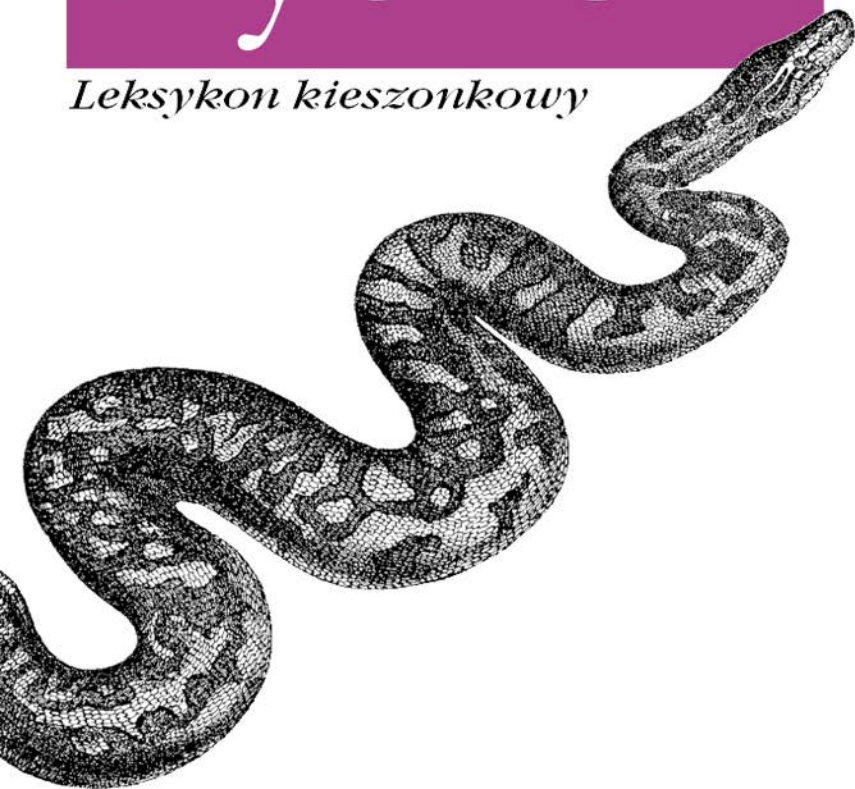


Python zawsze pod ręką!

Wydanie IV

Python

Leksykon kieszonkowy



O'REILLY®

Mark Lutz

HELION 

» Idź do

- Spis treści
- Przykładowy rozdział
- Skorowidz

» Katalog książek

- Katalog online
- Zamów drukowany katalog

» Twój koszyk

- Dodaj do koszyka

» Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

» Czytelnia

- Fragmenty książek online

» Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 32 230 98 63
e-mail: helion@helion.pl
© Helion 1991–2011

Python. Leksykon kieszonkowy. Wydanie IV

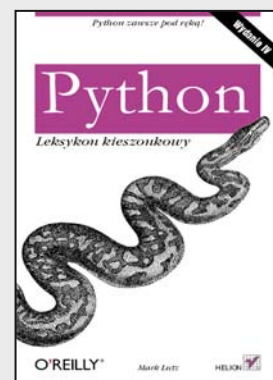
Autor: [Mark Lutz](#)

Tłumaczenie: Radosław Meryk

ISBN: 978-83-246-2686-1

Tytuł oryginału: [Python Pocket Reference](#)

Format: 122×194, stron: 200



- Jakie możliwości kryją standardowe moduły biblioteczne?
- Jak wykonywać operacje na plikach?
- Jak stworzyć graficzny interfejs użytkownika?

Python to wyjątkowo praktyczny język programowania, idealnie nadający się do szybkiego rozwiązywania niecodziennych problemów, z którymi często borykają się koderzy. Nie wymusza on stosowania jednego stylu programowania, co pozwala na dużo większą elastyczność w trakcie pisania kodu. Umożliwia programowanie obiektowe, strukturalne i funkcyjne, a ponadto udostępnia zaawansowane mechanizmy zarządzania pamięcią, zapewnia dynamiczne sprawdzanie typów oraz czytelną składnię. Te wszystkie zalety powodują, że Python ma grupę swoich wiernych fanów.

Niniejsza książka należy do popularnej serii „Leksykon kieszonkowy”, dzięki której zawsze i wszędzie możesz przypomnieć sobie wybrane zagadnienia, związane z różną tematyką. Pozycja, którą właśnie trzymasz w rękach, została poświęcona językowi Python. W trakcie jej lektury zapoznasz się z takimi zagadnieniami, jak sterowanie przepływem programu, wykorzystanie pętli, list, słowników oraz operacje na plikach. Ponadto w każdej chwili będziesz mógł sprawdzić składnię oraz sposoby wykorzystania funkcji i wyjątków wbudowanych. Książka stanowi znakomite kompendium wiedzy na temat języka Python. Sprawdzi się ona w rękach początkującego użytkownika – jako przewodnik, a w rękach zaawansowanego programisty – jako pomocnik.

- Wbudowane typy i operatory
- Działania na liczbach
- Operacje na łańcuchach znaków
- Wykorzystanie Unicode w Pythonie
- Obsługa list oraz słowników
- Operacje na zbiorach i plikach
- Sterowanie przepływem programu
- Konwersja typów
- Obsługa wyjątków
- Wykorzystanie przestrzeni nazw
- Zasięgi zmiennych
- Przeciążanie operatorów
- Standardowe moduły biblioteczne
- Zastosowanie wyrażeń regularnych
- Tworzenie graficznego interfejsu użytkownika

Wyciśnij jeszcze więcej z języka Python!

Spis treści

Wprowadzenie	7
Konwencje	8
Wykorzystanie przykładowego kodu	8
Opcje wiersza poleceń	9
Opcje Pythona	9
Specyfikacja programu	10
Zmienne środowiskowe	12
Zmienne operacyjne	12
Zmienne opcji wiersza poleceń	12
Wbudowane typy i operatory	13
Operatory i priorytet ich stosowania	13
Uwagi na temat stosowania operatorów	13
Operacje według kategorii	16
Uwagi na temat działań na sekwencjach	19
Specyficzne typy wbudowane	20
Liczby	20
Łańcuchy znaków	22
Łańcuchy znaków Unicode	36
Listy	39
Słowniki	43
Krotki	47
Pliki	48
Zbiory	52
Inne popularne typy	54
Konwersje typów	55
Instrukcje i ich składnia	56
Reguły składniowe	56
Reguły dotyczące nazw	57
Instrukcje	59
Instrukcja przypisania	59
Instrukcja wyrażeniowa	61
Instrukcja print	62

Instrukcja if	64
Instrukcja while	64
Instrukcja for	65
Instrukcja pass	65
Instrukcja break	65
Instrukcja continue	66
Instrukcja del	66
Instrukcja def	66
Instrukcja return	70
Instrukcja yield	70
Instrukcja global	71
Instrukcja nonlocal	72
Instrukcja import	72
Instrukcja from	74
Instrukcja class	75
Instrukcja try	77
Instrukcja raise	79
Instrukcja assert	81
Instrukcja with	81
Instrukcje w Pythonie 2.X	83
Przestrzenie nazw i reguły zasięgu	83
Nazwy kwalifikowane — przestrzenie nazw obiektów	83
Nazwy niekwalifikowane — zasięgi leksykalne	84
Zasięgi zagnieżdżone statycznie	84
Programowanie obiektowe	86
Klasy i egzemplarze	86
Atrybuty pseudoprywatne	87
Klasy nowego stylu	88
Metody przeciążające operatory	89
Wszystkie typy	89
Kolekcje (sekwencje, mapy)	93
Liczby (operatory dwuargumentowe)	95
Liczby (inne działania)	98
Deskryptory	98
Menedżery kontekstu	99
Metody przeciążające operatory w Pythonie 2.X	99
Funkcje wbudowane	102
Funkcje wbudowane w Pythonie 2.X	120

Wbudowane wyjątki	125
Klasy bazowe (kategorie)	125
Wyjątki szczegółowe	127
Wyjątki kategorii ostrzeżeń	130
Framework ostrzeżeń	131
Wbudowane wyjątki w Pythonie 2.X	132
Wbudowane atrybuty	132
Standardowe moduły biblioteczne	133
Moduł sys	134
Moduł string	140
Funkcje i klasy modułu	140
Stałe	141
Moduł systemowy os	142
Narzędzia administracyjne	142
Stałe wykorzystywane do zapewnienia przenośności	143
Polecenia powłoki	144
Narzędzia obsługi środowiska	145
Narzędzia obsługi deskryptorów plików	147
Narzędzia do obsługi nazw ścieżek	148
Zarządzanie procesami	151
Moduł os.path	154
Moduł dopasowywania wzorców re	157
Funkcje modułu	157
Obiekty wyrażeń regularnych	159
Obiekty dopasowania	160
Składnia wzorców	161
Moduły utrwalania obiektów	163
Moduły dbm i shelve	165
Moduł pickle	166
Moduł GUI tkinter i narzędzia	169
Przykład użycia modułu tkinter	169
Podstawowe widgety modułu tkinter	169
Wywołania okien dialogowych	171
Dodatkowe klasy i narzędzia modułu tkinter	171
Porównanie biblioteki Tcl/Tk z modulem tkinter Pythona	172
Moduły i narzędzia obsługi internetu	173
Powszechnie używane moduły biblioteczne	173

Inne standardowe moduły biblioteczne	176
Moduł math	176
Moduł time	176
Moduł datetime	177
Moduły obsługi wątków	177
Parsowanie danych binarnych	178
Przenośny interfejs API dostępu do baz danych SQL w Pythonie	179
Przykład użycia interfejsu API	179
Interfejs modułu	180
Obiekty połączeń	180
Obiekty kursora	181
Obiekty typów i konstruktory	182
Idiomy Pythona i wskazówki	182
Wskazówki dotyczące rdzenia języka	182
Wskazówki dotyczące środowiska	184
Wskazówki dotyczące użytkowania	185
Różne wskazówki	187
Skorowidz	189

Uwagi na temat działań na sekwencjach

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.
- $S[-2]$ pobiera przedostatni element ($S[\text{len}(S) - 2]$).

Wycinanie — $S[i:j]$

- Wyodrębnia ciągłą sekcję z sekwencji.
- Domyślnymi granicami wycinka są 0 oraz długość sekwencji.
- $S[1:3]$ pobiera elementy sekwencji S od 1. do 3. (bez trzeciego).
- $S[1:]$ pobiera elementy sekwencji od 1. do końca (długość -1).
- $S[: -1]$ pobiera elementy sekwencji od zerowego do przedostatniego elementu włącznie.
- $S[:]$ tworzy płytką kopię obiektu sekwencji S .
- Operacje przypisania wycinków są podobne do usuwania elementów i późniejszego ich wstawiania.

Wycinanie — $S[i:j:k]$

- Jeśli występuje trzeci argument k , to jest to krok; 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 4. do 1. (wyłącznie) w odwróconej kolejności.

Inne

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

Specyficzne typy wbudowane

W tym podrozdziale opisano liczby, ciągi znaków, listy, słowniki, krotki, pliki oraz inne zasadnicze typy wbudowane. 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

W tym punkcie opisano podstawowe typy liczbowe (całkowite, zmiennooprzecinkowe), a także bardziej zaawansowane typy danych (liczby zespolone, dziesiętne oraz ułamki). Liczby są zawsze niemutowalne (niezmienne).

Literały i ich tworzenie

Liczby zapisuje się w formie różnych stałych numerycznych:

1234, -24, 0

Liczby całkowite (nieograniczona precyzja)¹.

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

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

0o177, 0x9ff, 0b1111

Literały liczb całkowitych ósemkowe, szesnastkowe i dwójkowe².

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

Liczby zespolone.

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

Typy bazujące na modułach: liczby dziesiętne, ułamki.

¹ W Pythonie 2.6 dla liczb całkowitych o nieograniczonej precyzji występuje osobny typ o nazwie `long`. Typ `int` odpowiada zwykłemu liczbom całkowitym, o precyzji ograniczonej zazwyczaj do 32 bitów. Obiekty typu `long` mogą być kodowane z przyrostkiem „L” (np. 99999L). Zwykle liczby całkowite są automatycznie promowane do typu `long` w przypadku, gdy wymagają specjalnej dokładności. W Pythonie 3.0 typ `int` zapewnia nieograniczoną precyzję, a zatem zawiera w sobie zarówno typ `int`, jak i `long` z Pythona 2.6. W wersji 3.0 Pythona usunięto oznaczenie „L” z literałów liczbowych.

² W Pythonie 2.6 literały ósemkowe można zapisywać z wiodącym zerem — zapisy `0777` i `0o777` są sobie równoważne. W wersji 3.0 dla literałów ósemkowych dostępna jest tylko druga postać.


```
int(), float(), complex()
```

Tworzenie liczb na podstawie innych obiektów lub ciągów znaków z możliwością podania podstawy konwersji. Więcej informacji na ten temat można znaleźć w podrozdziale „Funkcje wbudowane” na stronie 102.

Działania

Typy liczbowe obsługują wszystkie działania liczbowe (patrz tabela 6. na stronie 18). W wyrażeniach z typami mieszanymi Python konwertuje operandy w górę, do „najwyższego” typu. W tej hierarchii liczby całkowite (`integer`) znajdują się niżej od zmiennoprzecinkowych, a te z kolei są niżej od liczb zespolonych. W wersjach Pythona 3.0 i 2.6 liczby całkowite i zmiennoprzecinkowe udostępniają również szereg **metod** oraz innych **atrybutów**. Więcej informacji na ten temat można znaleźć w podręczniku *Python Library Reference*.

```
>>> (2.5).as_integer_ratio()      # atrybuty liczb zmiennoprzecinkowych
(5, 2)
>>> (2.5).is_integer()
False
>>> (2).numerator, (2).denominator # atrybuty liczb całkowitych
(2, 1)
>>> (255).bit_length(), bin(255)  # 3.1+ bit_length()
(8, '0b11111111')
```

Liczby dziesiętne i ułamki

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

```
>>> 0.1 - 0.3
-0.19999999999999998

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

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

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

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

Inne typy numeryczne

W Pythonie jest dostępny również typ `set` (opisano go w podrozdziale „Zbiory” na stronie 52). Dodatkowe typy numeryczne, takie jak wektory i macierze, są dostępne za pośrednictwem zewnętrznych rozszerzeń *open source* (np. w pakiecie `NumPy`). Dostępne są również zewnętrzne pakiety do wizualizacji, obliczeń statystycznych i wiele innych.

Łańcuchy znaków

Standardowy obiekt `str` udostępniający własności łańcucha znaków to niemutowalna (tzn. niezmienna) tablica znaków, do której dostęp jest możliwy za pośrednictwem indeksu (przesunięcia). W Pythonie 3.0 dostępne są trzy typy łańcuchowe o bardzo podobnych interfejsach:

- `str` — niemutowalna sekwencja znaków używana w odniesieniu do wszystkich rodzajów tekstu: zarówno ASCII, jak i Unicode;
- `bytes` — niemutowalna sekwencja krótkich liczb całkowitych używana do reprezentacji binarnych danych bajtowych;
- `bytearray` — mutowalna wersja typu `bytes`.

Z kolei w Pythonie 2.X występują dwa niemutowalne typy łańcuchowe: `str` — do reprezentowania 8-bitowych danych tekstowych i binarnych — oraz `unicode` — dla danych tekstowych Unicode (więcej informacji na ich temat można znaleźć w podrozdziale „Łańcuchy znaków Unicode” na stronie 36. W Pythonie 2.6 występuje także typ `bytearray` z Pythona 3.0, ale nie wprowadza on tak ostrego rozgraniczenia pomiędzy danymi tekstowymi a binarnymi (w Pythonie 2.6 można je dowolnie łączyć z łańcuchami tekstowymi).

Większość materiału w tym podrozdziale dotyczy wszystkich typów łańcuchowych. Więcej informacji na temat typów `bytes` i `bytearray` można znaleźć w podrozdziałach „Metody klasy `String`” na stronie 30, „Łańcuchy znaków Unicode” na stronie 36 oraz „Funkcje wbudowane” na stronie 102.

Literały i ich tworzenie

Łańcuchy znaków zapisuje się w postaci serii znaków ujętych w cudzysłów (apostrofy). Opcjonalnie można je poprzedzić znakiem desygatora.

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

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

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

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

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

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

```
"To" "będzie" "scalone"
```

Śąsiednie stałe łańcuchowe są scalane. Jeśli zostaną ujęte w nawiasy, to mogą obejmować wiele wierszy.

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

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

Literały znakowe opisane poniżej to specjalizowane łańcuchy znaków. Szczegółowo opisano je w podrozdziale „Łańcuchy znaków Unicode” na stronie 36:

```
b'...'
```

Bajtowy literał tekstowy: sekwencja wartości 8-bitowych bajtów reprezentujących „surowe” dane binarne. Taki literał tworzy typ `bytes` w Pythonie 3.0 oraz zwykły ciąg `str` w Pythonie 2.6 (dla zachowania zgodności z wersją 3.0). Więcej informacji na ten temat można znaleźć w podrozdziałach „Metody klasy `String`” na stronie 30, „Łańcuchy znaków Unicode” na stronie 36 oraz „Funkcje wbudowane” na stronie 102.

`bytearray(...)`

Tekstowy literal `bytearray` — mutowalna wersja typu `bytes`, dostępna zarówno w Pythonie w wersji 2.6, jak i 3.0. Więcej informacji na ten temat można znaleźć w podrozdziałach „Metody klasy `String`” na stronie 30, „Łańcuchy znaków Unicode” na stronie 36 oraz „Funkcje wbudowane” na stronie 102.

`u'...'`

Literal łańcuchowy Unicode dostępny wyłącznie w Pythonie 2.X (w Pythonie 3 kodowanie Unicode jest obsługiwane przez standardowe obiekty `str`). Więcej informacji na ten temat można znaleźć w podrozdziale „Łańcuchy znaków Unicode” na stronie 36.

`str()`, `bytes()`, `bytearray()`

Ciągi znaków tworzone na podstawie obiektów. W Pythonie 3.0 dostępna jest opcja kodowania (dekodowania) Unicode. Więcej informacji na ten temat można znaleźć w podrozdziale „Funkcje wbudowane” na stronie 102.

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

Ciągi znaków zawierających tekstowe reprezentacje liczb w formacie ósemkowym, szesnastkowym i dwójkowym. Więcej informacji na ten temat można znaleźć w podrozdziale „Funkcje wbudowane” na stronie 102.

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

Działania

Wszystkie typy łańcuchowe obsługują mutowalne działania na sekwencjach (opisane wcześniej w tabeli 3. na stronie 17) oraz dodatkowo wywołania metod tekstowych (opisanych w dalszej części tego podrozdziału). Dodatkowo typ `str` obsługuje wyrażenia formatowania ciągów znaków z wykorzystaniem operatora `%` oraz zastępowanie szablonów. Ponadto typ `bytearray` obsługuje działania na sekwencjach mutowalnych (zestawione w tabeli 4. na stronie 17) oraz dodatkowe metody przetwarzania list. Warto również zapoznać się z modulem dopasowywania wzorców tekstowych *re* (jego opis znajduje się w podrozdziale „Moduł dopasowywania wzorców tekstowych *re*” na stronie 157), a także z tekstowymi funkcjami wbudowanymi, opisanymi w podrozdziale „Funkcje wbudowane” na stronie 102.

Tabela 7. Sekwencje specjalne dostępne dla stałych tekstowych

Sekwencja specjalna	Znaczenie	Sekwencja specjalna	Znaczenie
<code>\nowywiersz</code>	Kontynuacja w nowym wierszu	<code>\t</code>	Tabulacja pozioma
<code>\\</code>	Lewy ukośnik (<code>\</code>)	<code>\v</code>	Tabulacja pionowa
<code>\'</code>	Apostrof (<code>'</code>)	<code>\N{id}</code>	Znak o kodzie Unicode id
<code>\"</code>	Cudzysłów (<code>"</code>)	<code>\uhhhh</code>	16-bitowy szesnastkowy kod Unicode
<code>\a</code>	dzwonek (<code>bell</code>)	<code>\Uhhhhhhhhh</code>	32-bitowy szesnastkowy kod Unicode ^a
<code>\b</code>	Backspace	<code>\xhh</code>	Liczba szesnastkowa (najwyżej 2 cyfry)
<code>\f</code>	Wysuw strony	<code>\oooo</code>	Liczba ósemkowa (maksymalnie 3 znaki)
<code>\n</code>	Wysuw wiersza	<code>\0</code>	Znak <code>Null</code> (nie jest to koniec łańcucha znaków)
<code>\r</code>	Powrót karetki	<code>\inne</code>	To nie są sekwencje specjalne

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

Formatowanie łańcuchów znaków

Zarówno w Pythonie 2.6, jak i 3.0 standardowe łańcuchy znaków `str` obsługują dwa różne rodzaje formatowania:

- wyrażenie formatujące z wykorzystaniem operatora `%`: `fmt % (wartości)`,
- nowy sposób — wywołanie metody o składni: `fmt.format ↪(wartości)`.

Oba te rodzaje tworzą nowe ciągi znaków na podstawie kodów zastąpień. Kody te mogą być specyficzne dla różnych typów. Uzyskane wyniki można wyświetlić lub przypisać do zmiennych w celu późniejszego wykorzystania:

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

Chociaż w czasie powstawania tej książki sposób bazujący na wywołaniu metody wydaje się bardziej rozwojowy, to wyrażenia formatujące są powszechnie wykorzystywane w istniejącym kodzie.

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

Wyrażenia formatujące łańcuchy znaków

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

```
'Rycerze, którzy mówią %s!' % 'Nie'
```

```
Wynik: 'Rycerze, którzy mówią Nie!'
```

```
"%d %s %d you" % (1, 'spam', 4.0)
```

```
Wynik: '1 spam 4 you'
```

```
"%(n)d %(x)s" % {"n":1, "x":"spam"}
```

```
Wynik: '1 spam'
```

```
'%f, %.2f, %.*f' % (1/3.0, 1/3.0, 4, 1/3.0)
```

```
Wynik: '0.333333, 0.33, 0.3333'
```

Cele zastępowania po lewej stronie operatora % w wyrażeniu formatującym mają następujący ogólny format:

```
%(nazwaklucza)[flagi][szerokość][.precyzja]kodtypu
```

Człon *nazwaklucza* odwołuje się do elementu w odpowiednim słowniku; *flagi* to zbiór znaków o następującym znaczeniu: – (wyrównanie do lewej), + (znak liczby), spacja (pozostawienie pustego miejsca przed liczbami dodatnimi) oraz 0 (wypełnianie zerami); *szerokość* oznacza całkowitą szerokość pola; *precyzja* oznacza liczbę cyfr po kropce, natomiast *kodtypu* to znak z tabeli 8. Człony *szerokość* i *precyzja* można zakodować jako *. W tym przypadku ich wartości będą pobrane z następnego elementu występującego po prawej stronie operatora %.

Tabela 8. Kody typu formatowania łańcuchów znaków

Kod	Znaczenie	Kod	Znaczenie
s	String (lub dowolny obiekt, wykorzystuje <code>str()</code>)	X	typ x zapisany wielkimi literami
r	to samo co s, ale wykorzystuje metodę <code>repr()</code> zamiast <code>str()</code>	e	liczba zmiennoprzecinkowa w postaci wykładniczej
c	Character (<code>int</code> lub <code>str</code>)	E	e z wielkimi literami
d	Decimal (dziesiętna liczba całkowita)	f	dziesiętna liczba zmiennoprzecinkowa
i	Integer	P	f z wielkimi literami
u	to samo co d (przestarzałe)	g	zmiennoprzecinkowy typ e lub f
o	Octal (ósemkowa liczba całkowita)	G	zmiennoprzecinkowy typ E lub F
x	Hex (szesnastkowa liczba całkowita)	%	Litera ' % '

Jest to przydatne, gdy rozmiary nie są znane aż do fazy działania programu. **Wskazówka:** `%s` przekształca dowolny obiekt na postać jego reprezentacji tekstowej.

Wywołanie metody formatującej

Wywołanie metody formatującej działa podobnie jak opisany w poprzednim punkcie sposób bazujący na wyrażeniach, ale wykorzystuje standardową składnię wywołania metody obiektu formatowania łańcucha znaków. Cele zastępowania są natomiast identyfikowane za pomocą nawiasów klamrowych (`{}`), a nie operatora `%`. Cele zastępowania występujące w łańcuchu formatującym mogą odwoływać się do argumentów wywołania metody za pomocą pozycji lub nazwy słowa kluczowego. Dodatkowo można odwoływać się w nich do atrybutów argumentów, kluczy i przesunąć. Dozwolone jest posługiwanie się domyślnym formatowaniem lub jawne wprowadzanie kodów typu. Można również zagnieżdżać cele, dzięki czemu możliwe jest pobieranie wartości z list argumentów:

```
>>> '{0}, {jedzenie}'.format(42, jedzenie='spam')
'42, spam'

>>> import sys
>>> fmt = '{0.platform} {1[x]} {2[0]}' # attr,key,index
>>> fmt.format(sys, {'x': 'szynka', 'y': 'jajka'}, 'AB')
'win32 szynka A'

>>> '{0} {1:+.2f}'.format(1 / 3.0, 1 / 3.0)
'0.333333333333 +0.33'

>>> '{0:.{1}f}'.format(1 / 3.0, 4)
'0.3333'
```

Większość z pokazanych wywołań ma odpowiedniki we wzorcach użycia wyrażeń z operatorem % (np. odwołania do klucza i wartości słownika), chociaż w przypadku wyrażeń niektóre działania muszą być kodowane poza samym łańcuchem formatującym. Cele zastępowania, występujące w wywołaniach metody formatującej, mają następujący ogólny format:

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

Oto znaczenie poszczególnych członów powyższego celu zastępowania:

- *nazwapola* to liczba lub słowo kluczowe określające argument, za którym występuje opcjonalny atrybut ".name" lub odwołanie do komponentu "[index]".
- *flagakonwersji* to "r", "s" lub "a". Wywołują one odpowiednio metody `repr()`, `str()` lub `ascii()` dla wartości.
- *specyfikacjaformatu* określa sposób prezentacji wartości włącznie z takimi szczegółami, jak szerokość pola, wyrównania, wypełnienia, precyzji dziesiętnej itp. Ostatnim elementem jest opcjonalny kod typu danych.

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

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

Komponent *wyrównanie* może być jednym ze znaków: "<", ">", "=" lub "^", co oznacza odpowiednio wyrównanie do lewej, wyrównanie do prawej, wypełnienie za symbolem znaku lub wyrównanie do środka. W miejsce komponentu *znak* można wstawić +, – lub spację, natomiast *kodtypu*, ogólnie rzecz biorąc, ma takie samo znaczenie jak w przypadku wyrażeń z operatorem % z tabeli 8., z poniższymi wyjątkami:

- kody typów *i* oraz *u* są niedostępne; należy wykorzystać typ *d*, który wyświetla liczby całkowite w formacie dziesiętnym;
- dodatkowy kod typu *b* wyświetla liczby całkowite w formacie dwójkowym (działa podobnie jak wbudowana funkcja `bin()`);
- dodatkowy kod typu *%* wyświetla liczbę w postaci procenta.

Pojedyncze obiekty można również formatować za pomocą wbudowanej funkcji `format(obiekt, specyfikacjaformatu)` (więcej informacji można znaleźć w podrozdziale „Funkcje wbudowane” na stronie 102).

Formatowanie można dostosować do indywidualnych potrzeb określonych klas za pomocą metody przeciążania operatorów `__format__` (więcej informacji można znaleźć w podrozdziale „Metody przeciążające operatory” na stronie 89).

UWAGA

W Pythonie 3.1 i wersjach późniejszych znak `,` poprzedzający liczbę całkowitą lub zmiennoprzecinkową wewnątrz członu kodotypu powoduje wstawianie separatorów tysięcy (przecinków). Instrukcja:

```
'{0:,d}'.format(1000000)
```

tworzy ciąg `'1,000,000'`, natomiast:

```
'{0:13,.2f}'.format(1000000)
```

tworzy ciąg `'1,000,000.00'`.

Ponadto, począwszy od Pythona 3.1, w przypadku pominięcia pól w komponencie nazwapola numerom pól są automatycznie nadawane kolejne wartości. Poniższe trzy instrukcje dają ten sam efekt, choć w przypadku występowania wielu pól pola numerowane automatycznie mogą być mniej czytelne:

```
'{0}/{1}/{2}'.format(x, y, z) # jawne numerowanie  
'{}/{}/{}'.format(x, y, z)   # 3.1 — numerowanie automatyczne  
'%s/%s/%s' % (x, y, z)      # wyrażenie
```

Zastępowanie szablonów w łańcuchach znaków

W Pythonie 2.4 i wersjach późniejszych jako alternatywa wyrażen formatujących łańcuchy znaków oraz metod — zagadnienia opisane w poprzednich punktach — dostępny jest mechanizm prostego zastępowania szablonów. Standardowy sposób zastępowania zmiennych zapewnia operator `%`:

```
>>> '%(strona)i: %(tytul)s' % {'strona':2, 'tytul': 'PyRef4E'}  
'2: PyRef4E'
```

Do prostszych zadań formatowania na moduł `string` dodano klasę `Template`. W tej klasie zastąpienie jest wskazywane za pomocą symbolu `$`:

```
>>> import string  
>>> t = string.Template('$strona: $tytul')  
>>> t.substitute({'strona':2, 'tytul': 'PyRef4E'})  
'2: PyRef4E'
```

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

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

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

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

Metody klasy String

Oprócz metody `format()` opisanej wcześniej dostępne są inne metody klasy `string`. Są to wysokopoziomowe narzędzia przetwarzania tekstu zapewniające większe możliwości od wyrażeń łańcuchowych. Listę dostępnych metod klasy `String` zamieszczono w tabeli 9. `S` oznacza dowolny obiekt `string`. Metody klasy `String`, które modyfikują tekst, zawsze zwracają nowy łańcuch tekstowy. Nigdy nie modyfikują obiektu (obiekty `String` są niemutowalne). Warto również zapoznać się z modulem dopasowywania wzorców tekstowych *re* (jego opis znajduje się w podrozdziale „Moduł dopasowywania wzorców tekstowych *re*” na stronie 157). Można tam znaleźć bazujące na wzorach odpowiedniki niektórych metod klasy `String`.

Metody typów `byte` i `bytearray`

Występujące w Pythonie 3.0 typy łańcuchowe `bytes` i `bytearray` mają podobne, ale nie takie same zbiory metod jak standardowy typ łańcuchowy `str` (`str` to typ tekstowy Unicode, `bytes` to surowe dane binarne, natomiast typ `bytearray` jest mutowalny). W kodzie zamieszczonym poniżej wyrażenie `set(X) - set(Y)` wylicza elementy należące do zbioru `X`, których nie ma w zbiorze `Y`:

- typy `bytes` i `bytearray` nie obsługują kodowania Unicode (są „surowymi” bajtami, a nie dekodowanym tekstem) ani formatowania łańcuchów znaków (metoda `str.format` oraz operator `%` są zaimplementowane za pomocą metody `__mod__`);
- typ `str` nie obsługuje dekodowania Unicode (ten tekst już jest zdekodowany);

Tabela 9. Metody klasy String dostępne w Pythonie 3.0

```
S.capitalize()
S.center(width [, fill])
S.count(sub [, start [, end]])
S.encode([encoding [,errors]])
S.endswith(suffix [, start [, end]])
S.expandtabs([tabsize])
S.find(sub [, start [, end]])
S.format(fmtstr, *args, **kwargs)
S.index(sub [, start [, end]])
S.isalnum()
S.isalpha()
S.isdecimal()
S.isdigit()
S.isidentifier()
S.islower()
S.isnumeric()
S.isprintable()
S.isspace()
S.istitle()
S.isupper()
S.join(iterable)
S.ljust(width [, fill])
S.lower()
S.lstrip([chars])
S.maketrans(x[, y[, z]])
S.partition(sep)
S.replace(old, new [, count])
S.rfind(sub [,start [,end]])
S.rindex(sub [, start [, end]])
S.rjust(width [, fill])
S.rpartition(sep)
S.rsplit([sep[, maxsplit]])
S.rstrip([chars])
S.split([sep [,maxsplit]])
S.splitlines([keepends])
S.startswith(prefix [, start [, end]])
S.strip([chars])
S.swapcase()
S.title()
S.translate(map)
S.upper()
S.zfill(width)
```

- typ `bytearray` dostarcza unikatowych metod przetwarzania „w miejscu”, podobnych do tych, które są dostępne dla list:

```
>>> set(dir(str)) - set(dir(bytes))
{'isprintable', 'format', '__mod__', 'encode',
 'isidentifier', '_formatter_field_name_split',
 'isnumeric', '_rmod__', 'isdecimal',
 '_formatter_parser', 'maketrans'}
>>> set(dir(bytes)) - set(dir(str))
{'decode', 'fromhex'}

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

Oprócz metod typu `bytes` i `bytearray` udostępniają również standardowe działania na sekwencjach, zestawione w tabeli 3. na stronie 17. Dodatkowo typ `bytearray` obsługuje mutowalne działania na sekwencjach, zestawione w tabeli 4. na stronie 17. Więcej informacji na ten temat można znaleźć w podrozdziałach „Łańcuchy znaków Unicode” na stronie 36 oraz „Funkcje wbudowane” na stronie 102.

UWAGA

Zbiór metod typu `string` w Pythonie 2.6 jest nieco inny (np. istnieje metoda `decode` dla innego modelu typu `Unicode`, charakterystycznego dla wersji 2.6). Typ `unicode` Pythona 2.6 ma interfejs niemal taki sam jak obiekty `str` z tej wersji Pythona. Więcej informacji na ten temat można znaleźć w podręczniku *Python 2.6 Library Reference*. W trybie interaktywnym można też uruchomić polecenie `dir(str)` oraz `help(str.metoda)`.

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

[...], 13
__abs__(), 98
__all__, 87
__annotations__, 67
__bases__, 133
__bool__(), 91
__builtin__, 84
__call__(), 91
__class__, 133
__class__(), 88
__cmp__(), 100
__coerce__(), 101
__complex__(), 98
__contains__(), 94
__del__(), 90
__delattr__(), 91
__delete__(), 99
__delitem__(), 95
__delslice__(), 101
__dict__, 133
__dir__(), 93
__div__(), 101
__enter__(), 99
__eq__(), 92
__exit__(), 99
__float__(), 98
__format__(), 90
__ge__(), 92
__get__(), 99
__getattr__(), 88, 91
__getattribute__(), 88, 92
__getitem__(), 94
__getslice__(), 100
__gt__(), 92
__hash__(), 90
__hex__(), 101
__import__(), 107
__index__(), 98
__init__(), 89
__int__(), 98
__invert__(), 98
__iter__(), 71, 94
__le__(), 92
__len__(), 93
__long__(), 101
__lt__(), 92
__metaclass__(), 102
__name__, 133
__ne__(), 92
__neg__(), 98
__new__(), 89
__next__(), 71, 94
__nonzero__(), 100
__oct__(), 101
__pos__(), 98
__repr__(), 90
__reversed__(), 95
__round__(), 98
__set__(), 99
__setattr__(), 91
__setitem__(), 95
__setslice__(), 101
__slots__(), 93
__stderr__, 139
__stdin__, 139
__stdout__, 139

`__str__()`, 90
`__exit__()`, 152
`__getframe__()`, 136
`__thread`, 177

A

`abort()`, 152
`abs()`, 102
`abspath()`, 155
`access()`, 150
adnotacje funkcji, 67
`all()`, 102
`altsep`, 143
`any()`, 102
`anydbm`, 163
`append()`, 39
`apply()`, 120
argumenty, 62, 66
argumenty kluczowe, 67
`argv`, 134
`ArithmeticError`, 126
`as`, 74
`ascii()`, 102, 120
`ascii_letters`, 141
`ascii_lowercase`, 141
`ascii_uppercase`, 141
`assert`, 81
`AssertionError`, 81, 127
atributy, 51, 68, 83, 132
atributy egzemplarzy, 86
atributy klasy, 86
atributy pseudoprywatne, 87
`AttributeError`, 127

B

`base64`, 175
`BaseException`, 125
`basename()`, 155
`basestring()`, 121
baza danych, 179
białe spacje, 56
`bin()`, 102

`binascii`, 175
`binhex`, 175
bloki, 56
bogate porównania, 92
`bool`, 54
`bool()`, 103
`Boolean`, 16
`break`, 65
`buffer()`, 121
`builtin_module_names`, 134
`byte`, 30, 37
`bytearray`, 22, 30, 37
`bytearray()`, 103
`byteorder`, 134
`bytes`, 22
`bytes()`, 103
`BytesWarning`, 131

C

`callable()`, 121
`callproc()`, 181
`capitalize()`, 34
`capwords()`, 140
`center()`, 35
`cgi`, 174
`chdir()`, 146, 148
`chmod()`, 148
`chown()`, 149
`chr()`, 103
ciągi dokumentacyjne, 56
`class`, 75, 86
`classmethod()`, 104
`clear()`, 45
`clock()`, 176
`close()`, 50, 147, 180, 181
`cmp()`, 121
`codecs`, 36
`coerce()`, 121
`commit()`, 181
`commonprefix()`, 155
`compile()`, 104, 157
`complex()`, 21, 104
`connect()`, 180
`continue`, 66

copy(), 45
copyright, 134
count(), 33, 40
cPickle, 163
ctime(), 177
curdir, 143
cursor(), 181

D

datetime, 177
dbm, 163, 165
def, 66
defpath, 144
dekoratory, 69
dekoratory klas, 76
del, 66
delattr(), 104
DeprecationWarning, 131
description, 181
deskryptory, 98
deskryptory plików, 147
dict(), 104
digits, 141
dir(), 105
dirname(), 155
displayhook(), 134
divmod(), 105
dllhandle, 134
doctest, 185
dont_write_bytecode, 138
dopasowywanie wzorców, 157
dostęp do baz danych, 179
 obiekty kursora, 181
 obiekty połączeń, 180
 obiekty typów, 182
dup(), 147
dup2(), 147
dziedziczenie, 84, 86
dzielenie, 13

E

egzemplarz klasy, 86
eksperymentalne własności
 języka, 185
else, 78
email, 175
endpos, 160
endswith(), 33
enumerate(), 105
environ, 145
EnvironmentError, 126
EOFError, 127
error, 142
Error, 180
eval(), 105
exc_info(), 135
except, 77, 78
excepthook(), 134
Exception, 126
exec, 83
exec(), 83, 105, 120
exec_prefix, 135
execfile(), 121
execl(), 152
execle(), 152
execlp(), 152
executable, 135
execute(), 181
executemany(), 182
execv(), 152
execve(), 152
execvp(), 152
execvpe(), 152
exists(), 155
exit(), 135
expand(), 161
expandtabs(), 34
expanduser(), 155
expandvars(), 155
extend(), 40
extsep, 143

F

False, 16
fdopen(), 147
fetchall(), 182
fetchmany(), 182
fetchone(), 182
file(), 48, 122
fileno(), 50
filter(), 106
finally, 78
find(), 32
findall(), 159, 160
finditer(), 159, 160
flags, 159
float(), 21, 106
FloatingPointError, 127
flush(), 50
for, 65
fork(), 153
format nazwy, 57
format(), 30, 106
formatowanie łańcuchów, 25
Formatter, 141
framework ostrzeżeń, 131
from, 74
fromkeys(), 46
frozenset(), 106
fstat(), 147
ftplib, 174
ftruncate(), 147
funkcja-fabryka, 69
funkcje, 66
 adnotacje, 67
 argumenty, 66
 dekoratory, 69
 Python 2.X, 120
 Python 3.0, 120
 wbudowane funkcje, 84, 102
 wyrażenia lambda, 68
funkcje okalające, 84
FutureWarning, 131

G

GeneratorExit, 127
generatory, 43, 71
get(), 45
getatime(), 155
getattr(), 106
getcheckinterval(), 135
getcwd(), 146, 148
getdefaultencoding(), 135
geteuid(), 153
getfilesystemencoding(), 136
getmtime(), 155
getpid(), 153
getppid(), 153
getrecursionlimit(), 136
getrefcount(), 136
getsize(), 155
getsizeof(), 136
getuid(), 153
glob, 142
global, 71, 84
globals(), 107
gniazda, 88
graficzny interfejs użytkownika, 169
group(), 160
groupdict(), 161
groupindex, 159
groups(), 161
GUI, 169

H

has_key(), 46
hasattr(), 107
hash(), 107
help(), 56, 107
hex(), 107
hexdigits, 141
hexversion, 136
html, 175
htmllib, 175
http.client, 175
http.server, 175
httplib, 175

I

- id(), 107
- idiomy, 182
- IDLE, 186
- if, 64
- imaplib, 175
- import, 72, 133
- ImportError, 127
- importowanie modułów, 72
- importowanie pakietów, 73
- importowanie względem katalogu
 - pakietów, 75
- ImportWarning, 131
- indeksowanie, 19
- IndentationError, 127
- index(), 33, 40
- IndexError, 127
- input(), 108, 122
- insert(), 40
- instrukcje, 56, 59
 - Python 2.X, 83
- int(), 21, 108
- integer, 21
- interfejs API bazy danych, 179
- intern(), 122, 136
- internet, 173
- IOError, 127
- isabs(), 155
- isatty(), 50, 147
- isdir(), 156
- isfile(), 156
- isinstance(), 108
- islink(), 156
- ismount(), 156
- issubclass(), 108
- items(), 45
- iter(), 109
- iteratory, 65, 71
- iteritems(), 46
- iterkeys(), 46
- itervalues(), 46

J

- jednostka programu, 54
- join(), 33, 156

K

- KeyboardInterrupt, 128
- KeyError, 128
- keys(), 45
- kill(), 153
- klasy, 75, 86
 - atrybuty, 86
 - atrybuty prywatne, 87
 - dekoratory, 76
 - dziedziczenie, 86
 - egzemplarze, 86
 - metody, 86
 - nowy styl, 88
- klasy wyjątków, 80
- kody formatowania łańcuchów, 27
- kolekcje, 93
- komentarze, 56
- konwencje nazewnictwa, 57, 58
- konwersje typów, 55
- krotki, 47
- kursor, 181

L

- lambda, 68
- last_traceback, 137
- last_type, 137
- last_value, 137
- len(), 109
- liczby, 18, 20, 21, 95
 - działania, 21
 - literały, 20
 - ułamki, 21
- linesep, 144
- link(), 149
- list(), 109
- listdir(), 149
- listy, 39
 - wyrażenia generatorowe, 42

- listy składane, 13, 41
- literały łańcuchowe Unicode, 24
- ljust(), 34
- locals(), 109
- logiczny typ danych, 54
- long(), 122
- LookupError, 126
- lower(), 34
- lseek(), 147
- lstat(), 149
- lstrip(), 34

Ł

- łańcuchy znaków, 22, 23
 - byte, 30, 37
 - bytearray, 30, 37
 - działania, 24
 - formatowanie, 25, 34
 - literały, 23
 - łączenie, 33
 - rozdzielanie, 33
 - sekwencje specjalne, 25
 - string, 35
 - String, 30
 - testy zawartości, 35
 - Unicode, 36, 37, 38
 - wyrażenia formatujące, 26
 - wyszukiwanie, 32
 - zastępowanie szablonów, 29

M

- makedirs(), 149
- maketrans(), 141
- map(), 109
- mapy, 93
- match(), 158, 160
- math, 176
- max(), 110
- maxsize, 137
- maxunicode, 137
- MemoryError, 128
- memoryview(), 110, 120

- menedżery kontekstu, 51, 99
- metaklasy, 76
- metody, 86
- metody przeciążające operatory, 89
 - działania dwuargumentowe, 95
 - działania dwuargumentowe
 - z aktualizacją w miejscu, 97
 - liczby, 98
 - menedżery kontekstu, 99
 - prawostronne metody działań dwuargumentowych, 96
 - Python 2.X, 99, 100
 - Python 3.0, 99
- min(), 110
- makedirs(), 149
- mknfif(), 149, 153
- modele klas, 88
- modules, 137
- moduły, 72, 133
 - atrybuty prywatne, 87
 - datetime, 177
 - dbm, 163, 165
 - math, 176
 - obsługa internetu, 173
 - obsługa wątków, 177
 - os, 142
 - os.path, 154
 - pickle, 163, 166
 - re, 157
 - shelve, 164, 165
 - string, 140
 - sys, 134
 - time, 176
 - tkinter, 169
- multiprocessing, 142

N

- name, 143
- NameError, 128
- narzędzia administracyjne, 142
- narzędzia obsługi środowiska, 145
- nazwy, 57
 - kwalifikowane, 83
 - niekwalifikowane, 84

- nazwy ścieżek, 148
- next(), 111
- nice(), 153
- nntplib, 175
- nonlocal, 72
- normcase(), 156
- normpath(), 156
- NotImplementedError, 128
- NumPy, 186

O

- obiekty, 86
 - przestrzenie nazw, 83
- obiekty połączeń, 180
- obiekty składowe, 43
- obiekty wyrażeń regularnych, 159
- object(), 111
- obsługa środowiska, 145
- obsługa Unicode, 37, 38
- oct(), 111
- octdigits, 141
- odwzorowania, 18
- okna dialogowe, 171
- OODB, 186
- OOP, 86
- opcje wiersza poleceń, 9
- open(), 48, 111, 124, 148
- operacje logiczne, 16
- operatory, 13, 14
 - przeciążanie, 89
- ord(), 114
- os, 142
- os.path, 154
- OSError, 128, 142
- ostrzeżenia, 130
 - framework, 131
- OverflowError, 128

P

- paramstyle, 180
- paridir, 143
- parsowanie danych binarnych, 178

- pass, 65
- path, 137, 143
- pathsep, 143
- pattern, 160
- PendingDeprecationWarning, 131
- pętle, 65
- pickle, 163, 166
 - odtwarzanie, 167
 - utrwalanie, 167
- pipe(), 148, 153
- platform, 137
- pliki, 48, 111
 - atrybuty, 51
 - buforowanie, 52
 - menedżery kontekstu, 51
 - pliki wejściowe, 48
 - pliki wyjściowe, 49
 - tryby otwarcia, 51
- plock(), 153
- polecenia powłoki, 144
- pop(), 40, 46
- popen(), 144
- popitem(), 46
- poplib, 175
- porównania, 16
- pos, 160
- potoki, 144
- pow(), 114
- prefix, 138
- print, 62, 63, 114, 120
- printable, 141
- priorytet operatorów, 13
- procesy, 151
- programowanie obiektowe, 86
- property(), 115
- prywatne atrybuty klas, 87
- prywatne atrybuty modułów, 87
- przechwytywanie wyjątków, 77
- przeciążanie operatorów, 89
- przepływ sterowania, 56
- przestrzenie nazw, 71, 83, 86
- przypisanie, 59
 - przypisanie sekwencji, 60
 - przypisanie z aktualizacją, 60

- ps1, 138
- ps2, 138
- punctuation, 141
- putenv(), 146
- PyInstaller, 186
- PyQT, 169
- python, 9
- PYTHONCASEOK, 12
- PYTHONDEBUG, 12
- PYTHONDONTWRITE
 - ↳BYTECODE, 12
- PYTHONHOME, 12
- PYTHONINSPECT, 13
- PYTHONIOENCODING, 12
- PYTHONNOUSERSITE, 13
- PYTHONOPTIMIZE, 13
- PYTHONPATH, 12
- PYTHONSTARTUP, 12
- PYTHONUNBUFFERED, 13
- PYTHONVERBOSE, 13

Q

- queue, 142, 178
- quopri, 175

R

- raise, 79
 - Python 2.X, 80
- range(), 115
- raw strings, 23
- raw_input(), 123
- re, 157, 160
- read(), 48, 49, 148
- readline(), 49
- readlines(), 49
- readlink(), 149
- realpath(), 156
- reduce(), 123
- ReferenceError, 128
- reguły składniowe, 56
- reguły zasięgu, 83
- reload(), 123

- remove(), 40, 150
- removedirs(), 150
- rename(), 150
- renames(), 150
- replace(), 34
- repr(), 115
- return, 70
- reverse(), 40
- reversed(), 115
- rfind(), 33
- rindex(), 33
- rjust(), 35
- rmdir(), 150
- rollback(), 181
- round(), 116
- rowcount, 181
- rozszerzone instrukcje
 - przypisania sekwencji, 60
- rstrip(), 34
- RuntimeError, 129
- RuntimeWarning, 131

S

- samefile(), 156
- sameopenfile(), 156
- samestat(), 156
- SciPy, 186
- search(), 158, 160
- seek(), 50
- sekwencje, 17, 19, 93
- sekwencje mutowalne, 17
- select, 174
- sep, 143
- set, 22
- set(), 116
- setattr(), 116
- setcheckinterval(), 138
- setdefaultencoding(), 138
- setprofile(), 139
- setrecursionlimit(), 139
- settrace(), 139
- shelve, 164, 165
- signal, 142
- składnia języka, 56

- składnia wzorców wyrażeń
 - regularnych, 161, 162
- skróty, 90
- skrypty CGI, 174
- sleep(), 177
- slice(), 116
- słowa zarezerwowane, 57
- słowniki, 18, 43, 44
 - działania, 45
- słowniki składane, 43
- smtpplib, 175
- socket, 142, 173
- socketserver, 174
- sort(), 40
- sorted(), 116
- span(), 161
- spawn*(), 145, 153, 154
- specyfikacja programu, 10
- split(), 33, 157, 158, 160
- splitdrive(), 157
- splittext(), 157
- splitlines(), 34
- SQL, 179
- standardowe moduły biblioteczne, 133
- start(), 161
- startfile(), 144
- startswith(), 33
- stat(), 150
- staticmethod(), 117
- stderr, 139
- stdin, 11, 139
- stdout, 139
- StopIteration, 129
- str, 22
- str(), 117
- strerror(), 146
- string, 35, 140, 160
 - funkcje, 140
 - klasy, 140
 - stałe, 141
- String, 30, 31
- strip(), 34
- struct, 178
- strumienie, 134, 139
- sub(), 159, 160

- subn(), 159, 160
- subprocess, 142
- sum(), 118
- super(), 118
- surowe łańcuchy znaków, 23
- swapcase(), 34
- SWIG, 186
- symlink(), 150
- synonimy, 74
- SyntaxError, 129
- SyntaxWarning, 131
- sys, 134
- sys.exc_info(), 78
- system(), 144
- SystemError, 129
- SystemExit, 129

Ś

- ścieżki dostępu, 154

T

- TabError, 130
- Tcl/Tk, 172
- tell(), 50
- telnetlib, 175
- tempfile, 142
- Template, 141
- testy diagnostyczne, 81
- threading, 142, 178
- time, 176
- time(), 177
- times(), 146
- title(), 35
- tkColorChooser, 171
- tkFileDialog, 171
- tkinter, 169
 - klasy, 171
 - okna dialogowe, 171
 - użycie modułu, 169
 - widżety, 169
- tkinter.colorchooser, 171
- tkinter.filedialog, 171
- tkinter.messagebox, 171

- tkinter.simpledialog, 171
- tkMessageBox, 171
- tkSimpleDialog, 171
- tracebacklimit, 140
- translate(), 35
- True, 16
- truncate(), 50
- try, 77, 79
- tuple(), 118
- type(), 119
- TypeError, 130
- typy danych, 88
 - typy logiczne, 54
 - typy numeryczne, 22
 - typy wbudowane, 20

U

- ułamki, 21
- umask(), 146
- uname(), 146
- UnboundLocalError, 130
- unichr(), 123
- Unicode, 24, 36
- unicode(), 123
- UnicodeDecodeError, 130
- UnicodeEncodeError, 130
- UnicodeError, 130
- UnicodeTranslateError, 130
- UnicodeWarning, 131
- unittest, 185, 186
- unlink(), 150
- update(), 45
- upper(), 34
- urllib, 174
- urllib.parse, 174
- urllib.request, 174
- urllib2, 174
- urlparse, 174
- UserWarning, 130
- utime(), 150
- utrwalanie obiektów, 163
- uu, 175

V

- ValueError, 130
- values(), 45
- vars(), 119
- version, 140
- version_info, 140

W

- wait(), 154
- waitpid(), 154
- walk(), 151, 157
- Warning, 130, 180
- warnings.warn(), 131
- wartości domyślne, 68
- wątki, 177
- wbudowane atrybuty, 132
- wbudowane wyjątki, 125
- while, 64
- whitespace, 141
- widżety, 169
- wielokropek, 15
- wiersz poleceń, 9
- WindowsError, 130
- winver, 140
- with, 81
 - wiele menedżerów kontekstu, 82
- write(), 49, 148
- writelines(), 49
- wxPython, 169
- wycinanie, 19
- wyjątki, 77, 125
 - klasy wyjątków, 80, 125
 - ostrzeżenia, 130
 - przechwytywanie, 77
 - Python 2.X, 132
 - zgłaszanie, 79
- wyrażenia, 61
- wyrażenia formatujące łańcuchy
 - znaków, 26
- wyrażenia generatorowe, 42
- wyrażenia lambda, 68

wyrażenia regularne, 157

 obiekty, 159

 obiekty dopasowania, 160

 składnia wzorców, 161, 162

wywołanie, 61

wywołanie metody formatującej, 27

wzorce dziedziczenia

 wielokrotnego, 88

X

xdrlib, 174

xml, 175

xmlrpc, 175

xrange(), 124

Y

yield, 15, 43, 70

Z

zamknięcie pliku, 50

zarządzanie procesami, 151

zasięg, 83

 leksykalne, 84

 niekwalifikowane nazwy, 85

 zagnieżdżenie statyczne, 84

zastępowanie szablonów

 w łańcuchach znaków, 29

zbiory, 52, 53

zbiory składane, 15, 43

ZeroDivisionError, 130

zfill(), 35

zgłaszanie wyjątków, 79

zip(), 119

złożenia, 71

zmienne opcji wiersza poleceń, 12

zmienne środowiskowe, 12

ZODB, 186

Python. Leksykon kieszonkowy



Python to wyjątkowo praktyczny język programowania, idealnie nadający się do szybkiego rozwiązywania niecodziennych problemów, z którymi często borykają się koderzy. Nie wymusza on stosowania jednego stylu programowania, co pozwala na dużo większą elastyczność w trakcie pisania kodu. Umożliwia programowanie obiektowe, strukturalne i funkcyjne, a ponadto udostępnia zaawansowane mechanizmy zarządzania pamięcią, zapewnia dynamiczne sprawdzanie typów oraz czytelną składnię. Te wszystkie zalety powodują, że Python ma grupę swoich wiernych fanów.

Niniejsza książka należy do popularnej serii „Leksykon kieszonkowy”, dzięki której zawsze i wszędzie możesz przypomnieć sobie wybrane zagadnienia, związane z różną tematyką. Pozycja, którą właśnie trzymasz w rękach, została poświęcona językowi Python. W trakcie jej lektury zapoznasz się z takimi zagadnieniami, jak sterowanie przepływem programu, wykorzystanie pętli, list, słowników oraz operacje na plikach. Ponadto w każdej chwili będziesz mógł sprawdzić składnię oraz sposoby wykorzystania funkcji i wyjątków wbudowanych. Książka stanowi znakomite kompendium wiedzy na temat języka Python. Sprawdzi się ona w rękach początkującego użytkownika – jako przewodnik, a w rękach zaawansowanego programisty – jako pomocnik.

- Wbudowane typy i operatory
- Wykorzystanie przestrzeni nazw
- Działania na liczbach
- Zasięgi zmiennych
- Operacje na łańcuchach znaków
- Przeciążanie operatorów
- Wykorzystanie Unicode w Pythonie
- Standardowe moduły biblioteczne
- Obsługa list oraz słowników
- Zastosowanie wyrażeń regularnych
- Operacje na zbiorach i plikach
- Tworzenie graficznego interfejsu użytkownika
- Sterowanie przepływem programu
- Konwersja typów
- Obsługa wyjątków

Wyciśnij jeszcze więcej z języka Python!



Helion

Sprawdź najnowsze promocje:

➔ <http://helion.pl/promocje>

Książki najchętniej czytane:

➔ <http://helion.pl/bestsellery>

Zamów informacje o nowościach:

➔ <http://helion.pl/howosci>

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel.: 32 230 98 63

e-mail: helion@helion.pl

<http://helion.pl>

helion.pl
księgarnia
internetowa

Nr katalogowy: **5605**



Księgarnia internetowa:

<http://helion.pl>



Zamówienia telefoniczne:

0 801 339900



0 601 339900

Cena 29,90 zł

ISBN 978-83-246-2686-1



9 788324 626861

Informatyka w najlepszym wydaniu