Python

Zwięzłe kompendium dla programisty

Helion

David M. Beazley

Tytuł oryginału: Python Distilled

Tłumaczenie: Łukasz Wójcicki

ISBN: 978-83-283-9019-5

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

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
https://ftp.helion.pl/przyklady/pyzwko.zip

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

Printed in Poland.

Spis treści

2

Operatory, wyrażenia i manipulacja danymi

Ten rozdział opisuje wyrażenia, operatory oraz sposoby operowania na danych. Wyrażenia są podstawą wykonywania użytecznych obliczeń. Co więcej, biblioteki innych firm mogą dostosować zachowanie Pythona, aby zapewnić lepsze wrażenia użytkownika. Ten rozdział opisuje wyrażenia wysokiego poziomu. Rozdział 3. omawia podstawowe protokoły, których można użyć do dostosowania zachowania interpretera.

2.1. Literały

Literał to wartość wpisana bezpośrednio do programu, taka jak `42`, `4.2` lub `'czterdziesci-dwa'`.

Literały całkowite reprezentują liczbę całkowitą ze znakiem o dowolnym rozmiarze. Możliwe jest określenie liczb całkowitych w formacie binarnym, ósemkowym lub szesnastkowym:

```
42          # Dziesiętna liczba całkowita
0b101010    # Binarna liczba całkowita
0o52        # Ósemkowa liczba całkowita
0x2a        # Szesnastkowa liczba całkowita
```

Baza nie jest przechowywana jako część wartości całkowitej. Wszystkie powyższe literały, jeśli zostaną wyświetlone, będą wyświetlane jako liczba 42. Możesz użyć wbudowanych funkcji bin(x), oct(x) lub hex(x) do konwersji liczby całkowitej na łańcuch reprezentujący jej wartość w różnych podstawach.

Liczby zmiennoprzecinkowe można zapisać, dodając kropkę dziesiętną lub używając notacji naukowej, gdzie `e` lub `E` określa wykładnik. Wszystkie poniższe są liczbami zmiennoprzecinkowymi:

```
4.2
42.
0,42
```

```
4.2e+2
4.2E2
-4.2e-2
```

Wewnętrznie liczby zmiennoprzecinkowe są przechowywane jako wartości podwójnej precyzji IEEE 754 (64-bitowe).

W literałach numerycznych pojedynczy znak podkreślenia (_) może służyć jako wizualny separator między cyframi. Na przykład:

```
123_456_789
0x1234_5678
0b111_00_101
123.789_012
```

Separator cyfr nie jest przechowywany jako część liczby — służy tylko do ułatwienia odczytywania dużych literałów numerycznych w kodzie źródłowym.

Literały logiczne są zapisywane jako `True` i `False`.

Literały ciągów są zapisywane przez umieszczanie znaków w pojedynczych, podwójnych lub potrójnych cudzysłowach. Ciągi w cudzysłowie pojedynczym i podwójnym muszą się znajdować w tym samym wierszu. Ciągi w potrójnym cudzysłowie mogą obejmować wiele wierszy. Na przykład:

```
'Witaj, świecie'
"Witaj, świecie"
'''Witaj, świecie'''
"""Witaj, świecie"""
```

Literały krotki, listy, zbioru i słownika są zapisywane w następujący sposób:

```
(1, 2, 3)              # krotka
[1, 2, 3]              # lista
{1, 2, 3}              # zbiór
{'x':1, 'y':2, 'z':3}  # słownik
```

2.2. Wyrażenia i lokalizacje

Wyrażenie reprezentuje obliczenie, którego wynikiem jest konkretna wartość. Składa się z kombinacji literałów, nazw, operatorów i wywołań funkcji lub metod. Wyrażenie może się zawsze pojawić po prawej stronie instrukcji przypisania, może być używane jako operand w operacjach w innych wyrażeniach lub przekazywane jako argument funkcji. Na przykład:

```
value = 2 + 3 * 5 + sqrt(6+7)
```

Operatory, takie jak + (dodawanie) lub * (mnożenie), reprezentują operację wykonywaną na obiektach dostarczonych jako operandy. `sqrt()` to funkcja stosowana do argumentów wejściowych.

Lewa strona przypisania reprezentuje lokalizację, w której przechowywane jest odniesienie do obiektu. Ta lokalizacja, jak pokazano w poprzednim przykładzie, może być prostym identyfikatorem, takim jak `value`. Może to być również atrybut obiektu lub indeks w kontenerze. Przykład:

```
a = 4 + 2
b[1] = 4 + 2
c['key'] = 4 + 2
d.value = 4 + 2
```

Odczytywanie wartości z lokalizacji jest również wyrażeniem. Na przykład:

```
value = a + b[1] + c['key']
```

Przypisanie wartości i obliczanie wyrażenia to odrębne pojęcia. Nie możesz dołączyć operatora przypisania jako części wyrażenia:

```
while line=file.readline(): # Błąd składni
    print(line)
```

Jednak operator przypisania (`:=`) może być użyty do wykonania tej połączonej akcji obliczania wyrażenia i przypisania. Na przykład:

```
while (line:=file.readline()):
    print(line)
```

Operator `:=` jest zwykle używany w połączeniu z wyrażeniami takimi jak `if` i `while`.

W rzeczywistości użycie go jako normalnego operatora przypisania skutkuje błędem składni, chyba że umieścisz go w nawiasach.

2.3. Standardowe operatory

Obiekty Pythona mogą działać z dowolnymi operatorami z tabeli 2.1.

Zwykle mają one interpretację numeryczną. Istnieją jednak godne uwagi przypadki szczególne. Na przykład operator + jest również używany do łączenia sekwencji, operator * replikuje sekwencje, - jest używany do znajdowania różnic w zbiorach, a % wykonuje formatowanie ciągu:

```
[1,2,3] + [4,5]       # [1,2,3,4,5]
[1,2,3] * 4           # [1,2,3,1,2,3,1,2,3,1,2,3]
'%s ma %d wiadomości' % ('Dave', 37)
```

Sprawdzanie operatorów jest procesem dynamicznym. Operacje obejmujące mieszane typy danych często „działają”, jeśli istnieje intuicyjny sens działania. Na przykład możesz dodać liczby całkowite i ułamki:

```
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = 5
>>> a + b
Fraction(17, 3)
>>>
```

Tabela 2.1. *Operatory standardowe*

Operacja	Opis
`x + y`	Dodawanie
`x – y`	Odejmowanie
`x * y`	Mnożenie
`x / y`	Dzielenie
`x // y`	Dzielenie całkowite
`x @ y`	Mnożenie macierzy
`x ** y`	Potęga (`x` do potęgi `y`)
`x % y`	Modulo (`x` modulo `y`). Reszta
`x << y`	Przesunięcie w lewo
`x >> y`	Przesunięcie w prawo
`x & y`	Bitowe i
`x | y`	Bitowe lub
`x ^ y`	Bitowe xor (alternatywa wykluczająca)
`~x`	Negacja bitowa
`–x`	Jednoargumentowy minus
`+x`	Jednoargumentowy plus
`abs(x)`	Wartość bezwzględna
`divmod(x, y)`	Zwraca `(x // y, x % y)`
`pow(x, y [, modulo])`	Zwraca `(x ** y) % modulo`
`round(x [, n])`	Zaokrągla do `n` miejsc po przecinku

Takie operacje jednak nie zawsze działają — na przykład nie działają na ułamkach dziesiętnych.

```
>>> from decimal import Decimal
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Decimal('5')
>>> a + b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Fraction' and 'decimal.Decimal'
>>>
```

Jednak w przypadku większości kombinacji liczb Python stosuje standardową hierarchię wartości logicznych, liczb całkowitych, ułamków, liczb zmiennoprzecinkowych i liczb zespolonych. Operacje typu mieszanego po prostu zadziałają — nie musisz się tym martwić.

2.4. Modyfikacje w miejscu

Python udostępnia operacje modyfikacji „w miejscu” (ang. *in-place*) lub „rozszerzone”, wymienione w tabeli 2.2.

Tabela 2.2. *Rozszerzone operatory przypisania*

Operacja	Opis
`x += y`	`x = x + y`
`x –= y`	`x = x – y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x //= y`	`x = x // y`
`x **= y`	`x = x ** y`
`x %= y`	`x = x % y`
`x @= y`	`x = x @ y`
`x &= y`	`x = x & y`
`x |= y`	`x = x | y`
`x ^= y`	`x = x ^ y`
`x >>= y`	`x = x >> y`
`x <<= y`	`x = x << y`

Nie są one uważane za wyrażenia. Zamiast tego służą do aktualizacji wartości w miejscu. Na przykład:

```
a = 3
a = a + 1    # a = 4
a += 1       # a = 5
```

Obiekty mutowalne mogą używać tych operatorów do wykonywania mutacji danych w miejscu jako optymalizacji. Rozważ ten przykład:

```
>>> a = [1, 2, 3]
>>> b = a                 # Tworzy nowe odniesienie do a
>>> a += [4, 5]           # Aktualizacja w miejscu (nie tworzy nowej listy)
>>> a
[1, 2, 3, 4, 5]
>>> b
[1, 2, 3, 4, 5]
>>>
```

W tym przykładzie `a` i `b` są odwołaniami do tej samej listy. Gdy wykonywane jest `a += [4, 5]`, przeprowadzana jest aktualizacja obiektu listy w miejscu bez tworzenia nowej listy. W związku z tym `b` także widzi tę aktualizację. To często zaskakujące.

2.5. Porównywanie obiektów

Operator równości (`x == y`) testuje wartości `x` i `y` pod kątem równości. W przypadku list i krotek muszą one mieć taki sam rozmiar, posiadać takie same elementy i muszą być one umieszczone w tej samej kolejności. W przypadku słowników wartość `True` jest zwracana tylko wtedy, gdy `x` i `y` mają ten sam zestaw kluczy, a wszystkie obiekty z tym samym kluczem mają równe wartości. Dwa zbiory są równe, jeśli mają te same elementy.

Sprawdzenie równości między obiektami niezgodnych typów, takich jak plik i liczba zmiennoprzecinkowa, nie powoduje błędu, ale zwracana jest wartość `False`. Jednak czasami porównanie obiektów różnych typów da wynik `True`. Na przykład porównanie liczby całkowitej i liczby zmiennoprzecinkowej o tej samej wartości:

```
>>> 2 == 2.0
True
>>>
```

Operatory tożsamości (`x is y` i `x is not y`) testują dwie wartości, aby sprawdzić, czy odnoszą się one do tego samego obiektu w pamięci (np. `id(x) == id(y)`). Ogólnie może być tak, że `x == y`, ale `x is not y`. Na przykład:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
>>> a == b
True
>>>
```

W praktyce porównywanie obiektów za pomocą operatora `is` prawie nigdy nie jest tym, co chcesz uzyskać. Użyj operatora == dla wszystkich porównań, chyba że masz dobry powód, aby oczekiwać, że oba obiekty będą tożsame.

2.6. Operatory porównania porządkowego

Operatory porównania porządkowego z tabeli 2.3 mają standardową interpretację matematyczną dla liczb. Zwracają wartość logiczną.

Tabela 2.3. *Operatory porównania porządkowego*

Operacja	Opis
`x < y`	Mniejsze niż
`x > y`	Większe niż
`x >= y`	Większe lub równe
`x <= y`	Mniejsze lub równe

W przypadku zbiorów operacja `x < y` sprawdza, czy x jest ścisłym podzbiorem y (tzn. ma mniej elementów, ale nie jest równe y).

Podczas porównywania dwóch sekwencji porównywane są pierwsze elementy każdej z nich. Jeśli się różnią, to decyduje o wyniku całej operacji. Jeśli są takie same, porównanie przenosi się do drugiego elementu każdej sekwencji. Proces ten trwa, dopóki nie zostaną znalezione dwa różne elementy lub nie ma więcej elementów w żadnej z sekwencji. Jeżeli osiągnięto koniec obu sekwencji, sekwencje uważa się za równe. Jeśli `a` jest podciągiem `b`, to `a < b`.

Łańcuchy i bajty są porównywane za pomocą porządkowania leksykograficznego. Każdemu znakowi przypisany jest unikalny indeks numeryczny określony przez zbiór znaków (np. ASCII lub Unicode). Znak jest mniejszy niż inny znak, jeśli jego indeks jest mniejszy.

Nie wszystkie typy obsługują porównania porządkowe. Na przykład próba użycia < w słownikach jest niezdefiniowana i skutkuje błędem TypeError. Podobnie zastosowanie takich porównań do niezgodnych typów (takich jak ciąg i liczba) spowoduje wystąpienie TypeError.

2.7. Wyrażenia logiczne i wartości prawdziwe

Operatory and, or i not mogą tworzyć złożone wyrażenia logiczne. Zachowanie tych operatorów pokazano w tabeli 2.4.

Tabela 2.4. *Operatory logiczne*

Operator	Opis
x or y	Jeśli x ma wartość False, zwróć y; w przeciwnym razie zwróć x.
x and y	Jeśli x ma wartość False, zwróć x; w przeciwnym razie zwróć y.
not x	Jeśli x ma wartość False, zwróć True; w przeciwnym razie zwróć False.

Gdy używasz wyrażenia do określenia wartości True lub False, wartości True, dowolna niezerowa liczba, niepusty ciąg, lista, krotka lub słownik są uważane za wartość True. Wartości False, zero, None, puste listy, krotki i słowniki są oceniane jako wartość False.

Wyrażenia logiczne są liczone od lewej do prawej i wykorzystują prawy operand tylko wtedy, gdy jest to konieczne do określenia końcowej wartości. Na przykład a and b sprawdza wartość b tylko wtedy, gdy a ma wartość True. Ta technika nazywa się *skróconym wartościowaniem* (ang. *short-circuit evaluation*). Przydatne może być uproszczenie kodu obejmującego test i późniejszą operację. Na przykład:

```
if y != 0:
    result = x / y
else:
    result = 0

# Alternatywnie
result = y and x / y
```

W drugiej wersji podział x / y jest wykonywany tylko wtedy, gdy y jest niezerowe.

Poleganie na niejawnej „prawdziwości" obiektów może prowadzić do trudnych do znalezienia błędów. Rozważmy na przykład tę funkcję:

```
def f(x, items=None):
    if not items:
        items = []
    items.append(x)
    return items
```

Ta funkcja ma opcjonalny argument, który — jeśli nie zostanie podany — spowoduje utworzenie i zwrócenie nowej listy. Na przykład:

```
>>> foo(4)
[4]
>>>
```

Jednak funkcja zachowuje się naprawdę dziwnie, jeśli jako argument podasz jej istniejącą pustą listę:

```
>>> a = []
>>> foo(3, a)
[3]
>>> a  # Zwróć uwagę, że a NIE zostało zaktualizowane
[]
>>>
```

To błąd oparty na sprawdzaniu prawdziwości. Puste listy mają wartość `False`, więc kod utworzył nową listę, zamiast używać tej (`a`), która została przekazana jako argument. Aby to naprawić, musisz być bardziej precyzyjny w sprawdzaniu wartości `None`:

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

Dobrą praktyką jest zawsze precyzyjne stosowanie sprawdzeń warunkowych.

2.8. Wyrażenia warunkowe

Typowym wzorcem programowania jest warunkowe przypisanie wartości na podstawie wyniku wyrażenia. Na przykład:

```
if a <= b:
    minvalue = a
else:
    minvalue = b
```

Ten kod można skrócić za pomocą wyrażenia warunkowego. Na przykład:

```
minvalue = a if a <= b else b
```

W takich wyrażeniach najpierw oceniany jest warunek znajdujący się w środku. Jeśli wynikiem jest `True`, brane jest pod uwagę wyrażenie po lewej stronie `if`. W przeciwnym razie liczone jest wyrażenie po `else`. Klauzula `else` jest zawsze wymagana.

2.9. Operacje obejmujące elementy iterowalne

Iteracja jest ważną funkcją Pythona obsługiwaną przez wszystkie kontenery Pythona (listy, krotki, słowniki itd.), pliki, a także funkcje generatora. Operacje z tabeli 2.5 można zastosować do dowolnych iterowalnych obiektów.

Najważniejszą operacją na obiekcie iterowalnym jest pętla `for`. W ten sposób przechodzisz przez kolejne wartości. Wszystkie inne operacje opierają się na tym działaniu.

Tabela 2.5. *Operacje na obiektach iteracyjnych*

Operacja	Opis
`for vars in s:`	Iteracja
`v1, v2, ... = s`	Rozpakowywanie zmiennych
`x in s, x not in s`	Członkostwo
`[a, *s, b], (a, *s, b), {a, *s, b}`	Rozszerzanie list, krotek lub zbiorów

Operator `x in s` sprawdza, czy obiekt `x` pojawia się jako jeden z elementów generowanych przez iterowalne `s`, i zwraca `True` lub `False`. Operator `x not in s` jest tym samym co `not (x in s)`. W przypadku ciągów operatory `in` i `not in` akceptują podciągi. Na przykład `'hello' in 'hello world'` daje wartość `True`. Należy zauważyć, że operator `in` nie obsługuje symboli wieloznacznych ani żadnego rodzaju dopasowywania wzorców.

Każdy obiekt obsługujący iterację może mieć swoje wartości rozpakowane w serii lokalizacji. Na przykład:

```
items = [3, 4, 5]
x, y, z = items       # x = 3, y = 4, z = 5

letters = "abc"
x, y, z = letters     # x = 'a', y = 'b', z = 'c'
```

Lokalizacje po lewej stronie nie muszą być prostymi nazwami zmiennych. Dopuszczalna jest każda prawidłowa lokalizacja, która może się pojawić po lewej stronie znaku równości. Możesz więc napisać kod w ten sposób:

```
items = [3, 4, 5]
d = { }
d['x'], d['y'], d['z'] = items
```

Podczas rozpakowywania wartości do lokalizacji liczba lokalizacji po lewej stronie musi dokładnie odpowiadać liczbie elementów w iterowalnym obiekcie po prawej stronie. W przypadku zagnieżdżonych struktur danych dopasuj lokalizacje i dane, stosując ten sam wzorzec strukturalny. Rozważmy następujący przykład rozpakowywania dwóch zagnieżdżonych krotek:

```
datetime = ((5, 19, 2008), (10, 30, "am"))
(month, day, year), (hour, minute, am_pm) = datetime
```

Czasami zmienna _ jest używana do wskazania wartości do pominięcia podczas rozpakowywania. Jeśli na przykład zależy Ci tylko na dniu i godzinie, możesz użyć:

```
(_, day, _), (hour, _, _) = datetime
```

Jeśli liczba elementów do rozpakowania nie jest znana, możesz użyć rozszerzonej formy rozpakowywania, dołączając zmienną oznaczoną gwiazdką, taką jak `*extra` w poniższym przykładzie:

```
items = [1, 2, 3, 4, 5]
a, b, *extra = items        # a = 1, b = 2, extra = [3,4,5]
*extra, a, b                # extra = [1,2,3], a = 4, b = 5
a, *extra, b                # a = 1, extra = [2,3,4], b = 5
```

W tym przykładzie *extra otrzymuje wszystkie dodatkowe elementy. To zawsze jest lista. Podczas rozpakowywania pojedynczego elementu iteracyjnego można użyć nie więcej niż jednej zmiennej oznaczonej gwiazdką. Jednak podczas rozpakowywania bardziej złożonych struktur danych obejmujących różne iterowalne obiekty można użyć wielu zmiennych oznaczonych gwiazdką. Na przykład:

```
datetime = ((5, 19, 2008), (10, 30, "am"))
(month, *_), (hour, *_) = datetime
```

Dowolny element iteracyjny można rozwinąć podczas tworzenia list, krotek i zbiorów. Odbywa się to również za pomocą gwiazdki (*). Na przykład:

```
items = [1, 2, 3]
a = [10, *items, 11]          # a = [10, 1, 2, 3, 11] (lista)
b = (*items, 10, *items)      # b = [1, 2, 3, 10, 1, 2, 3] (krotka)
c = {10, 11, *items}          # c = {1, 2, 3, 10, 11} (zbiór)
```

W tym przykładzie zawartość elementów jest po prostu wklejana do listy, krotki lub tworzonego zbioru, tak jakbyś wpisywał ją w tym miejscu. Ta ekspansja określana jest jako *splatting*. Podczas definiowania literału możesz uwzględnić dowolną liczbę rozszerzeń *. Jednak wiele obiektów iterowalnych (takich jak pliki lub generatory) obsługuje tylko jednorazową iterację. Jeśli użyjesz *, zawartość zostanie zużyta, a element iteracyjny nie będzie generował więcej wartości w kolejnych iteracjach.

Wiele funkcji wbudowanych akceptuje jako dane wejściowe dowolne iterowalne dane. Tabela 2.6 przedstawia niektóre z tych operacji.

Tabela 2.6. *Funkcje operujące na danych iterowalnych*

Funkcja	Opis
`list(s)`	Utwórz listę z s.
`tuple(s)`	Utwórz krotkę z s.
`set(s)`	Utwórz zbiór z s.
`min(s [, key])`	Minimalna pozycja w s.
`max(s [, key])`	Maksymalna pozycja w s.
`any(s)`	Zwraca `True`, jeśli jakikolwiek element w s ma wartość `True`.
`all(s)`	Zwraca `True`, jeśli wszystkie elementy w s mają wartość `True`.
`sum(s [, initial])`	Suma pozycji z opcjonalną wartością początkową.
`sorted(s [, key])`	Tworzy posortowaną listę.

Dotyczy to również wielu innych funkcji bibliotecznych — na przykład funkcji w module `statistics`.

2.10. Operacje na sekwencjach

Sekwencja to iterowalny kontener, który ma rozmiar i umożliwia dostęp do elementów za pomocą indeksu liczb całkowitych zaczynającego się od 0. Przykładami są ciągi, listy i krotki. Oprócz wszystkich operacji związanych z iteracją operatory z tabeli 2.7 można zastosować do sekwencji.

Tabela 2.7. *Operacje na sekwencjach*

Operacja	Opis
`s + r`	Złączenie.
`s * n, n * s`	Tworzy `n` kopii `s`, gdzie `n` jest liczbą całkowitą.
`s[i]`	Indeksowanie.
`s[i:j]`	Wycinanie.
`s[i:j:step]`	Zaawansowane wycinanie.
`len(s)`	Długość.

Operator + łączy dwie sekwencje tego samego typu. Na przykład:

```
>>> a = [3, 4, 5]
>>> b = [6, 7]
>>> a + b
[3, 4, 5, 6, 7]
>>>
```

Operator `s * n` tworzy *n* kopii sekwencji. Są to jednak płytkie kopie, które replikują elementy tylko przez odniesienie. Spójrz na poniższy kod:

```
>>> a = [3, 4, 5]
>>> b = [a]
>>> c = 4 * b
>>> c
[[3, 4, 5], [3, 4, 5], [3, 4, 5], [3, 4, 5]]
>>> a[0] = -7
>>> c
[[-7, 4, 5], [-7, 4, 5], [-7, 4, 5], [-7, 4, 5]]
>>>
```

Zwróć uwagę, jak zmiana elementu `a` modyfikuje każdy element listy `c`. W tym przypadku odwołanie do listy `a` zostało umieszczone w liście `b`. Kiedy `b` zostało zreplikowane, stworzono cztery dodatkowe odniesienia do `a`. Wreszcie, gdy zmodyfikowano `a`, ta zmiana została rozesłana do wszystkich pozostałych kopii `a`. Takie zachowanie mnożenia sekwencji często nie jest intencją programisty. Jednym ze sposobów obejścia tego problemu jest ręczne skonstruowanie zreplikowanej sekwencji poprzez zduplikowanie zawartości `a`. Oto przykład:

```
a = [3, 4, 5]
c = [list(a) for _ in range(4)] # list() tworzy kopię listy
```

Operator indeksowania `s[n]` zwraca *n*-ty obiekt z sekwencji; `s[0]` to pierwszy obiekt. Ujemne indeksy mogą służyć do pobierania znaków z końca sekwencji. Na przykład `s[-1]` zwraca ostatni element. W przeciwnym razie próby uzyskania dostępu do elementów, które są poza zakresem, powodują wystąpienie wyjątku `IndexError`.

Operator wycinania `s[i:j]` wyodrębnia podciąg z `s`, składający się z elementów o indeksie `k`, gdzie `i <= k < j`. Zarówno `i`, jak i `j` muszą być liczbami całkowitymi. Jeżeli indeks początkowy lub końcowy zostanie pominięty, zakłada się odpowiednio początek lub koniec sekwencji. Ujemne indeksy są dozwolone i zakłada się, że odnoszą się do końca ciągu.

Operatorowi wycinania można nadać opcjonalny krok *step*: `s[i:j:step]`, który powoduje, że podczas wycinania zostaną pominięte niektóre elementy. Jednak zachowanie jest nieco bardziej subtelne. Jeśli podano krok, `i` jest indeksem początkowym, `j` jest indeksem końcowym, a otrzymany podciąg to elementy `s[i]`, `s[i+step]`, `s[i+2*step]` i tak dalej, aż do osiągnięcia indeksu `j` (który nie jest uwzględniony). Krok może być również ujemny. Jeśli początkowy indeks `i` zostanie pominięty, to jest ustawiany na początku (jeśli krok jest dodatni) lub na końcu sekwencji (jeśli krok jest ujemny). Jeśli indeks końcowy `j` zostanie pominięty, ustawiany jest na koniec (jeśli krok jest dodatni) lub początek sekwencji (jeśli krok jest ujemny). Oto kilka przykładów:

```
a = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

a[2:5]     # [2, 3, 4]
a[:3]      # [0, 1, 2]
a[-3:]     # [7, 8, 9]
a[::2]     # [0, 2, 4, 6, 8]
a[::-2]    # [9, 7, 5, 3, 1]
a[0:5:2]   # [0, 2, 4]
a[5:0:-2]  # [5, 3, 1]
a[:5:1]    # [0, 1, 2, 3, 4]
a[:5:-1]   # [9, 8, 7, 6]
a[5::1]    # [5, 6, 7, 8, 9]
a[5::-1]   # [5, 4, 3, 2, 1, 0]
a[5:0:-1]  # [5, 4, 3, 2, 1]
```

Skomplikowane wycinanie ciągu może skutkować kodem, który będzie później trudny do zrozumienia. W związku z tym zachowanie pewnego rozsądku jest uzasadnione. Wycinane podciągi można ponazywać za pomocą metody `slice()`. Na przykład:

```
firstfive = slice(0, 5)
s = 'hello world'
print(s[firstfive]) # Wyświetla 'hello'
```

2.11. Operacje na mutowalnych obiektach sekwencyjnych

Ciągi i krotki są niezmienne (ang. *immutable*) i nie można ich modyfikować po utworzeniu. Zawartość listy lub innej sekwencji mutowalnej można modyfikować w miejscu za pomocą operatorów z tabeli 2.8.

Operator `s[i] = x` zmienia element `i` sekwencji tak, aby odnosił się do obiektu `x`, zwiększając liczbę odwołań `x`. Indeksy ujemne są określane względem końca listy, a próba przypisania wartości do indeksu spoza zakresu powoduje wystąpienie wyjątku `IndexError`. Operator przypisania wycinka `s[i:j] = r` zastępuje elementy `k`, gdzie `i <= k < j`, elementami z sekwencji `r`. Indeksy mają takie samo znaczenie jak przy wycinaniu. W razie potrzeby sekwencja `s` może zostać rozszerzona lub zmniejszona, aby pomieścić wszystkie elementy w `r`. Oto przykład:

Tabela 2.8. *Operacje na mutowalnych obiektach sekwencyjnych*

Operacja	Opis
`s[i] = x`	Przypisanie indeksu
`s[i:j] = r`	Przypisanie wycinka
`s[i:j:step] = r`	Zaawansowane przypisanie wycinka
`del s[i]`	Usuwanie elementu
`del s[i:j]`	Usuwanie wycinka
`del s[i:j:step]`	Zaawansowane usuwanie wycinka

```
a = [1, 2, 3, 4, 5]
a[1] = 6                # a = [1, 6, 3, 4, 5]
a[2:4] = [10, 11]       # a = [1, 6, 10, 11, 5]
a[3:4] = [-1, -2, -3]   # a = [1, 6, 10, –1, –2, –3, 5]
a[2:] = [0]             # a = [1, 6, 0]
```

Przypisanie wycinka może być wykonane z opcjonalnym argumentem określającym krok. Zachowanie jest wtedy jednak bardziej ograniczone, ponieważ argument po prawej stronie musi mieć dokładnie taką samą liczbę elementów jak zastępowany wycinek. Oto przykład:

```
a = [1, 2, 3, 4, 5]
a[1::2] = [10, 11]          # a = [1, 10, 3, 11, 5]
a[1::2] = [30, 40, 50]      # ValueError. Tylko dwa elementy w wycinku po lewej stronie.
```

Operator `del s[i]` usuwa element `i` z sekwencji i zmniejsza jego liczbę odwołań. `del s[i:j]` usuwa wszystkie elementy z wycinka. Można również podać krok, jak w `del s[i:j:step]`.

Opisana tutaj semantyka dotyczy wbudowanego typu listy. Operacje obejmujące wycinanie sekwencji to bogaty obszar do dostosowywania w pakietach innych firm. Może się okazać, że operacje wycinania na obiektach spoza listy mają określone różne zasady dotyczące przypisywania, usuwania i współdzielenia obiektów. Na przykład popularny pakiet `numpy` ma inną semantykę wycinania niż listy Pythona.

2.12. Operacje na zbiorach

Zbiór to nieuporządkowana kolekcja unikalnych wartości. Na zbiorach można wykonywać operacje z tabeli 2.9.

Oto kilka przykładów:

```
>>> a = {'a', 'b', 'c'}
>>> b = {'c', 'd'}
>>> a | b
{'a', 'b', 'c', 'd'}
>>> a & b
>>> {'c'}
>>> a - b
{'a', 'b'}
>>> b - a
{'d'}
>>> a ^ b
{'a', 'b', 'd'}
>>>
```

Tabela 2.9. *Operacje na zbiorach*

Operacja	Opis
`s | t`	Złączenie `s` i `t`
`s & t`	Przecięcie `s` i `t`
`s – t`	Różnica zbioru (elementy w `s`, nie w `t`)
`s ^ t`	Różnica symetryczna (elementy niezawarte w `s` lub `t`)
`len(s)`	Liczba elementów w zbiorze
`item in s, item not in s`	Test członkostwa elementu
`s.add(item)`	Dodawanie elementu do zbioru `s`
`s.remove(item)`	Usuwanie elementu z `s`, jeśli istnieje (w przeciwnym razie błąd)
`s.discard(item)`	Odrzucanie elementu z `s`, jeśli istnieje

Operacje na zbiorach działają również na obiektach słownikowych kluczy i elementów. Aby na przykład dowiedzieć się, które klucze są wspólne dla dwóch słowników, wykonaj następujące czynności:

```
>>> a = {'x': 1, 'y': 2, 'z': 3}
>>> b = {'z': 3, 'w': 4, 'q': 5}
>>> a.keys() & b.keys()
{'z'}
>>>
```

2.13. Operacje na mapowaniach

Mapowanie to powiązanie kluczy i wartości. Przykładem jest wbudowany typ `dict`. Do mapowań można zastosować operacje z tabeli 2.10.

Tabela 2.10. *Operacje na mapowaniach*

Operacja	Opis
`x = m[k]`	Indeksowanie kluczem
`m[k] = x`	Przypisanie do klucza
`del m[k]`	Usuwanie elementu klucza
`k in m`	Testowanie członkostwa
`len(m)`	Liczba elementów w mapowaniu
`m.keys()`	Zwraca klucze
`m.values()`	Zwraca wartości
`m.items()`	Zwraca pary (`klucz, wartość`)

Wartości kluczy mogą być dowolnymi niezmiennymi obiektami, takimi jak ciągi, liczby i krotki. Używając krotki jako klucza, możesz pominąć nawiasy i wpisać wartości oddzielone przecinkami w następujący sposób:

```
d = { }
d[1,2,3] = "foo"
d[1,0,3] = "bar"
```

W takim przypadku wartości klucza reprezentują krotkę, dzięki czemu te przypisania są równoważne z następującymi:

```
d[(1,2,3)] = "foo"
d[(1,0,3)] = "bar"
```

Używanie krotki jako klucza jest powszechną techniką tworzenia kluczy złożonych w mapowaniu. Na przykład klucz może się składać z „imienia" i „nazwiska".

2.14. Lista, zbiór i słownik

Jedną z najczęstszych operacji na danych jest przekształcenie zbioru danych w inną strukturę danych. Na przykład tutaj bierzemy wszystkie elementy listy, stosujemy operację i tworzymy nową listę:

```
nums = [1, 2, 3, 4, 5]
squares = []
for n in nums:
    nums.append(n * n)
```

Ponieważ ten rodzaj operacji jest bardzo powszechny, jest dostępny jako operator zwany listą składaną. Oto bardziej kompaktowa wersja tego kodu:

```
nums = [1, 2, 3, 4, 5]
squares = [n * n for n in nums]
```

Możliwe jest również zastosowanie filtra do operacji:

```
squares = [n * n for n in nums if n > 2] # [9, 16, 25]
```

Ogólna składnia list składanych jest następująca:

```
[expression for item1 in iterable1 if condition1
            for item2 in iterable2 if condition2
            ...
            for itemN in iterableN if conditionN]
```

Ta składnia jest równoważna z następującym kodem:

```
result = []
for item1 in iterable1:
    if condition1:
        for item2 in iterable2:
            if condition2:
                ...
                for itemN in iterableN:
                    if conditionN:
                        result.append(expression)
```

Listy składane są bardzo przydatnym sposobem przetwarzania danych list w różnych formach. Oto kilka praktycznych przykładów:

```
# Niektóre dane (lista słowników)
portfolio = [
  {'name': 'IBM', 'shares': 100, 'price': 91.1},
```

```
  {'name': 'MSFT', 'shares': 50, 'price': 45.67},
  {'name': 'HPE', 'shares': 75, 'price': 34.51},
  {'name': 'CAT', 'shares': 60, 'price': 67.89},
  {'name': 'IBM', 'shares': 200, 'price': 95.25}
]

# Zbierz wszystkie nazwy ['IBM', 'MSFT', 'HPE', 'CAT', 'IBM']
name = [s['name'] for s in portfolio]

# Znajdź wszystkie wpisy z ponad 100 udziałami ['IBM']
more100 = [s['name'] for s in portfolio if s['shares'] > 100]

# Znajdź łączną wartość udziały*cena
cost = sum([s['shares']*s['price'] for s in portfolio])

# Zbierz krotki (nazwisko, udziały)
name_shares = [(s['name'], s['shares']) for s in portfolio]
```

Wszystkie zmienne użyte wewnątrz listy składanej są prywatne dla tej listy. Nie musisz się martwić, że takie zmienne nadpiszą inne zmienne o tej samej nazwie. Na przykład:

```
>>> x = 42
>>> squares = [x*x for x in [1,2,3]]
>>> squares
[1, 4, 9]
>>> x
42
>>>
```

Zamiast tworzyć listę, możesz również utworzyć zbiór, zmieniając nawiasy w nawiasy klamrowe. Nazywa się to zbiorem składanym. Zbiór składany da Ci zestaw odrębnych wartości. Na przykład:

```
# Zbiór składany
names = {s['name'] for s in portfolio}
# names = {'IBM', 'MSFT', 'HPE', 'CAT'}
```

Jeśli określisz pary klucz-wartość, zamiast tego utworzysz słownik. Jest to słownik składany. Na przykład:

```
prices = {s['name']:s['price'] for s in portfolio}
# prices = {'IBM': 95.25, 'MSFT': 45.67, 'HPE': 34.51, 'CAT': 67.89}
```

Tworząc zbiory i słowniki, należy pamiętać, że późniejsze wpisy mogą nadpisać wcześniejsze. Na przykład w słowniku `prices` otrzymujesz ostatnią cenę za `'IBM'`. Pierwsza cena przepada.

W ramach składania nie można obsłużyć wyjątków. Jeśli jest to problem, rozważ opakowanie obsługi wyjątków za pomocą funkcji, jak pokazano tutaj:

```
def toint(x):
    try:
        return int(x)
    except ValueError:
        return None

values = ['1', '2', '-4', 'n/a', '-3', '5']
```

```
data1 = [toint(x) for x in values]
# data1 = [1, 2, –4, None, –3, 5]

data2 = [toint(x) for x in values if toint(x) is not None]
# data2 = [1, 2, –4, –3, 5]
```

W ostatnim przykładzie można uniknąć podwójnego liczenia `toint(x)`, używając operatora `:=`. Na przykład:

```
data3 = [v for x in values if (v:=toint(x)) is not None]
# data3 = [1, 2, –4, –3, 5]
data4 = [v for x in values if (v:=toint(x)) is not None and v >= 0]
# data4 = [1, 2, 5]
```

2.15. Wyrażenia generujące

Wyrażenie generujące to obiekt, który wykonuje te same obliczenia co lista składana, ale iteracyjnie generuje wynik. Składnia jest taka sama jak w przypadku list składanych, z tym wyjątkiem, że zamiast nawiasów kwadratowych używa się nawiasów okrągłych. Oto przykład:

```
nums = [1,2,3,4]
squares = (x*x for x in nums)
```

W przeciwieństwie do list składanych wyrażenie generujące w rzeczywistości nie tworzy listy ani natychmiast nie liczy wyrażenia w nawiasach. Zamiast tego tworzy obiekt generatora, który generuje wartości na żądanie za pomocą iteracji. Jeśli spojrzysz na wynik powyższego przykładu, zobaczysz, co następuje:

```
>>> squares
<generator object at 0x590a8>
>>> next(squares)
1
>>> next(squares)
4
...
>>> for n in squares:
... print(n)
9
16
>>>
```

Wyrażenia generującego można użyć tylko raz. Jeśli spróbujesz powtórzyć iterację po raz drugi, nic nie otrzymasz:

```
>>> for n in squares:
... print(n)
...
>>>
```

Różnica między listą składaną a wyrażeniem generującym jest ważna, ale subtelna. Dzięki listom składanym Python faktycznie tworzy listę, która zawiera dane wynikowe. Za pomocą wyrażenia generującego Python tworzy generator, który jedynie wie, jak generować dane na żądanie. W niektórych aplikacjach może to znacznie poprawić wydajność i wykorzystanie pamięci. Oto przykład:

```
# Przeczytaj plik
f = open('data.txt')                        # Otwórz plik
lines = (t.strip() for t in f)              # Czytaj linie, usuń końcowe/początkowe białe znaki
comments = (t for t in lines if t[0] == '#') # Wszystkie komentarze
for c in comments:
    print(c)
```

W tym przykładzie wyrażenie generujące, które wyodrębnia wiersze z pliku i usuwa białe znaki, w rzeczywistości nie odczytuje i nie przechowuje całego pliku w pamięci. To samo dotyczy wyrażenia, które wyodrębnia komentarze. Zamiast tego wiersze pliku są odczytywane jeden po drugim, gdy program rozpoczyna iterację w pętli `for`. Podczas tej iteracji wiersze pliku są tworzone na żądanie i odpowiednio filtrowane. W rzeczywistości cały plik nie zostanie załadowany do pamięci podczas tego procesu. Jest to zatem bardzo wydajny sposób na odczytanie komentarzy z olbrzymiego pliku źródłowego Pythona.

W przeciwieństwie do list składanych wyrażenie generujące nie tworzy obiektu, który działa jak sekwencja. Nie może być indeksowany i żadna ze zwykłych operacji na listach (takich jak `append()`) nie zadziała. Jednak elementy tworzone przez wyrażenie generujące można przekonwertować na listę za pomocą `list()`:

```
clist = list(comments)
```

W przypadku przekazania pojedynczego argumentu funkcji jeden zestaw nawiasów może zostać usunięty. Na przykład następujące wyrażenia są równoważne:

```
sum((x*x for x in values))
sum(x*x for x in values)      # Usunięto dodatkowe nawiasy
```

W obu przypadkach tworzony jest generator (`x*x for x in values`), który jest przekazywany do funkcji `sum()`.

2.16. Operator atrybutu (.)

Operator kropki (`.`) służy do uzyskiwania dostępu do atrybutów obiektu. Oto przykład:

```
foo.x = 3
print(foo.y)
a = foo.bar(3,4,5)
```

W jednym wyrażeniu może się pojawić więcej niż jeden operator kropki, na przykład `foo.y.a.b`. Operator kropki można również zastosować do pośrednich wyników funkcji, na przykład `a = foo.bar(3,4,5).spam`. Takie długie łańcuchy pobrań atrybutów nie są jednak popularne.

2.17. Operator wywołania funkcji ()

Operator `f(args)` służy do wywołania funkcji `f`. Każdy argument funkcji jest wyrażeniem. Przed wywołaniem funkcji wszystkie wyrażenia argumentów są liczone od lewej do prawej. Określa się to jako aplikacyjny porządek liczenia. Więcej informacji o funkcjach można znaleźć w rozdziale 5.

2.18. Kolejność liczenia

Tabela 2.11 przedstawia kolejność działania (reguły pierwszeństwa) dla operatorów Pythona. Wszystkie operatory z wyjątkiem operatora potęgi (**) są liczone od lewej do prawej i są wymienione w tabeli od najwyższego do najniższego priorytetu. Oznacza to, że operatory wymienione jako pierwsze w tabeli są liczone przed operatorami wymienionymi później. Operatory zawarte razem w podsekcjach, takie jak `x * y`, `x / y`, `x // y`, `x @ y` i `x % y`, mają równy priorytet.

Kolejność liczenia z tabeli 2.11 nie zależy od typów `x` i `y`. Tak więc mimo że obiekty zdefiniowane przez użytkownika mogą przedefiniować poszczególne operatory, dostosowanie podstawowej kolejności liczenia, pierwszeństwa i reguł asocjacji nie jest możliwe.

Tabela 2.11. *Kolejność liczenia (od najwyższego priorytetu do najniższego)*

Operator	Nazwa
`(...), [...], {...}`	Tworzenie krotek, list i słowników
`s[i], s[i:j]`	Indeksowanie i wycinanie
`s.attr`	Wyszukiwanie atrybutów
`f(...)`	Wywołania funkcji
`+x, -x, ~x`	Operatory jednoargumentowe
`x ** y`	Potęga (liczenie od prawej strony)
`x * y, x / y, x // y, x % y, x @ y`	Mnożenie, dzielenie, dzielenie całkowite, modulo, mnożenie macierzy
`x + y, x - y`	Dodawanie, odejmowanie
`x << y, x >> y`	Przesunięcie bitowe
`x & y`	Bitowe i
`x ^ y`	Bitowa różnica symetryczna
`x | y`	Bitowe lub
`x < y, x <= y, x > y, x >= y, x == y, x !=y, x is y, x is not y, x in y, x not in y`	Porównanie, tożsamość, testy członkostwa
`not x`	Logiczna negacja
`x and y`	Logiczne i
`x or y`	Logiczne lub
`lambda args: expr`	Funkcja anonimowa
`expr if expr else expr`	Wyrażenie warunkowe
`x := expr`	Wyrażenie przypisania

Częstą pomyłką dotyczącą reguł pierwszeństwa jest sytuacja, gdy operatory bitowe i (`&`) i bitowe lub (`|`) są używane jak operatory logiczne `and` i `or`. Na przykład:

```
>>> a = 10
>>> a <= 10 and 1 < a
True
>>> a <= 10 & 1 < a
False
>>>
```

To ostatnie wyrażenie jest liczone jako `a <= (10 & 1) < a` lub `a <= 0 < a`. Możesz to naprawić, dodając nawiasy:

```
>>> (a <= 10) & (1 < a)
True
>>>
```

Takie operacje mogą się wydawać dziwnym przypadkiem brzegowym, ale pojawiają się z pewną częstotliwością w pakietach zorientowanych na dane, takich jak `numpy` i `pandas`. Operatory logiczne `and` lub `or` nie mogą być przerabiane do własnych potrzeb, więc zamiast nich używane są operatory bitowe — nawet jeśli mają wyższy poziom pierwszeństwa i dają inny wynik, gdy są wykorzystywane w relacjach logicznych.

2.19. Podsumowanie: sekretne życie danych

Jednym z najczęstszych zastosowań Pythona jest używanie go w aplikacjach związanych z manipulacją i analizą danych. W tych dziedzinach Python zapewnia rodzaj „języka dedykowanego" do rozwiązywania Twoich problemów. Wbudowane operatory i wyrażenia są rdzeniem tego języka i cała reszta jest „budowana" wokół nich. Tak więc kiedy kiedy już zdobędziesz orientację we wbudowanych obiektach i operacjach Pythona, przekonasz się, że możesz je stosować wszędzie.

Załóżmy na przykład, że pracujesz z bazą danych i chcesz iterować po rekordach zwróconych przez zapytanie. Są szanse, że użyjesz do tego instrukcji `for`. Lub załóżmy, że pracujesz z tablicami liczbowymi i chcesz wykonać operacje na tablicach, element po elemencie. Możesz pomyśleć, że standardowe operatory matematyczne zadziałają — Twoje założenia byłyby właściwe. Albo załóżmy, że używasz biblioteki do pobierania danych przez HTTP i chcesz uzyskać dostęp do zawartości nagłówków HTTP. Istnieje duża szansa, że dane będą prezentowane w sposób przypominający słownik.

Więcej informacji o wewnętrznych protokołach Pythona i sposobach ich dostosowywania znajduje się w rozdziale 4.

Python? Zacznij od tego, co w nim najważniejsze!

Przez ostatnie trzydzieści lat Python intensywnie się rozwijał. Stał się dojrzałym językiem programowania, nadającym się do tworzenia zarówno niewielkich, wyspecjalizowanych narzędzi, jak i złożonych systemów. Zakres jego możliwości poszerzają liczne biblioteki i narzędzia. Ta wszechstronność i bogactwo stanowią jednak duże wyzwanie dla programistów, którzy chcieliby nie tyle poznać poszczególne funkcje tego języka, ile dogłębnie go zrozumieć i nauczyć się „pythonicznego" sposobu myślenia.

Ta książka jest zwięzłym, skupionym na praktyce przewodnikiem po Pythonie w wersji 3.6 i nowszych. Dzięki niej skoncentrujesz się na rdzeniu języka i podstawowych zagadnieniach, które musisz doskonale opanować, jeśli chcesz pisać w nim dobry kod. Dowiesz się zatem, jak działa Python i jakich zasad należy przestrzegać podczas tworzenia programów, które później będą łatwe do przetestowania, debugowania i utrzymania. Dobrze zrozumiesz kluczowe kwestie, takie jak abstrakcja danych, kontrola przepływu programu, struktura programu, funkcje, obiekty i moduły. Poszczególne treści zostały zilustrowane przejrzystymi fragmentami kodu, pozwalającymi nie tylko łatwiej przyswoić opisane zagadnienia, ale i poczuć niezwykły urok Pythona — tę magię, która sprawia, że programowanie w tym języku daje mnóstwo przyjemności i satysfakcji!

- **czym jest rdzeń Pythona**
- **praca z danymi i ich analiza**
- **zasady tworzenia przejrzystego i niezawodnego kodu**
- **funkcje i idiomy w programowaniu funkcjonalnym**
- **generatory, klasy, moduły, pakiety**
- **prawidłowa obsługa I/O i korzyści ze stosowania słowników**

David Beazley jest amerykańskim inżynierem oprogramowania. Pythonem zajmuje się od 1996 roku i w znaczący sposób przyczynił się do jego rozwoju. Autor ważnych książek o Pythonie, napisał też kilka narzędzi programistycznych. Obecnie prowadzi zaawansowane kursy informatyki.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-283-9019-5
9 788328 390195
Cena: 69,00 zł
Pearson
Addison-Wesley