
Python Tutorial

Wydanie 3.8.20

**Guido van Rossum
and the Python development team**

września 08, 2024

**Python Software Foundation
Email: docs@python.org**

1	Na zaostwienie apetytu	3
2	Używanie Interpretera Pythona	5
2.1	Wywoływanie Interpretera	5
2.2	Interpreter i jego środowisko	7
3	Nieformalne wprowadzenie do Pythona	9
3.1	Używanie Pythona jako kalkulatora	10
3.2	Pierwsze kroki do programowania	16
4	Więcej narzędzi kontroli przepływu	19
4.1	Instrukcje <code>if</code>	19
4.2	Instrukcje <code>for</code>	20
4.3	Funkcja <code>range()</code>	20
4.4	<code>break</code> i <code>continue</code> oraz klauzula <code>else</code> w pętlach	21
4.5	Instrukcje <code>pass</code>	22
4.6	Definiowanie funkcji	23
4.7	Więcej o definiowaniu funkcji	24
4.8	Intermezzo: Styl kodowania	32
5	Struktury danych	35
5.1	Więcej na temat list	35
5.2	Instrukcja <code>del</code>	40
5.3	Krotki i sekwencje	40
5.4	Zbiory	41
5.5	Słowniki	42
5.6	Techniki pętli	43
5.7	Więcej na temat warunków	44
5.8	Porównywanie sekwencji i innych typów	45
6	Moduły	47
6.1	Więcej o modułach	48
6.2	Moduły standardowe	51
6.3	The <code>dir()</code> Function	51
6.4	Packages	52
7	Wejście i wyjście	57

7.1	Fancier Output Formatting	57
7.2	Reading and Writing Files	61
8	Błędy i Wyjątki	65
8.1	Błędy składni	65
8.2	Sytuacje Wyjątkowe	65
8.3	Obsługa wyjątków	66
8.4	Rzucanie wyjątków	68
8.5	Wyjątki zdefiniowane przez użytkownika	69
8.6	Defining Clean-up Actions	70
8.7	Predefined Clean-up Actions	71
9	Klasy	73
9.1	Kilka słów o nazwach i obiektach	74
9.2	Python Scopes and Namespaces	74
9.3	A First Look at Classes	76
9.4	Random Remarks	80
9.5	Inheritance	81
9.6	Private Variables	82
9.7	Odds and Ends	83
9.8	Iterators	84
9.9	Generators	85
9.10	Generator Expressions	85
10	Krótką wycieczką po Bibliotece Standardowej	87
10.1	Interfejs Systemu Operacyjnego	87
10.2	File Wildcards	88
10.3	Command Line Arguments	88
10.4	Error Output Redirection and Program Termination	88
10.5	String Pattern Matching	89
10.6	Mathematics	89
10.7	Internet Access	90
10.8	Dates and Times	90
10.9	Kompresja Danych	91
10.10	Performance Measurement	91
10.11	Kontrola Jakości	91
10.12	Dostarczone z bateriami	92
11	Brief Tour of the Standard Library — Part II	93
11.1	Output Formatting	93
11.2	Templating	94
11.3	Working with Binary Data Record Layouts	95
11.4	Multi-threading	96
11.5	Logging	96
11.6	Weak References	97
11.7	Tools for Working with Lists	98
11.8	Decimal Floating Point Arithmetic	99
12	Środowiska wirtualne i pakiety	101
12.1	Wprowadzenie	101
12.2	Tworzenie Środowisk Wirtualnych	101
12.3	Managing Packages with pip	102
13	Co dalej?	105

14 Interactive Input Editing and History Substitution	107
14.1 Tab Completion and History Editing	107
14.2 Alternatives to the Interactive Interpreter	107
15 Arytmetyka liczb zmiennoprzecinkowych: Problemy i ograniczenia	109
15.1 Representation Error	112
16 Załącznik	115
16.1 Tryb interaktywny	115
A Słownik	117
B O tej dokumentacji	131
B.1 Współtwórcy dokumentacji Pythona	131
C Historia i zapisy prawne	133
C.1 Historia programu	133
C.2 Zasady i warunki postępowania z programem języka pythonowskiego i ogólnie jego użycia.	134
C.3 Licenses and Acknowledgements for Incorporated Software	138
D Prawa autorskie	151
Indeks	153

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 być wszechstronny i umówić każdą pojedynczą cechę, lub nawet każdą często używaną cechę. Zamiast tego wprowadza wiele funkcji Pythona najbardziej wartych zauważenia i da ci dobre rozumienie smaku i stylu języka. Po przeczytaniu go, będziesz w stanie czytać i pisać moduły i programy Pythona, oraz będziesz gotowy uczyć się więcej o różnych modułach bibliotek Pythona opisanych w [library-index](#).

Warto również przejrzeć [Słownik](#).

Na zaostwienie 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, Mac OS X i uniksowe systemy operacyjne i pomoże ci wykonać zadanie 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 znaczny 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 pisany 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 nauczanie 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 dotycząc zaawansowanych pojęć, takich jak wyjątki i klasy zdefiniowane przez użytkownika.

Używanie Interpretera Pythona

2.1 Wywoływanie Interpretera

Interpreter Pythona jest zazwyczaj zainstalowany jako `/usr/local/bin/python3.8` na maszynach, na których jest dostępny; wstawienie `/usr/local/bin` w ścieżkę wyszukiwania uniksowego shella umożliwia jego uruchomienie przez wpisanie komendy:

```
python3.8
```

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 `python3.8`. 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 *Interactive Input Editing and History Substitution*. 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.

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.

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 pojedynczym cudzysłowie.

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 znalezionej 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:

```
$ python3.8
Python 3.8 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] 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("Be careful not to fall off!")
...
Be careful not to fall off!
```

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 -*-
```

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 tym manualu, także tych wpisywanych w interaktywnej konsoli, zawiera komentarze. Komentarze w Pythonie zaczynają się znakiem kratki `#` 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 kratki w ciągu znaków jest po prostu znakiem kratki. Jako że komentarze mają wyjaśniać kod i nie są interpretowane przez Pythona, można je ominąć przy wpisywaniu przykładów.

Trochę przykładów:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

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 `/` działają jak w większości innych języków programowania (na przykład Pascal lub C); nawiasy `()` można użyć do grupowania. Na przykład:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5 # division always returns a floating point number
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 (pomijając część ułamkową) możesz użyć operatora `//`; aby obliczyć resztę możesz użyć `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3 # floor division discards the fractional part
5
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # result * divisor + remainder
17
```

W Pythonie możesz użyć operatora `**`, aby obliczać potęgi¹:

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 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:

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`.


```
>>> n # try to access an undefined variable
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
>>> 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 złożonych i używa sufixów `j` lub `J` do wskazania części urojonej (`np. 3+5j`).

3.1.2 Ciągi znaków

Oprócz liczb Python może również manipulować ciągami znaków, które można wyrazić na parę sposobów. Mogą one być objęte pojedynczym ('...') lub podwójnym cudzysłowem ("...") z tym samym efektem². `\` można użyć, aby zapobiec domyślnej interpretacji znaku cudzysłowu:

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> 'doesn\'t' # use \' to escape the single quote...
"doesn't"
>>> "doesn't" # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> 'Isn\'t," they said.'
'Isn\'t," they said.'
```

W interaktywnym interpreterze wyjściowy ciąg znaków jest otoczony cudzysłowem a znaki specjalne są „uwolnione” ukośnikami wstecznymi. Mimo że może czasem wyglądać inaczej niż wejściowy ciąg znaków (cudzysłów okalający mógł się zmienić), te dwa ciągi znaków są równoważne. Ciąg znaków objęty jest podwójnym cudzysłowem, jeśli zawiera tylko pojedyncze cudzysłowy i nie zawiera podwójnych, w przeciwnym wypadku objęty jest pojedynczym. Funkcja `print()` produkuje bardziej czytelne wyjście, omijając okalający cudzysłów i drukując „uwolnione” i specjalne znaki:

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 escape'ować " (lecz musisz escape'ować `\`) i vice versa.

```
>>> "Isn't," they said.
Isn't," they said.
>>> print("Isn't," they said.)
Isn't," they said.
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print(), \n is included in the output
First line.\nSecond line.
>>> print(s) # with print(), \n produces a new line
First line.
Second line.
```

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

```
>>> print('C:\some\name') # here \n means newline!
C:\some
ame
>>> print(r'C:\some\name') # note the r before the quote
C:\some\name
```

Literały ciągów znaków mogą mieć wiele linii. Można je uzyskać używając potrójnych cudzysłowów: `"""..."""` lub `'''...'''`. Końce linii są automatycznie zawarte w ciągu znaków, ale można tego uniknąć dodając `\` na końcu linii. Następujący przykład:

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

produkuje następujące wyjście (zwróć uwagę, że nie ma pierwszej nowej linii):

```
Usage: thingy [OPTIONS]
    -h                        Display this usage message
    -H hostname              Hostname to connect to
```

Ciągi mogą być konkatelowane (sklejane) operatorem `+` i powtarzane przy użyciu `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Dwa lub więcej *literałów ciągów* (czyli te zawarte w cudzysłowach) obok siebie są automatycznie konkatelowane.

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

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

```
>>> text = ('Put several strings within parentheses '
...        'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

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

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
File "<stdin>", line 1
    prefix 'thon'
          ^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
File "<stdin>", line 1
    ('un' * 3) 'ium'
          ^
SyntaxError: invalid syntax
```

Jeśli chcesz skonkatelować 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] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

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

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

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

Oprócz indeksowania, dostępne jest także wykrawanie (ang. *slicing*). Indeksowania używamy, aby uzyskać pojedyncze znaki, wykrawanie pozwala uzyskać podciąg (ang. substring):

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

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'
```

Indeksy wykrawania mają przydatne wartości domyślne; pominięty pierwszy indeks domyślnie jest zerem, pominięty drugi indeks domyślnie ma wartość długości wykrawanego ciągu.

```
>>> word[:2]    # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]    # characters from position 4 (included) to the end
'on'
>>> word[-2:]   # characters from the second-last (included) to the end
'on'
```

Sposobem na zapamiętanie, jak działa wykrawanie, to myślenie o indeksach wskazujących *między* 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. Slice od i do j składa się ze wszystkich znaków pomiędzy krawędziami oznaczonymi kolejno i i j .

Dla nieujemnych indeksów długość slice'a 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]    # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Jednak indeksy wykrawania poza zakresem są obsługiwane bezpiecznie przy wykrawaniu:

```
>>> 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(-naś) stworzyć nowy:

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

Wbudowana funkcja `len()` zwraca długość ciągu:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

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 „wykrawać”:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

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

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

Listy wspierają też operacje takie jak konkatencja:

```
>>> 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] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

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

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

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

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

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

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

Można zagnieźdzać 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:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... 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 (wcięcie) 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('The value of i is', i)
The value of i is 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,
```

Więcej narzędzi kontroli przepływu

Oprócz wprowadzonej przed chwilą instrukcji `while`, Python używa zwykłych instrukcji kontroli przepływu znanych z innych języków, z drobnymi zmianami.

4.1 Instrukcje `if`

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

```
>>> x = int(input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print('Negative changed to zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Single')
... else:
...     print('More')
...
More
```

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.

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 jakiegokolwiek sekwencji (listy lub łańcucha znaków), w takim porządku, w jakim są one umieszczone w danej sekwencji. Na przykład (gra słów niezamierzona):

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print(w, len(w))
...
cat 3
window 6
defenestrate 12
```

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

```
# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        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”):

```
range(5, 10)
5, 6, 7, 8, 9

range(0, 10, 3)
0, 3, 6, 9
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
range(-10, -100, -30)
-10, -40, -70
```

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

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 had
2 a
3 little
4 lamb
```

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

Dzieje się dziwna rzecz jeśli po prostu wydrukujesz zakres:

```
>>> print(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 obejrzymy więcej funkcji, które zwracają iterable i biorą iterable jako argumenty. Może jesteś ciekawy(-a) jak uzyskać listę z zakresu. Tutaj jest rozwiązanie:

```
>>> list(range(4))
[0, 1, 2, 3]
```

W rozdziale [Struktury danych](#) omówimy bardziej szczegółowo `list()`.

4.4 break i continue oraz klauzula else w pętlach

Instrukcja `break`, tak jak w C, wychodzi z najbardziej wewnętrznej pętli `for` lub `while` zawierającej tę instrukcję.

Instrukcje pętli mogą mieć klauzulę `else`. Jest ona wykonywana, gdy pętla kończy się przez wyczerpanie się iterable'a (w pętli `for`) lub kiedy warunek staje się fałszywy (w pętli `while`), lecz nie gdy pętli jest przerwana przez instrukcję `break`. Ilustruje to poniższa pętla, która wyszukuje liczby pierwsze:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'equals', x, '*', n//x)
...             break
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
...     else:
...         # loop fell through without finding a factor
...         print(n, 'is a prime number')
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

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

Gdy używasz jej z pętlą, klauzula `else` ma więcej wspólnego z klauzulą `else` instrukcji `try` niż z tą z instrukcji `if`: klauzula `else` instrukcji `try` jest uruchamiana, gdy nie zostanie zgłoszony wyjątek, a klauzula `else` pętli jest uruchamiana, gdy nie wystąpi `break`. Więcej informacji na temat instrukcji `try` i wyjątków znajdziesz w [Obsługa wyjątków](#).

Instrukcja `continue`, również pożyczona z C, kontynuuje następną iterację pętli:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Found an even number", num)
...         continue
...     print("Found an odd number", num)
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

4.5 Instrukcje `pass`

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

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (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 # Remember to implement this!
... 
```

4.6 Definiowanie funkcji

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

```
>>> def fib(n): # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     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](#).) Istnieją pewne narzędzia, które używają docstringów do automatycznego tworzenia drukowanej lub dostępnej online dokumentacji albo pozwalają użytkownikowi na interaktywne przeglądanie kodu. Dobrym zwyczajem jest pisanie 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
```

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

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): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[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, która 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.

4.7 Więcej o definiowaniu funkcji

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

4.7.1 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='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```

if retries < 0:
    raise ValueError('invalid user response')
print(reminder)

```

Tę funkcję można wywołać na kilka sposobów:

- podając tylko wymagany argument: `ask_ok('Do you really want to quit?')`
- podając jeden z opcjonalnych argumentów: `ask_ok('OK to overwrite the file?', 2)`
- lub podając wszystkie argumenty: `ask_ok('OK to overwrite file?', 2, 'Come on, only yes or no!')`

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

Wartości domyślne są ewaluowane w momencie definiowania funkcji w scope *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

```

4.7.2 Argumenty nazwane

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

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", 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 positional argument
parrot(voltage=1000)        # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM') # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000) # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

ale wszystkie poniższe wywołania byłyby niepoprawne:

```
parrot()                    # required argument missing
parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
parrot(110, voltage=220)    # duplicate value for the same argument
parrot(actor='John Cleese') # unknown keyword argument
```

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 keyword 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("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
        print(kw, ":", keywords[kw])
```

Można ją wywołać w ten sposób:

Argumenty pozycyjne-lub-kluczowe

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

Parametry Tylko-Pozycyjne

Wchodząc w szczegóły, 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 keyword-only

Aby oznaczyć parametry jako *tylko słowa kluczowe* (*keyword-only*), to znaczy wskazać, że parametry muszą być przekazywane przez argument słowa kluczowego, umieścić * 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 wywołania, 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):
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got an unexpected keyword argument 'arg'
```

Trzecia z funkcji `kwd_only_args` pozwala tylko na wprowadzanie argumentów w postaci słów kluczowych z powodu `*` w swojej definicji:

```
>>> 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 an unexpected keyword argument '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):
  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 dostępna dla użytkownika. To może być pomocne gdy nazwy parametrów 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.

4.7.4 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 i sekwencje*). Przed zmienną liczbą argumentów, można wymusić jeden lub więcej argumentów.

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

Zazwyczaj takie wariadyczne argumenty będą na koniec listy formalnych parametrów, ponieważ zbierają one wszystkie pozostałe argumenty przekazane funkcji. Każdy formalny argument po `*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("earth", "mars", "venus")
'earth/mars/venus'
>>> concat("earth", "mars", "venus", sep=".")
'earth.mars.venus'
```

4.7.5 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 funkcję z operatorem `*` aby wypakować argumenty z listy lub krotki:

```
>>> list(range(3, 6))           # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))         # call with arguments unpacked from a list
[3, 4, 5]
```

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

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print("-- This parrot wouldn't", action, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     print("E's", state, "!")
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin'
↳demised !
```

4.7.6 Wyrażenia Lambda

Niewielkie anonimowe funkcje mogą być tworzone z wykorzystaniem słowa kluczowego `lambda`. Wykorzystując tę notację: `lambda a, b: a+b` powstanie funkcja sumująca podane argumenty. Funkcje `lambda` mogą być wykorzystywane zawsze wtedy gdy potrzebne są obiekty funkcji. Są syntaktycznie 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, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.7 Napisy dokumentujące

Oto kilka konwencji dotyczących zawartości i formatowania ciągów dokumentacji.

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 nazwenicze, 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 wcięć 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():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
>>> print(my_function.__doc__)
Do nothing, but document it.

No, really, it doesn't do anything.
```

4.7.8 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. Adnotacje do zwracanego wyniku definiuje się pomiędzy nawiasem z listą parametrów a dwukropkiem 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 = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
...
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

4.8 Intermezzo: Styl kodowania

Teraz, kiedy już jesteś gotowa 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 zagnieżdżenia) a głębokim wcięciem (jest łatwiejsze do przeczytania). Tabulatorów najlepiej nie używać, wprowadzają zamieszanie.

- 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 na 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 *[A First Look at Classes](#)* aby dowiedzieć się więcej o klasach i metodach).
- Nie wykorzystuj ambitnych zestawów znaków (encoding) 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.

Ten rozdział opisuje bardziej szczegółowo niektóre rzeczy, które już poznaliście, oraz również wprowadza nowe elementy.

5.1 Więcej na temat list

Typ danych listy ma kilka więcej metod. To wszystkie metody obiektów list:

`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 którym wstawiamy, więc `a.insert(0, x)` wstawia na początek listy a `a.insert(len(a), x)` odpowiada `a.append(x)`.

`list.remove(x)`

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

`list.pop([i])`

Usuwa element z podanej pozycji na liście i zwraca go. Jeśli indeks nie jest podany, `a.pop()` usuwa i zwraca ostatni element listy. (Nawiasy kwadratowe dookoła `i` w sygnaturze metody oznaczają, że ten parametr jest opcjonalny, nie że powinieneś wpisać nawiasy kwadratowe w tym miejscu. Taką notację będziesz często widzieć w dokumentacji biblioteki Pythona.)

`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 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 = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Mogłeś(-aś) zauważyć, że metod takie jak `insert`, `remove` lub `sort`, które tylko modyfikują listę, nie mają wyświetlonej zwracanej wartości – zwracają one domyślne `None`.¹ To zasada projektowa dla wszystkich mutowalnych typów 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.

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

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()
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")           # Terry arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()                 # The first to arrive now leaves
'Eric'
>>> queue.popleft()                 # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 List comprehensions

List comprehensions 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 założmy, że chcemy stworzyć listę kwadratów, jak tu:

```
>>> squares = []
>>> for x in range(10):
...     squares.append(x**2)
... 
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> 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 zwarte 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]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1, in <module>
[x, x**2 for x in range(6)]
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```

^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> 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, zagnieżdżone 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):
...     # the following 3 lines implement the nested listcomp
...     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 powinienes(-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 `typeseq`). 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, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
... 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 = 'hello',      # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

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 = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
>>> print(basket)                # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket           # fast membership testing
True
>>> 'crabgrass' in basket
False
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a
{'a', 'r', 'b', 'c', 'd'}
>>> a - b
{'r', 'd', 'b'}
>>> a | b
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b
{'a', 'c'}
>>> a ^ b
{'r', 'd', 'b', 'm', 'z', 'l'}
```

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

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', '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 umieszysz 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 z użyciem słownika:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> 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 = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.items():
...     print(k, v)
...
gallahad the pure
robin the brave
```

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

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

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

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print('What is your {0}? It is {1}'.format(q, a))
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

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 = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print(f)
...
apple
banana
orange
pear
```

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.

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

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 wyrażenia 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 mniejsza. 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]
'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`.

Jeśli wyjdiesz z interpretera Pythona i wejdiesz 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 także 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 `fibonacci.py` w bieżącym katalogu, z następującą zawartością:

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        a, b = b, a+b
    return result
```

Wejdź w interpreter Pythona i zaimportuj ten moduł następującą komendą:

```
>>> import fibo
```

Ta komenda nie umieszcza nazw funkcji określonych w `fibo` bezpośrednio w bieżącej tabeli symboli; umieszcza 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 `import`.¹ (Są one również wykonywane, jeśli plik jest wykonywany jako skrypt.)

Każdy moduł ma swoją własną tablicę symboli, która jest używana jako globalna tablica symboli 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 są umieszczane w globalnej tablicy symboli modułu importującego.

Istnieje wariant instrukcji `import`, który importuje nazwy z modułu bezpośrednio do tablicy symboli 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 tablicy symboli (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 (`_`). W większości przypadków programiści Pythona nie używają tego udogodnienia, ponieważ wprowadza ono nieznany zestaw nazw do interpretera, być może ukrywając niektóre już zdefiniowane rzeczy.

¹ In fact function definitions are also «statements» that are «executed»; the execution of a module-level function definition enters the function name in the module's global symbol table.

Zauważ, 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 modulem.

```
>>> 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 komendy 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. 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`).
- The installation-dependent default.

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 zawierający 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(-naś) 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`. Jedyną rzecz, która jest szybsza w plikach `.pyc` to szybkość, w jakiej są one ładowane.
- The module `compileall` can create `.pyc` files for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in [PEP 3147](#).

6.2 Moduły standardowe

Python comes with a library of standard modules, described in a separate document, the Python Library Reference („Library Reference” hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print('Yuck!')
Yuck!
C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter’s search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

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

6.3 The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__loader__', '__name__',
['__package__', '__stderr__', '__stdin__', '__stdout__',
['__clear_type_cache__', '__current_frames__', '__debugmallocstats__', '__getframe__',
['__home__', '__mercurial__', '__xoptions__', 'abiflags', 'api_version', 'argv',
'base_exec_prefix', 'base_prefix', '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',
'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',
'thread_info', 'version', 'version_info', 'warnoptions']
```

Without arguments, `dir()` lists the names you have defined currently:

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

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `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 Packages

Packages are a way of structuring Python's module namespace by using „dotted module names”. For example, the module name `A.B` designates a submodule named `B` in a package named `A`. Just like the use of modules saves the authors of different modules from having to worry about each other's global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or Pillow from having to worry about each other's module names.

Suppose you want to design a collection of modules (a „package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: `.wav`, `.aiff`, `.au`), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo,

applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here's a possible structure for your package (expressed in terms of a hierarchical filesystem):

sound/	Top-level package
__init__.py	Initialize the sound package
formats/	Subpackage for file format conversions
__init__.py	
wavread.py	
wavwrite.py	
aiffread.py	
aiffwrite.py	
auread.py	
auwrite.py	
...	
effects/	Subpackage for sound effects
__init__.py	
echo.py	
surround.py	
reverse.py	
...	
filters/	Subpackage for filters
__init__.py	
equalizer.py	
vocoder.py	
karaoke.py	
...	

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat directories containing the file as packages. This prevents directories with a common name, such as `string`, unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

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

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

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

An alternative way of importing the submodule is:

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

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

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

Yet another variation is to import the desired function or variable directly:

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

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

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

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an `ImportError` exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

6.4.1 Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the sub-module is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing `*` from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

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

This would mean that `from sound.effects import *` would import the three named submodules of the `sound` package.

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package `sound.effects` into the current namespace; it only ensures that the package `sound.effects` has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider this code:

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

In this example, the `echo` and `surround` modules are imported in the current namespace because they are defined in the `sound.effects` package when the `from . import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using `from package import specific_submodule`! In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

6.4.2 Intra-package References

When packages are structured into subpackages (as with the `sound` package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module `sound.filters.vocoder` needs to use the `echo` module in the `sound.effects` package, it can use `from sound.effects import echo`.

You can also write relative imports, with the `from module import name` form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

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

Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use absolute imports.

6.4.3 Packages in Multiple Directories

Packages support one more special attribute, `__path__`. This is initialized to be a list containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

7.1 Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the `print()` function. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use *formatted string literals*, begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The `str.format()` method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Trochę przykładów:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs'))))
'(32.5, 40000, ('spam', 'eggs'))'
```

The `string` module contains a `Template` class that offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary, but offers much less control of the formatting.

7.1.1 Formatted String Literals

Formatted string literals (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds `pi` to three places after the decimal:

```
>>> import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```
>>> 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
```

Other modifiers can be used to convert the value before it is formatted. `'!a'` applies `ascii()`, `'!s'` applies `str()`, and `'!r'` applies `repr()`:

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

For a reference on these format specifications, see the reference guide for the `formatspec`.

7.1.2 The String `format()` Method

Basic usage of the `str.format()` method looks like this:

```
>>> print('We are the {} who say "{}!".format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

```
>>> print('This {food} is {adjective}'.format(
...     food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred',
...                                                other='Georg'))
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets `'[]'` to access the keys.

```
>>> 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
```

This could also be done by passing the table as keyword arguments with the `«**»` notation.

```
>>> 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
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables.

As an example, the following lines produce a tidily-aligned set of columns giving integers and their squares and cubes:

```
>>> 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
```

For a complete overview of string formatting with `str.format()`, see [formatstrings](#).

7.1.3 Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     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
```

(Note that the one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

The `str.rjust()` method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, `str.zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

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

7.1.4 Old string formatting

The `%` operator (modulo) can also be used for string formatting. Given `'string' % values`, instances of `%` in `string` are replaced with zero or more elements of `values`. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

More information can be found in the [old-string-formatting](#) section.

7.2 Reading and Writing Files

`open()` returns a *file object*, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding. If encoding is not specified, the default is platform dependent (see `open()`). `'b'` appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try-finally` blocks:

```
>>> with open('workfile') as f:
...     read_data = f.read()

>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it.

Ostrzeżenie: Calling `f.write()` without using the `with` keyword or calling `f.close()` **might** result in the arguments of `f.write()` not being completely written to the disk, even if the program exits successfully.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> 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 Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* characters (in text mode) or *size* bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` will return an empty string ('').

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by '`\n`', a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use `f.seek(offset, whence)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *whence* argument. A *whence* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *whence* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5) # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid *offset* values are those returned from the `f.tell()`, or zero. Any other *offset* value produces undefined behaviour.

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

7.2.2 Saving structured data with json

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called **JSON (JavaScript Object Notation)**. The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

Informacja: The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a *text file*. So if `f` is a *text file* object opened for writing, we can do this:

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

To decode the object again, if `f` is a *text file* object which has been opened for reading:

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

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of this.

Zobacz także:

`pickle` - the `pickle` module

Contrary to *JSON*, *pickle* is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing `pickle` data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.

Do tej pory wiadomości o błędach były tylko wspomniane, ale jeśli próbowałeś 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, inaczej zwane błędami parsowania, to najczęstsze skargi jakie otrzymasz w swoim kierunku gdy wciąż uczysz się Pythona

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

The parser repeats the offending line and displays a little «arrow» pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function `print()`, since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2 Sytuacje Wyjątkowe

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>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are `ZeroDivisionError`, `NameError` and `TypeError`. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

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

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

`bltin-exceptions` lists the built-in exceptions and their meanings.

8.3 Obsługa wyjątków

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using `Control-C` or whatever the operating system supports); note that a user-generated interruption is signalled by raising the `KeyboardInterrupt` exception.

```
>>> while True:
...     try:
...         x = int(input("Please enter a number: "))
...         break
...     except ValueError:
...         print("Oops! That was no valid number. Try again...")
... 
```

The `try` statement works as follows.

- First, the *try clause* (the statement(s) between the `try` and `except` keywords) is executed.
- If no exception occurs, the *except clause* is skipped and execution of the `try` statement is finished.
- If an exception occurs during execution of the `try` clause, the rest of the clause is skipped. Then if its type matches the exception named after the `except` keyword, the `except` clause is executed, and then execution continues after the `try` statement.
- If an exception occurs which does not match the exception named in the `except` clause, it is passed on to outer `try` statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A `try` statement may have more than one `except` clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding `try` clause, not in other handlers of the same `try` statement. An `except` clause may name multiple exceptions as a parenthesized tuple, for example:


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

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around — an `except` clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

```
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")
```

Note that if the `except` clauses were reversed (with `except B` first), it would have printed B, B, B — the first matching `except` clause is triggered.

The last `except` clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except OSError as err:
    print("OS error: {0}".format(err))
except ValueError:
    print("Could not convert data to an integer.")
except:
    print("Unexpected error:", sys.exc_info()[0])
    raise
```

The `try ... except` statement has an optional *else clause*, which, when present, must follow all `except` clauses. It is useful for code that must be executed if the `try` clause does not raise an exception. For example:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
```

(ciąg dalszy na następnej stronie)

```
f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The `except` clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> try:
...     raise Exception('spam', 'eggs')
... except Exception as inst:
...     print(type(inst))    # the exception instance
...     print(inst.args)    # arguments stored in .args
...     print(inst)         # __str__ allows args to be printed directly,
...                           # but may be overridden in exception subclasses
...     x, y = inst.args    # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

If an exception has arguments, they are printed as the last part («detail») of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the `try` clause, but also if they occur inside functions that are called (even indirectly) in the `try` clause. For example:

```
>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division by zero
```

8.4 Rzucanie wyjątków

The `raise` statement allows the programmer to force a specified exception to occur. For example:

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

The sole argument to `raise` indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from `Exception`). If an exception class is passed, it will be implicitly instantiated by calling its constructor with no arguments:

```
raise ValueError # shorthand for 'raise ValueError()'
```

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the `raise` statement allows you to re-raise the exception:

```
>>> 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>
NameError: HiThere
```

8.5 Wyjątki zdefiniowane przez użytkownika

Programs may name their own exceptions by creating a new exception class (see *Klasy* for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
        self.expression = expression
        self.message = message

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """
```

(ciąg dalszy na następnej stronie)

```
def __init__(self, previous, next, message):
    self.previous = previous
    self.next = next
    self.message = message
```

Most exceptions are defined with names that end in „Error”, similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter *Klasy*.

8.6 Defining Clean-up Actions

The `try` statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

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

If a `finally` clause is present, the `finally` clause will execute as the last task before the `try` statement completes. The `finally` clause runs whether or not the `try` statement produces an exception. The following points discuss more complex cases when an exception occurs:

- If an exception occurs during execution of the `try` clause, the exception may be handled by an `except` clause. If the exception is not handled by an `except` clause, the exception is re-raised after the `finally` clause has been executed.
- An exception could occur during execution of an `except` or `else` clause. Again, the exception is re-raised after the `finally` clause has been executed.
- If the `try` statement reaches a `break`, `continue` or `return` statement, the `finally` clause will execute just prior to the `break`, `continue` or `return` statement's execution.
- If a `finally` clause includes a `return` statement, the returned value will be the one from the `finally` clause's `return` statement, not the value from the `try` clause's `return` statement.

Dla przykładu:

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

A more complicated example:

```

>>> 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
>>> 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>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'

```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

In real world applications, the `finally` clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

8.7 Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```

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

```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```

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

```

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

Klasy dostarczają sposoby powiązania informacji i funkcjonalności. Stworzenie nowej klasy, tworzy nowy typ obiektu, pozwalający na tworzenie nowych instancji tego typu. Każda instancja klasy może mieć atrybuty przypisane w celu utrzymania jej stanu. Instancje klas mogą również posiadać metody (zdefiniowane przez jej klasę) w celu modyfikacji 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 tworzenie wielu bazowych klas, pochodne klasy mogą nadpisać każdą metodę klasy lub klas bazowej i metoda może przywołać metody klas bazowych o tej samej nazwie. Obiekty mogą zawierać dowolną ilość i rodzaj danych. Zarówno klasy jak i moduły są częścią dynamicznej natury Python'a: 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 *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 wyrażen 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) mogą być zdefiniowane przez instancje klasy.

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

9.1 Kilka słów o nazwach i obiektach

Objects have individuality, and multiple names (in multiple scopes) can be bound to the same object. This is known as aliasing in other languages. This is usually not appreciated on a first glance at Python, and can be safely ignored when dealing with immutable basic types (numbers, strings, tuples). However, aliasing has a possibly surprising effect on the semantics of Python code involving mutable objects such as lists, dictionaries, and most other types. This is usually used to the benefit of the program, since aliases behave like pointers in some respects. For example, passing an object is cheap since only a pointer is passed by the implementation; and if a function modifies an object passed as an argument, the caller will see the change — this eliminates the need for two different argument passing mechanisms as in Pascal.

9.2 Python Scopes and Namespaces

Before introducing classes, I first have to tell you something about Python's scope rules. Class definitions play some neat tricks with namespaces, and you need to know how scopes and namespaces work to fully understand what's going on. Incidentally, knowledge about this subject is useful for any advanced Python programmer.

Let's begin with some definitions.

A *namespace* is a mapping from names to objects. Most namespaces are currently implemented as Python dictionaries, but that's normally not noticeable in any way (except for performance), and it may change in the future. Examples of namespaces are: the set of built-in names (containing functions such as `abs()`, and built-in exception names); the global names in a module; and the local names in a function invocation. In a sense the set of attributes of an object also form a namespace. The important thing to know about namespaces is that there is absolutely no relation between names in different namespaces; for instance, two different modules may both define a function `maximize` without confusion — users of the modules must prefix it with the module name.

By the way, I use the word *attribute* for any name following a dot — for example, in the expression `z.real`, `real` is an attribute of the object `z`. Strictly speaking, references to names in modules are attribute references: in the expression `modname.funcname`, `modname` is a module object and `funcname` is an attribute of it. In this case there happens to be a straightforward mapping between the module's attributes and the global names defined in the module: they share the same namespace!¹

Attributes may be read-only or writable. In the latter case, assignment to attributes is possible. Module attributes are writable: you can write `modname.the_answer = 42`. Writable attributes may also be deleted with the `del` statement. For example, `del modname.the_answer` will remove the attribute `the_answer` from the object named by `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `builtins`.)

The local namespace for a function is created when the function is called, and deleted when the function returns or raises an exception that is not handled within the function. (Actually, forgetting would be a better way to describe what actually happens.) Of course, recursive invocations each have their own local namespace.

A *scope* is a textual region of a Python program where a namespace is directly accessible. „Directly accessible” here means that an unqualified reference to a name attempts to find the name in the namespace.

Although scopes are determined statically, they are used dynamically. At any time during execution, there are 3 or 4 nested scopes whose namespaces are directly accessible:

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.

- the innermost scope, which is searched first, contains the local names
- the scopes of any enclosing functions, which are searched starting with the nearest enclosing scope, contains non-local, but also non-global names
- the next-to-last scope contains the current module's global names
- the outermost scope (searched last) is the namespace containing built-in names

If a name is declared `global`, then all references and assignments go directly to the middle scope containing the module's global names. To rebind variables found outside of the innermost scope, the `nonlocal` statement can be used; if not declared `nonlocal`, those variables are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Usually, the local scope references the local names of the (textually) current function. Outside functions, the local scope references the same namespace as the global scope: the module's namespace. Class definitions place yet another namespace in the local scope.

It is important to realize that scopes are determined textually: the global scope of a function defined in a module is that module's namespace, no matter from where or by what alias the function is called. On the other hand, the actual search for names is done dynamically, at run time — however, the language definition is evolving towards static name resolution, at „compile“ time, so don't rely on dynamic name resolution! (In fact, local variables are already determined statically.)

A special quirk of Python is that – if no `global` or `nonlocal` statement is in effect – assignments to names always go into the innermost scope. Assignments do not copy data — they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope.

The `global` statement can be used to indicate that particular variables live in the global scope and should be rebound there; the `nonlocal` statement indicates that particular variables live in an enclosing scope and should be rebound there.

9.2.1 Scopes and Namespaces Example

This is an example demonstrating how to reference the different scopes and namespaces, and how `global` and `nonlocal` affect variable binding:

```
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)
```

The output of the example code is:

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

Note how the *local* assignment (which is default) didn't change *scope_test*'s binding of *spam*. The *nonlocal* assignment changed *scope_test*'s binding of *spam*, and the *global* assignment changed the module-level binding.

You can also see that there was no previous binding for *spam* before the *global* assignment.

9.3 A First Look at Classes

Classes introduce a little bit of new syntax, three new object types, and some new semantics.

9.3.1 Class Definition Syntax

The simplest form of class definition looks like this:

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

Class definitions, like function definitions (`def` statements) must be executed before they have any effect. (You could conceivably place a class definition in a branch of an `if` statement, or inside a function.)

In practice, the statements inside a class definition will usually be function definitions, but other statements are allowed, and sometimes useful — we'll come back to this later. The function definitions inside a class normally have a peculiar form of argument list, dictated by the calling conventions for methods — again, this is explained later.

When a class definition is entered, a new namespace is created, and used as the local scope — thus, all assignments to local variables go into this new namespace. In particular, function definitions bind the name of the new function here.

When a class definition is left normally (via the `end`), a *class object* is created. This is basically a wrapper around the contents of the namespace created by the class definition; we'll learn more about class objects in the next section. The original local scope (the one in effect just before the class definition was entered) is reinstated, and the class object is bound here to the class name given in the class definition header (`ClassName` in the example).

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
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
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. (In Python, the term method is not unique to class instances: other object types can have methods as well. For example, list objects

have methods called `append`, `insert`, `remove`, `sort`, and so on. However, in the following discussion, we'll use the term method exclusively to mean methods of class instance objects, unless explicitly stated otherwise.)

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.

If you still don't understand how methods work, a look at the implementation can perhaps clarify matters. 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, a method object is created by packing (pointers to) the instance object and the function object just found together in an abstract object: this is the 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'
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> 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 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 <https://www.python.org/download/releases/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.

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 = []
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```

        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. An empty class definition will do nicely:

```

class Employee:
    pass

john = Employee()    # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
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
<iterator object at 0x00A1DB50>
>>> 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."""
    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
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
>>> 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']
```

Krótką wycieczka po Bibliotece Standardowej

10.1 Interfejs Systemu Operacyjnego

The `os` module provides dozens of functions for interacting with the operating system:

```
>>> import os
>>> os.getcwd()          # Return the current working directory
'C:\\Python38'
>>> os.chdir('/server/accesslogs')  # Change current working directory
>>> os.system('mkdir today')  # Run the command mkdir in the system shell
0
```

Be sure to use the `import os` style instead of `from os import *`. This will keep `os.open()` from shadowing the built-in `open()` function which operates much differently.

The built-in `dir()` and `help()` functions are useful as interactive aids for working with large modules like `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

For daily file and directory management tasks, the `shutil` module provides a higher level interface that is easier to use:

```
>>> import shutil
>>> shutil.copyfile('data.db', 'archive.db')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'installdir'
```

10.2 File Wildcards

The `glob` module provides a function for making file lists from directory wildcard searches:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3 Command Line Arguments

Common utility scripts often need to process command line arguments. These arguments are stored in the `sys` module's `argv` attribute as a list. For instance the following output results from running `python demo.py one two three` at the command line:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'one', 'two', 'three']
```

The `argparse` module provides a more sophisticated mechanism to process command line arguments. The following script extracts one or more filenames and an optional number of lines to be displayed:

```
import argparse

parser = argparse.ArgumentParser(prog = 'top',
                                description = 'Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

When run at the command line with `python top.py --lines=5 alpha.txt beta.txt`, the script sets `args.lines` to 5 and `args.filenames` to `['alpha.txt', 'beta.txt']`.

10.4 Error Output Redirection and Program Termination

The `sys` module also has attributes for `stdin`, `stdout`, and `stderr`. The latter is useful for emitting warnings and error messages to make them visible even when `stdout` has been redirected:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

The most direct way to terminate a script is to use `sys.exit()`.

10.5 String Pattern Matching

The `re` module provides regular expression tools for advanced string processing. For complex matching and manipulation, regular expressions offer succinct, optimized solutions:

```
>>> import re
>>> re.findall(r'\b[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'
```

When only simple capabilities are needed, string methods are preferred because they are easier to read and debug:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Mathematics

The `math` module gives access to the underlying C library functions for floating point math:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

The `random` module provides tools for making random selections:

```
>>> 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
0.17970987693706186
>>> random.randrange(6)             # random integer chosen from range(6)
4
```

The `statistics` module calculates basic statistical properties (the mean, median, variance, etc.) of numeric data:

```
>>> 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
```

The SciPy project <<https://scipy.org>> has many other modules for numerical computations.

10.7 Internet Access

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib.request` for retrieving data from URLs and `smtplib` for sending mail:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8') # Decoding the binary data to text.
...         if 'EST' in line or 'EDT' in line: # look for Eastern Time
...             print(line)

<BR>Nov. 25, 09:43:32 PM EST

>>> 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()
```

(Note that the second example needs a mailserver running on localhost.)

10.8 Dates and Times

The `datetime` module supplies classes for manipulating dates and times in both simple and complex ways. While date and time arithmetic is supported, the focus of the implementation is on efficient member extraction for output formatting and manipulation. The module also supports objects that are timezone aware.

```
>>> # 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 Performance Measurement

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)
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
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 ująć w zaawansowanych możliwościach jego większych package’y. 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 `smtpplib` 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.

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
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> print(textwrap.fill(doc, width=40))
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("%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

>>> 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)
        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:/python38/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):
        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

>>> sum([Decimal('0.1')] * 10) == Decimal('1.0')
True
>>> sum([0.1] * 10) == 1.0
False
```

The `decimal` module provides arithmetic with as much precision as needed:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

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 jest nazwane `venv`. `venv` zwykle instaluje najnowszą dostępną wersję Pythona. Jeśli masz w swoim systemie wiele wersji Pythona, możesz wybrać konkretną wersję Pythona, uruchamiając `python3` lub dowolną wersję którą sobie wybierzesz.

Aby stworzyć środowisko wirtualne, wybierz katalog, w którym chcesz je umieścić i uruchom moduł `venv` jako skrypt ze ścieżką do katalogu:

```
python3 -m venv tutorial-env
```

This will create the `tutorial-env` directory if it doesn't exist, and also create directories inside it containing a copy of the Python interpreter, the standard library, and various supporting files.

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.bat
```

Na systemie Unix albo MacOS, uruchom:

```
source tutorial-env/bin/activate
```

(This script is written for the bash shell. If you use the **csh** or **fish** shells, there are alternate `activate.csh` and `activate.fish` scripts you should use instead.)

Activating the virtual environment will change your shell's prompt to show what virtual environment you're using, and modify the environment so that running `python` will get you that particular version and installation of Python. For example:

```
$ 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']
>>>
```

12.3 Managing Packages with pip

You can install, upgrade, and remove packages using a program called **pip**. By default `pip` will install packages from the Python Package Index, <<https://pypi.org>>. You can browse the Python Package Index by going to it in your web browser.

`pip` has a number of subcommands: „install”, „uninstall”, „freeze”, etc. (Consult the installing-index guide for complete documentation for `pip`.)

You can install the latest version of a package by specifying a package's name:

```
(tutorial-env) $ 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
```

You can also install a specific version of a package by giving the package name followed by `==` and the version number:

```
(tutorial-env) $ 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
```

If you re-run this command, `pip` will notice that the requested version is already installed and do nothing. You can supply a different version number to get that version, or you can run `pip install --upgrade` to upgrade the package to the latest version:

```
(tutorial-env) $ 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
```

`pip uninstall` followed by one or more package names will remove the packages from the virtual environment.

`pip show` will display information about a particular package:

```
(tutorial-env) $ 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:
```

`pip list` will display all of the packages installed in the virtual environment:

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`pip freeze` will produce a similar list of the installed packages, but the output uses the format that `pip install` expects. A common convention is to put this list in a `requirements.txt` file:

```
(tutorial-env) $ pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

The `requirements.txt` can then be committed to version control and shipped as part of an application. Users can then install all the necessary packages with `install -r`:

```
(tutorial-env) $ 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
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```
Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` has many more options. Consult the [installing-index](#) guide for complete documentation for `pip`. When you've written a package and want to make it available on the Python Package Index, consult the [distributing-index](#) guide.

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>: The major Python Web site. It contains code, documentation, and pointers to Python-related pages around the Web. This Web site is mirrored in various places around the world, such as Europe, Japan, and Australia; a mirror may be faster than the main site, depending on your geographical location.
- <https://docs.python.org>: Szybki dostęp do dokumentacji Pythona.
- <https://pypi.org>: The Python Package Index, previously also nicknamed the Cheese Shop¹, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.
- <https://code.activestate.com/recipes/langs/python/>: The Python Cookbook is a sizable collection of code examples, larger modules, and useful scripts. Particularly notable contributions are collected in a book also titled Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

¹ „Cheese Shop” is a Monty Python's sketch: a customer enters a cheese shop, but whatever cheese he asks for, the clerk says it's missing.

- <http://www.pyvideo.org> zbiera linki do filmów związanych z Pythonem z konferencji oraz spotkań grup użytkowników.
- <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.

For Python-related questions and problem reports, you can post to the newsgroup *comp.lang.python*, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are hundreds of postings a day, asking (and answering) questions, suggesting new features, and announcing new modules. Mailing list archives are available at <https://mail.python.org/pipermail/>.

Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ). The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

Interactive Input Editing and History Substitution

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the [GNU Readline](#) library, which supports various styles of editing. This library has its own documentation which we won't duplicate here.

14.1 Tab Completion and History Editing

Completion of variable and module names is automatically enabled at interpreter startup so that the `Tab` key invokes the completion function; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression. The default configuration also saves your history into a file named `.python_history` in your user directory. The history will be available again during the next interactive interpreter session.

14.2 Alternatives to the Interactive Interpreter

This facility is an enormous step forward compared to earlier versions of the interpreter; however, some wishes are left: It would be nice if the proper indentation were suggested on continuation lines (the parser knows if an indent token is required next). The completion mechanism might use the interpreter's symbol table. A command to check (or even suggest) matching parentheses, quotes, etc., would also be useful.

One alternative enhanced interactive interpreter that has been around for quite some time is [IPython](#), which features tab completion, object exploration and advanced history management. It can also be thoroughly customized and embedded into other applications. Another similar enhanced interactive environment is [bpython](#).

Arytmetyka liczb zmiennoprzecinkowych: Problemy i ograniczenia

Liczby zmiennoprzecinkowe są reprezentowane w systemie komputerowym jako ułamki o podstawie 2 (binarne). Na przykład, ułamek dziesiętny

0.125

has value $1/10 + 2/100 + 5/1000$, and in the same way the binary fraction

0.001

has value $0/2 + 0/4 + 1/8$. These two fractions have identical values, the only real difference being that the first is written in base 10 fractional notation, and the second in base 2.

Unfortunately, most decimal fractions cannot be represented exactly as binary fractions. A consequence is that, in general, the decimal floating-point numbers you enter are only approximated by the binary floating-point numbers actually stored in the machine.

The problem is easier to understand at first in base 10. Consider the fraction $1/3$. You can approximate that as a base 10 fraction:

0.3

or, better,

0.33

or, better,

0.333

and so on. No matter how many digits you're willing to write down, the result will never be exactly $1/3$, but will be an increasingly better approximation of $1/3$.

In the same way, no matter how many base 2 digits you're willing to use, the decimal value 0.1 cannot be represented exactly as a base 2 fraction. In base 2, $1/10$ is the infinitely repeating fraction

```
0.00011001100110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation. On most machines today, floats are approximated using a binary fraction with the numerator using the first 53 bits starting with the most significant bit and with the denominator as a power of two. In the case of $1/10$, the binary fraction is $3602879701896397 / 2^{55}$ which is close to but not exactly equal to the true value of $1/10$.

Many users are not aware of the approximation because of the way values are displayed. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. On most machines, if Python were to print the true decimal value of the binary approximation stored for 0.1 , it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

That is more digits than most people find useful, so Python keeps the number of digits manageable by displaying a rounded value instead

```
>>> 1 / 10
0.1
```

Just remember, even though the printed result looks like the exact value of $1/10$, the actual stored value is the nearest representable binary fraction.

Interestingly, there are many different decimal numbers that share the same nearest approximate binary fraction. For example, the numbers 0.1 and 0.10000000000000001 and $0.1000000000000000055511151231257827021181583404541015625$ are all approximated by $3602879701896397 / 2^{55}$. Since all of these decimal values share the same approximation, any one of them could be displayed while still preserving the invariant `eval(repr(x)) == x`.

Historically, the Python prompt and built-in `repr()` function would choose the one with 17 significant digits, 0.10000000000000001 . Starting with Python 3.1, Python (on most systems) is now able to choose the shortest of these and simply display 0.1 .

Note that this is in the very nature of binary floating-point: this is not a bug in Python, and it is not a bug in your code either. You'll see the same kind of thing in all languages that support your hardware's floating-point arithmetic (although some languages may not *display* the difference by default, or in all output modes).

For more pleasant output, you may wish to use string formatting to produce a limited number of significant digits:

```
>>> 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'
```

It's important to realize that this is, in a real sense, an illusion: you're simply rounding the *display* of the true machine value.

One illusion may beget another. For example, since 0.1 is not exactly $1/10$, summing three values of 0.1 may not yield exactly 0.3 , either:

```
>>> .1 + .1 + .1 == .3
False
```

Also, since the 0.1 cannot get any closer to the exact value of $1/10$ and 0.3 cannot get any closer to the exact value of $3/10$, then pre-rounding with `round()` function cannot help:

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Though the numbers cannot be made closer to their intended exact values, the `round()` function can be useful for post-rounding so that results with inexact values become comparable to one another:

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

Binary floating-point arithmetic holds many surprises like this. The problem with „0.1” is explained in precise detail below, in the „Representation Error” section. See [The Perils of Floating Point](#) for a more complete account of other common surprises.

As that says near the end, „there are no easy answers.” Still, don’t be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2^{53} per operation. That’s more than adequate for most tasks, but you do need to keep in mind that it’s not decimal arithmetic and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you’ll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. `str()` usually suffices, and for finer control see the `str.format()` method’s format specifiers in formatstrings.

For use cases which require exact decimal representation, try using the `decimal` module which implements decimal arithmetic suitable for accounting applications and high-precision applications.

Another form of exact arithmetic is supported by the `fractions` module which implements arithmetic based on rational numbers (so the numbers like $1/3$ can be represented exactly).

If you are a heavy user of floating point operations you should take a look at the Numerical Python package and many other packages for mathematical and statistical operations supplied by the SciPy project. See <https://scipy.org>.

Python provides tools that may help on those rare occasions when you really *do* want to know the exact value of a float. The `float.as_integer_ratio()` method expresses the value of a float as a fraction:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Since the ratio is exact, it can be used to losslessly recreate the original value:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

The `float.hex()` method expresses a float in hexadecimal (base 16), again giving the exact value stored by your computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

This precise hexadecimal representation can be used to reconstruct the float value exactly:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Since the representation is exact, it is useful for reliably porting values across different versions of Python (platform independence) and exchanging data with other languages that support the same format (such as Java and C99).

Another helpful tool is the `math.fsum()` function which helps mitigate loss-of-precision during summation. It tracks „lost digits” as values are added onto a running total. That can make a difference in overall accuracy so that the errors do

not accumulate to the point where they affect the final total:

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

15.1 Representation Error

This section explains the „0.1” example in detail, and shows how you can perform an exact analysis of cases like this yourself. Basic familiarity with binary floating-point representation is assumed.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect.

Why is that? $1/10$ is not exactly representable as a binary fraction. Almost all machines today (November 2000) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 „double precision”. 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form $J/2^N$ where J is an integer containing exactly 53 bits. Rewriting

```
1 / 10 ~ J / (2**N)
```

as

```
J ~ 2**N / 10
```

and recalling that J has exactly 53 bits (is $\geq 2^{52}$ but $< 2^{53}$), the best value for N is 56:

```
>>> 2**52 <= 2**56 // 10 < 2**53
True
```

That is, 56 is the only value for N that leaves J with exactly 53 bits. The best possible value for J is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Since the remainder is more than half of 10, the best approximation is obtained by rounding up:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to $1/10$ in 754 double precision is:

```
7205759403792794 / 2 ** 56
```

Dividing both the numerator and denominator by two reduces the fraction to:

```
3602879701896397 / 2 ** 55
```

Note that since we rounded up, this is actually a little bit larger than $1/10$; if we had not rounded up, the quotient would have been a little bit smaller than $1/10$. But in no case can it be *exactly* $1/10$!

So the computer never „sees” $1/10$: what it sees is the exact fraction given above, the best 754 double approximation it can get:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

If we multiply that fraction by 10^{55} , we can see the value out to 55 decimal digits:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
100000000000000000055511151231257827021181583404541015625
```

meaning that the exact number stored in the computer is equal to the decimal value 0.10000000000000000055511151231257827021181583404541015625. Instead of displaying the full decimal value, many languages (including older versions of Python), round the result to 17 significant digits:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

The fractions and decimal modules make these calculations easy:

```
>>> 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('0.10000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17f')
'0.10000000000000001'
```


16.1 Tryb interaktywny

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 przerywania (zazwyczaj `Control-C` lub `Delete`) w pierwszorzędny lub drugorzędny prompt anuluje tryb wpisywania i wraca do pierwszorzędnego prompta.¹ Wpisanie znaku przerywania podczas wykonywania komendy wywołuje wyjątek `KeyboardInterrupt`, który może być obsługiwany 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 python3.5
```

(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`.

¹ Problem z pakietem GNU Readline może w tym przeszkodzić.

```
$ 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, 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.

If you want to read an additional start-up file from the current directory, you can program this in the global start-up file using code like `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. If you want to use the startup file in a script, you must do this explicitly in the script:

```
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 The Customization Modules

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/python3.5/site-packages'
```

Now you can create a file named `usercustomize.py` in that directory and put anything you want in it. It will affect every invocation of Python, unless it is started with the `-s` option to disable the automatic 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.

„»>” Domyślny znak zachęty powłoki interaktywnej w języku Python. Często spotykane w przypadku przykładów kodu, które mogą być wykonywane w interpreterze.

... Może służyć do:

- Domyślny znak zachęty powłoki interaktywnej Pythona przy wpisywaniu kodu dla wciętego bloku kodu, wewnątrz pary odpowiadających sobie ograniczników (nawiasów, nawiasów kwadratowych, nawiasów klamrowych lub potrójnych cudzysłówów) lub po użyciu dekoratora.
- Wbudowana stała `Ellipsis`.

2to3 A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See 2to3-reference.

abstract base class 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.

annotation A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a *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.

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](#).

attribut A value associated with an object which is referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

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.

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.

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).

coercion The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3, but in `3+4.5`, each argument is of a different type (one `int`, one `float`), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

complex number 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(...):  
    ...  
f = staticmethod(f)  
  
@staticmethod  
def f(...):  
    ...
```

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](#).

dictionary 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 [statements](#) 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*.

finder An object that tries to find the *loader* for a module that is being imported.

Since Python 3.3, 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 [PEP 302](#), [PEP 420](#) and [PEP 451](#) 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](#).

function 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.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter.

By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it becomes 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](#).

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.

Past efforts to create a „free-threaded” interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

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 Environment for Python. IDLE is a basic editor and interpreter environment which ships with the standard distribution of Python.

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)`).

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](#).

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, the `operator` module provides three key function constructors: `attrgetter()`, `itemgetter()`, and `methodcaller()`. 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 [PEP 302](#) for details and `importlib.abc.Loader` for an *abstract base class*.

magic method An informal synonym for *special method*.

mapping A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `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 [The Python 2.3 Method Resolution Order](#) for details of the algorithm used by the Python interpreter since the 2.3 release.

module 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`.

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 with the factory function `collections.namedtuple()`. The latter technique also adds 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, `__getattr__()`, class methods, and static methods.

object Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

package A Python [module](#) which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__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_hook` 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. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. The `sys` module defines a `getrefcount()` function that programmers can call 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*.

__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 *immutable* 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()`.

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.

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`.

statement 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`.

text encoding A codec which encodes Unicode strings to bytes.

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:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

could be made more readable like this:

```
from typing import List, Tuple

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 analysis tools, and 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.

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.

O tej dokumentacji

Dokumenty są wygenerowane ze źródeł [reStructuredText](#) przez [Sphinksa](#), procesor dokumentów napisany specjalnie dla dokumentacji Pythona.

Rozwój dokumentacji i jej oprządkowanie jest w całości wysiłkiem wolontarystycznym, 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 wzięło wiele dobrych pomysłów.

B.1 Współtwórcy dokumentacji Pythona

Wiele 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!

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 Pythonowskiego (PSF)	nie
2.0.1	2.0 i 1.6.1	2001	Fundacja Programu języka Pythonowskiego (PSF)	tak
2.1.1	2.1 i 2.0.1	2001	Fundacja Programu języka Pythonowskiego (PSF)	tak
2.1.2	2.1.1	2002	Fundacja Programu języka Pythonowskiego (PSF)	tak
2.1.3	2.1.2	2002	Fundacja Programu języka Pythonowskiego (PSF)	tak
2.2 and above	2.1.1	2001-now	Fundacja Programu języka Pythonowskiego (PSF)	tak

Informacja: Zgodność z uprawnieniami ogólnie powszechnymi (w skrócie - z ang. - GPL) nie oznacza, że rozprowadzamy język pythonowski z uprawnieniami ogólnie powszechnymi (w skrócie - z ang. - GPL). Wszystkie uprawnienia dostarczane z językiem pythonowskim, w przeciwieństwie do uprawnień ogólnie powszechnych (w skrócie - z ang. - GPL), pozwalają na rozpowszechnianie programów języka pythonowskiego 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 pythonowskiego 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 pythonowskiego.

C.2 Zasady i warunki postępowania z programem języka pythonowskiego 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 3.8.20

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),
 and
 the Individual or Organization ("Licensee") accessing and otherwise using
 Python
 3.8.20 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 3.8.20 alone or in any derivative
 version, provided, however, that PSF's License Agreement and PSF's notice
 of
 copyright, i.e., "Copyright © 2001-2023 Python Software Foundation; All
 Rights
 Reserved" are retained in Python 3.8.20 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 3.8.20 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
 3.8.20.
4. PSF is making Python 3.8.20 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 3.8.20 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.20
 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT
 OF
 MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.20, 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 3.8.20, 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.
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.
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.

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

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 3.8.20 DOCUMENTATION

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

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` 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.

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, <http://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
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

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 `asynchat` and `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 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
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 module 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.

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
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

C.3.8 test_epoll

The `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 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

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

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's 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/siphhash24/little)
    djb (supercop/crypto_auth/siphhash24/little2)
    Jean-Philippe Aumasson (https://131002.net/siphhash/siphhash24.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 <http://www.netlib.org/fp/>. 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
 * OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
 */
```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```

*
*****/

```

C.3.12 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```

LICENSE ISSUES
=====

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of
the OpenSSL License and the original SSLeay license apply to the toolkit.
See below for the actual license texts. Actually both licenses are BSD-style
Open Source licenses. In case of any license issues related to OpenSSL
please contact openssl-core@openssl.org.

OpenSSL License
-----

/* =====
 * Copyright (c) 1998-2008 The OpenSSL 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. All advertising materials mentioning features or use of this
 * software must display the following acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 * endorse or promote products derived from this software without
 * prior written permission. For written permission, please contact
 * openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 * nor may "OpenSSL" appear in their names without prior written
 * permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 * acknowledgment:
 * "This product includes software developed by the OpenSSL Project
 * for use in the OpenSSL Toolkit (http://www.openssl.org/)"
 *

```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```

* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED 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 OpenSSL PROJECT OR
* ITS 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.
*
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to. The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code. The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * 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 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. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *    "This product includes cryptographic software written by
 *     Eric Young (eay@cryptsoft.com)"
 * The word 'cryptographic' can be left out if the rouines from the library

```

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

```

*   being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*   the apps directory (application code) you must include an acknowledgement:
*   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``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
* SUCH DAMAGE.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

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` extension 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
``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
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` module is built using an included copy of the `libmpdec` library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 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.

(ciąg dalszy na następnej stronie)

(kontynuacja poprzedniej strony)

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 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.

Prawa autorskie

Python i ta dokumentacja jest:

Copyright © 2001-2023 Python Software Foundation. All rights reserved.

Copyright © 2000 BeOpen.com. All rights reserved.

Copyright © 1995-2000 Corporation for National Research Initiatives. All rights reserved.

Copyright © 1991-1995 Stichting Mathematisch Centrum. All rights reserved.

Patrz dział *Historia i zapisy prawne*, aby zobaczyć pełną informację na temat licencji i praw.

Niealfabetyczny

..., [117](#)
">>>", [117](#)
(*hash*)
 comment, [9](#)
* (*asterisk*)
 in function calls, [30](#)
**
 in function calls, [30](#)
2to3, [117](#)
: (*colon*)
 function annotations, [32](#)
->
 function annotations, [32](#)
__all__, [54](#)
__future__, [121](#)
__slots__, [127](#)

A

abstract base class, [117](#)
annotation, [117](#)
annotations
 function, [32](#)
argument, [117](#)
asynchronous context manager, [118](#)
asynchronous generator, [118](#)
asynchronous generator iterator, [118](#)
asynchronous iterable, [118](#)
asynchronous iterator, [118](#)
atrybut, [118](#)
awaitable, [118](#)

B

BDFL, [118](#)
binary file, [118](#)
builtins
 moduł, [52](#)
bytes-like object, [119](#)

C

C-contiguous, [119](#)
class, [119](#)
class variable, [119](#)
coding
 style, [32](#)
coercion, [119](#)
complex number, [119](#)
context manager, [119](#)
context variable, [119](#)
contiguous, [119](#)
coroutine, [119](#)
coroutine function, [119](#)
CPython, [120](#)

D

decorator, [120](#)
descriptor, [120](#)
dictionary, [120](#)
dictionary comprehension, [120](#)
dictionary view, [120](#)
docstring, [120](#)
docstrings, [23](#), [31](#)
documentation strings, [23](#), [31](#)
duck-typing, [120](#)

E

EAFP, [120](#)
expression, [120](#)

F

f-string, [121](#)
file
 obiekt, [61](#)
file object, [121](#)
file-like object, [121](#)
finder, [121](#)
floor division, [121](#)
for

- instrukcja, 20
- Fortran contiguous, 119
- function, 121
 - annotations, 32
- function annotation, 121
- funkcja wbudowana
 - help, 87
 - open, 61

G

- garbage collection, 121
- generator, 121
- generator expression, 122
- generator iterator, 122
- generic function, 122
- GIL, 122
- global interpreter lock, 122

H

- hash-based pyc, 122
- hashable, 122
- help
 - funkcja wbudowana, 87

I

- IDLE, 122
- immutable, 122
- import path, 122
- importer, 122
- importing, 122
- instrukcja
 - for, 20
- interactive, 123
- interpreted, 123
- interpreter shutdown, 123
- iterable, 123
- iterator, 123

J

- json
 - moduł, 63

K

- key function, 123
- keyword argument, 123
- kod bajtowy, 119

L

- lambda, 123
- LBYL, 124
- list, 124
- list comprehension, 124
- loader, 124

M

- magic
 - method, 124
- magic method, 124
- mangling
 - name, 82
- mapping, 124
- meta path finder, 124
- metaclass, 124
- method, 124
 - magic, 124
 - obiekt, 78
 - special, 128
- method resolution order, 124
- module, 124
 - search path, 50
- module spec, 124
- moduł
 - builtins, 52
 - json, 63
 - sys, 51
- moduł rozszerzenia, 121
- MRO, 124
- mutable, 124

N

- name
 - mangling, 82
- named tuple, 124
- namespace, 125
- namespace package, 125
- nested scope, 125
- new-style class, 125

O

- obiekt
 - file, 61
 - method, 78
- object, 125
- open
 - funkcja wbudowana, 61

P

- package, 125
- parameter, 125
- PATH, 50, 115
- path
 - module search, 50
- path based finder, 126
- path entry, 126
- path entry finder, 126
- path entry hook, 126
- path-like object, 126

PEP, [126](#)
 portion, [126](#)
 positional argument, [126](#)
 provisional API, [126](#)
 provisional package, [127](#)
 Python 3000, [127](#)
 Python Enhancement Proposals
 PEP 1, [126](#)
 PEP 8, [32](#)
 PEP 238, [121](#)
 PEP 278, [129](#)
 PEP 302, [121](#), [124](#)
 PEP 343, [119](#)
 PEP 362, [118](#), [126](#)
 PEP 411, [126](#)
 PEP 420, [121](#), [125](#), [126](#)
 PEP 443, [122](#)
 PEP 451, [121](#)
 PEP 484, [32](#), [117](#), [121](#), [128](#), [129](#)
 PEP 492, [118](#), [119](#)
 PEP 498, [121](#)
 PEP 519, [126](#)
 PEP 525, [118](#)
 PEP 526, [117](#), [129](#)
 PEP 3107, [32](#)
 PEP 3116, [129](#)
 PEP 3147, [50](#)
 PEP 3155, [127](#)
 Pythonic, [127](#)
 PYTHONPATH, [50](#), [51](#)
 PYTHONSTARTUP, [116](#)

Q

qualified name, [127](#)

R

reference count, [127](#)
 regular package, [127](#)
 RFC
 RFC 2822, [92](#)

S

search
 path, module, [50](#)
 sequence, [127](#)
 set comprehension, [128](#)
 single dispatch, [128](#)
 slice, [128](#)
 special
 method, [128](#)
 special method, [128](#)
 statement, [128](#)
 strings, documentation, [23](#), [31](#)
 style

 coding, [32](#)
 sys
 moduł, [51](#)

T

text encoding, [128](#)
 text file, [128](#)
 triple-quoted string, [128](#)
 type, [128](#)
 type alias, [128](#)
 type hint, [128](#)

U

universal newlines, [129](#)

V

variable annotation, [129](#)
 virtual environment, [129](#)
 virtual machine, [129](#)

W

wywołanie zwrotne, [119](#)

Z

Zen of Python, [129](#)
 zmienna środowiskowa
 PATH, [50](#), [115](#)
 PYTHONPATH, [50](#), [51](#)
 PYTHONSTARTUP, [116](#)