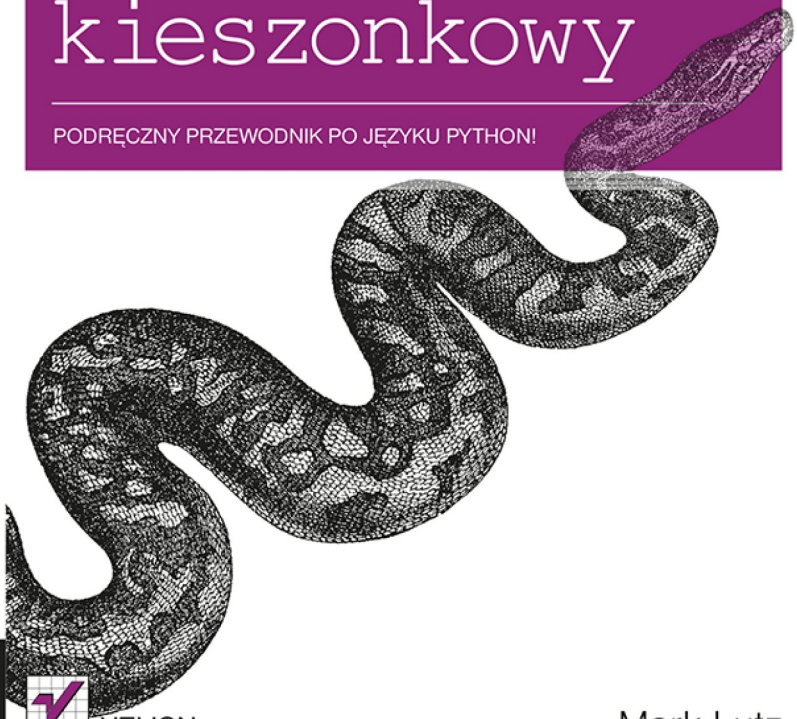
O'REILLY®

Wydanie V

Python

Leksykon kieszonkowy

PODRĘCZNY PRZEWODNIK PO JĘZYKU PYTHON!



HELION

Mark Lutz

Tytuł oryginału: Python Pocket Reference, Fifth Edition

Tłumaczenie: Radosław Meryk

ISBN: 978-83-246-9433-4

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/pythl5_ebook
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Spis treści

Python
Leksykon kieszonkowy

Wprowadzenie

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

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

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

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

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

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

Konwencje

W książce zastosowano następujące konwencje:

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

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

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

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

`Czcionka o stałej szerokości`
: Oznacza kod, polecenia i opcje wiersza poleceń, a także nazwy modułów, funkcji, atrybutów, zmiennych i metod.

`Czcionka o stałej szerokości – kursywa`
: W składni poleceń, wyrażeń, funkcji i metod oznacza nazwy parametrów, które można zastępować.

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

Patrz „Nagłówek podrozdziału”
: Odniesienia do innych podrozdziałów w tej książce są oznaczone za pomocą tekstu nagłówka podrozdziału umieszczonego w cudzysłowie.

UWAGA

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

Opcje wiersza poleceń Pythona

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

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

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

Opcje poleceń Pythona

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

`-b`

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

`-B`

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

`-d`

Włączenie wyjścia diagnostycznego (dla programistów rdzenia Pythona).

`-E`

Ignorowanie zmiennych środowiskowych Pythona opisanych w dalszej części tej książki (na przykład `PYTHONPATH`).

`-h`

Wyświetlenie komunikatu pomocy i zakończenie działania.

`-i`

Włączenie trybu interaktywnego po wykonaniu skryptu. Przydatne przy debugowaniu po awarii (w tzw. trybie „postmortem"). Zobacz też polecenie `pdb.pm` opisane w podręcznikach dotyczących bibliotek Pythona.

`-O`

Optymalizacja generowanego kodu bajtowego (tworzenie i wykorzystywanie plików z kodem bajtowym *.pyo*). W bieżącej wersji nieznacznie poprawia wydajność.

`-OO`

Działa podobnie jak opcja `–O`, opisana wcześniej, ale dodatkowo usuwa ciągi dokumentacyjne (ang. *docstring*) z kodu bajtowego.

`-q`

Wyłączenie wyświetlania komunikatu z informacją o wersji i prawach autorskich dla interaktywnego uruchamiania (począwszy od Pythona 3.2).

`-s`

Wyłączenie dodawania katalogu użytkownika do ścieżki wyszukiwania modułów `sys.path`.

`-S`

Wyłączenie dedukowania „ośrodka importu" podczas inicjalizacji.

`-u`

Wymuszenie braku buforowania i trybu binarnego dla urządzeń *stdout* i *stderr*.

`-v`

Wyświetlenie każdorazowo podczas inicjowania modułu komunikatu zawierającego lokalizację, z której moduł jest ładowany. Aby uzyskać obszerniejszy wynik, należy powtórzyć tę flagę.

`-V`

Wyświetlenie numeru wersji Pythona i zakończenie działania aplikacji.

`-W` *`arg`*

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

`-x`

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

`-X` *`opcja`*

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

Specyfikacja programu w wierszu polecenia

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

`nazwaplikuskryptu`

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

`-c` *`polecenie`*

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

`-m` *`moduł`*

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

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

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

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

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

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

Opcje poleceń Pythona 2.X

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

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

`-Q`

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

`-3`

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

`-R`

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

Zmienne środowiskowe

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

Zmienne operacyjne

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

`PYTHONPATH`

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

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

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

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

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

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

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

Zmienne opcji wiersza poleceń

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

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

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

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

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

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

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

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

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

Python Launcher dla systemu Windows

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

Dyrektywy plikowe launchera

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

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

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

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

Wiersz poleceń launchera

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

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

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

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

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

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

Zmienne środowiskowe launchera

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

`PY_PYTHON`	*Wersja wykorzystywana w przypadkach domyślnych (w pozostałych przypadkach 2)*
`PY_PYTHON3`	*Wersja wykorzystywana we fragmentach dotyczących Pythona 3 (np. 3.2)*
`PY_PYTHON2`	*Wersja wykorzystywana we fragmentach dotyczących Pythona 2 (np. 2.6)*

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

Wbudowane typy i operatory

Operatory i priorytet ich stosowania

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

Wyrażenia atomowe i dynamiczne określanie typów

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

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

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

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

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

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

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

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

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

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

Uwagi na temat stosowania operatorów

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

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

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

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

Operacje według kategorii

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

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

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

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

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

Tabela 2. Operacje porównań i operacje logiczne

Operator	Opis
`X < Y`	Ściśle mniejszy niż[1]
`X <= Y`	Mniejszy lub równy
`X > Y`	Ściśle większy niż
`X >= Y`	Większy lub równy
`X == Y`	Równość (ta sama wartość)
`X != Y`	Różny (odpowiednik działania `X<>Y` dostępnego wyłącznie w Pythonie 2.X)[2]
`X is Y`	Ten sam obiekt
`X is not Y`	Zanegowana tożsamość obiektu
`X < Y < Z`	Porównania łańcuchowe
`not X`	Jeśli `X` ma wartość `false`, działanie zwraca `True`, w przeciwnym razie `False`
`X or Y`	Jeśli `X` ma wartość `false`, działanie zwraca `Y`, w przeciwnym razie `X`
`X and Y`	Jeśli `X` ma wartość `false`, działanie zwraca `X`, w przeciwnym razie `Y`

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

Działanie	Opis	Metoda klasy
`X in S` `X not in S`	Test członkostwa	`__contains__`, `__iter__`, `__getitem__`[3]
`S1 + S2`	Konkatenacja	`__add__`
`S * N, N * S`	Repetycja	`__mul__`
`S[i]`	Indeksowanie według przesunięcia	`__getitem__`

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

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

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

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

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

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

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

Tabela 5. Działania odwzorowań (słowniki)

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

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

Tabela 5. Działania odwzorowań (słowniki) — ciąg dalszy

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

Tabela 6. Działania na liczbach (wszystkie typy liczbowe)

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

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

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

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

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

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

Uwagi na temat działań na sekwencjach

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

Indeksowanie — S[i]

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

Wycinki proste — S[i:j]

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

Wycinki rozszerzone — S[i:j:k]

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

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

Przypisanie wycinka — S[i:j:k] = I

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

Inne

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

Specyficzne typy wbudowane

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

Liczby

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

Literały i ich tworzenie

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