartość (gdzie *wartość* jest zawsze *odniesieniem* do obiektu, a nie samym obiektem).[1] Nowa tablica symboli tworzona jest również w przypadku, gdy funkcja wywołuje inną funkcję lub wywołuje się przez rekursję.

Definicja funkcji powiązuje nazwę funkcji z obiektem funkcji w aktualnej tablicy symboli. Interpreter rozpoznaje obiekt wskazany tą nazwą jako funkcję zdefiniowaną przez użytkownika. Inne nazwy też mogą wskazywać na ten sam obiekt funkcji i mogą być używane, aby dostać się do funkcji:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Przychodząc z innych języków, mógłbyś oponować, że `fib` nie jest funkcją, ale procedurą, jako że nie zwraca wartości. Tak naprawdę nawet funkcje bez instrukcji `return` zwracają wartość, chociaż dość nudną. Tę wartość nazywamy `None` (to wbudowana nazwa). Wypisywanie wartości `None` jest normalnie pomijane przez interpreter, jeśli miałaby to jedyna wypisywana wartość. Możesz ją zobaczyć, jeśli bardzo chcesz, używając `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

Prosto można napisać funkcję, która zwraca listę numerów ciągu Fibonnaciego zamiast go wyświetlać:

```
>>> def fib2(n):  # Zwróć ciąg Fibonacciego do n
...     """Zwróć listę zawierającą ciąg Fibonacciego do n."""
```

(ciąg dalszy na następnej stronie)

[1] Tak właściwie, *wywołanie przez referencję do obiektu* byłoby lepszym opisem, jako że jeśli mutowalny obiekt jest przekazany, wszystkie zmiany które wywoływany robi (elementy wstawiane na listę) będą widziane przez wywołującego.

(kontynuacja poprzedniej strony)

```
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # Zobacz poniżej
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # Wywołaj
>>> f100                # Wypisz wynik
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Ten przykład, jak zazwyczaj, prezentuje nowe cechy Pythona:

- Instrukcja `return` zwraca wartość funkcji. `return` bez wyrażenia podanego jako argument zwraca `None`. Dojście do końca funkcji również zwraca `None`.
- Instrukcja `result.append(a)` wywołuje *metodę* listy obiektów `result`. Metoda to funkcja, która „należy" do obiektu i jest nazwana `obj.methodname`, gdzie `obj` jest jakimś obiektem (może też być wyrażeniem) a `methodname` jest nazwą metody, które jest zdefiniowana przez typ obiektu. Różne typy definiują różne metody. Metody różnych typów mogą mieć te same nazwy bez powodowania dwuznaczności. (Da się definiować własne typy obiektów i metody, używając *klas*, patrz *Klasy*.) Metoda `append()` pokazana w przykładzie jest zdefiniowana dla listy obiektów; dodaje nowy element na końcu listy. W tym przykładzie jest równoważna `result = result + [a]`, ale bardziej wydajna.

4.9 Więcej o definiowaniu funkcji

Można też definiować funkcje ze zmienną liczbą argumentów. Są trzy sposoby, które można łączyć.

4.9.1 Domyślne wartości argumentów

Najbardziej przydatnym sposobem jest podanie domyślnej wartości dla jednego lub więcej argumentów. Tworzy to funkcję, która może zostać wywołana z mniejszą liczbą argumentów, niż jest podane w jej definicji. Na przykład:

```
def ask_ok(prompt, retries=4, reminder='Spróbuj ponownie!'):
    while True:
        reply = input(prompt)
        if reply in {'t', 'ta', 'tak'}:
            return True
        if reply in {'n', 'ni', 'nie'}:
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('Niepoprawna odpowiedź użytkownika')
        print(reminder)
```

Tę funkcję można wywołać na kilka sposobów:

- podając tylko wymagany argument: `ask_ok('Czy na pewno chcesz wyjść?')`
- podając jeden z opcjonalnych argumentów: `ask_ok('OK nadpisać plik?', 2)`
- lub podając wszystkie argumenty: `ask_ok('OK nadpisać plik?', 2, 'No coś ty, tylko tak lub nie!')`

Ten przykład wprowadza słowo kluczowe `in`. Sprawdza ono, czy sekwencja zawiera szczególną wartość.

Wartości domyślne są ewaluowane w momencie definiowania funkcji w scopie *defining*, więc

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

wyświetli `5`.

Ważna uwaga: Wartość domyślna jest wyliczana tylko raz. Ma to znaczenie, gdy domyślna wartość jest obiektem mutowalnym takim jak lista, słownik lub instancje większości klas. Na przykład następująca funkcja akumuluje argumenty przekazane do niej w kolejnych wywołaniach:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

To wyświetli

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

Jeśli nie chcesz, żeby domyślna wartość była współdzielona pomiędzy kolejnymi wywołaniami, możesz napisać funkcję w ten sposób:

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

4.9.2 Argumenty nazwane

Funkcje mogą być również wywoływane przy użyciu *argumentów nazwanych* w formie `kwarg=value`. Na przykład poniższa funkcja:

```
def parrot(voltage, state='jest sztywna', action='fru', type='norweska błękitna'):
    print("-- Ta papuga nie zrobi", action, end=' ')
    print("nawet jeśli podłączę ją do", voltage, "woltów.")
    print("-- Śliczne upierzenie,", type)
    print("-- Ona", state, "!")
```

akceptuje jeden wymagany argument (`voltage`) i trzy opcjonalne argumenty (`state`, `action` i `type`). Funkcja może być wywołana w dowolny z poniższych sposobów:

```
parrot(1000)                                          # 1 argument pozycyjny
parrot(voltage=1000)                                  # 1 argument nazwany
parrot(voltage=1000000, action='FRUUUUU')             # 2 argumenty nazwane
parrot(action='FRUUUUU', voltage=1000000)             # 2 argumenty nazwane
parrot('miliona', 'wyzionęła ducha', 'hop')          # 3 argumenty pozycyjne
parrot('tysiąca', state='wącha kwiatki')  # 1 pozycyjny, 1 nazwany
```

ale wszystkie poniższe wywołania byłyby niepoprawne:

```
parrot()                     # brakuje wymaganego argumentu
parrot(voltage=5.0, 'zdechła')  # argument nie-nazwany za argumentem nazwanym
parrot(110, voltage=220)     # zduplikowana wartość dla tego samego argumentu
parrot(actor='John Cleese')  # nieznany argument nazwany
```

W wywołaniu funkcji argumenty nazwane muszą znajdować się za argumentami pozycyjnymi. Wszystkie przekazane argumenty nazwane muszą pasować do jednego argumentu akceptowanego przez funkcję (na przykład `actor` nie jest poprawnym argumentem dla funkcji `parrot`) a ich kolejność nie ma znaczenia. Dotyczy to również nie-opcjonalnych argumentów (na przykład `parrot(voltage=1000)` też jest poprawne). Żaden argument nie może otrzymać wartości więcej niż raz. Tutaj jest przykład, który się nie powiedzie z powodu tego ograniczenia:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: function() got multiple values for argument 'a'
```

Kiedy ostatni formalny parametr ma postać `**nazwa`, to otrzymuje on słownik (zobacz typesmapping) zawierający wszystkie argumenty kluczowe oprócz tych które odpowiadają formalnym parametrom. Może to być połączone z formalnym parametrem o postaci `*nazwa` (opisanym w następnej subsekcji) który otrzymuje *krotkę* zawierającą argumenty pozycyjne z poza listy formalnych parametrów. (`*nazwa` musi występować przed `**nazwa`.) Dla przykładu możemy zdefiniować funkcję tak:

```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Czy jest może", kind, "?")
    print("-- Przykro mi, nie mamy już sera", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Można ją wywołać w ten sposób:

```
cheeseshop("limburger", "Jest bardzo płynny, proszę pana.",
           "Naprawdę jest bardzo, BARDZO płynny, proszę pana.",
           shopkeeper="Michael Palin",
           client="John Cleese",
           sketch="Sklep z serami")
```

i oczywiście wyświetli się nam:

```
-- Czy jest może limburger ?
-- Przykro mi, nie mamy już sera limburger
Jest bardzo płynny, proszę pana.
Naprawdę jest bardzo, BARDZO płynny, proszę pana.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Sklep z serami
```

Zwróć uwagę, że kolejność w jakim argumenty kluczowe są wyświetlane dokładnie odpowiada kolejności w jakim zostały one podane w wywołaniu funkcji.

4.9.3 Parametry specjalne

Domyślnie, argumenty mogą być przekazywane do funkcji Pythona albo poprzez swoje umiejscowienie (pozycję) albo wprost poprzez słowo kluczowe. Dla czytelności oraz wydajności, ma sens aby ograniczyć sposób w jaki argumenty mogą być przekazywane tak aby deweloper musiał tylko spojrzeć na definicję funkcji aby zrozumieć czy argumenty są przekazywane poprzez pozycję, pozycję i słowa kluczowe czy same słowa kluczowe.

Definicja funkcji może wyglądać w ten sposób:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
      -----------    ----------     ----------
        |             |                  |
        |        pozycyjny lub nazwany   |
        |                                - tylko nazwany
         -- tylko pozycyjne
```

gdzie `/` i `*` są opcjonalne. Jeśli są używane, te symbole wskazują rodzaj parametru poprzez sposób przekazywania argumentów do funkcji: tylko pozycyjne, pozycyjne lub słowo kluczowe i tylko słowo kluczowe. Parametry słów kluczowych są również nazywane parametrami nazwanymi.

Argumenty pozycyjne-lub-nazwane

Jeśli `/` czy `*` nie są obecne w definicji funkcji, argumenty mogą być przekazywane i poprzez pozycje i przez słowa kluczowe.

Parametry tylko-pozycyjne (positional-only)

Wchodząc w szegóły, można oznaczyć pewne parametry jako *tylko-pozycyjne*. Jeśli argument jest *tylko pozycyjny*, ma znaczenie kolejność parametrów a parametry te nie mogą być przekazywane przez słowa kluczowe. Argumenty tylko--pozycyjne są umieszczane przed `/` (forward-slash). `/` rozdziela parametry tylko-pozycyjne od reszty parametrów. Jeśli w definicji funkcji nie ma `/` , nie ma parametrów tylko pozycyjnych.

Argumenty po `/` mogą mieć charakter *pozycja-lub-słowo-kluczowe* lub *tylko-kluczowe*.

Argumenty tylko-nazwane (keyword-only)

Aby oznaczyć parametry jako *tylko słowa kluczowe* (*keyword-only*), to znaczy wskazać, że parametry muszą być przekazywane przez argument słowa kluczowego, umieść `*` na liście argumentów tuż przed pierwszym parametrem, który ma być przekazywany tylko przez słowo kluczowe.

Przykłady funkcji

Rozważ następujące przykłady definicji funkcji, przykładając szczególną uwagę do oznaczeń `/` oraz `*`:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

Pierwsza z definicji, `standard_arg`, mająca najbardziej znaną formę, nie wprowadza ograniczeń na sposób wywoływania, argumenty mogą być przekazywane poprzez pozycję lub słowo kluczowe:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

Druga z funkcji `pos_only_arg` jest ograniczona do wykorzystywania tylko argumentów pozycyjnych jako, że posiada `/` w swojej definicji:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got some positional-only arguments passed as keyword␣
↪arguments: 'arg'
```

The third function `kwd_only_arg` only allows keyword arguments as indicated by a `*` in the function definition:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

Ostatnia z funkcji wykorzystuje wszystkie trzy konwencje w swojej definicji:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got some positional-only arguments passed as keyword␣
↪arguments: 'pos_only'
```

Na koniec, przyjrzyj się definicji funkcji, która ma potencjalną kolizję pomiędzy pozycyjnym argumentem `name` a `**kwds` gdzie `name` jest kluczem:

```
def foo(name, **kwds):
    return 'name' in kwds
```

Nie jesteśmy w stanie sprawić, aby ta funkcja zwróciła `True`, jako że słowo kluczowe `'name'` zawsze powiązane będzie z pierwszym parametrem. Dla przykładu:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

Korzystając z / (argumentów tylko-pozycyjnych), da się to zrobić ponieważ pozwala ono na `name` jako argument pozycyjny i `'name'` jako klucz w argumentach kluczowych:

```
>>> def foo(name, /, **kwds):
...     return 'name' in kwds
...
>>> foo(1, **{'name': 2})
True
```

Innymi słowy, nazwy w argumentach pozycyjnych mogą być bez dwuznaczności wykorzystywane jako klucze w `**kwds`.

Podsumowanie

Przypadki użycia determinują, jakich typów parametrów kiedy w definicji funkcji:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

Kieruj się następującym:

- Wykorzystuj argumenty tylko-pozycyjne gdy chcesz aby nazwa parametrów nie była dostepna dla użytkownika. To może być pomocne gdy nazwy parametów nie mają specjalnego znaczenia a ty chcesz wymusić określoną kolejność argumentów gdy funkcja jest wywoływana. Ewentualnie gdy chcesz aby część argumentów była pozycyjna a część była jakimiś słowami kluczowymi.
- Wykorzystuj argumenty tylko-kluczowe gdy nazwy mają znaczenie i definicja funkcji jest łatwiejsza do zrozumienia poprzez wprost podawanie nazw, lub jeśli nie chcesz aby użytkownicy polegali na kolejności argumentów.
- W przypadku budowania API, wykorzystanie argumentów tylko-pozycyjnych pozwoli w przyszłości uniknąć problemów, gdy nazwa parametru jest zmieniona.

4.9.4 Arbitralne listy argumentów

Najmniej często wykorzystywaną opcją jest specyfikowanie, że funkcja może być wywoływana z arbitralną liczbą argumentów. Takie argumenty zostaną opakowane w krotkę (zobacz *krotki*). Przed zmienną liczbą argumentów, można wymusić jeden lub więcej argumentów.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Zazwyczaj takie argumenty będą ostatnie na liście formalnych parametrów, ponieważ zbierają one wszystkie pozostałe argumenty przekazane funkcji. Każdy formalny argument po parametrze `*args` może być «tylko-kluczowy», to znaczy da się go wprowadzić tylko poprzez słowo kluczowe a nie przez pozycję.

```
>>> def concat(*args, sep="/"):
...     return sep.join(args)
...
>>> concat("ziemia", "mars", "wenus")
'ziemia/mars/wenus'
>>> concat("ziemia", "mars", "wenus", sep=".")
'ziemia.mars.wenus'
```

4.9.5 Rozpakowywanie listy argumentów

Odwrotna sytuacja wystąpi, gdy argumenty już są listą albo krotką a muszą być rozpakowane gdyż funkcja wymaga argumentów przekazanych pozycyjnie, jeden po drugim . Dla przykładu, wbudowana funkcja `range()` oczekuje oddzielnych argumentów *start* oraz *stop*. Jeśli nie są dostępne oddzielnie, wywołaj funkcją z operatorem `*` aby wypakować argumenty z listy lub krotki:

```
>>> list(range(3, 6))            # zwykłe wywołanie z osobnymi argumentami
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))            # wywołanie z argumentami rozpakowanymi z listy
[3, 4, 5]
```

W podobny sposób, słowniki moga dostarczać argumentów kluczowych poprzez operator `**`:

```
>>> def parrot(voltage, state='sztywna', action='fru'):
...     print("-- Ta papuga nie zrobi", action, end=' ')
...     print("nawet jeśli podłączę ją do", voltage, "woltów.", end=' ')
...     print("Jest", state, "!")
...
>>> d = {"voltage": "czterech milionów", "state": "sztywna jak kłoda", "action":
↪"FRUU"}
>>> parrot(**d)
-- Ta papuga nie zrobi FRUU nawet jeśli podłączę ją do czterech milionów woltów.␣
↪Jest sztywna jak kłoda !
```

4.9.6 Wyrażenia Lambda

Niewielkie anonimowe funkcje mogą byś tworzone z wykorzystaniem słowa kluczowego `lambda`. Wykorzystując tą notację: `lambda a, b: a+b` powstanie funkcja sumująca podane argumenty. Funkcje lambda mogą być wykorzystywane zawsze wtedy gdy potrzebne są objekty funkcji. Są synaktycznie ograniczone do jednego wyrażenia. Semantycznie, jest to tylko lukier składniowy normalnej definicji funkcji. Podobnie jak funkcje zagnieżdżone funkcje lambda mogą odwoływać się do zmiennych z otaczającego zakresu

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Powyższy przykład wykorzystuje ekspresję lambda aby zwrócić funkcję. Inne wykorzystanie to przekazanie małej funkcji jako argumentu:

```
>>> pairs = [(1, 'jeden'), (2, 'dwa'), (3, 'trzy'), (4, 'cztery')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'cztery'), (2, 'dwa'), (1, 'jeden'), (3, 'trzy')]
```

4.9.7 Napisy dokumentujące (docstringi)

Oto kilka konwencji dotyczących zawartości i formatowania docstringów.

Pierwsza linijka powinna zawierać zwięzłe streszczenie sensu jaki stoi za obiektem. Nie powinna wprost zawierać nazwy obiektu ani typu, gdyż te są dostępne w inny sposób (chyba, że nazwa jest czasownikiem opisującym działanie funkcji). Ta linijka powinna zaczynać się z dużej litery i kończyć kropką.

Jeśli w docstringu jest więcej niż jedna linijka, druga linijka powinna być pusta, aby rozdzielić ją od reszty opisu. Następne linijki powinny zawierać konwencje nazewnicze, efekty uboczne itd.

Parser Pythona nie usuwa wcięć z literału stringu wielolinijkowego, więc jeśli to jest pożądane, to narzędzia obrabiające dokumentację powinny usuwać wcięcia. Robi się to, wykorzystując następujące. Pierwsza nie-pusta linijka *po* pierwszej określa jak dużo wcięcia jest w całym docstringu. (Nie można do tego wykorzystać pierwszej linijki, ponieważ ta zazwyczaj przylega do cudzysłowów, więc sposób wcięcia jest nieoczywisty.) „Ilość wcięcia" z tej linijki jest następnie usuwana z tej i wszystkich następnych linijek. W kolejnych linijkach nie powinno być mniej wcięć ale jeśli tak będzie to całość wcięcia powinna być usunięta. Ilość wcięcia powinna być usuwana po zamianie tabulatorów na spacje (zazwyczaj na 8 spacji).

Poniżej przykład wielolinijkowego docstringu:

```
>>> def my_function():
...     """Nie robi nic, ale dokumentuje to.
...
...     Nie, naprawdę, ona nic nie robi.
...     """
... pass
...
>>> print(my_function.__doc__)
Nie robi nic, ale dokumentuje to.

    Nie, naprawdę, ona nic nie robi.
```

4.9.8 Adnotacje funkcji

Adnotacje funkcji to całkowicie opcjonalne metadane dające informacje o funkcjach zdefiniowanych przez użytkowników (zobacz **PEP 3107** oraz **PEP 484** aby uzyskać więcej informacji).

Adnotacje przechowywane są w atrybucie `__annotations__` funkcji jako słownik i nie mają oprócz żadnego innego wpływu na nią. Adnotacje parametrów są definiowane po nazwie parametru i dwukropku, jako wyrażenie sprawdzające wartość argumentu. Adonotacje do zwracanego wyniku definiuje się pomiędzy nawiasem z listą parametrów a drukropkiem kończącym instrukcję `def` , poprzez literał `->`, z następującym po nim wyrażeniem. Poniższy przykład ma adnotację dotyczącą argumentu wymaganego, argumentu opcjonalnego oraz adnotację zwracanego wyniku:

```
>>> def f(ham: str, eggs: str = 'jajka') -> str:
...     print("adnotacje:", f.__annotations__)
...     print("argumenty:", ham, eggs)
...     return ham + ' i ' + eggs
...
>>> f('szynka')
adnotacje: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
argumenty: szynka jajka
'szynka i jajka'
```

4.10 Intermezzo: Styl kodowania

Teraz, kiedy już jesteś gotowa(-wy) aby pisać dłuższe, bardziej złożone pythonowe dzieła, dobrze żebyśmy porozmawiali o *stylu kodowania*. Większość języków może być pisana (a mówiąc precyzyjniej, *formatowana*) w różnym stylu; bardziej lub mniej czytelnym. Zawsze dobrze jest dążyć, aby twój kod był łatwy do czytania przez innych a w tym bardzo pomaga stosowanie fajnego stylu kodowania.

Dla Pythona **PEP 8** stał się wzorcem stylu, którego trzyma się większość projektów; szerzy czytelny i miły dla oka styl kodowania. W którymś momencie, powinien go przeczytać każdy developer Pythona, poniżej przedstawiliśmy jego najistotniejsze elementy:

- Jako wcięcie, wykorzystuj cztery spacje, nie tabulator.

Cztery spacje są dobrym kompromisem pomiędzy płytkim wcięciem (pozwala na więcej kroków zagnieżdżania) a głębokim wcięciem (jest łatwiejsze do przeczytania). Tabulatorów najlepiej nie używać, wprowadzają zamiesznie.

- Zawijaj linie tak, aby nie ich długość nie przekraczała 79 znaków.

  To pomoże użytkownikom z małymi wyświetlaczami a na większych ekranach pozwoli mieć kilka plików obok siebie na ekranie.

- Wstawiaj puste linie aby oddzielić od siebie funkcje, klasy lub większe bloki kodu wewnątrz funkcji.
- Jeśli jest to możliwe, wstawiaj komentarze w oddzielnej linii.
- Wykorzystuj docstringi
- Korzystaj ze spacji naokoło operatorów oraz za przecinkami, ale nie przy nawiasach: `a = f(1, 2) + g(3, 4)`.
- Nazywaj klasy i funkcje w konsekwentny sposób. Preferowaną konwencją jest „UpperCamelCase” dla nazw klas i „lowercase_with_underscores” dla funkcji i metod. Zawsze wykorzystuj „self” jako nazwę dla pierwszego argumentu metody (zobacz *Pierwsze spojrzenie na klasy* aby dowiedzieć się więcej o klasach i metodach).
- Nie wykorzystuj wymyślnych zestawów znaków jeśli twój kod będzie wykorzystywany międzynarodowo. Najlepiej trzymać się domyślnych w Pythonie: UTF-8 lub nawet ASCII.
- Podobnie, nie korzystaj ze znaków innych niż ASCII jako identyfikatorów, jeśli jest chociaż szansa, że osoby mówiące innym językiem będą czytać lub rozwijać Twój kod.

ROZDZIAŁ 5

Struktury danych

Ten rozdział opisuje bardziej szczegółowo niektóre rzeczy, które już poznaliście, a także dodaje kilka nowych.

5.1 Więcej na temat list

Typ danych listy ma kilka dodatkowych metod. Poniżej znajdują się wszystkie metody obiektów typu listy:

`list.`**`append`**`(`*x*`)`

Dodaje element na końcu listy. Ekwiwalent `a[len(a):] = [x]`.

`list.`**`extend`**`(`*iterable*`)`

Rozszerza listę przez dodanie wszystkich elementów iterable'a. Ekwiwalent `a[len(a):] = iterable`.

`list.`**`insert`**`(`*i*, *x*`)`

Wstawia element na podaną pozycję. Pierwszy argument jest indeksem elementu, przed który wstawiamy, więc `a.insert(0, x)` wstawia na początek listy a `a.insert(len(a), x)` odpowiada `a.append(x)`.

`list.`**`remove`**`(`*x*`)`

Usuwa pierwszy element z listy, którego wartość jest równa *x*. Rzuca `ValueError`, jeśli nie ma takiego elementu.

`list.`**`pop`**`(`[*i*]`)`

Usuwa element na podanej pozycji na liście i zwraca go. Jeśli nie podano indeksu, `a.pop()` usuwa i zwraca ostatnią pozycję na liście. Funkcja rzuca `IndexError`, jeśli lista jest pusta lub indeks znajduje się poza zakresem listy.

`list.`**`clear`**`()`

Usuwa wszystkie elementy z listy. Ekwiwalent `del a[:]`.

`list.`**`index`**`(`*x*[, *start*[, *end*]]`)`

Zwraca indeks (liczony od zera) pierwszego elementu na liście, którego wartość jest równa *x*. Rzuca `ValueError`, jeśli nie ma takiego elementu.

Opcjonalne argumenty *start* i *end* są interpretowane jak w notacji slice i służą do ograniczenia wyszukiwania do szczególnej podsekwencji listy. Zwracany indeks jest wyliczany względem początku pełnej sekwencji, nie względem argumentu *start*.

`list.`**`count`**`(`*x*`)`

Zwraca liczbę razy, jaką *x* występuje w liście.

`list.sort`(*, *key=None*, *reverse=False*)

Sortuje elementy listy w miejscu (argumenty mogą służyć do dostosowania sortowania, patrz `sorted()` po ich wyjaśnienie).

`list.reverse`()

Odwraca elementy listy w miejscu.

`list.copy`()

Zwraca płytką kopię listy. Ekwiwalent `a[:]`.

Przykład, który używa większość metod listy:

```
>>> fruits = ['pomarańcza', 'jabłko', 'winogrono', 'banan', 'kiwi', 'jabłko',
↪'banan']
>>> fruits.count('jabłko')
2
>>> fruits.count('mandarynka')
0
>>> fruits.index('banan')
3
>>> fruits.index('banan', 4)  # znajdź następnego banana zaczynając od 4. pozycji
6
>>> fruits.reverse()
>>> fruits
['banan', 'jabłko', 'kiwi', 'banan', 'winogrono', 'jabłko', 'pomarańcza']
>>> fruits.append('gruszka')
>>> fruits
['banan', 'jabłko', 'kiwi', 'banan', 'winogrono', 'jabłko', 'pomarańcza', 'gruszka
↪']
>>> fruits.sort()
>>> fruits
['banan', 'banan', 'gruszka', 'jabłko', 'jabłko', 'kiwi', 'pomarańcza', 'winogrono
↪']
>>> fruits.pop()
'winogrono'
```

Być może zauważyłeś(-łaś), że metody takie jak `insert`, `remove` czy `sort`, które tylko modyfikują listę, nie mają wypisanej wartości zwrotnej – zwracają domyślne `None`.[1] Jest to zasada projektowa dla wszystkich mutowalnych struktur danych w Pythonie.

Możesz też zauważyć, że nie wszystkie dane da się posortować lub porównać. Na przykład, `[None, 'hello', 10]` nie sortuje się, ponieważ liczby całkowite nie mogą być porównywane do ciągów znaków a `None` nie może być porównywany do innych typów. Są również typy, które nie mają określonej relacji porządku. Na przykład `3+4j < 5+7j` nie jest poprawnym porównaniem.

5.1.1 Używanie list jako stosów

Metody listy ułatwiają używanie listy jako stosu, gdzie ostatni element dodany jest pierwszym elementem pobieranym („last-in, first-out"). Aby dodać element na wierzch stosu, użyj `append()`. Aby pobrać element z wierzchu stosu, użyj `pop()` bez podanego indeksu. Na przykład:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
```

(ciąg dalszy na następnej stronie)

[1] Inne języki mogą zwracać zmieniony obiekt, co pozwala na łańcuchowanie metod, na przykład `d->insert("a")->remove("b")->sort();`.

(kontynuacja poprzedniej strony)

```
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2 Używanie list jako kolejek

Można też używać list jako kolejek, gdzie pierwszy element dodany jest pierwszym elementem pobieranym („first-in, first-out”); jednakże listy nie są wydajne do tego celu. Appendy i popy na końcu listy są szybkie, lecz inserty i popy na początku listy są wolne (ponieważ wszystkie inne elementy muszą zostać przesunięte o jeden).

Aby zaimplementować kolejkę, użyj `collections.deque`, która została zaprojektowana, by mieć szybkie appendy i popy na obu końcach. Na przykład:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # pojawia się Terry
>>> queue.append("Graham")          # pojawia się Graham
>>> queue.popleft()                 # pierwszy, który się pojawił, teraz wychodzi
'Eric'
>>> queue.popleft()                 # drugi, który się pojawił, teraz wychodzi
'John'
>>> queue                           # pozostała kolejka w kolejności przybycia
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Wyrażenia listowe

Wyrażenia listowe są zwięzłym sposobem na tworzenie list. Powszechne zastosowania to tworzenie nowych list, gdzie każdy element jest wynikiem jakichś operacji zastosowanych do każdego elementu innej sekwencji lub iterable'a lub do tworzenia podsekwencji tych elementów, które spełniają określony warunek.

Na przykład załóżmy, że chcemy stworzyć listę kwadratów, jak tu:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Zwróć uwagę, że ten kod tworzy (lub nadpisuje) zmienną o nazwie `x`, która wciąż istnieje po wykonaniu pętli. Możemy obliczyć listę kwadratów bez żadnych efektów ubocznych w następujący sposób:

```
squares = list(map(lambda x: x**2, range(10)))
```

lub równoważnie:

```
squares = [x**2 for x in range(10)]
```

co jest bardziej zwięzłe i czytelne.

Wyrażenie listowe składa się z nawiasów zawierających wyrażenie oraz klauzulę `for`, następnie zero lub więcej klauzul `for` lub `if`. Rezultatem będzie nowa lista powstała z obliczenia wyrażenia w kontekście klauzul `for` i `if`, które po nim następują. Na przykład to wyrażenie listowe łączy elementy dwóch list, jeśli nie są równe:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

i jest odpowiednikiem:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Zwróć uwagę, że kolejność instrukcji `for` i `if` jest taka sama w obu fragmentach kodu.

Jeśli wyrażenie jest krotką (tak jak `(x, y)` w poprzednim przykładzie), musi być wzięte w nawias.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # stwórz nową listę ze zdwojonymi wartościami
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # przefiltruj listę, by wykluczyć liczby ujemne
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # zastosuj funkcję do każdego elementu
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # wywołaj metodę na każdym elemencie
>>> freshfruit = ['  banan', '  malina ', 'marakuja  ']
>>> [weapon.strip() for weapon in freshfruit]
['banan', 'malina', 'marakuja']
>>> # stwórz listę dwukrotek jak (liczba, kwadrat)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # krotka musi być w nawiasach, inaczej dostajemy błąd
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1
    [x, x**2 for x in range(6)]
     ^^^^^^^
SyntaxError: did you forget parentheses around the comprehension target?
>>> # spłaszcz listę używając list comprehension z dwoma 'for-ami'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Wyrażenia listowe mogą zawierać złożone wyrażenia i zagnieżdżone funkcje:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4 Zagnieżdżone wyrażenia listowe

Wyrażeniem wyjściowym w wyrażeniu listowym może być każde arbitralne wyrażenie, włączając inne wyrażenie listowe.

Rozważmy następujący przykład macierzy 3-na-4 zaimplementowanej jako lista trzech list o długości 4:

```
>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

Następujące wyrażenie listowe przetransponuje wiersze i kolumny:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Jak widzieliśmy w poprzedniej sekcji, wewnętrzne wyrażenie listowe jest ewaluowane w kontekście `for`, które po nim następuje, więc ten przykład jest równoważny temu:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

który z kolei jest taki sam jak:

```
>>> transposed = []
>>> for i in range(4):
...     # następne 3 linie implementują zagnieżdżone list comprehension
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

W prawdziwym świecie powinieneś(-nnaś) preferować wbudowane funkcje w złożonych instrukcjach przepływu. Funkcja `zip()` bardzo się przyda w tym przypadku:

```
>>> list(zip(*matrix))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

W *Rozpakowywanie listy argumentów* znajdziesz wyjaśnienie znaku gwiazdki w tej linii.

5.2 Instrukcja `del`

Element można usunąć z listy mając jego indeks zamiast wartości: instrukcją `del`. Jest ona różna od metody `pop()`, która zwraca wartość. Instrukcji `del` można też użyć do usunięcia slice'ów lub wyczyszczenia całej listy (zrobiliśmy to wcześniej przypisując pustą listę do slice'a). Na przykład:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` można również użyć do usuwania całych zmiennych:

```
>>> del a
```

Odniesienie się do nazwy `a` odtąd jest błędem (przynajmniej dopóki nie przypisana jest do niej inna wartość). Później odnajdziemy więcej zastosowań dla `del`.

5.3 Krotki i sekwencje

Widzieliśmy, że listy i ciągi znaków mają wiele wspólnych własności, takich jak indeksowanie i operacje slice. Są one dwoma przykładami *sekwencyjnych* typów danych (patrz typesseq). Jako że Python jest ewoluującym językiem, mogą zostać dodane inne sekwencyjne typy danych. Jest też inny standardowy sekwencyjny typ danych: *krotka*.

Krotka składa się z kilku wartości rozdzielonych przecinkami, na przykład:

```
>>> t = 12345, 54321, 'dzień dobry!'
>>> t[0]
12345
>>> t
(12345, 54321, 'dzień dobry!')
>>> # krotki mogą być zagnieżdżane:
>>> u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'dzień dobry!'), (1, 2, 3, 4, 5))
>>> # krotki są niemutowalne
>>> t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # ale mogą zawierać mutowalne obiekty:
>>> v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Jak widzisz na wyjściu krotki zawsze są otoczone nawiasami, tak aby zagnieżdżone krotki były poprawnie interpretowane; wpisać je można z lub bez otaczających nawiasów, chociaż często nawiasy są i tak potrzebne (jeśli krotka jest częścią większego wyrażenia). Nie da się przypisać wartości do pojedynczych elementów krotki, ale da się stworzyć krotki, które zawierają mutowalne obiekty, takie jak listy.

Mimo że krotki mogą wydawać się podobne do list, często są używane w innych sytuacjach i do innych celów. Krotki są *niemutowalne* i zazwyczaj zawierają heterogeniczne sekwencje elementów, do których dostęp uzyskuje się przez rozpakowywanie (patrz później w tej sekcji) lub indeksowanie (lub nawet przez atrybut w przypadku `namedtuples`). Listy są *mutowalne* i ich elementy są zazwyczaj homogeniczne i dostęp do nich uzyskuje się przez iterowanie po liście.

Specjalnym problemem jest konstrukcja krotek zawierających 0 lub 1 element: składnia przewiduje na to kilka sposobów. Puste krotki można konstruować pustą parą nawiasów; krotkę z jednym elementem można skonstruować umieszczając przecinek za wartością (nie wystarczy otoczyć pojedynczej wartości nawiasami). Brzydkie, ale działa. Na przykład:

```
>>> empty = ()
>>> singleton = 'cześć', # <- zwróć uwagę na przecinek na końcu
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('cześć',)
```

Instrukcja `t = 12345, 54321, 'hello!'` jest przykładem *pakowania krotki*: wartości `12345`, `54321` i `'hello!'` są razem zapakowane w krotkę. Możliwa jest również odwrotna operacja:

```
>>> x, y, z = t
```

Takie coś nazywane jest, odpowiednio, *rozpakowywaniem sekwencji*. Takie rozpakowywanie wymaga, aby po lewej stronie znaku równości było tyle samo zmiennych, ile jest elementów w sekwencji. Zauważcie, że wielokrotne przypisanie jest kombinacją pakowania i rozpakowywania sekwencji.

5.4 Zbiory

Python ma również typ danych dla zbiorów. Zbiór jest nieuporządkowaną kolekcją bez zduplikowanych elementów. Podstawowe użycia to sprawdzenie zawierania i eliminacja duplikatów. Obiekty zbiorów wspierają też operacje matematyczne jak suma, iloczyn, różnica i różnica symetryczna zbiorów.

Zbiory można stworzyć używając nawiasów klamrowych lub funkcji `set()`. Uwaga: aby stworzyć pusty zbiór, musisz użyć `set()`, nie `{}`; to drugie tworzy pusty słownik, strukturę danych, którą omówimy w następnej sekcji.

Poniżej krótka demonstracja:

```
>>> basket = {'jabłko', 'pomarańcza', 'jabłko', 'gruszka', 'pomarańcza', 'banan'}
>>> print(basket) # pokaż, że duplikaty zostały usunięte
{'pomarańcza', 'banan', 'gruszka', 'jabłko'}
>>> 'pomarańcza' in basket # szybkie sprawdzenie zawierania
True
>>> 'wiechlina' in basket
False

>>> # demonstracja operacji na zbiorach dla unikalnych liter z dwóch słów
>>>
>>> a = set('abrakadabra')
>>> b = set('alakazam')
>>> a # unikalne litery w a
{'a', 'r', 'b', 'k', 'd'}
>>> a - b # litery w a ale nie w b
{'r', 'd', 'b'}
>>> a | b # litery w a lub b lub w obu
{'a', 'k', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b # litery w obu a i b
{'a', 'k'}
>>> a ^ b # litery w a lub b ale nie w obu
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Podobnie do *wyrażeń listowych*, są wspierane również wyrażenia zbiorów:

```
>>> a = {x for x in 'abrakadabra' if x not in 'abc'}
>>> a
{'r', 'k', 'd'}
```

5.5 Słowniki

Innym przydatnym typem danych wbudowanym w Pythona jest *słownik* (patrz typesmapping). Słowniki w innych językach czasem występują jako „pamięci asocjacyjne” albo „tablice asocjacyjne”. W przeciwieństwie do sekwencji, które są indeksowane zakresem liczb, słowniki są indeksowane przez *klucze*, które mogą być dowolnym niemutowalnym typem; ciągi znaków i liczby zawsze mogą być kluczami. Można użyć krotek, jeśli zawierają tylko ciągi znaków, liczby lub krotki; jeśli krotka zawiera choć jeden mutowalny obiekt, bezpośrednio lub pośrednio, nie można jej użyć jako klucza. Nie możesz używać list jako kluczy, jako że listy mogą być modyfikowane „w miejscu” przy użyciu przypisań do indeksu, przypisań do slice'ów lub metod jak `append()` i `extend()`.

Najlepiej jest myśleć o słowniku jako zbiorze par *klucz: wartość*, z wymaganiem aby klucze były unikalne (w obrębie jednego słownika). Para nawiasów klamrowych tworzy pusty słownik: `{}`. Umieszczenie listy par klucz:wartość

rozdzielonych przecinkami dodaje początkowe pary do słownika; w ten sposób również słowniki są wypisywane na wyjściu.

Głównymi operacjami na słowniku są umieszczanie wartości pod jakimś kluczem oraz wyciąganie wartości dla podanego klucza. Możliwe jest również usunięcie pary klucz:wartość przy użyciu `del`. Jeśli umieścisz wartość używając klucza, który już jest w użyciu, stara wartość powiązana z tym kluczem zostanie zapomniana. Próba wyciągnięcia wartości przy użyciu nieistniejącego klucza zakończy się błędem.

Wykonanie `list(d)` na słowniku zwraca listę wszystkich kluczy używanych w słowniku, w kolejności wstawiania (jeśli chcesz je posortować, użyj `sorted(d)`). Aby sprawdzić, czy pojedynczy klucz jest w słowniku, użyj słowa kluczowego `in`.

Mały przykład użycia słownika:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

Konstruktor `dict()` buduje słowniki bezpośrednio z sekwencji par klucz-wartość:

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dodatkowo można użyć wyrażeń słownikowych to tworzenia słowników z podanych wyrażeń klucza i wartości:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Kiedy klucze są prostymi ciągami znaków, czasem łatwiej jest podać pary używając argumentów nazwanych:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

5.6 Techniki pętli

Podczas iterowania po słownikach, klucz i odpowiadającą mu wartość można pobrać w tym samym czasie używając metody `items()`.

```
>>> knights = {'galahad': 'cnotliwy', 'robin': 'odważny'}
>>> for k, v in knights.items():
...     print(k, v)
...
galahad cnotliwy
robin odważny
```

Przy iterowaniu po sekwencji, indeks pozycyjny i odpowiadającą mu wartość można pobrać w tym samym czasie używając funkcji `enumerate()`.

```
>>> for i, v in enumerate(['kółko', 'i', 'krzyżyk']):
...     print(i, v)
...
0 kółko
1 i
2 krzyżyk
```

Aby przeiterować po dwóch lub więcej sekwencjach w tym samym czasie, elementy mogą zostać zgrupowane funkcją `zip()`.

```
>>> questions = ['imię', 'misja', 'ulubiony kolor']
>>> answers = ['lancelot', 'święty graal', 'niebieski']
>>> for q, a in zip(questions, answers):
...     print('Jego {0} to {1}.'.format(q, a))
...
Jego imię to lancelot.
Jego misja to święty graal.
Jego ulubiony kolor to niebieski.
```

Aby przeiterować po sekwencji od końca, najpierw określ sekwencję w kierunku „do przodu” a następnie wywołaj funkcję `reversed()`.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Aby przeiterować po sekwencji w posortowanej kolejności, użyj funkcji `sorted()`, która zwraca nową posortowaną listę pozostawiając listę źródłową niezmienioną.

```
>>> basket = ['banan', 'jabłko', 'banan', 'pomarańcza', 'jabłko', 'gruszka']
>>> for i in sorted(basket):
...     print(i)
...
banan
banan
gruszka
jabłko
jabłko
pomarańcza
```

Użycie `set()` na sekwencji eliminuje zduplikowane elementy. Użycie `sorted()` w połączeniu z `set()` na sekwencji jest idiomatycznym sposobem na przeiterowanie po unikalnych elementach sekwencji w posortowanej kolejności.

```
>>> basket = ['banan', 'jabłko', 'banan', 'pomarańcza', 'jabłko', 'gruszka']
>>> for f in sorted(set(basket)):
... print(f)
...
banan
gruszka
jabłko
pomarańcza
```

Czasem kusi, żeby zmienić listę podczas iterowania po niej; jednak często prościej i bezpieczniej jest stworzyć nową listę.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 Więcej na temat warunków

Warunki użyte w instrukcjach `while` i `if` mogą zawierać dowolne operatory, nie tylko porównania.

Operatory porównania `in` i `not in` są testami należenia, które ustalają, czy wartość występuje (nie występuje) w kontenerze. Operatory `is` i `is not` porównują, czy dwa obiekty są rzeczywiście tym samym obiektem. Wszystkie operatory porównań mają ten sam priorytet, który jest niższy niż ten, który mają wszystkie operatory numeryczne.

Porównania mogą być układane w łańcuchy. Na przykład `a < b == c` sprawdza, czy `a` jest mniejsze od `b` i ponadto czy `b` równa się `c`.

Porównania można łączyć używając operatorów boolowskich `and` i `or`. Wynik porównania (lub jakiegokolwiek innego wyrażenia boolowskiego) można zanegować używając `not`. Operatory te mają mniejszy priorytet niż operatory porównania; wśród nich `not` ma najwyższy priorytet a `or` najniższy, więc `A and not B or C` jest ekwiwalentem `(A and (not B)) or C`. Jak zwykle można użyć nawiasów, aby wyrazić pożądaną kolejność kompozycji wyrażenia.

Argumenty operatorów boolowskich `and` i `or` są ewaluowane od lewej do prawej. Ewaluacja kończy się w momencie ustalenia wyniku. Na przykład, jeśli `A` i `C` są prawdą, ale `B` jest fałszem, `A and B and C` nie zewaluuje wyrażenia `C`. Przy użyciu ogólnej wartości, nie jako boolean, wartość zwracana tych operatorów to ostatnio ewaluowany argument.

Da się przypisać wynik porównania lub inne wyrażenie boolowskie do zmiennej. Na przykład,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Zwróć uwagę, że w Pythonie, w przeciwieństwie do C, przypisanie wewnątrz wyrażeń musi być wyrażone bezpośrednio przez użycie walrus operatora `:=`. W ten sposób unikamy powszechnej klasy problemów spotykanych w programach C: wpisywania = w wyrażeniu, gdy miało się intencję wpisać ==.

5.8 Porównywanie sekwencji i innych typów

Obiekty sekwencji zazwyczaj mogą być porównywane do innych obiektów, o tym samym typie sekwencji. Porównanie używa porządku *leksykograficznego*: pierwszy albo pierwsze dwa elementy są porównywane, i jeśli się różnią, to determinuje wynik porównania; jeśli są równe, następne dwa elementy są porównywane, i tak dalej, dopóki któraś z sekwencji się nie skończy. Jeśli dwa elementy do porównania same są sekwencjami tego samego typu, porównanie leksykograficzne odbywa się rekursywnie. Jeśli wszystkie elementy dwóch sekwencji są równe, takie sekwencje są traktowane jako równe. Jeśli jedna sekwencja jest początkową sub-sekwencją drugiej, ta krótsza sekwencja jest mniejszą. Porządek leksykograficzny ciągów znaków używa liczby tablicy Unicode do wyznaczenia porządku pojedynczych znaków. Niektóre przykłady porównań pomiędzy sekwencjami takiego samego typu:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Zwróć uwagę, że porównywanie obiektów innych typów przy użyciu < lub > jest dozwolone pod warunkiem, że te obiekty mają odpowiednie metody porównań. Na przykład mieszane typy numeryczne są porównywane w oparciu o ich wartość numeryczną, tak że 0 równa się 0.0 i tak dalej. W innych przypadkach, zamiast zwracać arbitralny porządek, interpreter zgłosi wyjątek `TypeError`.

ROZDZIAŁ 6

Moduły

Jeśli wyjdziesz z interpretera Pythona i wejdziesz do niego z powrotem, zauważysz, że straciłeś(-łaś) definicje, które zrobiłeś(-łaś) (funkcje i zmienne). Z tego powodu, jeśli chcesz napisać nieco dłuższy program, lepiej będzie użyć edytora tekstu, aby przygotować wejście dla interpretera i uruchomić go z tym plikiem jako wejściem. Znane jest to jako tworzenie „skryptu". Kiedy twój program staje się dłuższy, możesz podzielić go na kilka plików dla łatwiejszego utrzymania. Możesz chcieć też użyć przydatnej funkcji, którą napisałeś, w kilku programach, bez kopiowania jej definicji do każdego programu.

By to zapewnić, Python ma możliwość umieszczania definicji w pliku i używania ich w skrypcie lub w interaktywnej instancji interpretera. Taki plik nazywa się *modułem*; definicje z modułu mogą być *importowane* do innych modułów lub do modułu *main* (zbiór zmiennych, do których masz dostęp w skrypcie wykonywanym na najwyższym poziomie i w trybie kalkulatora).

Moduł to plik zawierający definicje i instrukcje Pythona. Nazwą pliku jest nazwa modułu z dodanym sufiksem `.py`. Wewnątrz modułu, jego nazwa (jako ciąg znaków) jest dostępna jako wartość zmiennej globalnej `__name__`. Na przykład użyj swojego ulubionego edytora tekstu, by stworzyć plik o nazwie `fibo.py` w bieżącym katalogu, z następującą zawartością:

```
# moduł liczb Fibonacciego

def fib(n):    # wypisz ciąg Fibonacieggo do n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # zwróć ciąg Fibonacciego do n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Teraz wejdź w interpreter Pythona i zaimportuj ten moduł następującą komendą:

```
>>> import fibo
```

Ta komenda nie dodaje nazw funkcji określonych w `fibo` bezpośrednio w bieżącej *przestrzeni nazw* (więcej szczegółów w *Zasięgi widoczności i przestrzenie nazw w Pythonie*); dodaje ona tam tylko nazwę modułu `fibo`. Używając nazwy modułu możesz użyć funkcji:

```
>>> fibo.fib(1000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Jeśli zamierzasz używać funkcji często, możesz przypisać ją do lokalnej nazwy:

```
>>> fib = fibo.fib
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Więcej o modułach

Moduł może zawierać zarówno wykonywalne instrukcje jak i definicje funkcji. Instrukcje mają inicjalizować moduł. Są wykonywane tylko za *pierwszym* razem, gdy nazwa modułu zostanie napotkana w instrukcji importu.[1] (Są one również wykonywane, jeśli plik jest wykonywany jako skrypt.)

Każdy moduł ma swoją własną przestrzeń nazw, która jest używana jako globalna przestrzeń nazw przez wszystkie funkcje zdefiniowane w tym module. Tak więc autor modułu może używać globalnych zmiennych nie martwiąc się o przypadkowe konflikty z globalnymi zmiennymi użytkownika. Z drugiej strony, jeśli wiesz co robisz, możesz odnosić się do globalnych zmiennych modułu tą samą notacją, którą używa się do odniesień do jego funkcji, `nazwamodułu.nazwalementu`.

Moduły mogą importować inne moduły. Zwyczajowo (nieobowiązkowo) umieszcza się wszystkie instrukcje `import` na początku modułu (lub skryptu). Zaimportowane nazwy modułów, jeśli są umieszczone na najwyższym poziomie modułu (poza jakimikolwiek funkcjami lub klasami), są umieszczane w globalnej przestrzeni nazw modułu.

Istnieje wariant instrukcji `import`, który importuje nazwy z modułu bezpośrednio do przestrzeni nazw modułu importującego. Na przykład:

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Nie wprowadza to nazwy modułu, z którego wzięte są importy, do lokalnej przestrzeni nazw (więc w przykładzie `fibo` nie jest zdefiniowane).

Jest również wariant importujący wszystkie nazwy definiowane przez moduł:

```
>>> from fibo import *
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Importuje to wszystkie nazwy z wyjątkiem tych zaczynających się od podkreślenia (_). Najczęściej programiści Pythona nie używają tego udogodnienia, ponieważ wprowadza ono nieznany zestaw nazw do interpretera, być może ukrywając niektóre już zdefiniowane rzeczy.

Zwróć uwagę, że ogólnie praktyka importowania `*` z modułu lub pakietu jest niemile widziana, ponieważ często powoduje, że kod jest mało czytelny. Można jej jednak używać do oszczędzania pisania w sesjach interaktywnych.

Jeżeli po nazwie modułu następuje `as`, to nazwa następująca po `as` jest powiązana bezpośrednio z importowanym modułem.

[1] W rzeczywistości definicje funkcji są również „instrukcjami”, które są „wykonywane”; wykonanie definicji funkcji na poziomie modułu dodaje nazwę funkcji do globalnej przestrzeni nazw modułu.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

W praktyce jest to importowanie modułu w taki sam sposób, jak robi to `import fibo`, z tą różnicą, że jest on dostępny jako `fib`.

Można go również użyć przy użyciu `from` z podobnym efektem:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Informacja

Ze względów wydajnościowych każdy moduł jest importowany tylko raz dla sesji interpretera. W związku z tym jeśli zmienisz swoje moduły, musisz zrestartować interpreter – lub, jeśli to tylko jeden moduł, który chcesz testować interaktywnie, użyj `importlib.reload()`, na przykład `import importlib; importlib.reload(modulename)`.

6.1.1 Wykonywanie modułów jako skryptów

Kiedy uruchamiasz moduł Pythona:

```
python fibo.py <arguments>
```

kod w module zostanie wykonany, tak jakbyś go zaimportował, ale z `__name__` ustawionym na `"__main__"`. To oznacza, że dodając ten kod na końcu swojego modułu:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

możesz uczynić plik używalnym zarówno jako skrypt oraz jako importowalny moduł, ponieważ kod, który parsuje linię komend uruchamia się tylko jeśli moduł jest wykonywany jako plik „główny”:

```
$ python fibo.py 50
0 1 1 2 3 5 8 13 21 34
```

Jeśli moduł jest zaimportowany, kod nie jest uruchamiany:

```
>>> import fibo
>>>
```

Często się z tego korzysta, aby dodać wygodny interfejs użytkownika do modułu lub na potrzeby testów (uruchomienie modułu jako skryptu wykonuje zestaw testów).

6.1.2 Ścieżka wyszukiwania modułów

Kiedy importowany jest moduł o nazwie `spam`, interpreter najpierw szuka wbudowanego modułu o takiej nazwie. Te nazwy modułów znajdują się w `sys.builtin_module_names`. Jeśli nie znajdzie, wtedy szuka pliku o nazwie `spam.py` na liście katalogów danych w zmiennej `sys.path`. `sys.path` jest inicjalizowane z tych lokalizacji:

- Katalog zawierający skrypt wejściowy (lub bieżący katalog, jeśli plik nie jest określony).
- `PYTHONPATH` (lista nazw katalogów o takiej samej składni jak zmienna shell `PATH`).
- Wartość domyślna zależna od instalacji (przez konwencję zawierająca katalog `site-packages`, obsługiwany przez moduł `site`).

Więcej szczegółów można znaleźć w sys-path-init.

Informacja

W systemach plików wspierających dowiązania symboliczne, katalog zawierający skrypt wejściowy jest wyliczany po rozwiązaniu dowiązania symbolicznego. Innymi słowy katalog zawierający dowiązanie symboliczne **nie** jest dodany do ścieżki wyszukiwania modułów.

Po inicjalizacji programy pythonowe mogą modyfikować `sys.path`. Katalog zwierający uruchamiany skrypt jest umieszczony na początku ścieżki wyszukiwania, przed ścieżką biblioteki standardowej. To znaczy, że skrypty w tym katalogu zostaną załadowane zamiast modułów o tej samej nazwie w katalogu biblioteki. Jest to błąd, o ile taka zamiana jest zamierzona. Patrz sekcja *Moduły standardowe* po więcej informacji.

6.1.3 „Skompilowane" pliki Pythona

Aby przyspieszyć ładowanie modułów, Python cache'uje skompilowaną wersję każdego modułu w katalogu `__pycache__` pod nazwą `module.wersja.pyc`, gdzie wersja koduje format skompilowanego pliku; zazwyczaj zawiera numer wersji Pythona. Na przykład w wydaniu CPythona 3.3 skompilowana wersja spam.py zostałaby zcache'owana jako `__pycache__/spam.cpython-33.pyc`. Ta konwencja nazw pozwala na współistnienie skompilowanych modułów z różnych wydań i wersji Pythona.

Python porównuje datę modyfikacji źródła ze skompilowaną wersją, aby ustalić, czy jest przeterminowana i powinna zostać zrekompilowana. To całkowicie automatyczny proces. Skompilowane moduły są niezależne od platformy, więc ta sama biblioteka może być współdzielona pomiędzy systemami z innymi architekturami.

Python nie sprawdza cache'u w dwóch przypadkach. Po pierwsze zawsze rekompiluje i nie zapisuje wyniku dla modułu załadowanego bezpośrednio z linii komend. Po drugie nie sprawdza cache'u, kiedy nie ma modułu źródłowego. Dla wsparcia dystrybucji bez źródła (tylko kompilowany), skompilowany moduł musi być w katalogu źródłowym i nie może być modułu źródłowego.

Wskazówki dla ekspertów:

- Możesz użyć przełączników `-O` lub `-OO` na komendzie `python`, aby zmniejszyć rozmiar skompilowanego modułu. Przełącznik `-O` usuwa instrukcje assert. Przełącznik `-OO` usuwa zarówno instrukcje assert jak i docstringi. Jako że niektóre programy mogą polegać na dostępności powyższych, powinieneś(-nnaś) używać tylko jeśli wiesz, co robisz. „Zoptymalizowane" moduły mają tag `opt-` i zazwyczaj są mniejsze. Przyszłe wydania mogą zmienić efekty optymalizacji.
- Program nie wykonuje się ani chwili szybciej, gdy jest czytany z pliku `.pyc` niż gdy jest czytany z pliku `.py`. Jedyna rzecz, która jest szybsza w plikach `.pyc` to szybkość, w jakiej są one ładowane.
- Moduł `compileall` może stworzyć pliki .pyc dla wszystkich modułów w katalogu.
- Więcej szczegółów na temat tego procesu, w tym diagram przepływu decyzji, znajduje się w **PEP 3147**.

6.2 Moduły standardowe

Python zawiera bibliotekę modułów standardowych, opisaną w osobnym dokumencie, dokumentacji biblioteki standardowej. Niektóre moduły są wbudowane w interpreter; dają one dostęp do operacji, które nie są częścią trzonu języka, niemniej jednak są wbudowane, dla wydajności lub aby dać dostęp do elementów systemu operacyjnego jak wywołania systemowe. Zbiór takich modułów jest opcją konfiguracyjną, która zależy również od platformy. Na przykład moduł `winreg` jest dostępny tylko w systemach Windows. Jeden szczególny moduł zasługuje na uwagę: `sys`, który jest wbudowany w każdy interpreter Pythona. Zmienne `sys.ps1` i `sys.ps2` określają ciągi znaków używane jako znaki zachęty pierwszego i drugiego rzędu:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
'... '
>>> sys.ps1 = 'C> '
C> print('Ble!')
Ble!
C>
```

Te dwie zmienne są określone tylko jeśli interpreter jest w trybie interaktywnym.

Zmienna `sys.path` jest listą ciągów znaków, która określa ścieżkę wyszukań modułów interpretera. Jest inicjalizowana domyślą ścieżką braną ze zmiennej środowiskowej `PYTHONPATH` lub z wbudowanej wartości domyślnej, jeśli `PYTHONPATH` jest nieustawiona. Możesz ją modyfikować używając standardowych operacji na listach:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 Funkcja `dir()`

Wbudowana funkcja `dir()` jest używana do sprawdzenia, jakie nazwy definiuje moduł. Zwraca uporządkowaną listę ciągów znaków:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__breakpointhook__', '__displayhook__', '__doc__', '__excepthook__',
 '__interactivehook__', '__loader__', '__name__', '__package__', '__spec__',
 '__stderr__', '__stdin__', '__stdout__', '__unraisablehook__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_framework',
 '_getframe', '_git', '_home', '_xoptions', 'abiflags', 'addaudithook',
 'api_version', 'argv', 'audit', 'base_exec_prefix', 'base_prefix',
 'breakpointhook', 'builtin_module_names', 'byteorder', 'call_tracing',
 'callstats', 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
 'float_repr_style', 'get_asyncgen_hooks', 'get_coroutine_origin_tracking_depth',
 'getallocatedblocks', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencodeerrors', 'getfilesystemencoding', 'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'is_finalizing', 'last_traceback', 'last_type', 'last_value',
 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path', 'path_hooks',
 'path_importer_cache', 'platform', 'prefix', 'ps1', 'ps2', 'pycache_prefix',
 'set_asyncgen_hooks', 'set_coroutine_origin_tracking_depth', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace', 'stderr',
 'stdin', 'stdout', 'thread_info', 'unraisablehook', 'version', 'version_info',
 'warnoptions']
```

Bez argumentów, `dir()` wypisuje nazwy zdefiniowane w bieżącym kontekście:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Zauważ, że wylistowuje wszystkie typy nazw: zmienne, moduły, funkcje i tak dalej.

`dir()` nie wypisuje nazw wbudowanych funkcji i zmiennych. Jeśli chcesz ich listę, określone one są w module

standardowym `builtins`:

```
>>> import builtins
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
 'zip']
```

6.4 Pakiety

Pakiety to sposób strukturyzacji przestrzeni nazw modułów Pythona za pomocą „nazw modułów z kropkami”. Na przykład nazwa modułu `A.B` oznacza podmoduł `B` nazwany w pakiecie o nazwie `A`. Tak jak użycie modułów oszczędza autorom martwienia się o nazwy zmiennych globalnych w pozostałych modułach, użycie nazw modułów z kropkami oszczędza autorom pakietów wielomodułowych, takich jak NumPy lub Pillow, martwienie się o przestrzeń nazw innych podmodułów.

Załóżmy, że chcesz zaprojektować zbiór modułów („pakiet”) do jednolitej obsługi plików dźwiękowych i danych dźwiękowych. Istnieje wiele różnych formatów plików dźwiękowych (zwykle rozpoznawanych po ich rozszerzeniach, na przykład: `.wav`, `.aiff`, `.au`), więc może być konieczne utworzenie i utrzymywanie rosnącej kolekcji modułów do konwersji między różnymi formatami plików. Istnieje również wiele różnych operacji, które możesz chcieć wykonać na danych dźwiękowych (takich jak miksowanie, dodawanie echa, stosowanie funkcji korektora, tworzenie sztucznego efektu stereo), więc dodatkowo będziesz pisać niekończącą się liczbę modułów do wykonywania tych operacji. Oto możliwa struktura twojego pakietu (wyrażona jako hierarchiczny system plików):

```
sound/                          Pakiet najwyższego poziomu
      __init__.py               Inicjalizacja pakietu sound
      formats/                  Podpakiet do konwersji formatu plików
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                Podpakiet do efektów dźwiękowych
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                Podpakiet dla filtrów
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Podczas importowania pakietu Python przeszukuje katalogi zapisane w `sys.path` w poszukiwaniu podkatalogu pakietu.

Pliki `__init__.py` są wymagane, aby Python traktował katalogi zawierające ten plik jako pakiety (o ile nie używa się *namespace package*, dość zaawansowanej funkcji). Zapobiega to katalogom o często występującej nazwie, takiej jak `string`, przed niezamierzonym ukrywaniem prawidłowych modułów, które pojawiają się później w ścieżce wyszukiwania modułów. W najprostszym przypadku `__init__.py` może to być po prostu pustym plikiem, ale może też wykonać kod inicjujący pakiet lub ustawić zmienną `__all__`, która będzie opisana później.

Użytkownicy pakietu mogą importować z pakietu poszczególne moduły, na przykład:

```
import sound.effects.echo
```

Spowoduje to załadowanie podmodułu `sound.effects.echo`. Musi on być wywoływany używając jego pełnej nazwy.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Alternatywnym sposobem importowania podmodułu jest:

```
from sound.effects import echo
```

To również spowoduje załadowanie podmodułu `echo` lecz udostępnia go bez prefiksu pakietu, dzięki czemu można go użyć w następujący sposób:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Jeszcze innym sposobem jest bezpośredni import żądanej funkcji lub zmiennej:

```
from sound.effects.echo import echofilter
```

Ponownie, ładuje to podmoduł `echo`, ale dzięki temu jego funkcja `echofilter()` jest bezpośrednio dostępna:

```
echofilter(input, output, delay=0.7, atten=4)
```

Zauważ, że podczas używania `from package import item`, element może być podmodułem (lub podpakietem) pakietu lub inną nazwą zdefiniowaną w pakiecie, jak funkcją, klasą lub zmienną. Instrukcja `import` najpierw sprawdza, czy element jest zdefiniowany w paczce; jeśli nie, zakłada, że jest to moduł i próbuje go załadować. Jeśli go nie znajdzie, zgłaszany jest wyjątek `ImportError`.

W przeciwieństwie do tego, gdy używana jest składnia `import item.subitem.subsubitem`, każdy element oprócz ostatniego musi być pakietem; ostatni element może być modułem lub pakietem, ale nie może być klasą, funkcją lub zmienną zdefiniowaną w poprzednim elemencie.

6.4.1 Importowanie * z pakietu

Teraz, co się dzieje, gdy użytkownik pisze `from sound.effects import *`? W idealnym świecie, mamy nadzieję, że to w jakiś sposób wychodzi do systemu plików, wyszukuje, które podmoduły są obecne w pakiecie i importuje je wszystkie. Jednak może to zająć dużo czasu, a importowanie modułów podrzędnych może mieć niepożądane skutki uboczne, które powinny wystąpić tylko wtedy, gdy moduł podrzędny zostanie jawnie zaimportowany.

Jedynym rozwiązaniem jest podanie przez autora pakietu jawnego indeksu pakietu. Instrukcja `import` wykorzystuje następującą konwencję: jeśli kod pakietu definiuje listę o nazwie `__all__`, przyjmuje się, że jest to lista nazw modułów, które powinny zostać zaimportowane, gdy zostanie wywołane `from package import *`. Do autora pakietu należy aktualizowanie tej listy po wydaniu nowej wersji pakietu. Autorzy pakietów mogą również zdecydować, że nie będą go wspierać, jeśli nie widzą zastosowania do importowania * ze swojego pakietu. Na przykład plik `sound/effects/__init__.py` może zawierać następujący kod:

```
__all__ = ["echo", "surround", "reverse"]
```

To by znaczyło że `from sound.effects import *` zaimportuje trzy wymienione podmoduły z pakietu `sound.effects`.

Wróć uwagę na to, że podmoduły mogą być przysłaniane lokalnie zdefiniowanymi nazwami. Na przykład, jeśli dodasz funkcję `reverse` do pliku `sound/effects/__init__.py`, instrukcja `from sound.effects import *` zaimportuje tylko dwa podmoduły `echo` i `surround`, ale nie podmoduł `reverse`, ponieważ jest on przysłonięty lokalnie zdefiniowaną funkcją `reverse`:

```
__all__ = [
    "echo",      # odnosi się do pliku 'echo.py'
    "surround",  # odnosi się do 'surround.py'
    "reverse",   # !!! odnosi się teraz do funkcji 'reverse' !!!
]

def reverse(msg: str):  # <-- ta nazwa przysłania submoduł 'reverse.py'
    return msg[::-1]    #     w przypadku 'from sound.effects import *'
```

Jeśli zmienna `__all__` nie jest zdefiniowana, instrukcja `from sound.effects import *` *nie* importuje wszystkich podmodułów z pakietu `sound.effects` do bieżącej przestrzeni nazw; upewnia się tylko, że pakiet `sound.effects` został zaimportowany (ewentualnie uruchamiając dowolny kod inicjujący w `__init__.py`), a następnie importuje dowolne nazwy zdefiniowane w pakiecie. Obejmuje to wszelkie nazwy zdefiniowane (i jawnie załadowane moduły podrzędne) przez `__init__.py`. Obejmuje również wszelkie podmoduły pakietu, które zostały jawnie załadowane przez poprzednie instrukcje `import`. Rozważ ten kod:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

W tym przykładzie moduły `echo` i `surround` są importowane do bieżącej przestrzeni nazw, ponieważ są zdefiniowane w pakiecie `sound.effects` podczas wykonywania instrukcji `from...import`. (Działa to również, gdy zmienna `__all__` jest zdefiniowana.)

Chociaż niektóre moduły są zaprojektowane do eksportowania tylko nazw zgodnych z określonymi wzorcami podczas używania `import *`, nadal jest to uważane za złą praktykę w kodzie produkcyjnym.

Pamiętaj, nie ma nic złego w używaniu `from package import specific_submodule`! W rzeczywistości jest to zalecana notacja, chyba że moduł importujący musi używać podmodułów o tej samej nazwie z innych pakietów.

6.4.2 Referencje wewnątrz-pakietowe

Gdy pakiety są podzielone na podpakiety (jak w przypadku pakietu `sound` w przykładzie), możesz użyć importu bezwzględnego, aby odnieść się do podmodułów siostrzanych pakietów. Na przykład, jeśli moduł `sound.filters.vocoder` musi użyć modułu `echo` w pakiecie `sound.effects`, może użyć `from sound.effects import echo`.

Można również pisać importy względne w formie takiej: `from module import name` instrukcji importu. Te importy wykorzystują wiodące kropki do wskazania pakietów bieżących i nadrzędnych biorących udział w imporcie względnym. Na przykład z modułu `surround` możesz użyć:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Należy zauważyć, że importy względne są oparte na nazwie bieżącego modułu. Ponieważ nazwa głównego modułu to zawsze `"__main__"`, moduły przeznaczone do użycia jako główny moduł aplikacji w Python-ie muszą zawsze używać bezwzględnego importu.

6.4.3 Pakiety w wielu katalogach

Pakiety obsługują jeszcze jeden specjalny atrybut, `__path__`. Jest on inicjalizowany jako *sekwencja* ciągów znaków zawierająca nazwę katalogu zawierającego `__init__.py` przed wykonaniem kodu w tym pliku. Ta zmienna może być modyfikowana; ma to wpływ na przyszłe wyszukiwania modułów i podpakietów zawartych w pakiecie.

Chociaż ta funkcja nie jest często potrzebna, można jej użyć do rozszerzenia zestawu modułów znajdujących się w pakiecie.

ROZDZIAŁ 7

Wejście i wyjście

Wyjście programu można zaprezentować na różne sposoby; można wypisać dane w formie czytelnej dla człowieka lub zapisać do pliku do wykorzystania w przyszłości. Ten rozdział omówi niektóre z możliwości.

7.1 Wymyślniejsze formatowanie wyjścia

Do tej pory natknęliśmy się na dwa sposoby wypisywania wartości: *wyrażenia* i funkcję `print()`. (Trzecim sposobem jest użycie metody `write()` obiektu pliku; do pliku standardowego wyjścia można odwołać się używając `sys.stdout`. Więcej informacji na ten temat znajdziesz w dokumentacji biblioteki standardowej.)

Często będziesz chciał(a) mieć więcej kontroli nad formatowaniem swojego wyjścia niż proste wypisywanie wartości rozdzielonych spacją. Jest kilka sposobów na formatowanie wyjścia.

- Aby użyć *literału ciągu znaków*, rozpocznij ciąg znaków znakiem `f` lub `F` przed otwierającym znakiem cudzysłowu. Wewnątrz ciągu znaków, możesz wpisać wyrażenia Pythona pomiędzy znakami `{` i `}`, które mogą odnosić się do zmiennych lub wartości.

```
>>> year = 2016
>>> event = 'referendum'
>>> f'Wyniki {event} {year}'
'Wyniki referendum 2016'
```

- Metoda `str.format()` ciągów znaków wymaga większego ręcznego wysiłku. Nadal będziesz używał(a) `{` i `}` do zaznaczenia gdzie zmienna zostanie podstawiona i możesz podać dokładne dyrektywy formatowania, ale będziesz musiał(a) podać również informacje do sformatowania. W przedstawionym bloku kodu są dwa przykłady, jak formatować zmienne.

```
>>> yes_votes = 42_572_654
>>> total_votes = 85_705_149
>>> percentage = yes_votes / total_votes
>>> '{:-9} głosów na TAK  {:2.2%}'.format(yes_votes, percentage)
' 42572654 głosów na TAK  49.67%'
```

  Zauważ, że `yes_votes` są wypełnione spacjami i znakiem minus tylko dla liczb ujemnych. Przykład wypisuje również `percentage` pomnożony przez 100, z 2 miejscami po przecinku i ze znakiem procentu (szczegóły: formatspec).

- Wreszcie, możesz wykonać całą obsługę ciągów znaków samodzielnie, używając operacji krojenia (slicing) i konkatenacji, aby stworzyć dowolny układ, jaki możesz sobie wyobrazić. Typ ciągu znaków posiada kilka metod, które wykonują przydatne operacje do wypełniania ciągów znaków do danej szerokości kolumny.

Gdy nie potrzebujesz wymyślnego wyjścia, tylko chcesz szybko wyświetlić zmienne do debugowania, możesz zkonwertować dowolną wartość do ciągu znaków przy pomocy funkcji `repr()` lub `str()`.

Funkcja `str()` ma na celu zwrócenie reprezentacji wartości, które są dość czytelne dla człowieka, podczas gdy `repr()` ma na celu wygenerowanie reprezentacji, które mogą być odczytane przez interpreter (lub wymuszą `SyntaxError`, jeśli nie ma równoważnej składni). Dla obiektów, które nie mają określonej reprezentacji do użycia przez człowieka, `str()` zwróci taką samą wartość jak `repr()`. Wiele wartości, takich jak liczby lub struktury takie jak listy i słowniki, ma taką samą reprezentację przy użyciu obu funkcji. Ciągi znaków, w szczególności, mają dwie odrębne reprezentacje.

Trochę przykładów:

```
>>> s = 'Witaj, świecie.'
>>> str(s)
'Witaj, świecie.'
>>> repr(s)
"'Witaj, świecie.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Wartość x to ' + repr(x) + ', i y to ' + repr(y) + '...'
>>> print(s)
Wartość x to 32.5, i y to 40000...
>>> # repr() ciągu znaków dodaje cudzysłów i backslashe:
>>> hello = 'witaj, świecie\n'
>>> hellos = repr(hello)
>>> print(hellos)
'witaj, świecie\n'
>>> # argumentem repr() może być dowolny obiekt Pythona:
>>> repr((x, y, ('mielonka', 'jajka')))
"(32.5, 40000, ('mielonka', 'jajka'))"
```

Moduł `string` zawiera klasę `Template`, która oferuje jeszcze jeden sposób zastępowania wartości ciągami znaków, przy użyciu placeholderów takich jak `$x` i zastępując je wartościami ze słownika, ale daje mniejszą kontrolę nad formatowaniem.

7.1.1 f-stringi

Formatowane literały ciągów znaków (nazywane też krótko f-stringami) pozwalają zawrzeć wartość pythonowych wyrażeń wewnątrz ciągu znaków przez dodanie prefiksu `f` lub `F` i zapisanie wyrażenia jako `{expression}`.

Po wyrażeniu może nastąpić opcjonalny specyfikator formatu. Umożliwia on większą kontrolę nad sposobem formatowania wartości. Poniższy przykład zaokrągla pi do trzech miejsc po przecinku:

```
>>> import math
>>> print(f'Wartość pi wynosi w przybliżeniu {math.pi:.3f}.')
Wartość pi wynosi w przybliżeniu 3,142.
```

Przekazanie liczby całkowitej po znaku `':'` spowoduje, że pole będzie miało minimalną liczbę znaków szerokości. Jest to przydatne przy tworzeniu linii kolumn.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Sjoerd      ==>         4127
Jack        ==>         4098
Dcab        ==>         7678
```

Inne modyfikatory mogą być użyte do konwersji wartości przed jej sformatowaniem. `'!a'` stosuje `ascii()`, `'!s'` stosuje `str()`, a `'!r'` stosuje `repr()`:

```
>>> animals = 'węgorze'
>>> print(f'Na moim poduszkowcu są {animals}.')
Na moim poduszkowcu są węgorze.
>>> print(f'Na moim poduszkowcu są {animals!r}.')
Na moim poduszkowcu są 'węgorze'.
```

Specyfikator = może być użyty do rozwinięcia wyrażenia do tekstu wyrażenia, znaku równości, a następnie reprezentacji obliczonego wyrażenia:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

Zobacz wyrażenia samodokumentujące po więcej informacji na temat specyfikatora =. Informacje o specyfikacjach formatu znajdziesz w przewodniku referencyjnym dla mini-języka specyfikacji formatu.

7.1.2 Metoda format() ciągu znaków

Podstawowe użycie metody `str.format()` wygląda tak:

```
>>> print('Jesteśmy {}, którzy mówią „{}!”'.format('rycerzami', 'Ni'))
Jesteśmy rycerzami, którzy mówią „Ni!”
```

Nawiasy i znaki wewnątrz nich (zwane polami formatu) są zastępowane obiektami przekazywanymi do metody `str.format()`. Liczba w nawiasach może być użyta do odwołania się do pozycji obiektu przekazanego do metody `str.format()`.

```
>>> print('{0} i {1}'.format('mielonka', 'jajka'))
mielonka i jajka
>>> print('{1} i {0}'.format('mielonka', 'jajka'))
jajka i mielonka
```

Jeśli argumenty nazwane są używane w metodzie `str.format()`, ich wartości są przywoływane przy użyciu nazwy argumentu.

```
>>> print('Ta {food} jest {adjective}.'.format(
...       food='mielonka', adjective='absolutnie okropna'))
Ta mielonka jest absolutnie okropna.
```

Argumenty pozycyjne i nazwane mogą być dowolnie łączone:

```
>>> print('Historia {0}, {1} i {other}.'.format('Billa', 'Manfreda',
...                                            other='Georga'))
Historia of Billa, Manfreda i Georga.
```

Jeśli masz naprawdę długi ciąg znaków formatu, którego nie chcesz dzielić, byłoby miło, gdybyś mógł odwoływać się do zmiennych, które mają być sformatowane według nazwy zamiast według pozycji. Można to zrobić, po prostu przekazując dict i używając nawiasów kwadratowych `'[]'`, aby uzyskać dostęp do kluczy.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Można to również zrobić, przekazując dict `table` jako argumenty nazwane przy użyciu notacji `**`.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Jest to szczególnie przydatne w połączeniu z wbudowaną funkcją `vars()`, która zwraca słownik zawierający wszystkie zmienne lokalne:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: ' + '{' + k +'};' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

Jako przykład, poniższe wiersze tworzą uporządkowany zestaw kolumn zawierających liczby całkowite oraz ich kwadraty i sześciany:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Aby uzyskać pełny przegląd formatowania ciągów znaków za pomocą funkcji `str.format()`, zobacz formatstrings.

7.1.3 Ręczne formatowanie ciągów znaków

Oto ta sama tabela kwadratów i sześcianów, sformatowana ręcznie:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # zwróć uwagę na użycie 'end' w poprzedniej linii
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Zauważ, że jedna spacja między każdą kolumną została dodana przez sposób działania funkcji `print()`: zawsze dodaje ona spacje między swoimi argumentami.)

Metoda `str.rjust()` obiektów typu string wyrównuje ciąg znaków do prawej strony w polu o zadanej szerokości poprzez wypełnienie go spacjami po lewej stronie. Istnieją podobne metody `str.ljust()` i `str.center()`. Metody te niczego nie wypisują, a jedynie zwracają nowy ciąg znaków. Jeśli wejściowy ciąg jest zbyt długi, nie obcinają go, ale zwracają go bez zmian; spowoduje to bałagan w układzie kolumn, ale zwykle jest to lepsze niż alternatywa, która polegałaby na kłamaniu na temat wartości. (Jeśli naprawdę chcesz przycinania, zawsze możesz dodać operację wycinania, jak w `x.ljust(n)[:n]`.)

Istnieje jeszcze jedna metoda, `str.zfill()`, która wypełnia ciąg liczbowy po lewej stronie zerami. Rozumie ona znaki plus i minus:

```
>>> '12'.zfill(5)
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4 Stare formatowanie ciągów znaków

Operator % (modulo) może być również używany do formatowania ciągów znaków. Dla `format % values` (gdzie *format* jest ciągiem znaków), specyfikacje konwersji `%` w ciągu znaków *format* są zastępowane przez zero lub więcej elementów *values*. Operacja ta jest powszechnie znana jako interpolacja ciągów znaków. Na przykład:

```
>>> import math
>>> print('Wartość pi to około %5.3f.' % math.pi)
Wartość pi to około 3.142.
```

Więcej informacji można znaleźć w sekcji old-string-formatting.

7.2 Odczytywanie i zapisywanie plików

`open()` zwraca *obiekt pliku* i jest najczęściej używany z dwoma argumentami pozycyjnymi i jednym argumentem nazwanym: `open(filename, mode, encoding=None)`

```
>>> f = open('plikroboczy', 'w', encoding="utf-8")
```

Pierwszy argument to ciąg znaków zawierający nazwę pliku. Drugi argument to kolejny ciąg znaków zawierający kilka znaków opisujących sposób, w jaki plik będzie używany. *Tryb* może być `'r'` gdy plik będzie tylko do odczytu, `'w'` tylko do zapisu (istniejący plik o tej samej nazwie zostanie usunięty), a `'a'` otwiera plik do dołączania; wszelkie dane zapisane do pliku są automatycznie dodawane na koniec.“»r+»“ otwiera plik zarówno do odczytu, jak i zapisu. Argument *trybu* jest opcjonalny; jeśli zostanie pominięty, zakłada się, że domyślnie został wybrany `'r'`.

Zwykle pliki są otwierane w trybie tekstowym *text mode*, co oznacza, że czytasz i zapisujesz ciągi znaków do i z pliku, które są kodowane w określonym *kodowaniu*. Jeśli *kodowanie* nie jest określone, domyślne jest zależne od platformy (patrz `open()`). Ponieważ UTF-8 jest współczesnym standardem, zaleca się stosowanie `encoding="utf-8"`, chyba że wiesz, że musisz użyć innego kodowania. Dodanie `'b'` do trybu otwiera plik w trybie binarnym *binary mode*. Dane w trybie binarnym są odczytywane i zapisywane jako obiekty `bytes`. Nie możesz określić kodowania podczas otwierania pliku w trybie binarnym.

W trybie tekstowym domyślnym podczas odczytywania następuje konwersja końcówek linii specyficznych dla platformy (`\n` na systemie Unix, `\r\n` na systemie Windows) na pojedynczy znak `\n`. Podczas zapisu w trybie tekstowym domyślnym jest konwersja wystąpień `\n` z powrotem na końcówki linii specyficzne dla platformy. Ta niewidoczna modyfikacja danych pliku jest odpowiednia dla plików tekstowych, ale uszkodzi dane binarne, takie jak te w plikach `JPEG` lub `EXE`. Bądź bardzo ostrożny, używając trybu binarnego podczas odczytywania i zapisywania takich plików.

Używanie `with` podczas pracy z obiektami plików należy do dobrych praktyk. Zaletą tego podejścia jest to, że plik jest prawidłowo zamykany po zakończeniu jego bloku, nawet jeśli w pewnym momencie zostanie zgłoszony wyjątek.

Użycie `with` jest również znacznie krótsze niż pisanie równoważnych bloków `try` - `finally`:

```
>>> with open('plikroboczy', encoding="utf-8") as f:
...     read_data = f.read()

>>> # Możemy sprawdzić, że plik został automatycznie zamknięty.
>>> f.closed
True
```

Jeśli nie używasz `with` to powinieneś wywołać `f.close()` w celu zamknięcia pliku i natychmiastowego zwolnienia wszystkich zasobów systemowych wykorzystywanych przez ten plik.

Ostrzeżenie

Wywołanie `f.write()` bez użycia `with` lub `f.close()` **może** spowodować, że argumenty `f.write()` nie zostaną w pełni zapisane na dysku, nawet jeśli program zakończy się pomyślnie.

Po zamknięciu obiektu pliku, zarówno przez instrukcję `with`, jak i `f.close()`, wszystkie próby użycia obiektu pliku automatycznie się nie powiodą.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```

7.2.1 Metody obiektów plików

Reszta przykładów w tej sekcji zakłada, że obiekt pliku o nazwie `f` został już utworzony.

Aby odczytać zawartość pliku, należy wywołać polecenie `f.read(size)`, które odczytuje pewną ilość danych i zwraca je jako ciąg znaków (w trybie tekstowym) lub obiekt bajtowy (w trybie binarnym). *size* jest opcjonalnym argumentem numerycznym. Gdy *size* jest pominięty lub ujemny, zostanie odczytana i zwrócona cała zawartość pliku; to twój problem, jeśli plik jest dwa razy większy niż pamięć twojego komputera. W przeciwnym razie odczytane i zwrócone zostanie co najwyżej *size* znaków (w trybie tekstowym) lub *size* bajtów (w trybie binarnym). Jeśli został osiągnięty koniec pliku, `f.read()` zwróci pusty ciąg znaków (`''`).

```
>>> f.read()
'To jest cały plik.\n'
>>> f.read()
''
```

`f.readline()` odczytuje pojedynczą linię z pliku; znak nowej linii (`\n`) jest pozostawiony na końcu ciągu znaków i jest pomijany tylko w ostatniej linii pliku, jeśli plik nie kończy się nową linią. To sprawia, że wartość zwracana jest jednoznaczna; jeśli `f.readline()` zwróci pusty ciąg znaków, koniec pliku został osiągnięty, podczas gdy pusta linia jest reprezentowana przez `'\n'`, ciąg znaków zawierający tylko pojedynczą nową linię.

```
>>> f.readline()
'To jest pierwsza linia pliku.\n'
>>> f.readline()
'Druga linia pliku\n'
>>> f.readline()
''
```

Aby odczytać wiersze z pliku, można wykonać pętlę na obiekcie pliku. Jest to wydajne pamięciowo, szybkie i prowadzi do prostego kodu:

```
>>> for line in f:
...     print(line, end='')
...
To jest pierwsza linia pliku.
Druga linia pliku
```

Jeśli chcesz wczytać wszystkie wiersze pliku w listę, możesz również użyć `list(f)` lub `f.readlines()`.

`f.write(string)` zapisuje zawartość *string* do pliku, zwracając liczbę zapisanych znaków.

```
>>> f.write('To jest test\n')
13
```

Inne typy obiektów muszą zostać przekonwertowane – albo na ciąg znaków (w trybie tekstowym), albo na obiekt bajtowy (w trybie binarnym) – przed ich zapisaniem:

```
>>> value = ('odpowiedź', 42)
>>> s = str(value)  # konwersja krotki na ciąg znaków
>>> f.write(s)
17
```

`f.tell()` zwraca liczbę całkowitą podającą aktualną pozycję obiektu pliku w pliku reprezentowaną jako liczba bajtów od początku pliku w trybie binarnym i nieprzejrzystą liczbę w trybie tekstowym.

Aby zmienić pozycję obiektu pliku, należy użyć `f.seek(offset, whence)`. Pozycja jest obliczana przez dodanie *offset* do punktu odniesienia; punkt odniesienia jest wybierany przez argument *whence*. Wartość *whence* równa 0 mierzy od początku pliku, 1 używa bieżącej pozycji pliku, a 2 używa końca pliku jako punktu odniesienia. *whence* może zostać pominięty i domyślnie przyjmuje wartość 0, używając początku pliku jako punktu odniesienia.

```
>>> f = open('plikroboczy', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Idź do szóstego bajtu w pliku
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2)  # Idź do trzeciego bajtu przed końcem
13
>>> f.read(1)
b'd'
```

W plikach tekstowych (otwieranych bez `b` w ciągu znaków trybu) dozwolone jest tylko wyszukiwanie względem początku pliku (wyjątkiem jest wyszukiwanie do samego końca pliku z `seek(0, 2)`), a jedynymi poprawnymi wartościami *offset* są te zwrócone z `f.tell()` lub zero. Każda inna wartość *offset* powoduje niezdefiniowane zachowanie.

Obiekty plikowe mają kilka dodatkowych metod, takich jak `isatty()` i `truncate()`, które są rzadziej używane; zapoznaj się z Library Reference, aby uzyskać pełny przewodnik po obiektach plikowych.

7.2.2 Zapisywanie struktur danych przy użyciu modułu `json`

Ciągi znaków można łatwo zapisywać i odczytywać z pliku. Liczby wymagają nieco więcej wysiłku, ponieważ metoda `read()` zwraca tylko ciągi znaków, które będą musiały zostać przekazane do funkcji takiej jak `int()`, która pobiera ciąg znaków taki jak `'123'` i zwraca jego wartość liczbową 123. W przypadku zapisywania bardziej złożonych typów danych, takich jak zagnieżdżone listy i słowniki, ręczne parsowanie i serializowanie staje się skomplikowane.

Zamiast zmuszać użytkowników do ciągłego pisania i debugowania kodu w celu zapisania skomplikowanych typów danych do plików, Python pozwala na użycie popularnego formatu wymiany danych zwanego JSON (JavaScript Object Notation). Standardowy moduł o nazwie `json` może pobierać hierarchie danych Pythona i konwertować je na reprezentacje ciągów znaków; proces ten nazywany jest *serializacją*. Rekonstrukcja danych z reprezentacji

łańcuchowej nazywana jest *deserializacją*. Pomiędzy serializacją i deserializacją, ciąg znaków reprezentujący obiekt może być przechowywany w pliku lub bazie danych, lub wysłany przez połączenie sieciowe do odległej maszyny.

Informacja

Format JSON jest powszechnie używany przez nowoczesne aplikacje w celu umożliwienia wymiany danych. Wielu programistów jest już z nim zaznajomionych, co czyni go dobrym wyborem dla interoperacyjności.

Jeśli masz obiekt `x`, możesz wyświetlić jego reprezentację JSON za pomocą prostej linii kodu:

```
>>> import json
>>> x = [1, 'prosta', 'lista']
>>> json.dumps(x)
'[1, "prosta", "lista"]'
```

Inny wariant funkcji `dumps()`, zwany `dump()`, po prostu serializuje obiekt do *pliku tekstowego*. Jeśli więc `f` jest obiektem *pliku tekstowego* otwartym do zapisu, możemy zrobić tak:

```
json.dump(x, f)
```

Aby ponownie zdekodować obiekt, jeśli `f` jest obiektem *pliku binarnego* lub *pliku tekstowego*, który został otwarty do odczytu:

```
x = json.load(f)
```

Informacja

Pliki JSON muszą być zakodowane w UTF-8. Użyj `encoding="utf-8"` podczas otwierania pliku JSON jako *pliku tekstowego* zarówno do odczytu jak i zapisu.

Ta prosta technika serializacji może obsługiwać listy i słowniki, ale serializacja dowolnych instancji klas w JSON wymaga nieco dodatkowego wysiłku. Wyjaśnienie tej kwestii znajduje się w dokumentacji modułu `json`.

Zobacz także

`pickle` – moduł pickle

W przeciwieństwie do *JSON*, *pickle* jest protokołem, który pozwala na serializację dowolnie złożonych obiektów Pythona. Jako taki, jest specyficzny dla Pythona i nie może być używany do komunikacji z aplikacjami napisanymi w innych językach. Jest on również domyślnie niezabezpieczony: deserializacja danych pickle pochodzących z niezaufanego źródła może wykonać dowolny kod, jeśli dane zostały spreparowane przez doświadczonego atakującego.

ROZDZIAŁ 8

Błędy i wyjątki

Do tej pory wiadomości o błędach były tylko wspomniane, ale jeśli próbowałaś(-łeś) przykładów to pewnie udało ci się na nie natknąć. Występują (przynajmniej) dwa charakterystyczne typy błędów: *błędy składni* (syntax errors) oraz *wyjątki* (exceptions).

8.1 Błędy składni

Błędy składni, znane również jako błędy parsowania, są prawdopodobnie najczęstszym rodzajem skarg, które pojawiają się podczas nauki Pythona:

```
>>> while True print('Witaj świecie')
  File "<stdin>", line 1
    while True print('Witaj świecie')
               ^^^^^
SyntaxError: invalid syntax
```

Parser powtarza błędną linię i wyświetla małe „strzałki" wskazujące token w linii, w której wykryto błąd. Błąd może być spowodowany brakiem tokenu *przed* wskazywanym tokenem. W przykładzie błąd jest wykryty na funkcji `print()`, ponieważ brakuje przed nią dwukropka (`':'`). Nazwa pliku i numer linii są drukowane, abyś wiedział(a), gdzie szukać, w przypadku, gdy dane wejściowe pochodzą ze skryptu.

8.2 Wyjątki

Nawet jeśli instrukcja lub wyrażenie jest poprawne składniowo, może ona wywołać błąd podczas próby jej wykonania. Błędy zauważone podczas wykonania programu są nazywane *wyjątkami* (exceptions) i nie zawsze są niedopuszczalne: już niedługo nauczysz w jaki sposób je obsługiwać. Większość wyjątków nie jest jednak obsługiwana przez program przez co wyświetlane są informacje o błędzie jak pokazano poniżej:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    10 * (1/0)
         ~^~
ZeroDivisionError: division by zero
>>> 4 + spam*3
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    4 + spam*3
        ^^^^
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    '2' + 2
    ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

Ostatni wiersz komunikatu o błędzie wskazuje, co się stało. Wyjątki występują w różnych typach, a typ jest drukowany jako część komunikatu: typy w przykładzie to `ZeroDivisionError`, `NameError` i `TypeError`. Ciąg wydrukowany jako typ wyjątku jest nazwą wbudowanego wyjątku, który wystąpił. Jest to prawdą dla wszystkich wbudowanych wyjątków, ale nie musi być prawdą dla wyjątków zdefiniowanych przez użytkownika (choć jest to przydatna konwencja). Standardowe nazwy wyjątków są wbudowanymi identyfikatorami (nie zarezerwowanymi słowami kluczowymi).

Pozostała część linii dostarcza szczegółów na temat typu wyjątku oraz informacji, co go spowodowało.

Wcześniejsza część komunikatu o błędzie pokazuje kontekst, w którym wystąpił wyjątek, w postaci śladu stosu. Ogólnie rzecz biorąc, zawiera on ślad stosu z listą linii źródłowych; jednak nie wyświetli linii odczytanych ze standardowego wejścia.

bltin-exceptions wymienia wbudowane wyjątki i ich znaczenie.

8.3 Obsługa wyjątków

Możliwe jest pisanie programów, które obsługują wybrane wyjątki. Spójrzmy na poniższy przykład, który prosi użytkownika o wprowadzenie danych, dopóki nie zostanie wprowadzona poprawna liczba całkowita, ale pozwala użytkownikowi na przerwanie programu (przy użyciu `Control-C` lub czegokolwiek innego obsługiwanego przez system operacyjny); zauważ, że przerwanie wygenerowane przez użytkownika jest sygnalizowane przez podniesienie wyjątku `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(input("Proszę podaj liczbę: "))
...         break
...     except ValueError:
...         print("Ups! To nie była poprawna liczba. Spróbuj ponownie...")
...
```

Instrukcja `try` działa następująco.

- W pierwszej kolejności wykonywane są instrukcje pod klauzulą *try* - pomiędzy słowami kluczowymi `try` i `except`.
- Jeżeli nie wystąpi żaden wyjątek, klauzula *except* jest pomijana i zostaje zakończone wykonywanie instrukcji `try`.
- Jeśli wyjątek wystąpi podczas wykonywania klauzuli `try`, reszta klauzuli jest pomijana. Następnie, jeśli jego typ pasuje do wyjątku nazwanego po słowie kluczowym `except`, wykonywana jest *klauzula except*, a następnie wykonanie jest kontynuowane po bloku try/except.
- Jeśli wystąpi wyjątek, który nie pasuje do wyjątku nazwanego w *klauzuli except*, jest on przekazywany do zewnętrznych instrukcji `try`; jeśli nie zostanie znaleziona obsługa, jest to *nieobsłużony wyjątek* i wykonanie zatrzymuje się z komunikatem błędu.

Instrukcja `try` może mieć więcej niż jedną *klauzulę except*, aby określić programy obsługi dla różnych wyjątków. Wykonany zostanie co najwyżej jeden handler. Obsługiwane są tylko wyjątki, które występują w odpowiadających im *klauzulach try*, a nie w kodzie obsługi tej samej instrukcji `try`. *Klauzula except* może określać wiele wyjątków krotką w nawiasach, na przykład:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Klasa w klauzuli `except` obsługuje wyjątki, które są instancjami samej klasy lub jednej z jej klas pochodnych (ale nie na odwrót — *klauzula except* wymieniająca klasę pochodną nie obsłuży instancji jej klas bazowych). Na przykład, poniższy kod wypisze B, C, D w tej kolejności:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Zauważ, że jeśli *klauzule except* byłyby odwrócone (z `except B` na pierwszym miejscu), wypisane zostałoby B, B, B — uruchamiana jest pierwsza pasująca *klauzula except*.

Gdy wystąpi wyjątek, może on mieć powiązane wartości, znane również jako *argumenty* wyjątku. Obecność i typy argumentów zależą od typu wyjątku.

Klauzula except może określać nazwę zmiennej po nazwie wyjątku. Zmienna jest powiązana z instancją wyjątku, która zazwyczaj posiada atrybut `args` przechowujący argumenty. Dla wygody, typy wyjątków wbudowanych definiują `__str__()` drukującą wszystkie argumenty bez odwoływania się do `.args`.

```
>>> try:
...     raise Exception('szynka', 'jajka')
... except Exception as inst:
...     print(type(inst))    # typ wyjątku
...     print(inst.args)     # argumenty przechowywane w .args
...     print(inst)          # metoda __str__ pozwala na wypisanie args␣
↪bezpośrednio,
...                          # ale jej definicja może być nadpisana w klasach␣
↪dziedziczących
...     x, y = inst.args     # rozpakowanie argumentów
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('szynka', 'jajka')
('szynka', 'jajka')
x = szynka
y = jajka
```

Dane wyjściowe metody `__str__()` wyjątku są drukowane jako ostatnia część («szczegóły») komunikatu dla nieobsłużonych wyjątków.

`BaseException` jest wspólną klasą bazową wszystkich wyjątków. Jedna z jej podklas, `Exception`, jest klasą bazową wszystkich nie-fatalnych wyjątków. Wyjątki, które nie są podklasami `Exception` nie są zazwyczaj obsługiwane, ponieważ są używane do wskazania, że program powinien się zakończyć. Obejmują one `SystemExit`, który jest rzucany przez `sys.exit()` i `KeyboardInterrupt`, które są rzucane, gdy użytkownik chce przerwać program.

`Exception` może być używany jako symbol wieloznaczny, który przechwytuje (prawie) wszystko. Dobrą praktyką jest jednak jak najdokładniejsze określenie typów wyjątków, które zamierzamy obsługiwać, i umożliwienie propagacji wszelkim nieoczekiwanym wyjątkom.

Najczęstszym wzorcem obsługi `Exception` jest drukowanie lub umieszczenie wyjątku w logach, a następnie ponowne rzucenie go (umożliwiając również obsługę wyjątku funkcji, w której znajduje się wywołanie):

```
import sys

try:
    f = open('moj_plik.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("Błąd OS:", err)
except ValueError:
    print("Nie udało się przekonwertować danych na liczbę całkowitą.")
except Exception as err:
    print(f"Nieoczekiwany {err=}, {type(err)=}")
    raise
```

Instrukcja `try … except` posiada opcjonalną *klauzulę else*, która, gdy jest obecna, musi następować po wszystkich *klauzulach except*. Jest to przydatne w przypadku kodu, który musi zostać wykonany, jeśli *klauzula try* nie rzuci wyjątku. Na przykład:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('nie mogę otworzyć', arg)
    else:
        print(arg, 'ma', len(f.readlines()), 'linii')
        f.close()
```

Użycie klauzuli `else` jest lepsze niż dodanie dodatkowego kodu do klauzuli `try`, ponieważ pozwala uniknąć przypadkowego wychwycenia wyjątku, który nie został rzucony przez kod chroniony instrukcją `try… except`.

Instrukcja *try … catch* nie obsługuje tylko wyjątków, które występują bezpośrednio w *klauzuli try*, ale także te, które występują wewnątrz funkcji, które są wywoływane (nawet pośrednio) w *klauzuli try*. Na przykład:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Błąd w czasie wykonania:', err)
...
Błąd w czasie wykonania: division by zero
```

8.4 Rzucanie wyjątków

Instrukcja `raise` pozwala programiście wymusić wystąpienie żądanego wyjątku. Na przykład:

```
>>> raise NameError('CześćiCzołem')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    raise NameError('CześćiCzołem')
NameError: CześćiCzołem
```

Jedyny argument do `raise` wskazuje wyjątek, który ma być rzucony. Musi to być albo instancja wyjątku, albo klasa wyjątku (klasa dziedzicząca z `BaseException`, taka jak `Exception` lub jedna z jej podklas). Jeśli przekazana zostanie klasa wyjątku, zostanie ona niejawnie zainicjowana przez wywołanie jej konstruktora bez argumentów:

```
raise ValueError  # skrót dla 'raise ValueError()'
```

Jeśli chcesz rozpoznać, czy wyjątek został rzucony, ale nie zamierzasz go obsługiwać, prostsza forma instrukcji `raise` pozwala na ponowne rzucenie wyjątku:

```
>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print('An exception flew by!')
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise NameError('HiThere')
NameError: HiThere
```

8.5 Łańcuch wyjątków

Jeśli nieobsłużony wyjątek wystąpi wewnątrz sekcji `except`, zostanie do niego dołączony obsługiwany wyjątek i uwzględniony w komunikacie o błędzie:

```
>>> try:
...     open("database.sqlite")
... except OSError:
...     raise RuntimeError("unable to handle error")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    open("database.sqlite")
    ~~~~^^^^^^^^^^^^^^^^^^^
FileNotFoundError: [Errno 2] No such file or directory: 'database.sqlite'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError("unable to handle error")
RuntimeError: unable to handle error
```

Aby wskazać, że wyjątek jest bezpośrednią konsekwencją innego, instrukcja `raise` dopuszcza opcjonalną klauzulę `from`:

```
# exc musi być instancją klasy Exception albo mieć wartość None
raise RuntimeError from exc
```

Może to być przydatne podczas przekształcania wyjątków. Na przykład:

```
>>> def func():
...     raise ConnectionError
...
>>> try:
...     func()
... except ConnectionError as exc:
...     raise RuntimeError('Failed to open database') from exc
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    func()
    ~~~~^^
  File "<stdin>", line 2, in func
ConnectionError

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError('Failed to open database') from exc
RuntimeError: Failed to open database
```

Umożliwia również wyłączenie automatycznego łączenia wyjątków przy użyciu idiomu `from None`:

```
>>> try:
...     open('bazadanych.sqlite')
... except OSError:
...     raise RuntimeError from None
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
    raise RuntimeError from None
RuntimeError
```

Więcej informacji na temat mechaniki łączenia w łańcuchy można znaleźć w rozdziale bltin-exceptions.

8.6 Wyjątki zdefiniowane przez użytkownika

Programy mogą nazywać własne wyjątki, tworząc nową klasę wyjątków (więcej informacji na temat klas Python można znaleźć w rozdziale *Klasy*). Wyjątki powinny zazwyczaj dziedziczyć z klasy `Exception`, bezpośrednio lub pośrednio.

Można zdefiniować klasy wyjątków, które robią wszystko, co może zrobić każda inna klasa, ale zwykle są one proste, często oferując tylko kilka atrybutów, które pozwalają na wyodrębnienie informacji o błędzie przez kod obsługi wyjątku.

Większość wyjątków ma nazwy kończące się na „Error”, podobnie jak w przypadku standardowych wyjątków.

Wiele standardowych modułów definiuje własne wyjątki w celu zgłaszania błędów, które mogą wystąpić w zdefiniowanych w nich funkcjach.

8.7 Definiowanie działań porządkujących

Instrukcja `try` ma inną opcjonalną klauzulę, która jest przeznaczona do definiowania działań porządkujących, które muszą być wykonane w każdych okolicznościach. Na przykład:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Goodbye, world!')
...
Goodbye, world!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise KeyboardInterrupt
KeyboardInterrupt
```

Jeśli klauzula `finally` jest obecna, klauzula `finally` wykona się jako ostatnie zadanie przed zakończeniem instrukcji `try`. Klauzula `finally` uruchomi się niezależnie od tego, czy instrukcja `try` spowoduje wyjątek. Poniższe punkty omawiają bardziej złożone przypadki wystąpienia wyjątku:

- Jeśli podczas wykonywania klauzuli `try` wystąpi wyjątek, może on zostać obsłużony przez klauzulę `except`. Jeśli wyjątek nie zostanie obsłużony przez klauzulę `except`, zostanie on ponownie rzucony po wykonaniu klauzuli `finally`.
- Wyjątek może wystąpić podczas wykonywania klauzul `except` lub `else`. Ponownie, wyjątek jest ponownie rzucany po wykonaniu klauzuli `finally`.
- Jeśli klauzula `finally` wykonuje instrukcje `break`, `continue` lub `return`, wyjątki nie są ponownie rzucane.
- Jeśli instrukcja `try` osiągnie instrukcję `break`, `continue` lub `return`, klauzula `finally` wykona się tuż przed wykonaniem instrukcji `break`, `continue` lub `return`.
- Jeśli klauzula `finally` zawiera instrukcję `return`, zwróconą wartością będzie ta z instrukcji `return` klauzuli `finally`, a nie wartość z instrukcji `return` klauzuli `try`.

Na przykład:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

Bardziej skomplikowany przykład:

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print("division by zero!")
...     else:
...         print("result is", result)
...     finally:
...         print("executing finally clause")
...
>>> divide(2, 1)
result is 2.0
executing finally clause
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    divide("2", "0")
    ~~~~~~^^^^^^^^^^
  File "<stdin>", line 3, in divide
    result = x / y
             ~~^~~
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Jak widać, klauzula `finally` jest wykonywana w każdym przypadku. `TypeError` rzucony przez dzielenie dwóch ciągów znaków nie jest obsłużony przez klauzulę `except` i dlatego jest ponownie rzucony po wykonaniu klauzuli `finally`.

W prawdziwych aplikacjach, klauzula `finally` jest przydatna do zwalniania zewnętrznych zasobów (takich jak pliki lub połączenia sieciowe), niezależnie od tego, czy użycie zasobu zakończyło się powodzeniem.

8.8 Predefiniowane akcje porządkujące

Niektóre obiekty definiują standardowe akcje porządkujące, które mają zostać podjęte, gdy obiekt nie jest już potrzebny, niezależnie od tego, czy operacja przy użyciu obiektu powiodła się, czy nie. Spójrz na poniższy przykład, który próbuje otworzyć plik i wydrukować jego zawartość na ekranie:

```
for line in open("moj_plik.txt"):
    print(line, end="")
```

Problem z tym kodem polega na tym, że pozostawia on plik otwarty przez nieokreślony czas po zakończeniu wykonywania tej części kodu. Nie jest to problemem w prostych skryptach, ale może być problemem dla większych aplikacjach. Instrukcja `with` pozwala na używanie obiektów takich jak pliki w sposób, który zapewnia, że są one zawsze czyszczone szybko i poprawnie:

```
with open("moj_plik.txt") as f:
    for line in f:
        print(line, end="")
```

Po wykonaniu instrukcji, plik *f* jest zawsze zamykany, nawet jeśli napotkano problem podczas przetwarzania linii. Obiekty, które, podobnie jak pliki, zapewniają predefiniowane akcje czyszczenia, wskażą to w swojej dokumentacji.

8.9 Rzucanie i obsługa wielu niepowiązanych wyjątków

Istnieją sytuacje, w których konieczne jest zgłoszenie kilku wyjątków, które wystąpiły. Dzieje się tak często w przypadku frameworków współbieżności, gdy kilka zadań może zakończyć się niepowodzeniem równolegle, ale istnieją również inne przypadki użycia, w których pożądane jest kontynuowanie wykonywania i zbieranie wielu błędów zamiast rzucenia pierwszego wyjątku.

Wbudowana `ExceptionGroup` zawiera listę instancji wyjątków, dzięki czemu mogą one być rzucone razem. Sama w sobie jest wyjątkiem, więc może być przechwycona jak każdy inny wyjątek:

```
>>> def f():
...     excs = [OSError('error 1'), SystemError('error 2')]
...     raise ExceptionGroup('there were problems', excs)
...
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> f()
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 3, in f
  |     raise ExceptionGroup('there were problems', excs)
  | ExceptionGroup: there were problems (2 sub-exceptions)
  +-+---------------- 1 ----------------
    | OSError: error 1
    +---------------- 2 ----------------
    | SystemError: error 2
    +------------------------------------
>>> try:
...     f()
... except Exception as e:
...     print(f'caught {type(e)}: e')
...
caught <class 'ExceptionGroup'>: e
>>>
```

Używając `except*` zamiast `except`, możemy selektywnie obsługiwać tylko te wyjątki w grupie, które pasują do określonego typu. W poniższym przykładzie, który pokazuje zagnieżdżoną grupę wyjątków, każda klauzula `except*` wyodrębnia z grupy wyjątki określonego typu, pozwalając wszystkim innym wyjątkom propagować się do innych klauzul i ostatecznie być ponownie zgłaszanymi.

```
>>> def f():
...     raise ExceptionGroup(
...         "group1",
...         [
...             OSError(1),
...             SystemError(2),
...             ExceptionGroup(
...                 "group2",
...                 [
...                     OSError(3),
...                     RecursionError(4)
...                 ]
...             )
...         ]
...     )
...
>>> try:
...     f()
... except* OSError as e:
...     print("There were OSErrors")
... except* SystemError as e:
...     print("There were SystemErrors")
...
There were OSErrors
There were SystemErrors
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 2, in <module>
  |     f()
  |     ~^^
  |   File "<stdin>", line 2, in f
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
  |     raise ExceptionGroup(
  |     ...<12 lines>...
  |     )
  | ExceptionGroup: group1 (1 sub-exception)
  +-+---------------- 1 ----------------
    | ExceptionGroup: group2 (1 sub-exception)
    +-+---------------- 1 ----------------
      | RecursionError: 4
      +------------------------------------
>>>
```

Należy pamiętać, że wyjątki zagnieżdżone w grupie wyjątków muszą być instancjami, a nie typami. Wynika to z faktu, że w praktyce wyjątki są zazwyczaj tymi, które zostały już rzucone i przechwycone przez program, zgodnie z następującym wzorcem:

```
>>> excs = []
... for test in tests:
...     try:
...         test.run()
...     except Exception as e:
...         excs.append(e)
...
>>> if excs:
...    raise ExceptionGroup("Niepowodzenia testów", excs)
...
```

8.10 Wzbogacanie wyjątków o notatki

Gdy wyjątek jest tworzony w celu rzucenia, jest on zwykle inicjowany informacjami opisującymi błąd, który wystąpił. Istnieją przypadki, w których przydatne jest dodanie informacji po przechwyceniu wyjątku. W tym celu wyjątki mają metodę `add_note(note)`, która akceptuje ciąg znaków i dodaje go do listy notatek wyjątku. Standardowe renderowanie tracebacku zawiera wszystkie notatki, w kolejności, w jakiej zostały dodane, po wyjątku.

```
>>> try:
...     raise TypeError('bad type')
... except Exception as e:
...     e.add_note('Add some information')
...     e.add_note('Add some more information')
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
    raise TypeError('bad type')
TypeError: bad type
Add some information
Add some more information
>>>
```

Na przykład zbierając wyjątki w grupę wyjątków, możemy chcieć dodać informacje kontekstowe dla poszczególnych błędów. Poniżej każdy wyjątek w grupie ma notatkę wskazującą, kiedy wystąpił.

```
>>> def f():
...     raise OSError('operation failed')
...
>>> excs = []
>>> for i in range(3):
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
...     try:
...         f()
...     except Exception as e:
...         e.add_note(f'Happened in Iteration {i+1}')
...         excs.append(e)
...
>>> raise ExceptionGroup('We have some problems', excs)
  + Exception Group Traceback (most recent call last):
  |   File "<stdin>", line 1, in <module>
  |     raise ExceptionGroup('We have some problems', excs)
  | ExceptionGroup: We have some problems (3 sub-exceptions)
  +-+---------------- 1 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 1
    +---------------- 2 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 2
    +---------------- 3 ----------------
    | Traceback (most recent call last):
    |   File "<stdin>", line 3, in <module>
    |     f()
    |     ~^^
    |   File "<stdin>", line 2, in f
    |     raise OSError('operation failed')
    | OSError: operation failed
    | Happened in Iteration 3
    +------------------------------------
>>>
```

ROZDZIAŁ 9

Klasy

Klasy umożliwiają łączenie danych i funkcjonalności. Tworzenie nowej klasy, tworzy nowy *typ* obiektu, umożliwiając tworzenie nowych *instancji* tego typu. Do każdej instancji klasy można przypisać atrybuty służące do utrzymywania jej stanu. Instancje klas mogą również posiadać metody (zdefiniowane przez klasę) umożliwiające modyfikację ich stanu.

W porównaniu do innych języków programowania, w Pythonie, mechanizm dodawania nowych klas wymaga niewielkiej ilości nowej składni i semantyki. Jest to połączenie mechanizmu klas, które można znaleźć w C++ i Modula-3. Klasy w Pythonie dostarczają wszystkie standardowe cechy programowania obiektowego: mechanizm dziedziczenia klas pozwala na wiele klas bazowych, klasy pochodne mogą nadpisać każdą metodę klasy lub klas bazowych i metoda może wywołać metody klas bazowych o tej samej nazwie. Obiekty mogą zawierać dowolną ilość i rodzaj danych. Zarówno klasy jak i moduły są częścią dynamicznej natury Pythona: są tworzone w trakcie działania programu i mogą być modyfikowane później, po stworzeniu.

Korzystając z terminologii C++, składniki klas (także pola) są *publiczne* (z wyjątkiem zobacz poniżej *Private Variables*), a wszystkie metody są *wirtualne*. Podobnie jak w Moduli-3, nie ma skrótów pozwalających na odnoszenie się do składników klas z ich metod: metoda jest deklarowana poprzez podanie wprost jako pierwszego argumentu obiektu, który w czasie wywołania metody zostanie jej przekazany niejawnie. Podobnie jak w Smalltalku, same klasy także są obiektami. Dostarcza nam to wyrażeń semantycznych pozwalających na importowanie i zmianę nazw klasy. Inaczej niż w C++ i Moduli-3 wbudowane typy mogą stanowić klasy, z których klasa użytkownika będzie dziedziczyć. Podobnie jak w C++, większość wbudowanych operatorów ze specjalną składnią (operatory arytmetyczne, indeksowanie) może być przedefiniowane przez instancje klasy.

(Z powodu braku ogólnie zaakceptowanej terminologii w kontekście klas, będę używał terminów ze Smalltalk i C++. Użyłbym Modula-3 ponieważ semantyka jego programowania obiektowego jest bliższa Pythonowi niż C++ ale zakładam, że mniej czytelników o nim słyszało.)

9.1 Kilka słów o nazwach i obiektach

Obiekty mają indywidualność, a wiele nazw (w wielu zakresach) może być powiązanych z tym samym obiektem. Jest to znane jako aliasing w innych językach. Zwykle nie jest to doceniane na pierwszy rzut oka w Pythonie i można je bezpiecznie zignorować, gdy mamy do czynienia z niezmiennymi typami podstawowymi (liczby, ciągi znaków, krotki). Jednak aliasing ma prawdopodobnie zaskakujący wpływ na semantykę kodu Pythona, który obejmuje zmienne obiekty, takie jak listy, słowniki i większość innych typów. Jest to zwykle wykorzystywane z korzyścią dla programu, ponieważ aliasy pod pewnymi względami zachowują się jak wskaźniki. Na przykład przekazanie obiektu jest tanie, ponieważ implementacja przekazuje tylko wskaźnik; a jeśli funkcja modyfikuje obiekt przekazany jako argument, wywołujący zobaczy zmianę — eliminuje to potrzebę stosowania dwóch różnych mechanizmów przekazywania argumentów, jak w Pascalu.

9.2 Zasięgi widoczności i przestrzenie nazw w Pythonie

Przed wprowadzeniem klas, najpierw muszę powiedzieć ci coś o zasadach zakresu Pythona. Definicje klas stosują kilka zgrabnych sztuczek z przestrzeniami nazw, a żeby w pełni zrozumieć, co się dzieje, trzeba wiedzieć, jak działają zakresy i przestrzenie nazw.

Zacznijmy od kilku definicji.

Przestrzeń nazw to odwzorowanie z nazw na obiekty. Większość przestrzeni nazw jest obecnie implementowana jako słowniki Pythona, ale to zwykle nie jest zauważalne w żaden sposób (z wyjątkiem wydajności), a to może się zmienić w przyszłości. Przykładami przestrzeni nazw są: zbiór nazw wbudowanych (zawierający funkcje np. `abs()` i nazwy wbudowanych wyjątków); nazwy globalne w module; oraz nazwy lokalne w wywołaniu funkcji. W pewnym sensie zbiór atrybutów obiektu również tworzy przestrzeń nazw. Ważną rzeczą, którą należy wiedzieć o przestrzeniach nazw, jest to, że nie ma absolutnie żadnych relacji między nazwami w różnych przestrzeniach nazw; na przykład, dwa różne moduły mogą zdefiniować funkcję `maximize` bez zamieszania — użytkownicy modułów muszą poprzedzić go nazwą modułu.

Nawiasem mówiąc, używam słowa *atrybut* dla każdej nazwy następującej po kropce — na przykład w wyrażeniu `z.real`, `real` jest atrybutem obiektu `z`. Ściśle mówiąc, odniesienia do nazw w modułach są odniesieniami atrybutowymi: w wyrażeniu `modname.funcname`, `modname` jest obiektem modułu, a `funcname` jest jego atrybutem. W tym przypadku istnieje proste odwzorowanie między atrybutami modułu i nazwami globalnymi zdefiniowanymi w module: mają tę samą przestrzeń nazw![1]

Atrybuty mogą być tylko do odczytu lub zapisywalne. W tym drugim przypadku możliwe jest przypisanie do atrybutu. Atrybuty modułu są zapisywalne: można zapisać `modname.the_answer = 42`. Zapisywalne atrybuty można również usunąć za pomocą instrukcji `del`. Na przykład, `del modname.the_answer` usunie atrybut `the_answer` z obiektu o nazwie `modname`.

Przestrzenie nazw są tworzone w różnych momentach i mają różny czas życia. Przestrzeń nazw zawierająca nazwy wbudowane jest tworzona podczas uruchamiania interpretera Pythona i nigdy nie jest usuwana. Globalna przestrzeń nazw dla modułu jest tworzona, gdy wczytywana jest definicja modułu; zwykle przestrzenie nazw modułu również trwają do zakończenia działania interpretera. Instrukcje wykonywane przez wywołanie interpretera najwyższego poziomu, zarówno odczytane z pliku skryptu, jak i interaktywnie, są uważane za część modułu o nazwie `__main__`, więc mają swoją własną globalną przestrzeń nazw. (Nazwy wbudowane w rzeczywistości również znajdują się w module; nazwany jest on `builtins`).

Lokalna przestrzeń nazw dla funkcji jest tworzona przy wywołaniu funkcji i usuwana, gdy funkcja zwraca wynik lub rzuca wyjątek, którego nie obsługuje. (Właściwie, zapominanie byłoby lepszym słowem na opisanie tego, co faktycznie się dzieje). Oczywiście, każde wywołanie rekurencyjne ma swoją własną lokalną przestrzeń nazw.

Zakres to tekstowy obszar programu Python, w którym przestrzeń nazw jest bezpośrednio dostępna. „Bezpośrednio dostępna” oznacza tutaj, że niekwalifikowane odwołanie do nazwy próbuje znaleźć ją w przestrzeni nazw.

Chociaż zakresy są określane statycznie, są używane dynamicznie. Cały czas w trakcie wykonywania programu istnieją 3 lub 4 zagnieżdżone zakresy, których przestrzenie nazw są bezpośrednio dostępne:

- najbardziej wewnętrzny zakres, który jest przeszukiwany jako pierwszy, zawiera nazwy lokalne
- zakresy wszystkich otaczających funkcji, które są przeszukiwane począwszy od najbliższego otaczającego zakresu, zawierają nazwy nielokalne, ale także nieglobalne
- przedostatni zakres zawiera globalne nazwy bieżącego modułu
- najbardziej zewnętrznym zakresem (przeszukiwanym jako ostatni) jest przestrzeń nazw zawierająca nazwy wbudowane

Jeśli nazwa jest zadeklarowana jako globalna, wtedy wszystkie referencje i przypisania przechodzą bezpośrednio do przedostatniego zakresu zawierającego globalne nazwy modułu. Aby ponownie powiązać zmienne znajdujące się poza najbardziej wewnętrznym zakresem, można użyć instrukcji `nonlocal`; jeśli nie są zadeklarowane jako nielokalne, zmienne te są tylko do odczytu (próba zapisu do takiej zmiennej po prostu utworzy *nową* zmienną lokalną w najbardziej wewnętrznym zakresie, pozostawiając identycznie nazwaną zmienną zewnętrzną bez zmian).

[1] Except for one thing. Module objects have a secret read-only attribute called `__dict__` which returns the dictionary used to implement the module's namespace; the name `__dict__` is an attribute but not a global name. Obviously, using this violates the abstraction of namespace implementation, and should be restricted to things like post-mortem debuggers.

Zazwyczaj zakres lokalny odwołuje się do nazw lokalnych (tekstowo) bieżącej funkcji. Poza funkcjami, zakres lokalny odwołuje się do tej samej przestrzeni nazw, co zakres globalny: przestrzeni nazw modułu. Definicje klas umieszczają jeszcze jedną przestrzeń nazw w zakresie lokalnym.

Ważne jest, aby zdać sobie sprawę, że zakresy są określane tekstowo: globalny zakres funkcji zdefiniowany w module jest przestrzenią nazw tego modułu, bez względu na to, skąd lub przez jaki alias funkcja jest wywoływana. Z drugiej strony, rzeczywiste wyszukiwanie nazw odbywa się dynamicznie, w czasie wykonywania — jednak definicja języka ewoluuje w kierunku statycznego rozpoznawania nazw, w czasie „kompilacji", więc nie należy polegać na dynamicznym rozpoznawaniu nazw! (W rzeczywistości zmienne lokalne są już określane statycznie).

Szczególnym dziwactwem Pythona jest to, że – jeśli nie działa instrukcja `global` lub `nonlocal` – przypisanie do nazw zawsze trafia do najbardziej wewnętrznego zakresu. Przypisania nie kopiują danych – po prostu wiążą nazwy z obiektami. To samo dotyczy usuwania: instrukcja `del x` usuwa wiązanie `x` z przestrzeni nazw, do której odwołuje się zakres lokalny. W rzeczywistości wszystkie operacje, które wprowadzają nowe nazwy, używają zakresu lokalnego: w szczególności instrukcje `import` i definicje funkcji wiążą nazwę modułu lub funkcji w zakresie lokalnym.

Instrukcja `global` może być użyta do wskazania, że określone zmienne znajdują się w zakresie globalnym i powinny być tam ponownie wiązane; instrukcja `nonlocal` wskazuje, że określone zmienne znajdują się w zakresie otaczającym i powinny być tam ponownie wiązane.

9.2.1 Przykład zakresów i przestrzeni nazw

Oto przykład pokazujący, jak odwoływać się do różnych zakresów i przestrzeni nazw oraz jak `global` i `nonlocal` wpływają na wiązanie zmiennych:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"

    def do_global():
        global spam
        spam = "global spam"

    spam = "test spam"
    do_local()
    print("After local assignment:", spam)
    do_nonlocal()
    print("After nonlocal assignment:", spam)
    do_global()
    print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Wyjście przykładowego kodu to:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Zauważ, że *lokalne* przypisanie (które jest domyślne) nie zmieniło wiązania *spam* w *scope_test*. Przypisanie `nonlocal` zmieniło wiązanie *spam* w *scope_test*, a przypisanie `global` zmieniło wiązanie na poziomie modułu.

Można również zauważyć, że nie było wcześniejszego powiązania dla *spam* przed przypisaniem `global`.

9.3 Pierwsze spojrzenie na klasy

Klasy wprowadzają trochę nowej składni, trzy nowe typy obiektów i trochę nowej semantyki.

9.3.1 Składnia definicji klasy

Najprostsza forma definicji klasy wygląda następująco:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Definicje klas, podobnie jak definicje funkcji (instrukcje `def`), muszą zostać wykonane, zanim będą miały jakikolwiek efekt. (Można sobie wyobrazić umieszczenie definicji klasy w gałęzi instrukcji `if` lub wewnątrz funkcji.)

W praktyce, instrukcje wewnątrz definicji klasy będą zwykle definicjami funkcji, ale inne instrukcje są dozwolone, a czasem przydatne — wrócimy do tego później. Definicje funkcji wewnątrz klasy zwykle mają specyficzną formę listy argumentów, podyktowaną konwencjami wywoływania metod — ponownie, zostanie to wyjaśnione później.

Po wejściu w definicję klasy tworzona jest nowa przestrzeń nazw i używana jako zakres lokalny — a zatem wszystkie przypisania do zmiennych lokalnych trafiają do tej nowej przestrzeni nazw. W szczególności, definicje funkcji wiążą nazwę nowej funkcji w tej przestrzeni nazw.

Kiedy definicja klasy jest opuszczana normalnie (przez koniec), tworzony jest *obiekt klasy*. Jest to w zasadzie opakowanie wokół zawartości przestrzeni nazw utworzonej przez definicję klasy; dowiemy się więcej o obiektach klas w następnej sekcji. Oryginalny zakres lokalny (ten, który obowiązywał tuż przed wprowadzeniem definicji klasy) zostaje przywrócony, a obiekt klasy jest powiązany z nazwą klasy podaną w nagłówku definicja klasy (`ClassName` w przykładzie).

9.3.2 Class Objects

Class objects support two kinds of operations: attribute references and instantiation.

Attribute references use the standard syntax used for all attribute references in Python: `obj.name`. Valid attribute names are all the names that were in the class's namespace when the class object was created. So, if the class definition looked like this:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

then `MyClass.i` and `MyClass.f` are valid attribute references, returning an integer and a function object, respectively. Class attributes can also be assigned to, so you can change the value of `MyClass.i` by assignment. `__doc__` is also a valid attribute, returning the docstring belonging to the class: `"A simple example class"`.

Class *instantiation* uses function notation. Just pretend that the class object is a parameterless function that returns a new instance of the class. For example (assuming the above class):

```
x = MyClass()
```

creates a new *instance* of the class and assigns this object to the local variable `x`.

The instantiation operation („calling" a class object) creates an empty object. Many classes like to create objects with instances customized to a specific initial state. Therefore a class may define a special method named `__init__()`, like this:

```
def __init__(self):
    self.data = []
```

When a class defines an `__init__()` method, class instantiation automatically invokes `__init__()` for the newly created class instance. So in this example, a new, initialized instance can be obtained by:

```
x = MyClass()
```

Of course, the `__init__()` method may have arguments for greater flexibility. In that case, arguments given to the class instantiation operator are passed on to `__init__()`. For example,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Instance Objects

Now what can we do with instance objects? The only operations understood by instance objects are attribute references. There are two kinds of valid attribute names: data attributes and methods.

data attributes correspond to „instance variables” in Smalltalk, and to „data members” in C++. Data attributes need not be declared; like local variables, they spring into existence when they are first assigned to. For example, if `x` is the instance of `MyClass` created above, the following piece of code will print the value `16`, without leaving a trace:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print(x.counter)
del x.counter
```

The other kind of instance attribute reference is a *method*. A method is a function that „belongs to” an object.

Valid method names of an instance object depend on its class. By definition, all attributes of a class that are function objects define corresponding methods of its instances. So in our example, `x.f` is a valid method reference, since `MyClass.f` is a function, but `x.i` is not, since `MyClass.i` is not. But `x.f` is not the same thing as `MyClass.f` — it is a *method object*, not a function object.

9.3.4 Method Objects

Usually, a method is called right after it is bound:

```
x.f()
```

In the `MyClass` example, this will return the string `'hello world'`. However, it is not necessary to call a method right away: `x.f` is a method object, and can be stored away and called at a later time. For example:

```
xf = x.f
while True:
    print(xf())
```

will continue to print `hello world` until the end of time.

What exactly happens when a method is called? You may have noticed that `x.f()` was called without an argument above, even though the function definition for `f()` specified an argument. What happened to the argument? Surely

Python raises an exception when a function that requires an argument is called without any — even if the argument isn't actually used…

Actually, you may have guessed the answer: the special thing about methods is that the instance object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's instance object before the first argument.

In general, methods work as follows. When a non-data attribute of an instance is referenced, the instance's class is searched. If the name denotes a valid class attribute that is a function object, references to both the instance object and the function object are packed into a method object. When the method object is called with an argument list, a new argument list is constructed from the instance object and the argument list, and the function object is called with this new argument list.

9.3.5 Class and Instance Variables

Generally speaking, instance variables are for data unique to each instance and class variables are for attributes and methods shared by all instances of the class:

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

As discussed in *Kilka słów o nazwach i obiektach*, shared data can have possibly surprising effects with involving *mutable* objects such as lists and dictionaries. For example, the *tricks* list in the following code should not be used as a class variable because just a single list would be shared by all *Dog* instances:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Correct design of the class should use an instance variable instead:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 Random Remarks

If the same attribute name occurs in both an instance and in a class, then attribute lookup prioritizes the instance:

```
>>> class Warehouse:
...     purpose = 'storage'
...     region = 'west'
...
>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

Data attributes may be referenced by methods as well as by ordinary users („clients”) of an object. In other words, classes are not usable to implement pure abstract data types. In fact, nothing in Python makes it possible to enforce data hiding — it is all based upon convention. (On the other hand, the Python implementation, written in C, can completely hide implementation details and control access to an object if necessary; this can be used by extensions to Python written in C.)

Clients should use data attributes with care — clients may mess up invariants maintained by the methods by stamping on their data attributes. Note that clients may add data attributes of their own to an instance object without affecting the validity of the methods, as long as name conflicts are avoided — again, a naming convention can save a lot of headaches here.

There is no shorthand for referencing data attributes (or other methods!) from within methods. I find that this actually increases the readability of methods: there is no chance of confusing local variables and instance variables when glancing through a method.

Often, the first argument of a method is called `self`. This is nothing more than a convention: the name `self` has absolutely no special meaning to Python. Note, however, that by not following the convention your code may be less readable to other Python programmers, and it is also conceivable that a *class browser* program might be written that relies upon such a convention.

Any function object that is a class attribute defines a method for instances of that class. It is not necessary that the function definition is textually enclosed in the class definition: assigning a function object to a local variable in the class is also ok. For example:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Now `f`, `g` and `h` are all attributes of class `C` that refer to function objects, and consequently they are all methods of instances of `C` — `h` being exactly equivalent to `g`. Note that this practice usually only serves to confuse the reader of a program.

Methods may call other methods by using method attributes of the `self` argument:

```
class Bag:
    def __init__(self):
        self.data = []

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Methods may reference global names in the same way as ordinary functions. The global scope associated with a method is the module containing its definition. (A class is never used as a global scope.) While one rarely encounters a good reason for using global data in a method, there are many legitimate uses of the global scope: for one thing, functions and modules imported into the global scope can be used by methods, as well as functions and classes defined in it. Usually, the class containing the method is itself defined in this global scope, and in the next section we'll find some good reasons why a method would want to reference its own class.

Each value is an object, and therefore has a *class* (also called its *type*). It is stored as `object.__class__`.

9.5 Inheritance

Of course, a language feature would not be worthy of the name „class" without supporting inheritance. The syntax for a derived class definition looks like this:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

The name `BaseClassName` must be defined in a namespace accessible from the scope containing the derived class definition. In place of a base class name, other arbitrary expressions are also allowed. This can be useful, for example, when the base class is defined in another module:

```
class DerivedClassName(modname.BaseClassName):
```

Execution of a derived class definition proceeds the same as for a base class. When the class object is constructed, the base class is remembered. This is used for resolving attribute references: if a requested attribute is not found in the

class, the search proceeds to look in the base class. This rule is applied recursively if the base class itself is derived from some other class.

There's nothing special about instantiation of derived classes: `DerivedClassName()` creates a new instance of the class. Method references are resolved as follows: the corresponding class attribute is searched, descending down the chain of base classes if necessary, and the method reference is valid if this yields a function object.

Derived classes may override methods of their base classes. Because methods have no special privileges when calling other methods of the same object, a method of a base class that calls another method defined in the same base class may end up calling a method of a derived class that overrides it. (For C++ programmers: all methods in Python are effectively `virtual`.)

An overriding method in a derived class may in fact want to extend rather than simply replace the base class method of the same name. There is a simple way to call the base class method directly: just call `BaseClassName.methodname(self, arguments)`. This is occasionally useful to clients as well. (Note that this only works if the base class is accessible as `BaseClassName` in the global scope.)

Python has two built-in functions that work with inheritance:

- Use `isinstance()` to check an instance's type: `isinstance(obj, int)` will be `True` only if `obj.__class__` is `int` or some class derived from `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(float, int)` is `False` since `float` is not a subclass of `int`.

9.5.1 Multiple Inheritance

Python supports a form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For most purposes, in the simplest cases, you can think of the search for attributes inherited from a parent class as depth-first, left-to-right, not searching twice in the same class where there is an overlap in the hierarchy. Thus, if an attribute is not found in `DerivedClassName`, it is searched for in `Base1`, then (recursively) in the base classes of `Base1`, and if it was not found there, it was searched for in `Base2`, and so on.

In fact, it is slightly more complex than that; the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

Dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see python_2.3_mro.

9.6 Private Variables

„Private" instance variables that cannot be accessed except from inside an object don't exist in Python. However, there is a convention that is followed by most Python code: a name prefixed with an underscore (e.g. `_spam`) should be treated as a non-public part of the API (whether it is a function, a method or a data member). It should be considered an implementation detail and subject to change without notice.

Since there is a valid use-case for class-private members (namely to avoid name clashes of names with names defined by subclasses), there is limited support for such a mechanism, called *name mangling*. Any identifier of the form `__spam` (at least two leading underscores, at most one trailing underscore) is textually replaced with `_classname__spam`, where `classname` is the current class name with leading underscore(s) stripped. This mangling is done without regard to the syntactic position of the identifier, as long as it occurs within the definition of a class.

Zobacz także

The private name mangling specifications for details and special cases.

Name mangling is helpful for letting subclasses override methods without breaking intraclass method calls. For example:

```
class Mapping:
    def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)

    __update = update   # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

The above example would work even if `MappingSubclass` were to introduce a `__update` identifier since it is replaced with `_Mapping__update` in the `Mapping` class and `_MappingSubclass__update` in the `MappingSubclass` class respectively.

Note that the mangling rules are designed mostly to avoid accidents; it still is possible to access or modify a variable that is considered private. This can even be useful in special circumstances, such as in the debugger.

Notice that code passed to `exec()` or `eval()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

9.7 Odds and Ends

Sometimes it is useful to have a data type similar to the Pascal „record” or C „struct”, bundling together a few named data items. The idiomatic approach is to use `dataclasses` for this purpose:

```
from dataclasses import dataclass

@dataclass
class Employee:
    name: str
    dept: str
    salary: int
```

```
>>> john = Employee('john', 'computer lab', 1000)
>>> john.dept
'computer lab'
>>> john.salary
1000
```

A piece of Python code that expects a particular abstract data type can often be passed a class that emulates the methods of that data type instead. For instance, if you have a function that formats some data from a file object, you can define a class with methods `read()` and `readline()` that get the data from a string buffer instead, and pass it as an argument.

Instance method objects have attributes, too: `m.__self__` is the instance object with the method `m()`, and `m.__func__` is the function object corresponding to the method.

9.8 Iterators

By now you have probably noticed that most container objects can be looped over using a `for` statement:

```
for element in [1, 2, 3]:
    print(element)
for element in (1, 2, 3):
    print(element)
for key in {'one':1, 'two':2}:
    print(key)
for char in "123":
    print(char)
for line in open("myfile.txt"):
    print(line, end='')
```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `__next__()` which accesses elements in the container one at a time. When there are no more elements, `__next__()` raises a `StopIteration` exception which tells the `for` loop to terminate. You can call the `__next__()` method using the `next()` built-in function; this example shows how it all works:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<str_iterator object at 0x10c90e650>
>>> next(it)
'a'
>>> next(it)
'b'
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `__next__()` method. If the class defines `__next__()`, then `__iter__()` can just return `self`:

```
class Reverse:
    """Iterator for looping over a sequence backwards."""
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
    def __init__(self, data):
        self.data = data
        self.index = len(data)

    def __iter__(self):
        return self

    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
```

```
>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print(char)
...
m
a
p
s
```

9.9 Generators

Generators are a simple and powerful tool for creating iterators. They are written like regular functions but use the `yield` statement whenever they want to return data. Each time `next()` is called on it, the generator resumes where it left off (it remembers all the data values and which statement was last executed). An example shows that generators can be trivially easy to create:

```
def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
```

```
>>> for char in reverse('golf'):
...     print(char)
...
f
l
o
g
```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `__next__()` methods are created automatically.

Another key feature is that the local variables and execution state are automatically saved between calls. This made the function easier to write and much more clear than an approach using instance variables like `self.index` and `self.data`.

In addition to automatic method creation and saving program state, when generators terminate, they automatically raise `StopIteration`. In combination, these features make it easy to create iterators with no more effort than writing a regular function.

9.10 Generator Expressions

Some simple generators can be coded succinctly as expressions using a syntax similar to list comprehensions but with parentheses instead of square brackets. These expressions are designed for situations where the generator is used right away by an enclosing function. Generator expressions are more compact but less versatile than full generator definitions and tend to be more memory friendly than equivalent list comprehensions.

Examples:

```
>>> sum(i*i for i in range(10))                 # sum of squares
285

>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # dot product
260

>>> unique_words = set(word for line in page  for word in line.split())

>>> valedictorian = max((student.gpa, student.name) for student in graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

ROZDZIAŁ 10

Krótka wycieczka po Bibliotece Standardowej

10.1 Interfejs Systemu Operacyjnego

Moduł `os` udostępnia wiele funkcji do interakcji z systemem operacyjnym:

```
>>> import os
>>> os.getcwd()      # Zwróć aktualny folder roboczy
'C:\\Python313'
>>> os.chdir('/server/accesslogs')   # Zmień aktualny folder roboczy
>>> os.system('mkdir today')   # Uruchom komendę mkdir w powłoce
0
```

Upewnij się, że używasz stylu `import os`, a nie `from os import *`. W tym drugim przypadku funkcja `os.open()` przesłoni wbudowaną funkcję `open()`, która działa w skrajnie inny sposób.

Wbudowane funkcje `dir()` i `help()`, są przydatne jako interaktywne pomoce do pracy z dużymi modułami, takimi jak `os`:

```
>>> import os
>>> dir(os)
<zwraca listę wszystkich funkcji modułu>
>>> help(os)
<zwraca pełną instrukcję utworzoną z dokumentacji modułu>
```

Do codziennego zarządzania plikami i katalogami, moduł `shutil` zapewnia interfejs wyższego poziomu, który jest łatwiejszy w użyciu:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 Symbole wieloznaczne plików

Moduł `glob` udostępnia funkcję do tworzenia list plików korzystając z wyszukiwania na symbolach wieloznacznych:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Argumenty linii polecenia

Typowe skrypty narzędziowe często wymagają przetwarzania argumentów wiersza poleceń. Te argumenty są przechowywane w atrybucie *argv* modułu `sys` jako lista. Na przykład, weźmy następujący plik `demo.py`:

```
# Plik demo.py
import sys
print(sys.argv)
```

Oto wynik uruchomienia `python demo.py one two three` w wierszu poleceń:

```
['demo.py', 'one', 'two', 'three']
```

Moduł `argparse` zapewnia bardziej wyrafinowane mechanizmy do przetwarzania argumentów wiersza poleceń. Poniższy skrypt wyodrębnia jedną lub więcej nazw plików i opcjonalną liczbę linii do wyświetlenia:

```
import argparse

parser = argparse.ArgumentParser(
    prog='top',
    description='Wypisz początkowe linie każdego z plików')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

Po uruchomieniu w wierszu poleceń `python top.py --lines=5 alpha.txt beta.txt`, skrypt ustawia `args.lines` na `5` i `args.filenames` na `['alpha.txt', 'beta.txt']`.

10.4 Przekierowanie wyjścia błędu i zakończenie programu

Moduł `sys` ma również atrybuty dla *stdin*, *stdout* i *stderr* (standardowe wejście, standardowe wyjście i standardowe wyjście błędu). To ostatnie jest przydatne do emitowania ostrzeżeń i komunikatów o błędach, aby były widoczne nawet wtedy, gdy *stdout* został przekierowany:

```
>>> sys.stderr.write('Ostrzeżenie, nie odnaleziono pliku rejestru, tworzę nowy␣
↪plik\n')
Ostrzeżenie, nie odnaleziono pliku rejestru, tworzę nowy plik
```

Najbardziej bezpośrednim sposobem na zakończenie skryptu jest użycie `sys.exit()`.

10.5 Dopasowywanie wzorców w napisach

Moduł `re` zapewnia narzędzia wyrażeń regularnych do zaawansowanego przetwarzania napisów. W przypadku złożonych operacji dopasowywania i manipulacji na ciągach znaków, wyrażenia regularne pozwalają na zwięzłe, zoptymalizowane rozwiązania:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Gdy potrzebne są tylko proste funkcje, preferowane są funkcje standardowe napisów, ponieważ są łatwiejsze do odczytania i debugowania:

```
>>> 'herbata dla dwa'.replace('dwa', 'dwojga')
'herbata dla dwojga'
```

10.6 Funkcje matematyczne

Moduł `math` zapewnia dostęp do funkcji bazujących na standardzie języka C dla obliczeń zmiennoprzecinkowych:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Moduł `random` dostarcza narzędzi do dokonywania wyborów losowych:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(100), 10)   # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()    # random float from the interval [0.0, 1.0)
0.17970987693706186
>>> random.randrange(6)    # random integer chosen from range(6)
4
```

Moduł `statistics` oblicza podstawowe wskaźniki statystyczne (średnią, medianę, wariancję itp.) z danych liczbowych:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

Projekt SciPy <https://scipy.org> posiada wiele innych modułów służących do obliczeń numerycznych.

10.7 Dostęp do internetu

Istnieje wiele modułów umożliwiających dostęp do Internetu i przetwarzanie protokołów internetowych. Dwa z najprostszych to `urllib.request` do pobierania danych z adresów URL i `smtplib` do wysyłania poczty:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://worldtimeapi.org/api/timezone/etc/UTC.txt') as response:
...     for line in response:
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
...         line = line.decode()              # Convert bytes to a str
...         if line.startswith('datetime'):
...             print(line.rstrip())          # Remove trailing newline
...
datetime: 2022-01-01T01:36:47.689215+00:00

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Zauważ, że drugi przykład wymaga serwera pocztowego działającego lokalnie pod adresem *localhost*).

10.8 Daty i czas

Moduł `datetime` dostarcza klasy do manipulowania datami i godzinami zarówno w prosty, jak i złożony sposób. Podczas gdy obsługiwana jest arytmetyka daty i czasu, implementacja koncentruje się na wydajnym wyodrębnianiu danych w celu formatowania i manipulacji. Moduł obsługuje również obiekty uwzględniające strefę czasową

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Kompresja Danych

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` and `tarfile`.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Mierzenie wydajności

Some Python users develop a deep interest in knowing the relative performance of different approaches to the same problem. Python provides a measurement tool that answers those questions immediately.

For example, it may be tempting to use the tuple packing and unpacking feature instead of the traditional approach to swapping arguments. The `timeit` module quickly demonstrates a modest performance advantage:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrast to `timeit`'s fine level of granularity, the `profile` and `pstats` modules provide tools for identifying time critical sections in larger blocks of code.

10.11 Kontrola jakości

One approach for developing high quality software is to write tests for each function as it is developed and to run those tests frequently during the development process.

The `doctest` module provides a tool for scanning a module and validating tests embedded in a program's docstrings. Test construction is as simple as cutting-and-pasting a typical call along with its results into the docstring. This improves the documentation by providing the user with an example and it allows the doctest module to make sure the code remains true to the documentation:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print(average([20, 30, 70]))
    40.0
    """
    return sum(values) / len(values)

import doctest
doctest.testmod()   # automatically validate the embedded tests
```

The `unittest` module is not as effortless as the `doctest` module, but it allows a more comprehensive set of tests to be maintained in a separate file:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main()  # Calling from the command line invokes all tests
```

10.12 Dostarczone z bateriami

Python trzyma się filozofii „dostarczone z bateriami”. Można to najłatwiej ujrzeć w zaawansowanych możliwościach jego większych pakietów. Dla przykładu:

- The `xmlrpc.client` and `xmlrpc.server` modules make implementing remote procedure calls into an almost trivial task. Despite the modules» names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other **RFC 2822**-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the email package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.
- The `json` package provides robust support for parsing this popular data interchange format. The `csv` module supports direct reading and writing of files in Comma-Separated Value format, commonly supported by databases and spreadsheets. XML processing is supported by the `xml.etree.ElementTree`, `xml.dom` and `xml.sax` packages. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- The `sqlite3` module is a wrapper for the SQLite database library, providing a persistent database that can be updated and accessed using slightly nonstandard SQL syntax.
- Internationalization is supported by a number of modules including `gettext`, `locale`, and the `codecs` package.

ROZDZIAŁ 11

Brief Tour of the Standard Library — Part II

This second tour covers more advanced modules that support professional programming needs. These modules rarely occur in small scripts.

11.1 Output Formatting

The `reprlib` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
"{'a', 'c', 'd', 'e', 'f', 'g', ...}"
```

The `pprint` module offers more sophisticated control over printing both built-in and user defined objects in a way that is readable by the interpreter. When the result is longer than one line, the „pretty printer” adds line breaks and indentation to more clearly reveal data structure:

```
>>> import pprint
>>> t = [[[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]]
...
>>> pprint.pprint(t, width=30)
[[[['black', 'cyan'],
   'white',
   ['green', 'red']],
  [['magenta', 'yellow'],
   'blue']]]
```

The `textwrap` module formats paragraphs of text to fit a given screen width:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

The `locale` module accesses a database of culture specific data formats. The grouping attribute of locale's format function provides a direct way of formatting numbers with group separators:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()          # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format_string("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Templating

The `string` module includes a versatile `Template` class with a simplified syntax suitable for editing by end-users. This allows users to customize their applications without having to alter the application.

The format uses placeholder names formed by `$` with valid Python identifiers (alphanumeric characters and underscores). Surrounding the placeholder with braces allows it to be followed by more alphanumeric letters with no intervening spaces. Writing `$$` creates a single escaped `$`:

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

The `substitute()` method raises a `KeyError` when a placeholder is not supplied in a dictionary or a keyword argument. For mail-merge style applications, user supplied data may be incomplete and the `safe_substitute()` method may be more appropriate — it will leave placeholders unchanged if data is missing:

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Template subclasses can specify a custom delimiter. For example, a batch renaming utility for a photo browser may elect to use percent signs for placeholders such as the current date, image sequence number, or file format:

```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
...
>>> fmt = input('Enter rename style (%d-date %n-seqnum %f-format):  ')
Enter rename style (%d-date %n-seqnum %f-format):  Ashley_%n%f
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> t = BatchRename(fmt)
>>> date = time.strftime('%d%b%y')
>>> for i, filename in enumerate(photofiles):
...     base, ext = os.path.splitext(filename)
...     newname = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(filename, newname))

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Another application for templating is separating program logic from the details of multiple output formats. This makes it possible to substitute custom templates for XML files, plain text reports, and HTML web reports.

11.3 Working with Binary Data Record Layouts

The `struct` module provides `pack()` and `unpack()` functions for working with variable length binary record formats. The following example shows how to loop through header information in a ZIP file without using the `zipfile` module. Pack codes `"H"` and `"I"` represent two and four byte unsigned numbers respectively. The `"<"` indicates that they are standard size and in little-endian byte order:

```
import struct

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):                      # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    start += 16
    filename = data[start:start+filenamesize]
    start += filenamesize
    extra = data[start:start+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    start += extra_size + comp_size     # skip to the next header
```

11.4 Multi-threading

Threading is a technique for decoupling tasks which are not sequentially dependent. Threads can be used to improve the responsiveness of applications that accept user input while other tasks run in the background. A related use case is running I/O in parallel with computations in another thread.

The following code shows how the high level `threading` module can run tasks in background while the main program continues to run:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
        self.infile = infile
        self.outfile = outfile

    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print('The main program continues to run in foreground.')

background.join()    # Wait for the background task to finish
print('Main program waited until background was done.')
```

The principal challenge of multi-threaded applications is coordinating threads that share data or other resources. To that end, the threading module provides a number of synchronization primitives including locks, events, condition variables, and semaphores.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the `queue` module to feed that thread with requests from other threads. Applications using `Queue` objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5 Logging

The `logging` module offers a full featured and flexible logging system. At its simplest, log messages are sent to a file or to `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

This produces the following output:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

By default, informational and debugging messages are suppressed and the output is sent to standard error. Other output options include routing messages through email, datagrams, sockets, or to an HTTP Server. New filters can select different routing based on message priority: `DEBUG`, `INFO`, `WARNING`, `ERROR`, and `CRITICAL`.

The logging system can be configured directly from Python or can be loaded from a user editable configuration file for customized logging without altering the application.

11.6 Weak References

Python does automatic memory management (reference counting for most objects and *garbage collection* to eliminate cycles). The memory is freed shortly after the last reference to it has been eliminated.

This approach works fine for most applications but occasionally there is a need to track objects only as long as they are being used by something else. Unfortunately, just tracking them creates a reference that makes them permanent.

The `weakref` module provides tools for tracking objects without creating a reference. When the object is no longer needed, it is automatically removed from a weakref table and a callback is triggered for weakref objects. Typical applications include caching objects that are expensive to create:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                    # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a             # does not create a reference
>>> d['primary']                 # fetch the object if it is still alive
10
>>> del a                        # remove the one reference
>>> gc.collect()                 # run garbage collection right away
0
>>> d['primary']                 # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                # entry was automatically removed
  File "C:/python313/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Tools for Working with Lists

Many data structure needs can be met with the built-in list type. However, sometimes there is a need for alternative implementations with different performance trade-offs.

The `array` module provides an `array` object that is like a list that stores only homogeneous data and stores it more compactly. The following example shows an array of numbers stored as two byte unsigned binary numbers (typecode `"H"`) rather than the usual 16 bytes per entry for regular lists of Python int objects:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

The `collections` module provides a `deque` object that is like a list with faster appends and pops from the left side but slower lookups in the middle. These objects are well suited for implementing queues and breadth first tree searches:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print("Handling", d.popleft())
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
        if is_goal(m):
            return m
        unsearched.append(m)
```

In addition to alternative list implementations, the library also offers other tools such as the `bisect` module with functions for manipulating sorted lists:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

The `heapq` module provides functions for implementing heaps based on regular lists. The lowest valued entry is always kept at position zero. This is useful for applications which repeatedly access the smallest element but do not want to run a full list sort:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data)                      # rearrange the list into heap order
>>> heappush(data, -5)                 # add a new entry
>>> [heappop(data) for i in range(3)]  # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Decimal Floating-Point Arithmetic

The `decimal` module offers a `Decimal` datatype for decimal floating-point arithmetic. Compared to the built-in `float` implementation of binary floating point, the class is especially helpful for

- financial applications and other uses which require exact decimal representation,
- control over precision,
- control over rounding to meet legal or regulatory requirements,
- tracking of significant decimal places, or
- applications where the user expects the results to match calculations done by hand.

For example, calculating a 5% tax on a 70 cent phone charge gives different results in decimal floating point and binary floating point. The difference becomes significant if the results are rounded to the nearest cent:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

The `Decimal` result keeps a trailing zero, automatically inferring four place significance from multiplicands with two place significance. Decimal reproduces mathematics as done by hand and avoids issues that can arise when binary floating point cannot exactly represent decimal quantities.

Exact representation enables the `Decimal` class to perform modulo calculations and equality tests that are unsuitable for binary floating point:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
```

The `decimal` module provides arithmetic with as much precision as needed:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

ROZDZIAŁ 12

Środowiska wirtualne i pakiety

12.1 Wprowadzenie

Aplikacje Pythonowe często używają pakietów oraz modułów które nie są dołączone do standardowej biblioteki. Potrzebują one czasami konkretnej wersji biblioteki, ponieważ mogą wymagać naprawionego określonego błędu lub mogą być napisane przy użyciu przestarzałej wersji interfejsu biblioteki.

Oznacza to, że może nie być możliwe aby jedna instalacja Pythona spełniała wymagania każdej aplikacji. Jeżeli aplikacja A potrzebuje wersji 1.0 danego modułu, a aplikacja B potrzebuje wersji 2.0, wtedy wymagania są sprzeczne i zainstalowanie wersji 1.0 lub 2.0 uniemożliwi uruchomienie którejś z aplikacji.

Rozwiązaniem tego problemu jest stworzenie *środowiska wirtualnego*, samodzielnej struktury katalogów, które zawierają instalację Pythona dla określonej wersji oraz dodatkowych pakietów.

Różne aplikacje mogą wtedy używać różnych środowisk wirtualnych. Aby rozwiązać wcześniej przytoczony przykład konfliktujących wymagań, aplikacja A może mieć swoje własne środowisko wirtualne z zainstalowaną wersją 1.0 w momencie gdy aplikacja B ma inne środowisko wirtualne z zainstalowaną wersją 2.0. Jeżeli w pewnym momencie aplikacja B będzie wymagała zaktualizowania modułu do wersji 3.0, nie wpłynie to na środowisko aplikacji A.

12.2 Tworzenie Środowisk Wirtualnych

Moduł używany do tworzenia i zarządzania środowiskami wirtualnymi nazywa się `venv`. `venv` zainstaluje wersję Pythona, z której polecenie zostało uruchomione (zgodnie z opcją `--version`). Na przykład wykonanie polecenia z `python3.12` zainstaluje wersję 3.12.

Aby stworzyć środowisko wirtualne, wybierz katalog, w którym chcesz je umieścić i uruchom moduł `venv` jako skrypt ze ścieżką do katalogu:

```
python -m venv tutorial-env
```

Spowoduje to utworzenie katalogu `tutorial-env`, jeśli nie istnieje, a także utworzy w nim katalogi zawierające kopię interpretera Pythona i różne pliki pomocnicze.

Popularną lokalizacją katalogu dla środowiska wirtualnego jest `.venv`. Nazwa ta sprawia, że katalog jest zwykle ukryty, a więc nie wchodzi w drogę, jednocześnie nadając mu nazwę, która wyjaśnia, po co dany katalog istnieje. Zapobiega to również kolizji z plikami `.env` definicji zmiennych środowiskowych, które są obsługiwane przez niektóre narzędzia.

Po utworzeniu środowiska wirtualnego możesz go aktywować.

Na systemie Windows, uruchom:

```
tutorial-env\Scripts\activate
```

Na systemie Unix albo MacOS, uruchom:

```
source tutorial-env/bin/activate
```

(Ten skrypt jest napisany dla powłoki bash. Jeżeli używasz powłok **csh** albo **fish**, istnieją skrypty `activate.csh` i `activate.fish` których powinieneś użyć.)

Uruchomienie środowiska wirtualnego zmieni wygląd powłoki tak aby pokazywała z którego środowiska wirtualnego aktualnie korzystasz, oraz zmieni parametry środowiska tak że uruchomienie `python` wywoła określoną wersję i instalacje Pythona. Dla przykładu:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May  6 2016, 10:59:36)
  ...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
'~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

Aby deaktywować środowisko wirtualne, wpisz:

```
deactivate
```

w terminalu.

12.3 Zarządzanie pakietami używając pip

Możesz instalować, aktualizować oraz usuwać pakiety korzystając z pragramu nazywanego **pip**. Domyślnie `pip` zainstaluje pakiety z Python Package Index. Możesz przeglądać dostępne tam pakiety wchodząc na ich stronę internetową.

`pip` ma kilka podkomend: „install”, „uninstall”, „freeze” itp. (Zapoznaj się z przewodnikiem installing-index, aby uzyskać pełną dokumentację dotyczącą `pip`.)

Możesz zainstalować najnowszą wersję pakietu, podając jego nazwę:

```
(tutorial-env) $ python -m pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Możesz również zainstalować konkretną wersję pakietu, podajac jego nazwę z dopiskiem == oraz numerem wersji:

```
(tutorial-env) $ python -m pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Jeśli ponownie uruchomisz to polecenie, `pip` zauważy, że żądana wersja jest już zainstalowana i nic nie zrobi. Możesz podać inny numer wersji, aby pobrać tę wersję, lub uruchomić `python -m pip install --upgrade`, aby zaktualizować pakiet do najnowszej wersji:

```
(tutorial-env) $ python -m pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

Wpisanie `python -m pip uninstall`, a następnie jednej lub więcej nazw pakietów, usunie te pakiety ze środowiska wirtualnego.

`python -m pip show` wyświetli informacje na temat określonego pakietu:

```
(tutorial-env) $ python -m pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`python -m pip list` wyświetli listę wszystkich pakietów zainstalowanych w środowisku wirtualnym:

```
(tutorial-env) $ python -m pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`python -m pip freeze` wygeneruje podobną listę zainstalowanych pakietów, ale dane wyjściowe będą miały format, którego oczekuje `python -m pip install`. Powszechną konwencją jest umieszczenie tej listy w pliku `requirements.txt`:

```
(tutorial-env) $ python -m pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

Plik `requirements.txt` można następnie przekazać do systemu kontroli wersji i wysłać jako część aplikacji. Użytkownicy mogą następnie zainstalować wszystkie niezbędne pakiety za pomocą `install -r`:

```
(tutorial-env) $ python -m pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
  ...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
  ...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
  ...
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` ma o wiele więcej opcji. Zapoznaj się z przewodnikiem installing-index, aby uzyskać pełną dokumentację dotyczącą `pip`. Kiedy napiszesz pakiet i będziesz chciał udostępnić go w indeksie pakietów Pythona, zapoznaj się z Python packaging user guide.

ROZDZIAŁ 13

Co dalej?

Przeczytanie tego samouczka prawdopodobnie zwiększyło twoje zainteresowanie używaniem Pythona — powinieneś chcieć zastosować Pythona do rozwiązywania swoich rzeczywistych problemów. Gdzie powinieneś się udać, aby dowiedzieć się więcej?

Ten tutorial jest częścią zbioru dokumentacji Pythona. Inne dokumenty zawarte w tym zbiorze to:

- library-index:

  Powinieneś przejrzeć ten podręcznik, który zawiera kompletny (choć zwięzły) materiał referencyjny na temat typów, funkcji i modułów w standardowej bibliotece. Standardowa dystrybucja Pythona zawiera *dużo* dodatkowego kodu. Istnieją moduły do czytania skrzynek pocztowych Unix, pobierania dokumentów przez HTTP, generowania liczb losowych, analizowania opcji wiersza poleceń, pisania programów CGI, kompresji danych i wielu innych zadań. Przejrzenie informacji o bibliotece da ci wyobrażenie o tym, co jest dostępne.

- installing-index tłumaczy w jaki sposób instalować dodatkowe moduły napisane przez innych użytkowników Pythona.

- reference-index: Szegółowe wyjaśnienie składni i semantyki Pythona. To ciężka lektura, ale bardzo przydatna jako kompletny przewodnik po samym języku.

Więcej zasobów Pythona:

- https://www.python.org: Główna strona internetowa poświęcona językowi Python. Zawiera kod, dokumentację oraz odnośniki do innych stron związanych z Python-em.

- https://docs.python.org: Szybki dostęp do dokumentacji Pythona.

- https://pypi.org: Indeks pakietów Pythona (Python Package Index), wcześniej nazywany także Sklep z Serem (Cheese Shop)[1], to indeks utworzonych przez użytkowników modułów Pythona, które można pobrać. Gdy zaczniesz udostępniać kod, możesz go tutaj zarejestrować, aby inni mogli go znaleźć.

- https://code.activestate.com/recipes/langs/python/: The Python Cookbook to pokaźny zbiór przykładów kodu, większych modułów i przydatnych skryptów po angielsku. Szczególnie godne uwagi fragmenty zostały zebrane i przetłumaczone w książce zatytułowanej „Python. Receptury” (Wydawnicto Helion/O'Reilly, ISBN: 978-83--246-8180-8).

- https://pyvideo.org zbiera linki do filmów związanych z Pythonem z konferencji oraz spotkań grup użytkowników.

[1] „Sklep z Serami” (Cheese Shop) to skecz Monty Pythona: klient wchodzi do sklepu z serami, ale bez względu na to, o jaki ser prosi, sprzedawca mówi, że go nie ma.

- https://scipy.org: Projekt „Scientific Python” zawiera moduły do szybkich obliczeń oraz modyfikacji tablicowych oraz wiele pakietów do takich rzeczy jak algebra liniowa, transformata Fouriera, rozwiązania nieliniowe, rozkłady liczb losowych, analiza statystyczna i tym podobne.

Pytania związane z Pythonem i raporty o problemach można przesyłać na grupę dyskusyjną `comp.lang.python` lub na listę mailingową python-list@python.org. Grupa dyskusyjna i lista mailingowa są połączone, więc wiadomości wysłane na jedną z nich będą automatycznie przekazywane na drugą. Codziennie pojawiają się tam setki postów z pytaniami (i odpowiedziami), sugestiami nowych funkcji i zapowiedziami nowych modułów. Archiwa list mailingowych są dostępne pod adresem https://mail.python.org/pipermail/.

Przed wysłaniem zapytania, warto sprawdzić listę Często zadawanych pytań (zwaną również FAQ). FAQ odpowiada na wiele często pojawiających się pytań i może już zawierać rozwiązanie twojego problemu.

ROZDZIAŁ 14

Interaktywna edycja danych wejściowych oraz podstawianie z historii

Niektóre wersje interpretera Python, podobnie jak ułatwienia znajdowane w powłokach Korn i GNU Bash, wspierają edycję bieżącej linii danych wejściowych oraz podstawianie z historii. Do implementacji użyto biblioteki GNU Readline, która wspiera różne style edytowania. Biblioteka ta ma własną dokumentację, której nie będziemy tutaj powielać.

14.1 Uzupełnianie z tabulatorem oraz edycja historii

Uzupełnianie zmiennych i nazw modułów jest włączane automatycznie podczas uruchamiania interpretera w taki sposób, że klawisz `Tab` wywołuje funkcję uzupełniania; sprawdza nazwy instrukcji Pythona, bieżące zmienne lokalne oraz dostępne nazwy modułów. Dla wyrażeń kropkowanych takich jak `string.a`, ewaluuje wyrażenie do ostaniej `'.'`, a następnie sugeruje uzupełnienie z atrybutów obiektu wynikowego. Zauważ, że może to wykonać zdefiniowany w aplikacji kod, jeżeli obiekt z metodą `__getattr__()` jest częścią wyrażenia. Domyślna konfiguracja zapisuje także twoją historię do pliku `.python_history` w katalogu użytkownika. Historia będzie ponownie dostępna podczas następnej interaktywnej sesji interpretera.

14.2 Alternatywy dla interaktywnego interpretera

To ułatwienie jest olbrzymim krokiem naprzód w porównaniu z wcześniejszymi wersjami interpretera, jednakże, niektóre życzenia pozostają: byłoby dobrze, gdyby prawidłowe wcięcia były podpowiadane w kontynuowanych liniach (analizator składni wie, jeżeli token wcięcia jest dalej wymagany). Mechanizm uzupełniania mógłby używać tabeli symboli interpretera. Polecenie, aby sprawdzić (lub nawet podpowiedzieć) pasujący nawias, cudzysłów itp. także byłoby użyteczne.

Jednym z alternatywnych udoskonalonych interaktywnych interpreterów dostępnych od pewnego czasu jest IPython, który posiada funkcję uzupełniania tabulatorem, eksplorację obiektów oraz zaawansowane zarządzanie historią. Można go także wszechstronnie dostosowywać oraz osadzać w innych aplikacjach. Innym podobnym udoskonalonym środowiskiem interaktywnego interpretera jest bpython.

ROZDZIAŁ 15

Arytmetyka liczb zmiennoprzecinkowych: problemy i ograniczenia

Liczby zmiennoprzecinkowe są reprezentowane w komputerze jako ułamki o podstawie 2 (binarne). Na przykład **dziesiętny** ułamek `0.625` ma wartość 6/10 + 2/100 + 5/1000 i analogicznie **binarny** ułamek `0.101` ma wartość 1/2 + 0/4 + 1/8. Te dwa ułamki mają identyczne wartości, a jedyną prawdziwą różnicą jest to, że pierwszy jest zapisany w notacji ułamkowej o podstawie 10, a drugi o podstawie 2.

Niestety, większości ułamków dziesiętnych nie można przedstawić dokładnie jako ułamków binarnych. Konsekwencją jest to, że ogólnie wprowadzane dziesiętne liczby zmiennoprzecinkowe są jedynie przybliżane przez binarne liczby zmiennoprzecinkowe faktycznie przechowywane w maszynie.

Problem jest łatwiejszy do zrozumienia na początku przy podstawie 10. Rozważ ułamek 1/3. Możesz go przybliżyć jako ułamek o podstawie 10:

```
0.3
```

albo lepiej:

```
0.33
```

albo lepiej:

```
0.333
```

i tak dalej. Bez względu na to, ile cyfr jesteś w stanie zapisać, wynik nigdy nie będzie dokładnie 1/3, ale będzie coraz lepszym przybliżeniem 1/3.

W ten sam sposób, bez względu na to, ile cyfr o podstawie 2 chcesz użyć, wartość dziesiętna 0,1 nie może być dokładnie przedstawiona jako ułamek o podstawie 2. W podstawie 2, 1/10 to ułamek okresowy

```
0.0001100110011001100110011001100110011001100110011...
```

Zatrzymaj się na dowolnej skończonej liczbie bitów, a otrzymasz przybliżenie. Na większości dzisiejszych maszyn liczby zmiennoprzecinkowe są aproksymowane przy użyciu ułamka binarnego z licznikiem wykorzystującym pierwsze 53 bity, zaczynając od najbardziej znaczącego bitu i mianownikiem jako potęgą dwójki. W przypadku 1/10 ułamek binarny jest równy `3602879701896397 / 2 ** 55` i jest zbliżony do prawdziwej wartości 1/10, ale nie do końca jej równy.

Wielu użytkowników nie jest świadomych przybliżenia ze względu na sposób wyświetlania wartości. Python wypisuje tylko przybliżenie dziesiętne do prawdziwej wartości dziesiętnej przybliżenia binarnego zapisanego przez maszynę.

Na większości maszyn, gdyby Python miał wydrukować prawdziwą wartość dziesiętną przybliżenia binarnego zapisanego dla 0,1 musiałby wyświetlić:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

To więcej cyfr, niż większość ludzi uważa za przydatne, więc Python utrzymuje liczbę cyfr tak by były one do opanowania, wyświetlając zamiast tego zaokrągloną wartość:

```
>>> 1 / 10
0.1
```

Pamiętaj tylko, że chociaż wydrukowany wynik wygląda jak dokładna wartość 1/10, rzeczywista zapisana wartość to najbliższa reprezentatywna część binarna.

Co ciekawe, istnieje wiele różnych liczb dziesiętnych, które mają ten sam najbliższy przybliżony ułamek binarny. Na przykład liczby `0.1`, `0.10000000000000001` i `0.1000000000000000055511151231257827021181583404541015625` są wszystkie przybliżone przez `3602879701896397 / 2 ** 55`. Ponieważ wszystkie te wartości dziesiętne mają to samo przybliżenie, każda z nich może zostać wyświetlona przy jednoczesnym zachowaniu niezmiennika `eval(repr(x)) == x`.

W przeszłości, interaktywny prompt oraz wbudowana funkcja Pythona `repr()` wybierały tę z 17 cyframi znaczącymi, `0.10000000000000001`. Począwszy od Pythona 3.1, Python (w większości systemów) może teraz wybrać najkrótszy z nich i po prostu wyświetlić `0.1`.

Zauważ, że leży to w samej naturze binarnej liczby zmiennoprzecinkowej: nie jest to błąd w Pythonie ani w twoim kodzie. Zobaczysz to samo we wszystkich językach obsługujących arytmetykę zmiennoprzecinkową twojego sprzętu (chociaż niektóre języki mogą nie *wyświetlać* różnicy domyślnie lub we wszystkich trybach wyjściowych).

Aby uzyskać przyjemniejszy wynik, możesz użyć formatowania ciągu znaków w celu uzyskania ograniczonej liczby cyfr znaczących:

```
>>> format(math.pi, '.12g')  # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')   # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

Ważne jest, aby zdać sobie sprawę, że w rzeczywistości jest to złudzenie: po prostu zaokrąglasz *wyświetlanie* prawdziwej wartości zapisanej w komputerze.

Jedna iluzja może zrodzić kolejną. Na przykład, z powodu że 0,1 nie jest dokładnie 1/10, zsumowanie trzech wartości 0,1 może nie dać dokładnie 0,3:

```
>>> 0.1 + 0.1 + 0.1 == 0.3
False
```

Ponadto, ponieważ 0,1 nie może zbliżyć się do dokładnej wartości 1/10, a 0,3 nie może zbliżyć się do dokładnej wartości 3/10, to wstępne zaokrąglenie za pomocą funkcji `round()` nie może pomóc:

```
>>> round(0.1, 1) + round(0.1, 1) + round(0.1, 1) == round(0.3, 1)
False
```

Chociaż liczb nie można już bardziej przybliżyć do ich dokładnych wartości, funkcja `math.isclose()` może być przydatna do porównywania niedokładnych wartości:

```
>>> math.isclose(0.1 + 0.1 + 0.1, 0.3)
True
```

Alternatywnie do porównania zgrubnych przybliżeń można użyć funkcji `round()`:

```
>>> round(math.pi, ndigits=2) == round(22 / 7, ndigits=2)
True
```

Binarna arytmetyka zmiennoprzecinkowa kryje w sobie wiele takich niespodzianek. Problem z „0,1” został dokładnie wyjaśniony poniżej, w sekcji „Błąd reprezentacji”. Zobacz Przykłady problemów zmiennoprzecinkowych dla przyjemnego podsumowania jak działa binarna arytmetyka zmiennoprzecinkowa i jakie rodzaje problemów często spotyka się w praktyce. Zobacz także The Perils of Floating Point dla bardziej kompletnego opisu innych typowych niespodzianek.

Jak jest tam napisane pod koniec, „nie ma łatwych odpowiedzi”. Mimo to nie należy nadmiernie uważać na liczby zmiennoprzecinkowe! Błędy w operacjach zmiennoprzecinkowych Pythona są dziedziczone z budowy komputera i na większości maszyn są rzędu nie więcej niż 1 przez 2**53 na operację. Jest to więcej niż wystarczające dla większości zadań, ale należy pamiętać, że nie jest to arytmetyka dziesiętna i że każda operacja zmiennoprzecinkowa może napotkać nowy błąd zaokrąglenia.

Chociaż istnieją przypadki skrajne, w większości przypadkowych zastosowań arytmetyki zmiennoprzecinkowej zobaczysz oczekiwany wynik, jeśli po prostu zaokrąglisz wyświetlanie końcowych wyników do oczekiwanej liczby cyfr dziesiętnych. `str()` zwykle wystarcza, lecz dla dokładniejszej kontroli możesz zobaczyć opis formatowania tekstu za pomocą metody `str.format()` w formatstrings.

W przypadkach użycia, które wymagają dokładnej reprezentacji dziesiętnej, spróbuj użyć modułu `decimal`, który implementuje arytmetykę dziesiętną odpowiednią dla aplikacji księgowych i aplikacji o wysokiej precyzji.

Inną formą wspierającą arytmetykę dokładną jest moduł `fractions` realizujący arytmetykę opartą na liczbach wymiernych (aby liczby takie jak 1/3 mogły być reprezentowane dokładnie).

Jeśli często korzystasz z operacji zmiennoprzecinkowych, powinieneś rzucić okiem na pakiet NumPy i wiele innych pakietów do operacji matematycznych i statystycznych dostarczonych przez projekt SciPy. Zobacz <https://scipy.org>.

Python udostępnia narzędzia, które mogą pomóc w tych rzadkich przypadkach, gdy naprawdę *musisz* poznać dokładną wartość liczby zmiennoprzecinkowej. Metoda `float.as_integer_ratio()` wyraża wartość float jako ułamek:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Ponieważ stosunek jest dokładny, można go użyć do bezstratnego odtworzenia oryginalnej wartości:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

Metoda `float.hex()` wyraża liczbę zmiennoprzecinkową w systemie hexadecymalnym (podstawa 16), ponownie podając dokładną wartość przechowywaną przez komputer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Ta precyzyjna reprezentacja szesnastkowa może być wykorzystana do dokładnego zrekonstruowania wartości liczby zmiennoprzecinkowej:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Ponieważ ta reprezentacja jest dokładna, przydatna jest ona do niezawodnego przenoszenia wartości między różnymi wersjami Pythona (niezależność od platformy) i wymiany danych z innymi językami obsługującymi ten sam format (takimi jak Java czy C99).

Kolejnym pomocnym narzędziem jest funkcja `sum()`, która pomaga złagodzić utratę precyzji podczas sumowania. Używa rozszerzonej precyzji dla pośrednich kroków zaokrąglania, gdy wartości są dodawane do bieżącej sumy. Może

to mieć wpływ na ogólną dokładność, aby błędy nie kumulowały się do punktu, w którym wpływają na ostateczny wynik:

```
>>> 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 == 1.0
False
>>> sum([0.1] * 10) == 1.0
True
```

The `math.fsum()` goes further and tracks all of the „lost digits" as values are added onto a running total so that the result has only a single rounding. This is slower than `sum()` but will be more accurate in uncommon cases where large magnitude inputs mostly cancel each other out leaving a final sum near zero:

```
>>> arr = [-0.10430216751806065, -266310978.67179024, 143401161448607.16,
...         -143401161400469.7, 266262841.31058735, -0.003244936839808227]
>>> float(sum(map(Fraction, arr)))   # Exact summation with single rounding
8.042173697819788e-13
>>> math.fsum(arr)                   # Single rounding
8.042173697819788e-13
>>> sum(arr)                         # Multiple roundings in extended precision
8.042178034628478e-13
>>> total = 0.0
>>> for x in arr:
...     total += x                   # Multiple roundings in standard precision
...
>>> total                            # Straight addition has no correct digits!
-0.0051575902860057365
```

15.1 Błąd reprezentacji

Ta sekcja wyjaśnia szczegółowo przykład „0,1" i pokazuje, jak samodzielnie przeprowadzić dokładną analizę takich przypadków. Zakładamy że masz podstawową znajomość binarnej reprezentacji liczb zmiennoprzecinkowych.

Błąd reprezentacji odnosi się do faktu, że niektóre (właściwie większość) ułamków dziesiętnych nie może być reprezentowane dokładnie jako ułamki binarne (o podstawie 2). Jest to główny powód, dla którego Python (lub Perl, C, C++, Java, Fortran i wiele innych) często nie wyświetla dokładnie takiej liczby dziesiętnej, jakiej oczekujesz.

Dlaczego? 1/10 nie jest dokładnie reprezentowalna jako ułamek binarny. Od co najmniej 2000 roku, prawie wszystkie dzisiejsze maszyny używają binarnej arytmetyki zmiennoprzecinkowej IEEE-754 i prawie wszystkie platformy odwzorowują liczby zmiennoprzecinkowe Pythona na „podwójną precyzję" IEEE-754 binary64. Wartości IEEE 754 binary64 zawierają 53 bity dokładności, więc na wejściu komputer stara się zamienić 0,1 na najbliższy możliwy ułamek w postaci *J*/2***N*, gdzie *J* jest liczbą całkowitą zawierającą dokładnie 53 bity. Zapisując

```
1 / 10 ~= J / (2**N)
```

jako

```
J ~= 2**N / 10
```

i pamiętając, że *J* ma dokładnie 53 bity (jest `>= 2**52` ale `< 2**53`), najlepszą wartością dla *N* jest 56:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

Oznacza to, że 56 jest jedyną wartością dla *N*, która pozostawia *J* dokładnie 53 bity. Najlepszą możliwą wartością dla *J* jest zatem zaokrąglony iloraz:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Ponieważ reszta jest większa niż połowa z 10, najlepsze przybliżenie uzyskuje się zaokrąglając w górę:

```
>>> q+1
7205759403792794
```

Dlatego najlepszym możliwym przybliżeniem do 1/10 w arytmetyce podwójnej precyzji typu IEEE 754 jest:

```
7205759403792794 / 2 ** 56
```

Podzielenie licznika i mianownika przez dwa zmniejsza ułamek do:

```
3602879701896397 / 2 ** 55
```

Zauważ, że ponieważ zaokrągliliśmy w górę, jest to w rzeczywistości trochę więcej niż 1/10; gdybyśmy nie zaokrąglili w górę, iloraz byłby nieco mniejszy niż 1/10. Ale w żadnym wypadku nie może to być *dokładnie* 1/10!

Tak więc komputer nigdy nie „widzi” 1/10: to, co widzi, to dokładny ułamek podany powyżej, najlepsze przybliżenie w standardzie podwójnej precyzji IEEE 754, jakie może uzyskać:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Jeśli pomnożymy ten ułamek przez 10**55, otrzymamy wartość z dokładnością do 55 cyfr dziesiętnych:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

co oznacza, że dokładna liczba zapisana w komputerze jest równa wartości dziesiętnej 0,1000000000000000055511151231257827021181583404541015625. Zamiast wyświetlać pełną wartość dziesiętną, wiele języków (w tym starsze wersje Pythona) zaokrągla wynik do 17 cyfr znaczących:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Moduły `fractions` oraz `decimal` ułatwiają tego typu obliczenia:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```

ROZDZIAŁ 16

Dodatek

16.1 Tryb interaktywny

There are two variants of the interactive *REPL*. The classic basic interpreter is supported on all platforms with minimal line control capabilities.

On Windows, or Unix-like systems with `curses` support, a new interactive shell is used by default. This one supports color, multiline editing, history browsing, and paste mode. To disable color, see using-on-controlling-color for details. Function keys provide some additional functionality. `F1` enters the interactive help browser `pydoc`. `F2` allows for browsing command-line history with neither output nor the »> and … prompts. `F3` enters „paste mode", which makes pasting larger blocks of code easier. Press `F3` to return to the regular prompt.

When using the new interactive shell, exit the shell by typing `exit` or `quit`. Adding call parentheses after those commands is not required.

If the new interactive shell is not desired, it can be disabled via the `PYTHON_BASIC_REPL` environment variable.

16.1.1 Przechwytywanie błędów

Gdy wystąpi błąd, interpreter wyświetla komunikat błędu i ślad stosu. W trybie interaktywnym zwraca je do wiersza polecenia; jeżeli wejście pochodzi z pliku, na wyjściu generowany jest kod wyjścia różny od zera po wyświetleniu śladu stosu. (Błędy przechwycone przez słowo kluczowe `except` w poleceniu `try` nie są błędami w tym kontekście). Niektóre błędy są bezwarunkowo krytyczne i wywołują wyjście z programu z kodem wyjścia różnym od zera; dotyczy to wewnętrznych niezgodności i - w niektórych przypadkach - przepełnienia pamięci. Wszystkie komunikaty błędów są wyświetlane na standardowym wyjściu błędów; zazwyczaj jest to normalne wyjście z wykonywania komend.

Wpisanie znaku przerwania (zazwyczaj `Control-C` lub `Delete`) w pierwszorzędny lub drugorzędny prompt anuluje tryb wpisywania i wraca do pierwszorzędnego prompta.[1] Wpisanie znaku przerwania podczas wykonywania komendy wywołuje wyjątek `KeyboardInterrupt`, który może być obsłużony instrukcją `try`.

16.1.2 Wykonywalne skrypty Pythona

W systemach uniksowych podobnych do BSD, skrypty Pythona można uczynić bezpośrednio wykonywalnymi, jak skrypty shell, przez dodanie linii

```
#!/usr/bin/env python3
```

[1] Problem z pakietem GNU Readline może w tym przeszkodzić.

(zakładając, że interpreter jest na zmiennej `PATH` użytkownika) na początku skryptu i nadając plikowi tryb wykonywalny. `#!` muszą być pierwszymi dwoma znakami pliku. Na niektórych platformach ta pierwsza linia musi kończyć się uniksowym zakończeniem linii (`'\n'`), nie windowsowym zakończeniem linii (`'\r\n'`). Zwróć uwagę, że znak kratki, czy krzyżyka, `'#'`, jest używany do rozpoczęcia komentarza w Pythonie.

Skryptowi można nadać tryb wykonywalny, lub permission, przy użyciu komendy **chmod**.

```
$ chmod +x myscript.py
```

W systemach Windows nie wyróżnia się „trybu wykonywalnego”. Instalator Pythona automatycznie powiązuje pliki `.py` z `python.exe`, żeby podwójne kliknięcie w plik Pythona uruchomiło go jako skrypt. Rozszerzeniem może być również `.pyw`. W tym przypadku okno konsoli, które normalnie się pojawia, zostanie ukryte.

16.1.3 Plik interaktywnego uruchomienia

Gdy używasz Pythona w trybie interaktywnym, często wygodne jest wykonywać jakieś standardowe komendy za każdym razem, gdy uruchamia się interpreter. Możesz to zrobić ustawiając zmienną środowiskową o nazwie `PYTHONSTARTUP` na nazwę pliku zawierającego twoje komendy uruchomieniowe. Jest to podobne do funkcji `.profile` shellów uniksowych.

Ten plik jest czytany tylko w sesjach interaktywnych, nie gdy Python czyta komendy ze skryptu i nie gdy `/dev/tty` jest podany jako źródło komend (które zachowuje się jak sesja interaktywna). Jest wykonywany w tym samym namespace'ie, w którym wykonują się interaktywne komendy, więc obiekty, które definiuje lub importuje mogą być używane bezpośrednio w interaktywnej sesji. Możesz również zmienić znaki prompt `sys.ps1` i `sys.ps2` w tym pliku.

Jeśli chcesz odczytać dodatkowy plik rozruchowy z bieżącego katalogu, możesz zaprogramować to w globalnym pliku rozruchowym używając kodu takiego jak `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. Jeśli chcesz użyć pliku rozruchowego w skrypcie, musisz to zrobić jawnie w skrypcie:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    with open(filename) as fobj:
        startup_file = fobj.read()
    exec(startup_file)
```

16.1.4 Moduły dostosowywania

Python provides two hooks to let you customize it: sitecustomize and usercustomize. To see how it works, you need first to find the location of your user site-packages directory. Start Python and run this code:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.x/site-packages'
```

Teraz możesz stworzyć plik o nazwie `usercustomize.py` w tym katalogu i umieścić w nim co tylko chcesz. Będzie to miało wpływ na każde wywołanie Pythona, chyba że zostanie on uruchomiony z opcją `-s`, aby wyłączyć automatyczny import.

sitecustomize works in the same way, but is typically created by an administrator of the computer in the global site-packages directory, and is imported before usercustomize. See the documentation of the `site` module for more details.

DODATEK A

Słownik

>>>
: The default Python prompt of the *interactive* shell. Often seen for code examples which can be executed interactively in the interpreter.

...
: Może odnosić się do:

  - The default Python prompt of the *interactive* shell when entering the code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.
  - Wbudowanej stałej `Ellipsis`.

abstrakcyjna klasa bazowa
: Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.

adnotacja
: Etykieta powiązana ze zmienną, atrybutem klasy lub parametrem funkcji lub wartością zwracaną, używana zgodnie z konwencją jako *type hint*.

  Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

  See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

argument
: A value passed to a *function* (or *method*) when calling the function. There are two kinds of argument:

  - *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, `3` and `5` are both keyword arguments in the following calls to `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an *iterable* preceded by `*`. For example, `3` and `5` are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the *parameter* glossary entry, the FAQ question on the difference between arguments and parameters, and **PEP 362**.

asynchronous context manager
An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by **PEP 492**.

asynchronous generator
A function which returns an *asynchronous generator iterator*. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.

asynchronous generator iterator
An object created by a *asynchronous generator* function.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See **PEP 492** and **PEP 525**.

asynchronous iterable
An object, that can be used in an `async for` statement. Must return an *asynchronous iterator* from its `__aiter__()` method. Introduced by **PEP 492**.

asynchronous iterator
An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__()` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by **PEP 492**.

atrybut
A value associated with an object which is usually referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

It is possible to give an object an attribute whose name is not an identifier as defined by identifiers, for example using `setattr()`, if the object allows it. Such an attribute will not be accessible using a dotted expression, and would instead need to be retrieved with `getattr()`.

awaitable
An object that can be used in an `await` expression. Can be a *coroutine* or an object with an `__await__()` method. See also **PEP 492**.

BDFL
Benevolent Dictator For Life, a.k.a. Guido van Rossum, Python's creator.

binary file
A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode (`'rb'`, `'wb'` or `'rb+'`), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also *text file* for a file object able to read and write `str` objects.

borrowed reference
In Python's C API, a borrowed reference is a reference to an object, where the code using the object does not own the reference. It becomes a dangling pointer if the object is destroyed. For example, a garbage collection can remove the last *strong reference* to the object and so destroy it.

Calling `Py_INCREF()` on the *borrowed reference* is recommended to convert it to a *strong reference* in-place, except when the object cannot be destroyed before the last usage of the borrowed reference. The `Py_NewRef()` function can be used to create a new *strong reference*.

bytes-like object
An object that supports the bufferobjects and can export a C-*contiguous* buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as „read-write bytes-like objects". Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects („read-only bytes-like objects"); examples of these include `bytes` and a `memoryview` of a `bytes` object.

kod bajtowy
Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This „intermediate language" is said to run on a *virtual machine* that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

A list of bytecode instructions can be found in the documentation for the dis module.

callable
A callable is an object that can be called, possibly with a set of arguments (see *argument*), with the following syntax:

```
callable(argument1, argument2, argumentN)
```

A *function*, and by extension a *method*, is a callable. An instance of a class that implements the `__call__()` method is also a callable.

wywołanie zwrotne
A subroutine function which is passed as an argument to be executed at some point in the future.

class
A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

class variable
A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

closure variable
A *free variable* referenced from a *nested scope* that is defined in an outer scope rather than being resolved at runtime from the globals or builtin namespaces. May be explicitly defined with the `nonlocal` keyword to allow write access, or implicitly defined if the variable is only being read.

For example, in the `inner` function in the following code, both `x` and `print` are *free variables*, but only `x` is a *closure variable*:

```
def outer():
    x = 0
    def inner():
        nonlocal x
        x += 1
        print(x)
    return inner
```

Due to the `codeobject.co_freevars` attribute (which, despite its name, only includes the names of closure variables rather than listing all referenced free variables), the more general *free variable* term is sometimes used even when the intended meaning is to refer specifically to closure variables.

liczba zespolona
An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of `-1`), often written `i` in mathematics or `j` in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a `j` suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

context manager
An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See **PEP 343**.

context variable
A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See `contextvars`.

contiguous
A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

coroutine
Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also **PEP 492**.

coroutine function
A function which returns a *coroutine* object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by **PEP 492**.

CPython
The canonical implementation of the Python programming language, as distributed on python.org. The term „CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

decorator
A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(arg):
    ...
f = staticmethod(f)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
@staticmethod
def f(arg):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

descriptor
Any object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using *a.b* to get, set or delete an attribute looks up the object named *b* in the class dictionary for *a*, but if *b* is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors» methods, see descriptors or the Descriptor How To Guide.

słownik
An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary comprehension
A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See comprehensions.

dictionary view
The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

docstring
A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing
A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used („If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP
Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

expression
A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statement*s which cannot be used as expressions, such as `while`. Assignments are also statements, not expressions.

moduł rozszerzenia
A module written in C or C++, using Python's C API to interact with the core and with user code.

f-string
String literals prefixed with `'f'` or `'F'` are commonly called „f-strings” which is short for formatted string literals. See also **PEP 498**.

file object
An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw *binary files*, buffered *binary files* and *text files*. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object
A synonym for *file object*.

filesystem encoding and error handler
Encoding and error handler used by Python to decode bytes from the operating system and encode Unicode to the operating system.

The filesystem encoding must guarantee to successfully decode all bytes below 128. If the file system encoding fails to provide this guarantee, API functions can raise `UnicodeError`.

The `sys.getfilesystemencoding()` and `sys.getfilesystemencodeerrors()` functions can be used to get the filesystem encoding and error handler.

The *filesystem encoding and error handler* are configured at Python startup by the `PyConfig_Read()` function: see `filesystem_encoding` and `filesystem_errors` members of `PyConfig`.

See also the *locale encoding*.

finder
An object that tries to find the *loader* for a module that is being imported.

There are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See finders-and-loaders and `importlib` for much more detail.

floor division
Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to `2` in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See **PEP 238**.

free threading
A threading model where multiple threads can run Python bytecode simultaneously within the same interpreter. This is in contrast to the *global interpreter lock* which allows only one thread to execute Python bytecode at a time. See **PEP 703**.

free variable
Formally, as defined in the language execution model, a free variable is any variable used in a namespace which is not a local variable in that namespace. See *closure variable* for an example. Pragmatically, due to the name of the `codeobject.co_freevars` attribute, the term is also sometimes used as a synonym for *closure variable*.

funkcja
A series of statements which returns some value to a caller. It can also be passed zero or more *arguments* which may be used in the execution of the body. See also *parameter*, *method*, and the function section.

function annotation
An *annotation* of a function parameter or return value.

Function annotations are usually used for *type hints*: for example, this function is expected to take two `int` arguments and is also expected to have an `int` return value:

```
def sum_two_numbers(a: int, b: int) -> int:
   return a + b
```

Function annotation syntax is explained in section function.

See *variable annotation* and **PEP 484**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

__future__
A future statement, `from __future__ import <feature>`, directs the compiler to compile the current module using syntax or semantics that will become standard in a future release of Python. The `__future__` module documents the possible values of *feature*. By importing this module and evaluating its variables, you can see when a new feature was first added to the language and when it will (or did) become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

garbage collection
The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles. The garbage collector can be controlled using the `gc` module.

generator
A function which returns a *generator iterator*. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

generator iterator
An object created by a *generator* function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression
An *expression* that returns an *iterator*. It looks like a normal expression followed by a `for` clause defining a loop variable, range, and an optional `if` clause. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))         # sum of squares 0, 1, 4, ... 81
285
```

generic function
A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the *single dispatch* glossary entry, the `functools.singledispatch()` decorator, and **PEP 443**.

generic type
A *type* that can be parameterized; typically a container class such as `list` or `dict`. Used for *type hints* and *annotations*.

For more details, see generic alias types, **PEP 483**, **PEP 484**, **PEP 585**, and the `typing` module.

GIL
See *global interpreter lock*.

global interpreter lock
The mechanism used by the *CPython* interpreter to assure that only one thread executes Python *bytecode* at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier

for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

As of Python 3.13, the GIL can be disabled using the `--disable-gil` build configuration. After building Python with this option, code must be run with `-X gil=0` or after setting the `PYTHON_GIL=0` environment variable. This feature enables improved performance for multi-threaded applications and makes it easier to use multi-core CPUs efficiently. For more details, see **PEP 703**.

hash-based pyc
A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See pyc-invalidation.

hashable
An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. Objects which are instances of user-defined classes are hashable by default. They all compare unequal (except with themselves), and their hash value is derived from their `id()`.

IDLE
An Integrated Development and Learning Environment for Python. idle is a basic editor and interpreter environment which ships with the standard distribution of Python.

immortal
Immortal objects are a CPython implementation detail introduced in **PEP 683**.

If an object is immortal, its *reference count* is never modified, and therefore it is never deallocated while the interpreter is running. For example, `True` and `None` are immortal in CPython.

immutable
An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

import path
A list of locations (or *path entries*) that are searched by the *path based finder* for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

importing
The process by which Python code in one module is made available to Python code in another module.

importer
An object that both finds and loads a module; both a *finder* and *loader* object.

interactive
Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`). For more on interactive mode, see *Tryb interaktywny*.

interpreted
Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also *interactive*.

interpreter shutdown
When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the *garbage collector*. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

iterable
An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *sequence* semantics.

Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, …). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also *iterator*, *sequence*, and *generator*.

iterator
An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `__next__()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

More information can be found in typeiter.

Szczegół implementacyjny CPythona: CPython does not consistently apply the requirement that an iterator define `__iter__()`. And also please note that the free-threading CPython does not guarantee the thread-safety of iterator operations.

key function
A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.merge()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example. the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, a key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, `operator.attrgetter()`, `operator.itemgetter()`, and `operator.methodcaller()` are three key function constructors. See the Sorting HOW TO for examples of how to create and use key functions.

keyword argument
See *argument*.

lambda
An anonymous inline function consisting of a single *expression* which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

LBYL
Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the *EAFP* approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between „the looking" and „the leaping". For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

list
A built-in Python *sequence*. Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

list comprehension
A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ['{:#04x}'.format(x) for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader
An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a *finder*. See also:

- finders-and-loaders
- `importlib.abc.Loader`
- **PEP 302**

locale encoding
On Unix, it is the encoding of the LC_CTYPE locale. It can be set with `locale.setlocale(locale.LC_CTYPE, new_locale)`.

On Windows, it is the ANSI code page (ex: `"cp1252"`).

On Android and VxWorks, Python uses `"utf-8"` as the locale encoding.

`locale.getencoding()` can be used to get the locale encoding.

See also the *filesystem encoding and error handler*.

magic method
An informal synonym for *special method*.

mapping
A container object that supports arbitrary key lookups and implements the methods specified in the `collections.abc.Mapping` or `collections.abc.MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

meta path finder
A *finder* returned by a search of `sys.meta_path`. Meta path finders are related to, but different from *path entry finders*.

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.

metaclass
The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

More information can be found in metaclasses.

method
A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first *argument* (which is usually called `self`). See *function* and *nested scope*.

method resolution order
Method Resolution Order is the order in which base classes are searched for a member during lookup. See python_2.3_mro for details of the algorithm used by the Python interpreter since the 2.3 release.

moduł
An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of *importing*.

See also *package*.

module spec
A namespace containing the import-related information used to load a module. An instance of `importlib.machinery.ModuleSpec`.

See also module-specs.

MRO
See *method resolution order*.

mutable
Mutable objects can change their value but keep their `id()`. See also *immutable*.

named tuple
The term „named tuple” applies to any type or class that inherits from tuple and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```
>>> sys.float_info[1]                    # indexed access
1024
>>> sys.float_info.max_exp               # named field access
1024
>>> isinstance(sys.float_info, tuple)    # kind of tuple
True
```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand, or it can be created by inheriting `typing.NamedTuple`, or with the factory function `collections.namedtuple()`. The latter techniques also add some extra methods that may not be found in hand-written or built-in named tuples.

namespace
The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `builtins.open` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

namespace package
A **PEP 420** *package* which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a *regular package* because they have no `__init__.py` file.

See also *module*.

nested scope
The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

new-style class
Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes

could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattribute__()`, class methods, and static methods.

obiekt
Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

optimized scope
A scope where target local variable names are reliably known to the compiler when the code is compiled, allowing optimization of read and write access to these names. The local namespaces for functions, generators, coroutines, comprehensions, and generator expressions are optimized in this fashion. Note: most interpreter optimizations are applied to all scopes, only those relying on a known set of local and nonlocal variable names are restricted to optimized scopes.

package
A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with a `__path__` attribute.

See also *regular package* and *namespace package*.

parameter
A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are five kinds of parameter:

- *positional-or-keyword*: specifies an argument that can be passed either *positionally* or as a *keyword argument*. This is the default kind of parameter, for example *foo* and *bar* in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example *posonly1* and *posonly2* in the following:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example *kw_only1* and *kw_only2* in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example *args* in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example *kwargs* in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, the `inspect.Parameter` class, the function section, and **PEP 362**.

path entry
A single location on the *import path* which the *path based finder* consults to find modules for importing.

path entry finder
A *finder* returned by a callable on `sys.path_hooks` (i.e. a *path entry hook*) which knows how to locate modules given a *path entry*.

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.

path entry hook
A callable on the `sys.path_hooks` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

path based finder
One of the default *meta path finders* which searches an *import path* for modules.

path-like object
An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by **PEP 519**.

PEP
Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible for building consensus within the community and documenting dissenting opinions.

See **PEP 1**.

portion
A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in **PEP 420**.

positional argument
See *argument*.

provisional API
A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously – they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a „solution of last resort" - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See **PEP 411** for more details.

provisional package
See *provisional API*.

Python 3000
Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated „Py3k".

Pythonic
An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print(food[i])
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print(piece)
```

qualified name
A dotted name showing the „path" from a module's global scope to a class, function or method defined in that module, as defined in **PEP 3155**. For top-level functions and classes, the qualified name is the same as the object's name:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

reference count
The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Some objects are *immortal* and have reference counts that are never modified, and therefore the objects are never deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. Programmers can call the `sys.getrefcount()` function to return the reference count for a particular object.

regular package
A traditional *package*, such as a directory containing an `__init__.py` file.

See also *namespace package*.

REPL
An acronym for the „read–eval–print loop", another name for the *interactive* interpreter shell.

__slots__
A declaration inside a class that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

sequence
An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `__len__()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `bytes`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *hashable* keys rather than integers.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`. For more documentation on sequence methods generally, see Common Sequence Operations.

set comprehension
A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See comprehensions.

single dispatch
A form of *generic function* dispatch where the implementation is chosen based on the type of a single argument.

slice
An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally.

soft deprecated
A soft deprecated API should not be used in new code, but it is safe for already existing code to use it. The API remains documented and tested, but will not be enhanced further.

Soft deprecation, unlike normal deprecation, does not plan on removing the API and will not emit warnings.

See PEP 387: Soft Deprecation.

special method
A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in specialnames.

instrukcja
A statement is part of a suite (a „block" of code). A statement is either an *expression* or one of several constructs with a keyword, such as `if`, `while` or `for`.

static type checker
An external tool that reads Python code and analyzes it, looking for issues such as incorrect types. See also *type hints* and the `typing` module.

strong reference
In Python's C API, a strong reference is a reference to an object which is owned by the code holding the reference. The strong reference is taken by calling `Py_INCREF()` when the reference is created and released with `Py_DECREF()` when the reference is deleted.

The `Py_NewRef()` function can be used to create a strong reference to an object. Usually, the `Py_DECREF()` function must be called on the strong reference before exiting the scope of the strong reference, to avoid leaking one reference.

See also *borrowed reference*.

text encoding
A string in Python is a sequence of Unicode code points (in range `U+0000–U+10FFFF`). To store or transfer a string, it needs to be serialized as a sequence of bytes.

Serializing a string into a sequence of bytes is known as „encoding", and recreating the string from the sequence of bytes is known as „decoding".

There are a variety of different text serialization codecs, which are collectively referred to as „text encodings".

text file
A *file object* able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the *text encoding* automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also *binary file* for a file object able to read and write *bytes-like objects*.

triple-quoted string
A string which is bound by three instances of either a quotation mark (") or an apostrophe («). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type
The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

type alias
A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying *type hints*. For example:

```
def remove_gray_shades(
        colors: list[tuple[int, int, int]]) -> list[tuple[int, int, int]]:
    pass
```

could be made more readable like this:

```
Color = tuple[int, int, int]

def remove_gray_shades(colors: list[Color]) -> list[Color]:
    pass
```

See `typing` and **PEP 484**, which describe this functionality.

type hint
An *annotation* that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to *static type checkers*. They can also aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See `typing` and **PEP 484**, which describe this functionality.

universal newlines
A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See **PEP 278** and **PEP 3116**, as well as `bytes.splitlines()` for an additional use.

variable annotation
An *annotation* of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:
    field: 'annotation'
```

Variable annotations are usually used for *type hints*: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section annassign.

See *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality. Also see annotations-howto for best practices on working with annotations.

virtual environment
A cooperatively isolated runtime environment that allows Python users and applications to install and upgrade Python distribution packages without interfering with the behaviour of other Python applications running on the same system.

See also `venv`.

virtual machine
A computer defined entirely in software. Python's virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python
Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing „`import this`" at the interactive prompt.

DODATEK B

O tej dokumentacji

Dokumenty są wygenerowane ze źródeł reStructuredText przez Sphinksa, procesor dokumentów napisany specjalnie dla dokumentacji Pythona.

Rozwój dokumentacji i jej oprzyrządowania jest w całości wysiłkiem wolontariackim, tak samo jak sam Python. Jeśli chcesz wnieść swój wkład, na stronie reporting-bugs znajdziesz informacje jak to zrobić. Nowi wolontariusze są zawsze mile widziani!

Ogromne podziękowania dla:

- Freda L. Drake'a, Jr., twórcy oryginalnego zestawu narzędzi dokumentacji Pythona i autora dużej części jej treści;
- projektu Docutils za stworzenie reStructuredText i pakietu Docutils;
- Fredrika Lundha za jego projekt Alternative Python Reference, z którego Sphinx wziął wiele dobrych pomysłów.

B.1 Współtwórcy dokumentacji Pythona

Wielu ludzi rozwija język Python, bibliotekę standardową Pythona i dokumentację. W Misc/ACKS w źródłach Pythona znajdziesz częściową listę kontrybutorów.

Tylko dzięki wkładowi społeczności Python ma tak wspaniałą dokumentację – dziękujemy!

DODATEK C

Historia i zapisy prawne

C.1 Historia programu

Python was created in the early 1990s by Guido van Rossum at Stichting Mathematisch Centrum (CWI, see https://www.cwi.nl/) in the Netherlands as a successor of a language called ABC. Guido remains Python's principal author, although it includes many contributions from others.

In 1995, Guido continued his work on Python at the Corporation for National Research Initiatives (CNRI, see https://www.cnri.reston.va.us/) in Reston, Virginia where he released several versions of the software.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see https://www.zope.org/). In 2001, the Python Software Foundation (PSF, see https://www.python.org/psf/) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

All Python releases are Open Source (see https://opensource.org/ for the Open Source Definition). Historically, most, but not all, Python releases have also been GPL-compatible; the table below summarizes the various releases.

wydanie	Pochodne po	Rok	Właściciel	Zgodne z Uprawnieniami Ogólnie Powszechnymi (GPL)?
od 0.9.0 do 1.2	nie podano	od 1991 do 1995	CWI	tak
od 1.3 do 1.5.2	1.2	od 1995 do 1999	CNRI	tak
1.6	1.5.2	2000	CNRI	nie
2.0	1.6	2000	BeOpen.com	nie
1.6.1	1.6	2001	CNRI	nie
2.1	2.0 i 1.6.1	2001	Fundacja Programu języka Pytonowskiego (PSF)	nie
2.0.1	2.0 i 1.6.1	2001	Fundacja Programu języka Pytonowskiego (PSF)	tak
2.1.1	2.1 i 2.0.1	2001	Fundacja Programu języka Pytonowskiego (PSF)	tak
2.1.2	2.1.1	2002	Fundacja Programu języka Pytonowskiego (PSF)	tak
2.1.3	2.1.2	2002	Fundacja Programu języka Pytonowskiego (PSF)	tak
2.2 and above	2.1.1	2001-now	Fundacja Programu języka Pytonowskiego (PSF)	tak

Informacja

Zgodność z uprawnieniami ogólnie powszechnymi (w skrócie - z ang. - GPL) nie oznacza, że rozprowadzamy język pytonowski z uprawnieniami ogólnie powszechnymi (w skrócie - z ang. - GPL). Wszystkie uprawnienia dostarczane z językiem pytonowskim, w przeciwieństwie do uprawnień ogólnie powszechnych (w skrócie - z ang. - GPL), pozwalają na rozpowszechnianie programów języka pytonowskiego z wprowadzonymi zmianami bez ustanawiania tych zmian w ramach otwartych źródeł. Uprawnienia zgodne z ogólnie powszechnymi uprawnieniami (w skrócie - z ang. - GPL) pozwalają na łączenie wydań programu języka pytonowskiego z innymi programami które są wydane z uprawnieniami ogólnie powszechnymi (w skrócie - z ang. - GPL). Inne uprawnienia, niezgodne z ogólnie powszechnymi, na to nie zezwalają.

Podziękowania dla wielu ochotników przychodzących z zewnątrz, którzy pracowali pod kierunkiem Gwidona aby umożliwić te wydania programu języka pytonowskiego.

C.2 Zasady i warunki postępowania z Pythonem i ogólnie jego użycia

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Licenses and Acknowledgements for Incorporated Software* for an incomplete list of these licenses.

C.2.1 PSF LICENSE AGREEMENT FOR PYTHON 3.13.0

```
1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and
   the Individual or Organization ("Licensee") accessing and otherwise using Python
   3.13.0 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby
```

```
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 3.13.0 alone or in any derivative
   version, provided, however, that PSF's License Agreement and PSF's notice of
   copyright, i.e., "Copyright © 2001-2024 Python Software Foundation; All Rights
   Reserved" are retained in Python 3.13.0 alone or in any derivative version
   prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 3.13.0 or any part thereof, and wants to make the
   derivative work available to others as provided herein, then Licensee hereby
   agrees to include in any such work a brief summary of the changes made to Python
   3.13.0.

4. PSF is making Python 3.13.0 available to Licensee on an "AS IS" basis.
   PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF PYTHON 3.13.0 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.13.0
   FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.13.0, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship
   of agency, partnership, or joint venture between PSF and Licensee.  This License
   Agreement does not grant permission to use PSF trademarks or trade name in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.

8. By copying, installing or otherwise using Python 3.13.0, Licensee agrees
   to be bound by the terms and conditions of this License Agreement.
```

C.2.2 BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0

BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

```
1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at
   160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization
   ("Licensee") accessing and otherwise using this software in source or binary
   form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement,
   BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license
   to reproduce, analyze, test, perform and/or display publicly, prepare derivative
   works, distribute, and otherwise use the Software alone or in any derivative
   version, provided, however, that the BeOpen Python License is retained in the
   Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis.
   BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF
   EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR
   WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE
   USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING,
   MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF
   ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects
   by the law of the State of California, excluding conflict of law provisions.
   Nothing in this License Agreement shall be deemed to create any relationship of
   agency, partnership, or joint venture between BeOpen and Licensee.  This License
   Agreement does not grant permission to use BeOpen trademarks or trade names in a
   trademark sense to endorse or promote products or services of Licensee, or any
   third party.  As an exception, the "BeOpen Python" logos available at
   http://www.pythonlabs.com/logos.html may be used according to the permissions
   granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be
   bound by the terms and conditions of this License Agreement.
```

C.2.3 CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

```
1. This LICENSE AGREEMENT is between the Corporation for National Research
   Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191
   ("CNRI"), and the Individual or Organization ("Licensee") accessing and
   otherwise using Python 1.6.1 software in source or binary form and its
   associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby
   grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce,
   analyze, test, perform and/or display publicly, prepare derivative works,
   distribute, and otherwise use Python 1.6.1 alone or in any derivative version,
   provided, however, that CNRI's License Agreement and CNRI's notice of copyright,
   i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All
   Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version
   prepared by Licensee.  Alternately, in lieu of CNRI's License Agreement,
   Licensee may substitute the following text (omitting the quotes): "Python 1.6.1
   is made available subject to the terms and conditions in CNRI's License
   Agreement.  This Agreement together with Python 1.6.1 may be located on the
   internet using the following unique, persistent identifier (known as a handle):
   1895.22/1013.  This Agreement may also be obtained from a proxy server on the
   internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or
   incorporates Python 1.6.1 or any part thereof, and wants to make the derivative
   work available to others as provided herein, then Licensee hereby agrees to
   include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis.  CNRI
   MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED.  BY WAY OF EXAMPLE,
   BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY
   OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF
   PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR
   ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF
   MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE
   THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of
   its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property
   law of the United States, including without limitation the federal copyright
   law, and, to the extent such U.S. federal law does not apply, by the law of the
   Commonwealth of Virginia, excluding Virginia's conflict of law provisions.
   Notwithstanding the foregoing, with regard to derivative works based on Python
   1.6.1 that incorporate non-separable material that was previously distributed
   under the GNU General Public License (GPL), the law of the Commonwealth of
   Virginia shall govern this License Agreement only as to issues arising under or
   with respect to Paragraphs 4, 5, and 7 of this License Agreement.  Nothing in
   this License Agreement shall be deemed to create any relationship of agency,
   partnership, or joint venture between CNRI and Licensee.  This License Agreement
   does not grant permission to use CNRI trademarks or trade name in a trademark
   sense to endorse or promote products or services of Licensee, or any third
   party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing
   or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and
   conditions of this License Agreement.
```

C.2.4 CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

```
Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The
Netherlands.  All rights reserved.

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted, provided that
the above copyright notice appear in all copies and that both that copyright
notice and this permission notice appear in supporting documentation, and that
the name of Stichting Mathematisch Centrum or CWI not be used in advertising or
publicity pertaining to distribution of the software without specific, written
prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO
EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT
OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE,
DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS
SOFTWARE.
```

C.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.13.0 DOCUMENTATION

```
Permission to use, copy, modify, and/or distribute this software for any
purpose with or without fee is hereby granted.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH
REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT,
INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM
LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR
OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3 Licenses and Acknowledgements for Incorporated Software

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

C.3.1 Mersenne Twister

The `_random` C extension underlying the `random` module includes code based on a download from http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Any feedback is very welcome.
http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo()`, and `getnameinfo()`, which are coded in separate source files from the WIDE Project, https://www.wide.ad.jp/.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.3 Asynchronous socket services

The `test.support.asynchat` and `test.support.asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

C.3.4 Cookie management

The `http.cookies` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

C.3.5 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

C.3.6 UUencode and UUdecode functions

The `uu` codec contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.7 XML Remote Procedure Calls

The `xmlrpc.client` module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

C.3.8 test_epoll

The `test.test_epoll` module contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.9 Select kqueue

The `select` module contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS ``AS IS'' AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

C.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski» implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013  Marek Majkowski <marek@popcount.org>

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in
all copies or substantial portions of the Software.
</MIT License>

Original location:
   https://github.com/majek/csiphash/

Solution inspired by code from:
   Samuel Neves (supercop/crypto_auth/siphash24/little)
   djb (supercop/crypto_auth/siphash24/little2)
   Jean-Philippe Aumasson (https://131002.net/siphash/siphash24.c)
```

C.3.11 strtod and dtoa

The file `Python/dtoa.c`, which supplies C functions dtoa and strtod for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from https://web.archive.org/web/20220517033456/http://www.netlib.org/fp/dtoa.c. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/****************************************************************
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
 * WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
 * REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 *
 ************************************************************/
```

C.3.12 OpenSSL

The modules `hashlib`, `posix` and `ssl` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and macOS installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here. For the OpenSSL 3.0 release, and later releases derived from that, the Apache License v2 applies:

```
                                 Apache License
                           Version 2.0, January 2004
                        https://www.apache.org/licenses/

TERMS AND CONDITIONS FOR USE, REPRODUCTION, AND DISTRIBUTION

1. Definitions.

   "License" shall mean the terms and conditions for use, reproduction,
   and distribution as defined by Sections 1 through 9 of this document.

   "Licensor" shall mean the copyright owner or entity authorized by
   the copyright owner that is granting the License.

   "Legal Entity" shall mean the union of the acting entity and all
   other entities that control, are controlled by, or are under common
   control with that entity. For the purposes of this definition,
   "control" means (i) the power, direct or indirect, to cause the
   direction or management of such entity, whether by contract or
   otherwise, or (ii) ownership of fifty percent (50%) or more of the
   outstanding shares, or (iii) beneficial ownership of such entity.

   "You" (or "Your") shall mean an individual or Legal Entity
   exercising permissions granted by this License.

   "Source" form shall mean the preferred form for making modifications,
   including but not limited to software source code, documentation
   source, and configuration files.

   "Object" form shall mean any form resulting from mechanical
   transformation or translation of a Source form, including but
   not limited to compiled object code, generated documentation,
   and conversions to other media types.

   "Work" shall mean the work of authorship, whether in Source or
   Object form, made available under the License, as indicated by a
   copyright notice that is included in or attached to the work
   (an example is provided in the Appendix below).

   "Derivative Works" shall mean any work, whether in Source or Object
   form, that is based on (or derived from) the Work and for which the
   editorial revisions, annotations, elaborations, or other modifications
   represent, as a whole, an original work of authorship. For the purposes
   of this License, Derivative Works shall not include works that remain
   separable from, or merely link (or bind by name) to the interfaces of,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
   the Work and Derivative Works thereof.

   "Contribution" shall mean any work of authorship, including
   the original version of the Work and any modifications or additions
   to that Work or Derivative Works thereof, that is intentionally
   submitted to Licensor for inclusion in the Work by the copyright owner
   or by an individual or Legal Entity authorized to submit on behalf of
   the copyright owner. For the purposes of this definition, "submitted"
   means any form of electronic, verbal, or written communication sent
   to the Licensor or its representatives, including but not limited to
   communication on electronic mailing lists, source code control systems,
   and issue tracking systems that are managed by, or on behalf of, the
   Licensor for the purpose of discussing and improving the Work, but
   excluding communication that is conspicuously marked or otherwise
   designated in writing by the copyright owner as "Not a Contribution."

   "Contributor" shall mean Licensor and any individual or Legal Entity
   on behalf of whom a Contribution has been received by Licensor and
   subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   copyright license to reproduce, prepare Derivative Works of,
   publicly display, publicly perform, sublicense, and distribute the
   Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of
   this License, each Contributor hereby grants to You a perpetual,
   worldwide, non-exclusive, no-charge, royalty-free, irrevocable
   (except as stated in this section) patent license to make, have made,
   use, offer to sell, sell, import, and otherwise transfer the Work,
   where such license applies only to those patent claims licensable
   by such Contributor that are necessarily infringed by their
   Contribution(s) alone or by combination of their Contribution(s)
   with the Work to which such Contribution(s) was submitted. If You
   institute patent litigation against any entity (including a
   cross-claim or counterclaim in a lawsuit) alleging that the Work
   or a Contribution incorporated within the Work constitutes direct
   or contributory patent infringement, then any patent licenses
   granted to You under this License for that Work shall terminate
   as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the
   Work or Derivative Works thereof in any medium, with or without
   modifications, and in Source or Object form, provided that You
   meet the following conditions:

   (a) You must give any other recipients of the Work or
       Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices
       stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works
       that You distribute, all copyright, patent, trademark, and
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
      attribution notices from the Source form of the Work,
      excluding those notices that do not pertain to any part of
      the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its
      distribution, then any Derivative Works that You distribute must
      include a readable copy of the attribution notices contained
      within such NOTICE file, excluding those notices that do not
      pertain to any part of the Derivative Works, in at least one
      of the following places: within a NOTICE text file distributed
      as part of the Derivative Works; within the Source form or
      documentation, if provided along with the Derivative Works; or,
      within a display generated by the Derivative Works, if and
      wherever such third-party notices normally appear. The contents
      of the NOTICE file are for informational purposes only and
      do not modify the License. You may add Your own attribution
      notices within Derivative Works that You distribute, alongside
      or as an addendum to the NOTICE text from the Work, provided
      that such additional attribution notices cannot be construed
      as modifying the License.

   You may add Your own copyright statement to Your modifications and
   may provide additional or different license terms and conditions
   for use, reproduction, or distribution of Your modifications, or
   for any such Derivative Works as a whole, provided Your use,
   reproduction, and distribution of the Work otherwise complies with
   the conditions stated in this License.

5. Submission of Contributions. Unless You explicitly state otherwise,
   any Contribution intentionally submitted for inclusion in the Work
   by You to the Licensor shall be under the terms and conditions of
   this License, without any additional terms or conditions.
   Notwithstanding the above, nothing herein shall supersede or modify
   the terms of any separate license agreement you may have executed
   with Licensor regarding such Contributions.

6. Trademarks. This License does not grant permission to use the trade
   names, trademarks, service marks, or product names of the Licensor,
   except as required for reasonable and customary use in describing the
   origin of the Work and reproducing the content of the NOTICE file.

7. Disclaimer of Warranty. Unless required by applicable law or
   agreed to in writing, Licensor provides the Work (and each
   Contributor provides its Contributions) on an "AS IS" BASIS,
   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
   implied, including, without limitation, any warranties or conditions
   of TITLE, NON-INFRINGEMENT, MERCHANTABILITY, or FITNESS FOR A
   PARTICULAR PURPOSE. You are solely responsible for determining the
   appropriateness of using or redistributing the Work and assume any
   risks associated with Your exercise of permissions under this License.

8. Limitation of Liability. In no event and under no legal theory,
   whether in tort (including negligence), contract, or otherwise,
   unless required by applicable law (such as deliberate and grossly
   negligent acts) or agreed to in writing, shall any Contributor be
   liable to You for damages, including any direct, indirect, special,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
   incidental, or consequential damages of any character arising as a
   result of this License or out of the use or inability to use the
   Work (including but not limited to damages for loss of goodwill,
   work stoppage, computer failure or malfunction, or any and all
   other commercial damages or losses), even if such Contributor
   has been advised of the possibility of such damages.

9. Accepting Warranty or Additional Liability. While redistributing
   the Work or Derivative Works thereof, You may choose to offer,
   and charge a fee for, acceptance of support, warranty, indemnity,
   or other liability obligations and/or rights consistent with this
   License. However, in accepting such obligations, You may act only
   on Your own behalf and on Your sole responsibility, not on behalf
   of any other Contributor, and only if You agree to indemnify,
   defend, and hold each Contributor harmless for any liability
   incurred by, or claims asserted against, such Contributor by reason
   of your accepting any such warranty or additional liability.

END OF TERMS AND CONDITIONS
```

C.3.13 expat

The `pyexpat` extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```
Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
                               and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.14 libffi

The `_ctypes` C extension underlying the `ctypes` module is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008  Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT.  IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.15 zlib

The `zlib` extension is built using an included copy of the zlib sources if the zlib version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2011 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty.  In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly        Mark Adler
jloup@gzip.org          madler@alumni.caltech.edu
```

C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` is based on the cfuhash project:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

Redistribution and use in source and binary forms, with or without
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
modification, are permitted provided that the following conditions
are met:

  * Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

  * Redistributions in binary form must reproduce the above
    copyright notice, this list of conditions and the following
    disclaimer in the documentation and/or other materials provided
    with the distribution.

  * Neither the name of the author nor the names of its
    contributors may be used to endorse or promote products derived
    from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
(INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED
OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.17 libmpdec

The `_decimal` C extension underlying the `decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2020 Stefan Krah. All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
SUCH DAMAGE.
```

C.3.18 W3C C14N test suite

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at https://www.w3.org/TR/xml-c14n2-testcases/ and is distributed under the 3-clause BSD license:

```
Copyright (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang),
All Rights Reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

* Redistributions of works must retain the original copyright notice,
  this list of conditions and the following disclaimer.
* Redistributions in binary form must reproduce the original copyright
  notice, this list of conditions and the following disclaimer in the
  documentation and/or other materials provided with the distribution.
* Neither the name of the W3C nor the names of its contributors may be
  used to endorse or promote products derived from this work without
  specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT
OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT
LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

C.3.19 mimalloc

MIT License:

```
Copyright (c) 2018-2021 Microsoft Corporation, Daan Leijen

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

C.3.20 asyncio

Parts of the `asyncio` module are incorporated from uvloop 0.16, which is distributed under the MIT license:

```
Copyright (c) 2015-2021 MagicStack Inc.  http://magic.io

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.21 Global Unbounded Sequences (GUS)

The file `Python/qsbr.c` is adapted from FreeBSD's „Global Unbounded Sequences" safe memory reclamation scheme in subr_smr.c. The file is distributed under the 2-Clause BSD License:

```
Copyright (c) 2019,2020 Jeffrey Roberson <jeff@FreeBSD.org>

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice unmodified, this list of conditions, and the following
   disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR
IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED.
IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT,
INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT
NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY
THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
(INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

DODATEK D

Prawa autorskie

Patrz dział *Historia i zapisy prawne*, aby zobaczyć pełną informację na temat licencji i praw.

Indeks

Niealfabetyczny

A

B

C

D

E

F

G