helion.pl

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

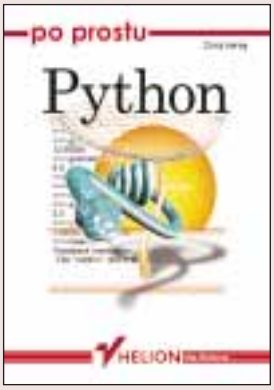
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Po prostu Python



Autor: Chris Fehily
Tłumaczenie: Zygmunt Wereszczyński
ISBN: 83-7197-684-4
Tytuł oryginału: Python Visual QuickStart Guide
Format: B5, stron: 436

Witamy w świecie Pythona, czyli popularnego języka programowania obiektowego, którego wersja źródłowa jest powszechnie dostępna. Python może być stosowany do wszystkiego – począwszy od prostych skryptów, aż do programowania witryn WWW i skomplikowanych aplikacji. W celu utrzymania rozsądnej objętości książki, autor ograniczył się do opisania rdzenia samego języka i podstawowych bibliotek. Po ukończeniu lektury czytelnik będzie znał sposoby wykorzystania Pythona zarówno w przypadku typowych, jak i w zaawansowanych zadań stawianych programiście. Na końcu książki przedstawiono sposoby, w jakie można uzyskać dostęp do źródeł informacji dotyczących różnych zagadnień. Czy to wystarczy?

Książka jest odpowiednia dla osób, które uczą się języka Python z czyjąś pomocą lub trochę programowały w innym języku programowania (przynajmniej na tyle, że wiedzą dobrze, co to są zmienne, pętle i instrukcje warunkowe). Książka nie nadaje się do wykorzystania jako podręcznik do samodzielnej nauki pierwszego języka programowania lub do nauki technik programowania stosowanych w administrowaniu systemem, programowaniu aplikacji WWW lub innych specjalistycznych dziedzinach (należy o tym pamiętać przesyłając recenzję do Amazon.com).

Książka jest wyczerpującym przewodnikiem po języku Python. Jej zakres tematyczny został ograniczony do samego rdzenia języka i najczęściej stosowanych modułów bibliotecznych. Informacje o innych modułach można znaleźć w podręczniku „Python Library Reference”, który zwykle jest dołączony do Pythona. Krótki przegląd częściej stosowanych modułów można także znaleźć w dodatku A.

Wydawnictwo Helion
ul. Chopina 6
44-100 Gliwice
tel. (32)230-98-63
e-mail: helion@helion.pl

Spis treści

Wprowadzenie **11**
Informacja o Pythonie ... 12
Informacja o książce ... 16
Krótki przegląd ... 18
Typy wbudowane ... 22

Rozdział 1. **Pierwsze kroki** **27**
Dostęp do dystrybucji Pythona ... 28
Dokumentacja języka Python ... 31
Otwieranie okna z wierszem poleceń ... 33
Definiowanie ścieżki ... 34
Zastosowanie zmiennych środowiskowych Pythona ... 37
Uruchamianie programów w trybie interakcyjnym ... 38
Uruchamianie programów jako skryptów ... 40
Zastosowanie IDLE ... 42
Określanie opcji wiersza poleceń ... 44
Przekazywanie argumentów do skryptu ... 45

Rozdział 2. **Wyrażenia i instrukcje** **47**
Dokumentowanie programów ... 48
Nadawanie nazw zmiennym ... 50
Tworzenie wyrażeń ... 51
Tworzenie instrukcji z wyrażeń ... 52
Tworzenie zmiennych ... 54
Usuwanie zmiennych ... 57
Drukowanie obiektów ... 58
Określanie tożsamości obiektu ... 60
Tworzenie odwołań do tego samego obiektu ... 61
Określanie typu obiektu ... 63
Użycie operatorów logicznych ... 67
Użycie operatorów porównania ... 69
Porównania kaskadowe ... 72
Kolejność obliczeń ... 73
Podsumowanie informacji o obiektach ... 74

Rozdział 3. **Praca z liczbami** **75**
Informacje o typach liczb....76
Zasady rozszerzania typów....78
Podstawowe działania arytmetyczne....79
Potęgowanie....81
Reszta pozostała z dzielenia....82
Uzyskiwanie ilorazu przy dzieleniu....83
Kontrola kolejności obliczeń....84
Przypisania przyrostowe....85
Przekształcanie typów liczb....86
Porównywanie liczb....87
Korzystanie z funkcji matematycznych....88
Zastosowanie zaawansowanych funkcji matematycznych....90
Generowanie liczb losowych....92

Rozdział 4. **Praca z łańcuchami** **95**
Tworzenie łańcucha....96
Wstawianie znaków specjalnych do łańcucha....98
Łańcuchy Unicode....100
Długość łańcucha....106
Indeksowanie łańcucha (pobieranie znaku)....107
Wycinanie łańcucha (pobieranie części łańcucha)....108
Łączenie łańcuchów....110
Powtarzanie łańcucha....112
Zastosowanie metod i funkcji łańcuchowych....114
Zmiana wielkości liter w łańcuchu....115
Testowanie łańcuchów....117
Przycinanie i wyrównywanie łańcucha....120
Wyszukiwanie łańcuchów podrzędnych....122
Zamiana łańcuchów podrzędnych....126
Konwersja łańcucha....128
Rozdzielanie i łączenie łańcuchów....129
Operacje listowe stosowane w łańcuchach....131
Przekształcenia łańcuchów....132
Porównywanie łańcuchów....135
Wyświetlanie sformatowanych łańcuchów....139

Rozdział 5. **Praca z listami i krotkami** **143**
Tworzenie listy lub krotki....144
Określanie rozmiaru (długości) listy lub krotki....146
Indeksowanie listy lub krotki (pobieranie pozycji)....147
Wycinanie listy lub krotki (pobieranie segmentu)....149
Kopiowanie listy lub krotki....151

Łączenie list lub krotek 154
Powtarzanie listy lub krotki 155
Przekształcanie listy lub krotki 157
Porównywanie list lub krotek 159
Sprawdzanie przynależności do listy lub krotki 161
Modyfikacja listy 162
Zastępowanie pozycji listy 163
Zliczanie pasujących wartości w liście 165
Przeszukiwanie listy 166
Dołączanie pozycji do listy 167
Usuwanie pozycji z listy 169
Sortowanie listy 172
Wstawianie pozycji do posortowanej listy 173
Samodzielne określanie kolejności sortowania listy 175
Odwracanie kolejności listy 177

Rozdział 6. **Praca ze słownikami** **179**
Użycie operatorów i metod słownikowych 180
Tworzenie słownika 181
Wyświetlanie (drukowanie) słownika 183
Uzyskiwanie wartości za pomocą klucza 185
Uzyskiwanie wszystkich wartości ze słownika 187
Uzyskiwanie wszystkich kluczy ze słownika 188
Uzyskiwanie wszystkich par klucz-wartość ze słownika 189
Sprawdzanie tego, czy klucz istnieje 190
Zliczanie par klucz-wartość w słowniku 191
Wstawianie lub zastępowanie pary klucz-wartość 192
Usuwanie pary klucz-wartość 193
Usuwanie przypadkowej pary klucz-wartość 194
Oczyszczanie lub usuwanie słownika 195
Łączenie słowników 196
Kopiowanie słownika 197
Przekształcanie słownika 198
Porównywanie słowników 199
Sortowanie słownika 201
Przechowywanie wyliczonych wartości w słowniku 202

Rozdział 7. **Instrukcje sterujące** **205**
Podział instrukcji na wiersze 206
Tworzenie instrukcji złożonych 208
Zastosowanie pass do tworzenia pustej instrukcji 209
Wcinanie bloków instrukcji 210
Wiele instrukcji w jednym wierszu 212

Zastosowanie warunków if 213
Zastosowanie warunków if-else 214
Zastosowanie warunków if-elif-else 215
Zastosowanie pętli while 217
Zastosowanie pętli while-else 219
Zastosowania pętli for 220
Zastosowania pętli for-else 224
Pętla w zakresie liczb całkowitych 225
Pomijanie części iteracji pętli 228
Wychodzenie z pętli 230

Rozdział 8. **Funkcje** **233**
Definiowanie funkcji 234
Dokumentowanie funkcji 236
Wywoływanie funkcji 238
Zwracanie wartości przez funkcję 240
Zwracanie wielu wartości funkcji 244
Określanie argumentów pozycyjnych 245
Określanie domyślnej wartości parametru 246
Określanie argumentów jako słów kluczowych 248
Określanie dowolnej liczby argumentów pozycyjnych 250
Określanie dowolnej liczby argumentów jako słów kluczowych 252
Mieszane techniki przekazywania argumentów 253
Tworzenie funkcji rekurencyjnych 254
Przekazywanie zmiennych i niezmiennych argumentów do funkcji 256
Deklaracje zmiennych globalnych 258
Przypisanie funkcji do zmiennej 260
Zastosowania narzędzi programowania funkcjonalnego 262
Zastosowanie wyrażeń lambda do tworzenia funkcji 264
Zastosowanie apply w celu wywołania funkcji 266
Użycie map w celu zastosowania funkcji do pozycji sekwencji 268
Zastosowanie zip do grupowania pozycji sekwencji 270
Zastosowanie filter do warunkowego usuwania pozycji sekwencji 271
Zastosowanie reduce do redukcji sekwencji 272
Zastosowanie skrótów listy do tworzenia list 273

Rozdział 9. **Moduły** **277**
Struktura modułu 278
Tworzenie modułu 279
Dokumentowanie modułu 281
Ładowanie modułu za pomocą import 282
Dostęp do atrybutów 284
Tworzenie listy atrybutów obiektu 286

Działania na atrybutach ... 287
Ładowanie specyficznych nazw z modułu za pomocą instrukcji from ... 288
Ładowanie modułu pod inną nazwą za pomocą instrukcji as ... 291
Udostępnianie właściwości języka ... 292
Ładowanie modułu za pomocą funkcji reload ... 293
Określanie ścieżki wyszukiwania modułów ... 295
Określanie tego, czy moduł działa jako program ... 296
Określanie tego, które moduły są załadowane ... 298
Przestrzenie nazw ... 299
Dostęp do przestrzeni nazw ... 301
Reguły zasięgu ... 304
Zagnieżdżanie funkcji ... 305
Grupowanie modułów w pakiety ... 307
Jawne zamykanie programu ... 308
Programowe uruchamianie kodu ... 309

Rozdział 10. **Pliki** **311**

Otwieranie pliku ... 312
Odczyt z pliku ... 314
Zapisywanie do pliku ... 318
Zamykanie pliku ... 320
Zmiana pozycji w pliku ... 321
Obcinanie pliku ... 323
Uzyskiwanie informacji o obiekcie pliku ... 324
„Drukowanie” do pliku ... 325
Dostęp do standardowych plików wejściowych i wyjściowych ... 326
Znak zachęty dla użytkownika ... 330
Korzystanie z systemu plików ... 331
Dostęp do zmiennych środowiskowych ... 333
Zmiana katalogu roboczego ... 334
Wypisywanie zawartości katalogu ... 335
Tworzenie katalogu ... 336
Usuwanie katalogu ... 337
Zmiana nazwy pliku lub katalogu ... 338
Usuwanie pliku ... 339
Uzyskiwanie informacji o ścieżce ... 340
Uzyskiwanie informacji o pliku ... 342
Dzielenie ścieżek ... 344
Łączenie ścieżek ... 346
Zachowywanie obiektów w postaci plików ... 347

Rozdział 11. **Wyjątki** **351**
Hierarchia wyjątków 353
Obsługa wyjątku....357
Ignorowanie wyjątku....362
Pobieranie argumentu wyjątku....363
Obsługa wszystkich wyjątków....365
Uruchamianie kodu przy braku wyjątku....366
Obsługa wielu wyjątków....367
Uruchamianie obowiązkowego kodu oczyszczającego....370
Jawne wywoływanie wyjątku....371
Samodzielne tworzenie wyjątków....372
Tworzenie asercji....375

Rozdział 12. **Klasy** **377**
Terminologia programowania obiektowego....378
Definiowanie klasy....379
Dokumentowanie klasy....380
Dostęp do wbudowanych atrybutów klasy....381
Tworzenie egzemplarza....382
Dostęp do wbudowanych atrybutów egzemplarza....383
Tworzenie zmiennych klasy....384
Zastosowanie metod specjalnych do przeciążania standardowego zachowania .386
Tworzenie zmiennych egzemplarza za pomocą metody inicjującej....388
Usuwanie egzemplarza....391
Tworzenie łańcuchowej reprezentacji egzemplarza....393
Określanie prawdziwości egzemplarza....395
Porównywanie egzemplarzy....396
Dostęp do atrybutów egzemplarza....398
Egzemplarz jako lista lub słownik....400
Operacje matematyczne na egzemplarzach....402
Wywoływanie egzemplarza....405
Samodzielne definiowanie i wywoływanie metod....406
Wyprowadzanie nowych klas z klas istniejących....408
Ukrywanie danych prywatnych....411
Określanie członkostwa klasy....412

Dodatek A **Zasoby poświęcone językowi Python** **413**
Nauka języka Python....414
Uzyskiwanie pomocy....415
Edycja i usuwanie błędów z kodu....417
Zasoby dla programistów....418

Skorowidz **421**

8 Funkcje

Python pozwala na samodzielne tworzenie funkcji przez użytkownika. Kilka funkcji wbudowanych omówiono już w poprzednich rozdziałach (na przykład `id()`, `str()`, `len()` i `range()`). Wspomniano także o niektórych funkcjach znanych ze standardowych modułów (na przykład `random.random()` i `copy.deepcopy()`). Dzięki zastosowaniu funkcji struktura programu może być modułowa i łatwo można nią zarządzać, bowiem opakowują one obliczenia i oddzielają logikę wysokiego poziomu od szczegółów. Funkcji można używać wielokrotnie, ponieważ pozwalają one na wielokrotne wykonywanie dobrze określonych zadań w wielu miejscach z różnymi wartościami początkowymi.

W tym rozdziale omówiono sposoby samodzielnego tworzenia funkcji. Opisano także narzędzia programowania funkcjonalnego w Pythonie, które umożliwiają zastosowanie funkcji dla pozycji sekwencji.

Definiowanie funkcji

Funkcja jest nazwanym blokiem instrukcji, które wykonują jakąś operację. Instrukcja `def` jest instrukcją złożoną (patrz: podrozdział „Tworzenie instrukcji złożonych" w rozdziale 7.), która tworzy nową funkcję (definiuje jej nazwę, parametry i blok instrukcji wykonywanych po wywołaniu funkcji).

Jak zdefiniować funkcję?

1. Wpisz:

   ```
   def func(param_list):
     blok
   ```

   Po uruchomieniu ta złożona instrukcja tworzy nowy obiekt funkcji i przypisuje go do nazwy *func* (rysunek 8.1). Nazwa *func* jest poprawną nazwą Pythona (patrz: podrozdział „Nadawanie nazw zmiennym" w rozdziale 2.), *param_list* reprezentuje zero lub więcej parametrów oddzielonych przecinkami (są one opisane w dalszej części tego rozdziału), zaś *blok* jest wciętym blokiem instrukcji.

Wskazówki

- Zdefiniowanie funkcji nie oznacza jej uruchomienia, bowiem funkcja jest uruchamiana tylko wtedy, gdy zostanie wywołana (patrz: podrozdział „Wywoływanie funkcji" w dalszej części tego rozdziału).
- Podobnie jak w przypadku wszystkich innych obiektów w języku Python nie trzeba deklarować typów obiektów zwracanej wartości, parametrów lub zmiennych funkcji. Taka elastyczność oznacza, że można zdefiniować jedną funkcję i wykorzystywać ją z obiektami różnego typu (na przykład funkcja `len()`działa na łańcuchach, listach, krotkach i słownikach).
- Instrukcja `def` jest instrukcją złożoną, zatem krótką definicję funkcji można zapisać w jednym wierszu:

  ```
  def square(x): return x * x.
  ```

- Funkcje mogą być definiowane wewnątrz innych funkcji (patrz: podrozdział „Zagnieżdżanie funkcji" w rozdziale 9.).

```
>>> def do_nothing():
...   pass
...
>>> def print_msg():
...   print "Mimetyczny wielostop"
...
>>> def iseven(num):
...   print num % 2 == 0
...
>>>
```

Rysunek 8.1. *Powyższe instrukcje definiują trzy proste funkcje. Trafnie nazwana instrukcja do_nothing jest najprostszą funkcją Pythona. Funkcja print_msg wyświetla łańcuch. Funkcja iseven ma listę parametrów; wyświetla ona 1, jeśli jej argument jest liczbą parzystą, albo 0 w przeciwnym wypadku*

- Funkcje są obiektami i mogą być przypisywane do zmiennych (patrz: podrozdział „Przypisywanie funkcji do zmiennej" w dalszej części tego rozdziału).
- Można utworzyć niewielkie anonimowe funkcje, które nie będą związane z żadnymi nazwami (patrz: podrozdział „Zastosowanie wyrażeń lambda do tworzenia funkcji" w dalszej części tego rozdziału).
- Do funkcji można przypisać dowolne atrybuty (patrz: podrozdział „Dostęp do atrybutów" w rozdziale 9.).
- **J** Funkcja języka Python jest równoważna podprogramowi stosowanemu w języku Perl lub funkcji stosowanej w języku C. Do tworzenia podprogramów w języku Perl wykorzystywana jest instrukcja `sub`. W odróżnieniu od definicji funkcji w językach Perl i C instrukcja `def` Pythona jest instrukcją wykonywalną (czyli tworzącą obiekt). W językach C i Java wymagane jest określenie typu danych wartości zwracanych przez funkcje i wartości każdego parametru (w języku Python nie jest to wymagane).

Parametry nie są argumentami

W literaturze informatycznej można czasami spotkać się z zamiennie stosowanymi określeniami parametr i argument, które w rzeczywistości znaczą co innego. Parametr jest nazwą, która pojawia się na liście parametrów w nagłówku definicji funkcji (czyli w instrukcji `def`). Uzyskuje on wartość z wywołania danej funkcji. Argument natomiast jest faktyczną wartością lub odwołaniem przekazanym do funkcji przez wywołanie.

Na przykład w poniższej definicji funkcji:

```
def sum(x, y):
  return x + y
```

`x` i `y` są parametrami, natomiast w wywołaniu funkcji:

```
z = sum(2, 3 * 5)
```

`2` i `3 * 5` są argumentami. Więcej informacji na temat wywoływania funkcji można znaleźć w podrozdziale „Wywoływanie funkcji" w dalszej części tego rozdziału.

Dokumentowanie funkcji

W przedstawionych dotąd przykładach oznaczania komentarzy opisujących kod stosowano znak # (patrz: podrozdział „Dokumentowanie programów" w rozdziale 2.). Python pozwala również na opcjonalne użycie literału łańcuchowego w pierwszym wierszu bloku funkcji. Łańcuch ten (zwany łańcuchem dokumentacyjnym) powinien zawierać informację o sposobie wywołania funkcji i krótko opisywać jej działanie.

Łańcuch dokumentacyjny pełni specjalną rolę, bowiem jest przypisywany do atrybutu `__doc__` funkcji i można uzyskać do niego dostęp za pomocą operatora kropki w fazie działania programu. Nie jest to możliwe w przypadku komentarzy poprzedzonych znakiem #.

Dostęp do łańcucha dokumentacyjnego funkcji

1. Wpisz *func*`.__doc__`.

   Nazwa *func* jest nazwą funkcji, która może być kwalifikowana za pomocą nazwy modułu. Po nazwie *func* nie należy używać nawiasów (rysunek 8.2).

```
>>> def do_nothing():
...    "Nic nie robi."
...    pass
...
>>> def print_msg():
...    """Drukuje łańcuch."""
...    print "Mimetyczny wielostop"
...
>>> def iseven(num):
...    """iseven(integer) -> Boolean
...
... Drukuje 1, gdy num jest parzyste,
↳albo 0 w przeciwnym wypadku."""
...    print num % 2 == 0
...
>>> do_nothing.__doc__
'Nic nie robi.'
>>> s = print_msg.__doc__
>>> print s
Drukuje łańcuch.
>>> print iseven.__doc__
iseven(integer) -> Boolean

Drukuje 1, gdy num jest parzyste, albo 0
↳w przeciwnym wypadku.
>>>
```

Rysunek 8.2. *Tutaj dodano łańcuchy dokumentacyjne do funkcji zdefiniowanych na rysunku 8.1. Łańcuchy dokumentacyjne w potrójnych cudzysłowach mogą zajmować wiele wierszy i zachowywać ich formatowanie przy wyświetlaniu*

```
>>> print str.__doc__
str(object) -> string

Return a nice string representation of the
↳object. If the argument is a string, the
↳return value is the same object.
>>> print "".join.__doc__
S.join(sequence) -> string

Return a string which is the concatenation
↳of the strings in the sequence. The
↳separator between elements is S.
>>> [].append.__doc__
'L.append(object) - append object to end'
>>> import math
>>> print math.sqrt.__doc__
sqrt(x)

Return the square root of x.
```

Rysunek 8.3. *Wyświetlenie łańcucha dokumentacyjnego wbudowanej funkcji pozwala szybko uzyskać pomoc*

Wskazówki

- Łańcuch dokumentacyjny musi być umieszczony w pierwszym wierszu za instrukcją `def`.
- Jeśli łańcuch dokumentacyjny nie zostanie podany, to domyślną wartością atrybutu `__doc__` staje się `None`.
- W przedstawionych w książce przykładach często pomijane są łańcuchy dokumentacyjne ze względu na oszczędność miejsca, lecz do dobrej praktyki należy używanie ich we wszystkich funkcjach. Za takie podejście będą wdzięczni użytkownicy. Warto również pamiętać o tym, że niektóre programy narzędziowe korzystają z łańcuchów dokumentacyjnych przy tworzeniu dokumentacji pomocniczej.
- Łańcuchy dokumentacyjne są także dostępne w funkcjach i metodach wbudowanych (patrz: rysunek 8.3).
- Łańcuch dokumentacyjny można również dodać do modułu (patrz: podrozdział „Dokumentowanie modułu” w rozdziale 9.) oraz do klasy (patrz: podrozdział „Dokumentowanie klasy” w rozdziale 12.).

Wywoływanie funkcji

Funkcje nie działają zaraz po ich zdefiniowaniu. Aby tak się stało, należy je jawnie wywołać. *Wywołanie funkcji* jest wyrażeniem, które uruchamia funkcję. Część programu, z którego funkcja jest wywoływana, bywa nazywana *wywołującym*. W tej książce przedstawiano już wywołania wbudowanych funkcji Pythona (na przykład `len()`). Funkcje zdefiniowane przez użytkownika są wywoływane w taki sam sposób: poprzez użycie nazwy funkcji, za którą w nawiasach podana jest lista jej argumentów.

Jak wywołać funkcję?

1. Wpisz:

   func`(`*arg_list*`)`.

   Podczas wykonywania program wywołujący zatrzymuje się i trwa w tym stanie do momentu, gdy *func* zakończy działanie i sterowanie powróci znów do wywołującego (rysunek 8.4).

   Nazwa *func* jest nazwą funkcji, zaś *arg_list* oznacza zero lub więcej argumentów oddzielonych przecinkami, które są przekazywane do funkcji (zostanie to omówione później).

```
>>> do_nothing()
>>> print_msg()
Mimetyczny wielostop
>>> num = 9
>>> iseven(num)
0
```

Rysunek 8.4. *Wywołania trzech funkcji zdefiniowanych na rysunku 8.1*

```
>>> def f1():
...   print "f1"
...   f2()
...
>>> def f2():
...   print "f2"
...
>>> f1()
f1
f2
```

Rysunek 8.5. *Dozwolone jest odwołanie do funkcji przed ich zdefiniowaniem, lecz nie ich wywoływanie. Funkcja f2 może pojawić się w definicji f1, zanim sama f2 zostanie zdefiniowana, ale dopiero po zdefiniowaniu f2 można wywołać f1*

Wskazówki

- Nawet wtedy, gdy do funkcji nie są przekazywane żadne argumenty, w wywołaniu należy wpisać puste nawiasy.
- Funkcja musi być zdefiniowana przed jej wywołaniem. Można jednak odwołać się do funkcji przed jej zdefiniowaniem (rysunek 8.5).
- Opis wywołania funkcji zwracającej jakąś wartość można znaleźć w podrozdziale „Zwracanie wartości przez funkcję" w dalszej części tego rozdziału.
- To, czy dany obiekt jest wywoływalny, można stwierdzić za pomocą wbudowanej funkcji `callable(`*`obiekt`*`)`. Funkcja ta zwraca 1 (prawda), jeśli argument *`obiekt`* jest wywoływalny, a w przeciwnym wypadku zwraca 0 (fałsz). Do obiektów wywoływalnych zaliczane są funkcje zdefiniowane przez użytkownika, funkcje wbudowane, metody obiektów wbudowanych, obiekty klas oraz metody egzemplarzy klas.

Zwracanie wartości przez funkcję

Wartość zwracana jest tym wynikiem działania funkcji, który jest przesyłany do programu wywołującego daną funkcję. Wartość tę można określić za pomocą instrukcji `return`.

Jak uzyskać wartość funkcji?

1. W ciele funkcji wpisz:

   `return` *expr*.

   Ta instrukcja kończy bieżące wywołanie funkcji. Oblicza ona wyrażenie *expr* i wysyła jego wartość (jako wartość zwracaną) do programu wywołującego.

   Wyrażenie *expr* jest wyrażeniem. Jeśli *expr* zostanie pominięte, to podstawione tu będzie `None`.

```
>>> def iseven(num):
...   return num % 2 == 0
...
>>> def isprime(n):
...   count = n / 2
...   while count > 1:
...     if n % count == 0:
...       return 0
...     count -= 1
...   return 1
...
>>> def primes(m):
...   lista = []
...   for i in range(1, m):
...     if isprime(i):
...       lista.append(i)
...   return lista
...
>>> def vowels(s):
...   v = ""
...   for c in s:
...     if c in "aeiouAEIOU":
...       v += c
...   return v
...
>>> def ascii(s):
...   asc = {}
...   for c in s:
...     if not asc.has_key(c):
...       asc[c] = ord(c)
...   return asc
...
```

Rysunek 8.6. *— ciąg dalszy na następnej stronie*

```
>>> a = iseven(40)
>>> b = isprime(27)
>>> c = primes(20)
>>> d = vowels("sekwoja")
>>> e = ascii("ick")
>>> print a
1
>>> print b
0
>>> print c
[1, 2, 3, 5, 7, 11, 13, 17, 19]
>>> print d
eoa
>>> print e
{'c': 99, 'k': 107, 'i': 105}
```

Rysunek 8.6. *Pokazane tu funkcje zwracają wartości różnych typów. Funkcja iseven określa to, czy jej argument jest liczbą parzystą i zwraca 1 (prawda) albo 0 (fałsz). Funkcja isprime określa to, czy jej argument jest liczbą pierwszą i zwraca 1 albo 0. Zwróćmy uwagę na to, że isprime ma dwie instrukcje return (sterowanie powraca do programu wywołującego wtedy, gdy którakolwiek z tych instrukcji zostanie wykonana). Funkcja primes zwraca listę zawierającą liczby pierwsze, które są mniejsze niż jej argument. Funkcja vowels zwraca łańcuch zawierający samogłoski pochodzące z jej argumentu. Funkcja ascii zwraca słownik zawierający kody ASCII (wartości) znaków (kluczy) jej argumentu*

Wywoływanie funkcji, która zwraca wartość

1. Wpisz

zmienna = *func*(*arg_list*).

Podczas wykonywania tego wyrażenia, program wywołujący jest zatrzymywany do momentu, gdy *func* zakończy działanie. Wówczas sterowanie powraca do wywołującego. Zmienna *zmienna* uzyskuje wartość zwróconą przez funkcję (rysunek 8.6).

Zmienna *zmienna* jest zmienną, *func* jest nazwą funkcji, zaś *arg_list* reprezentuje argumenty przekazywane do funkcji.

Na rysunkach przedstawionych w rozdziale 7. (od 7.22 do 7.26) pokazano kilka przykładów działania pętli `for`. Skrypty od 8.1 do 8.5 zawierają te same przykłady przerobione na funkcje, które można wielokrotnie wykorzystywać. Na rysunku 8.7 pokazano kilka przykładów wywołań tych funkcji.

Skrypt 8.1. *Funkcja unique pobiera sekwencję i zwraca listę, która zawiera wszystkie pozycje jej argumentu (bez duplikatów)*

Skrypt

```
def unique(seq):
  result = []
  for x in seq:
    if x not in result:
      result.append(x)
  return result
```

Skrypt 8.2. *Funkcja intersection pobiera dwie sekwencje i zwraca listę, która zawiera pozycje występujące jednocześnie w obydwu jej argumentach*

Skrypt

```
def intersection(seq1, seq2):
  result = []
  for x in seq1:
    if x in seq2:
      if x not in result:
        result.append(x)
  return result
```

Skrypt 8.3. *Funkcja union pobiera dwie sekwencje i zwraca listę, zawierającą wszystkie pozycje, które pojawiają się w obu jej argumentach (bez duplikatów)*

Skrypt

```
def union(seq1, seq2):
  result = seq1[:]  # Wykonanie kopii
  for x in seq2:
      if not x in seq1:
        result.append(x)
  return result
```

Skrypt 8.4. *Funkcja combine pobiera dwie sekwencje i zwraca listę, która zawiera każdą pozycję pierwszego argumentu połączoną w parę z każdą pozycją drugiego argumentu w postaci dwupozycyjnych krotek*

Skrypt

```
def combine(seq1, seq2):
  result = []
  for x in seq1:
    for y in seq2:
      result.append((x, y))
  return result
```

Skrypt 8.5. *Funkcja all_in pobiera dwie sekwencje i zwraca 1 (prawda), jeśli wszystkie pozycje pierwszego argumentu występują w drugim argumencie, albo 0 (fałsz) w przeciwnym wypadku*

Skrypt

```
def all_in(seq1, seq2):
  result = 1
  for x in seq1:
    if x not in seq2:
      result = 0
      break
  return result
```

```
>>> s = [1, 2, 2.0, "a", "A", "a", (3, 4),
↳(6/2, 8/2)]
>>> print unique(s)
[1, 2, 'a', 'A', (3, 4)]
>>>
>>> s1 = "mimetyczny wielostop"
>>> s2 = "sekwoja"
>>> print intersection(s1, s2)
['e', 'w', 'o', 's']
>>>
>>> s1 = [2, 3, 4]
>>> s2 = [1, 2, 3, 9]
>>> print union(s1, s2)
[2, 3, 4, 1, 9]
>>>
>>> s1 = (1, 5)
>>> s2 = (2, 6, 7)
>>> print combine(s1, s2)
[(1, 2), (1, 6), (1, 7), (5, 2), (5, 6),
↳(5, 7)]
>>>
>>> s1 = "aeo"
>>> s2 = "sekwoja"
>>> all_in(s1, s2)
1
```

Rysunek 8.7. *Przykładowe wywołania funkcji ze skryptów od 8.1 do 8.5*

```
>>> a = do_nothing()
>>> b = print_msg()
Mimetyczny wielostop
>>> c = iseven(2)
1
>>> print a, b, c
None None None
```

Rysunek 8.8. *Wszystkie funkcje zdefiniowane na rysunku 8.1 zwracają None, czyli domyślną wartość zwracaną przez funkcję nie posiadającą instrukcji return (lub przez funkcję, w której return istnieje, lecz nie jest uruchamiane)*

Wskazówki

- Jeśli sterowanie wyjdzie poza funkcję bez uruchamiania instrukcji `return`, to zwracana jest wartość `None` (rysunek 8.8).
- Instrukcja `return` może występować tylko wewnątrz definicji funkcji.
- Zwracanie wielu wartości przez funkcję opisano w następnym podrozdziale.
- **J** Instrukcja `return` jest używana do zwracania wartości funkcji także w językach Perl i C. Wartość `None` zwracana przez funkcję w Pythonie jest podobna do `void` w języku C. W języku Perl wartością domyślną jest wartość ostatniego wyrażenia obliczonego w bloku, zaś w języku Python wartością domyślną jest `None`.

Zwracanie wielu wartości funkcji

Jeśli liczba zwracanych wartości nie jest duża, to należy zwracać je w postaci krotki. Zastosowanie krotki umożliwia wówczas łatwe przypisanie każdej zwracanej wartości do jej własnej zmiennej za pomocą mechanizmu pakowania i rozpakowywania (patrz: podrozdział „Tworzenie zmiennych" w rozdziale 2.). Jeśli w instrukcji `return` poda się wiele wyrażeń, będą one zwrócone jako krotka.

Jak uzyskać wiele wartości z funkcji?

1. W ciele funkcji wpisz:

   `return` *`expr1, expr2,...`*

   Taka instrukcja kończy bieżące wywołanie funkcji. Wyrażenia *`expr1, expr2,...`* są obliczane i odsyłane do programu wywołującego jako krotka zawierająca zwracane wartości.

   Wyrażenia *`expr1, expr2,...`* są dwoma (lub więcej) wyrażeniami oddzielonymi za pomocą przecinków.

Wywoływanie funkcji, która zwraca wiele wartości

1. Wpisz:

   `zmienna1, zmienna2,... = func(arg_list)`.

   Po uruchomieniu takiego wyrażenia program wywołujący zatrzymuje się aż do momentu zakończenia działania *`func`*. Później sterowanie powraca do programu wywołującego. Pierwsza pozycja zwróconej krotki jest przypisywana do zmiennej *`zmienna1`*, druga do zmiennej *`zmienna2`* itd. (patrz: rysunek 8.9). Nazwa *`func`* jest nazwą funkcji, *`arg_list`* reprezentuje argumenty przekazywane do funkcji, zaś *`zmienna1, zmienna2,...`* oznaczają jedną lub więcej zmiennych oddzielonych przecinkami. Liczba zmiennych musi być taka jak liczba pozycji w zwracanej krotce.

```
>>> def fracint(x):
...     i = int(x)
...     f = x - i
...     return float(f), float(i)
...
>>> frac, int = fracint(-5.5)
>>> print frac
-0.5
>>> print int
-5.0
>>>
>>> def letters(s):
...     from string import letters, whitespace
...     v = c = w = o = ""
...     s = str(s)
...     for ch in s:
...       if ch in "aeiouyAEIOUY":
...         v += ch
...       elif ch in letters:
...         c += ch
...       elif ch in whitespace:
...         w += ch
...       else:
...         o += ch
...     return v, c, w, o
...
>>> s = "mimetyczny wielostop"
>>> v, c, w, o = letters(s)
>>> v, c, w, o
('ieyyieoo', 'mmtcznwlstp', ' ', '')
```

Rysunek 8.9. *Funkcja fracint zwraca część ułamkową i część całkowitą liczby w postaci dwupozycyjnej krotki (symulując działanie funkcji math.modf). Funkcja letters zwraca samogłoski, spółgłoski, znaki odstępu i inne znaki pochodzące z jej argumentu w postaci krotki łańcuchów*

Wskazówki

- Pozycje zwracanej krotki mogą być obiektami różnych typów.
- Jeśli liczba zmiennych i liczba pozycji w zwracanej krotce nie są sobie równe, Python wywołuje wyjątek `ValueError`.

```
>>> def get_color(red, green, blue):
...   kolory = {
...    (0,   0,   0)  : "black",
...    (0,   0,   255): "blue",
...    (0,   255, 0)  : "green",
...    (0,   255, 255): "cyan",
...    (255, 0,   0)  : "red",
...    (255, 0,   255): "magenta",
...    (255, 255, 0)  : "yellow",
...    (255, 255, 255): "white"
...    }
...   rgb = (red, green, blue)
...   if kolory.has_key(rgb):
...     return kolory[rgb]
...   else:
...     return rgb
...
>>> print get_color(255, 0, 255)
magenta
>>> print get_color(0, 0, 0)
black
>>> print get_color(0, 25, 18)
(0, 25, 18)
>>> print get_color(0, 255)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: get_color() takes exactly 3
↳arguments (2 given)
```

Rysunek 8.10. *Funkcja get_color pobiera trzy argumenty pozycyjne, które reprezentują składowe koloru (red, green i blue) i zwracają jego nazwę, jeśli jest ona znana (w przeciwnym wypadku funkcja zwraca krotkę zawierającą wartości argumentu)*

Określanie argumentów pozycyjnych

Argumenty pozycyjne muszą być przekazane do funkcji dokładnie w takiej kolejności, w jakiej podane są w definicji funkcji. Jest to normalny sposób przekazywania argumentów, który był prezentowany już wcześniej w przypadku funkcji wbudowanych Pythona. Przy braku domyślnych wartości parametrów (patrz: następny podrozdział), liczba przekazywanych argumentów musi być równa liczbie parametrów określonych w definicji funkcji. Na rysunku 8.10 pokazano kilka przykładów ilustrujących to zagadnienie.

Definiowanie parametrów pozycyjnych w nagłówku funkcji

1. Wpisz:

   `def` *`func`*`(`*`param1, param2,...`*`):`.

   Nazwa *`func`* jest nazwą funkcji, a *`param1, param2,...`* oznaczają nazwy parametrów pozycyjnych, które są oddzielone przecinkami.

Wywoływanie funkcji z argumentami pozycyjnymi

1. Wpisz:

   `func``(`*`arg1, arg2,...`*`)`.

   Nazwa *`func`* jest nazwą funkcji, a *`arg1, arg2,...`* są argumentami (wyrażeniami) oddzielonymi przecinkami, które odpowiadają parametrom pozycyjnym funkcji.

Wskazówka

- Jeśli liczba argumentów pozycyjnych i liczba parametrów pozycyjnych nie są sobie równe, Python wywołuje wyjątek `TypeError`.

Określanie domyślnej wartości parametru

Wartość domyślna jest przypisywana do parametru automatycznie, gdy w wywołaniu funkcji nie zostanie przekazana żadna wartość. Wartości domyślne wbudowanych funkcji Pythona pokazano już wcześniej (na przykład dla funkcji `round(x, [,n])` można podać tylko wartość *x*, bowiem wartością domyślną *n* jest 0).

Wartości domyślne są określane na liście parametrów w nagłówku funkcji (patrz: poprzedni podrozdział). Jeśli trzeba określić wartość domyślną parametru, to w nagłówku — zamiast wielkości *param* definiującej określony parametr — należy wpisać *param = wartość_domyślna* (*wartość_domyślna* jest tu wyrażeniem).

Parametry ze zdefiniowanymi wartościami domyślnymi muszą być podawane po parametrach bez tych wartości. Oto przykład błędnego nagłówka funkcji:

```
def f(x = 1, y):   # Niepoprawny nagłówek
```

W wywołaniu funkcji można podać opcjonalnie argument, który zastąpi wartość domyślną. Na rysunku 8.11 pokazano kilka przykładów ilustrujących to zagadnienie.

```
>>> def get_color(red = 0, green = 0,blue = 0):
...   kolory = {
...    (0,   0,   0)  : "black",
...    (0,   0,   255): "blue",
...    (0,   255, 0)  : "green",
...    (0,   255, 255): "cyan",
...    (255, 0,   0)  : "red",
...    (255, 0,   255): "magenta",
...    (255, 255, 0)  : "yellow",
...    (255, 255, 255): "white"
...    }
...   rgb = (red, green, blue)
...   if kolory.has_key(rgb):
...     return kolory[rgb]
...   else:
...     return rgb
...
>>> print get_color()
black
>>> print get_color(255)
red
>>> print get_color(0, 255)
green
>>> print get_color(0, 0, 255)
blue
>>> print get_color(25)
(25, 0, 0)
>>>
>>> def repeat(x, razy = 1):
...   return str(x) * razy
...
>>> print repeat("*")
*
>>> print repeat("-", 25)
-------------------------
>>> print repeat()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: repeat() takes at least 1 argument
↳(0 given)
```

Rysunek 8.11. *Tutaj zmieniono nagłówek funkcji get_color z rysunku 8.10. Wszystkie jej parametry mają teraz wartość domyślną równą zero. Pierwszy argument funkcji repeat jest wymagany, a drugi argument jest opcjonalny*

```
>>> def affix(x, lista = []):
...   lista.append(x)
...   return lista
...
>>> print affix(1)
[1]
>>> print affix(2)
[1, 2]
>>> print affix(3)
[1, 2, 3]
>>>
>>> def affix2(x, lista = None):
...   if lista is None: lista = []
...   lista.append(x)
...   return lista
...
>>> print affix2(1)
[1]
>>> print affix2(2)
[2]
>>> print affix2(3)
[3]
```

Rysunek 8.12. *Funkcja affix dodaje x do listy lista (modyfikując ją w miejscu). Python oblicza wartość domyślną argumentu lista tylko raz (gdy definiowana jest funkcja affix), a zatem lista zachowuje swoją wartość między kolejnymi wywołaniami affix. W funkcji affix2 przeniesiono wartość domyślną do ciała funkcji, zatem wartość lista jest obliczana przy każdym nowym wywołaniu (a nie współużytkowana przez kolejne wywołania)*

Wskazówki

- Nie ma sensu stwierdzenie, że parametr jest opcjonalny (przypisanie mu jakiejś wartości jest opcjonalne).
- Jeśli zostanie określony domyślny argument, to należy również określić wszystkie argumenty leżące po jego lewej stronie. Nie można pominąć argumentu domyślnego poprzez pozostawienie pustego miejsca w wywołaniu funkcji. Przy nagłówku funkcji określonym jak poniżej:

  ```
  def f(a = 1, b = 2, c = 3):
  ```

  nie można wywoływać funkcji, jeśli pominie się `b` i poda `c`. Gdy tak się stanie, Python zgłosi wyjątek `SyntaxError`:

  ```
  f(5,,10)    # Błąd składni
  ```

- Wartość domyślna jest obliczana i zachowywana tylko raz podczas definiowania funkcji, a nie podczas jej wywołania. Takie zachowanie oznacza, że jeśli wartość domyślna jest obiektem zmiennym (listą lub słownikiem), będzie ona gromadzić wszystkie zmiany dokonywane w miejscu przy kolejnych wywołaniach funkcji. Czasem może to być zgodne z oczekiwaniami, ale nie zawsze tak jest. Na rysunku 8.12 pokazano kilka przykładów współużytkowania wartości domyślnej przez kolejne wywołania funkcji, a także przykłady pokazujące, jak unikać tego współużytkowania poprzez zastosowanie `None` jako wartości domyślnej. Jeśli wartość domyślna jest obiektem niezmiennym (liczbą, łańcuchem lub krotką), to nie występuje efekt jej współużytkowania przy różnych wywołaniach funkcji, ponieważ żadna próba modyfikacji nie spowoduje utworzenia nowego obiektu ani zmiany już istniejącego. Więcej informacji na temat przekazywania argumentów można znaleźć w podrozdziale „Przekazywanie argumentów zmiennych i niezmiennych do funkcji” w dalszej części tego rozdziału.

Określanie argumentów jako słów kluczowych

Do funkcji można przekazywać *argumenty jako słowa kluczowe* za pomocą nazw odpowiednich parametrów (a nie na podstawie ich położenia). Wystarczy po prostu w wywołaniu funkcji wpisać *nazwa = arg*. Definicja funkcji nie wymaga żadnych zmian, by można było używać takiego sposobu przekazywania argumentów.

Przekazywanie argumentów jako słów kluczowych przydaje się najbardziej wtedy, gdy funkcja ma dużą liczbę parametrów, z których większość ma taką samą wartość domyślną. Załóżmy, że trzeba wywołać fikcyjną funkcję wyszukującą dany tekst w pliku:

```
def search(tekst, plik,
  match_case = 0,
  match_whole_word = 0,
  match_wildcards = 0,
  reverse = 0):
```

Parametr `reverse` zmienia kolejność przeszukiwania. Jeśli trzeba będzie rozpocząć przeszukiwanie wstecz, a nie do przodu i pozostałe wartości domyślne będą mogły być przyjęte, wywołanie tej funkcji z użyciem argumentów pozycyjnych może być następujące:

```
search("sekwoja", "drzewa.txt", 0, 0, 0, 1)
```

Oto jego równoważnik z użyciem argumentów jako słów kluczowych:

```
search("sekwoja", "drzewa.txt", reverse = 1)
```

Warto zwrócić uwagę na zalety wywołania ze słowami kluczowymi: jest ono bardziej czytelne, można pominąć opcjonalne argumenty i można umieszczać słowa kluczowe w dowolnej kolejności.

```
>>> def get_color(red = 0, green = 0,
↳blue = 0):
...   kolory = {
...    (0,   0,   0)  : "black",
...    (0,   0,   255): "blue",
...    (0,   255, 0)  : "green",
...    (0,   255, 255): "cyan",
...    (255, 0,   0)  : "red",
...    (255, 0,   255): "magenta",
...    (255, 255, 0)  : "yellow",
...    (255, 255, 255): "white"
...    }
...   rgb = (red, green, blue)
...   if kolory.has_key(rgb):
...     return kolory[rgb]
...   else:
...     return rgb
...
>>> print get_color(red = 255)
red
>>> print get_color(green = 255,
↳blue = 255, red = 255)
white
>>> print get_color(255, 255,
↳blue = 255)
white
>>> print get_color(18, blue = 255)
(18, 0, 255)
>>> print get_color(red = 255, 0, 255)
SyntaxError: non-keyword arg after keyword
↳arg
>>> print get_color(255, red = 255)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: get_color() got multiple values
↳for keyword argument 'red'
>>> print get_color(puce = 255)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: get_color() got an unexpected
↳keyword argument 'puce'
```

Rysunek 8.13. *Przykłady wywołań funkcji z argumentami w postaci słów kluczowych i kilka przykładów błędów popełnianych przy takich wywołaniach*

Wywoływanie funkcji z argumentami w postaci słów kluczowych

1. Wpisz:

func(*pos_args*, *keyword1* = *arg1*, *keyword2* = *arg2*,...)

Nazwa *func* jest nazwą funkcji, *pos_args* to argumenty pozycyjne (jeśli takie istnieją), zaś *arg*, *arg2*,... są argumentami (wyrażeniami) przekazywanymi do parametrów nazwanych *keyword1*, *keyword2* w definicji funkcji (rysunek 8.13).

Oto kilka reguł stosowania argumentów w postaci słów kluczowych:

- w odróżnieniu od argumentów pozycyjnych argumenty w postaci słów kluczowych nie muszą być podawane w tej samej kolejności, w jakiej występują na liście argumentów w definicji funkcji;
- argument pozycyjny nie może następować po argumencie w postaci słowa kluczowego (patrz: rysunek 8.13);
- nie można używać argumentów zduplikowanych, równocześnie podając je jako pozycyjne i jako słowa kluczowe (patrz: rysunek 8.13).

Wskazówka

- Użycie argumentów w postaci słów kluczowych w wywołaniach funkcji wbudowanych nie jest dozwolone. W takiej sytuacji Python zgłasza wyjątek `TypeError`.

Określanie dowolnej liczby argumentów pozycyjnych

Funkcja może pobierać zmienną (czyli taką, która nie jest ustalona z góry) liczbę argumentów pozycyjnych. Kilka z takich funkcji już omawiano (na przykład `max()` i `min()`, które pobierają listy argumentów o zmiennej długości). Przy wywołaniu takiej funkcji Python dopasowuje normalne argumenty pozycyjne (począwszy od lewego), a później umieszcza każdy nadmiarowy argument w krotce, która może być wykorzystana w funkcji.

Definiowanie funkcji pobierającej dowolną liczbę argumentów pozycyjnych

1. Wpisz:

   `def func(pos_params, *args):`.

   Nazwa *func* jest nazwą funkcji, *pos_params* są parametrami pozycyjnymi (jeśli takie istnieją), a *args* jest krotką, która uzyskuje każde nadmiarowe argumenty pozycyjne (* nie jest częścią nazwy parametru).

Wywołanie funkcji z dowolną liczbą argumentów pozycyjnych

1. Wpisz:

   func(*pos_args*, *arg1*, *arg2*,...).

   Nazwa *func* jest nazwą funkcji, *pos_args* są zwykłymi argumentami pozycyjnymi (jeśli takie występują), zaś *arg1*, *arg2*,... są argumentami nadmiarowymi (wyrażeniami), które są umieszczone w krotce.

Na rysunku 8.14 pokazano funkcję obliczającą średnią arytmetyczną dowolnej liczby swoich argumentów. W skryptach 8.2 i 8.3 przedstawionych w tym rozdziale pokazano funkcje, które odpowiednio zwracają przecięcie i złączenie dwóch sekwencji.

```
>>> def mean(*nums):
...   if len(nums) == 0:
...     return 0.0
...   else:
...     suma = 0.0
...     for x in nums: suma += x
...     return suma/len(nums)
...
>>> print mean()
0.0
>>> print mean(4)
4.0
>>> print mean(-1, 0, 2)
0.333333333333
```

Rysunek 8.14. *Funkcja mean pobiera dowolną liczbę argumentów i zwraca ich średnią arytmetyczną*

Skrypt 8.6. *Funkcja intersection pobiera dowolną liczbę sekwencji i zwraca listę, która zawiera pozycje wspólne dla wszystkich jej argumentów*

Skrypt

```
def intersection(*seqs):
  wynik = []
  for x in seqs[0]:
    for y in seqs[1:]:
      if x not in y:
        break
      else:
        wynik.append(x)
  return wynik
```

Skrypt 8.7. *Funkcja union pobiera dowolną liczbę sekwencji i zwraca listę, która zawiera wszystkie pozycje pojawiające się w jej argumentach (bez duplikatów)*

Skrypt

```
def union(*seqs):
  wynik = []
  for seq in seqs:
    for x in seq:
      if not x in wynik:
        wynik.append(x)
  return wynik
```

```
>>> s1 = "sekwoja"
>>> s2 = "wielostop"
>>> s3 = "formaldehyd"
>>> print intersection(s1, s2, s3)
['s', 'e', 'e', 'w', 'o', 'o']
>>> print union(s1, s2, s3)
['s', 'e', 'k', 'w', 'o', 'j', 'a', 'i',
'l', 't', 'p', 'f', 'r', 'm', 'd', 'h',
'y']
```

Rysunek 8.15. *Przykładowe wywołania funkcji ze skryptów 8.6 i 8.7*

Skrypty 8.6 i 8.7 zawierają zmodyfikowane funkcje obliczające przecięcie i złączenie dowolnej liczby sekwencji. Na rysunku 8.15 pokazano przykłady wywołań skryptów 8.6 i 8.7.

Wskazówki

- Jeśli do funkcji nie przekazano żadnych argumentów nadmiarowych, to *args* staje się domyślnie pustą krotką.
- Opis wprowadzania dowolnej liczby argumentów w postaci słów kluczowych podano w następnym podrozdziale.
- N Zapis *args wprowadzono w wersji 2.0 języka Python.
- J W języku Perl argumenty podprogramu są gromadzone w tablicy o nazwie @_. W języku C do przechowywania list argumentów o zmiennej długości używa się ... i va_list.

Określanie dowolnej liczby argumentów jako słów kluczowych

Funkcja może pobierać zmienną (czyli taką, która nie jest ustalona z góry) liczbę argumentów w postaci słów kluczowych, jeśli nazwa jej ostatniego parametru rozpoczyna się od znaków **. Przy wywołaniu takiej funkcji Python dopasowuje normalne argumenty pozycyjne (począwszy od lewego), a potem dopasowuje nadmiarowe argumenty pozycyjne (patrz: poprzedni podrozdział) i umieszcza każdy nadmiarowy argument podany jako słowo kluczowe w słowniku, który może być wykorzystany w funkcji. Na rysunku 8.16 pokazano przykład takiej funkcji.

```
>>> def ksiazka(tytul, autor, **inne):
...   print "tytul :", tytul
...   print "autor :", autor
...   for (k, v) in inne.items():
...     print k, ":", v
...
>>>
>>> ksiazka("Therapy",
...         "Lodge, David",
...         wydawca = "Penguin",
...         format = "miekka oprawa",
...         stron = 336,
...         rok = 1996)
tytul : Therapy
autor : Lodge, David
format : miekka oprawa
wydawca : Penguin
stron : 336
rok : 1996
```

Rysunek 8.16. *Funkcja książka gromadzi w słowniku inne nadmiarowe argumenty w postaci słów kluczowych i wyświetla pary klucz-wartość należące do tego słownika*

Definiowanie funkcji pobierającej dowolną liczbę argumentów w postaci słów kluczowych

1. Wpisz nagłówek definicji funkcji:

   ```
   def func(pos_params, *args, **kwargs):.
   ```

   Nazwa *func* jest nazwą funkcji, *pos_params* to zwykłe parametry pozycyjne funkcji (jeśli takie występują), *args* jest krotką, która odbiera nadmiarowe argumenty pozycyjne (jeśli takie istnieją), zaś *kwargs* jest słownikiem, który odbiera każde nadmiarowe argumenty w postaci słów kluczowych (** nie jest częścią nazwy parametru).

Wywoływanie funkcji z dowolną liczbą argumentów w postaci słów kluczowych

1. Wpisz:

   ```
   func(pos_args, keyword1 = arg1,
   keyword2 = arg2,...)
   ```

   Nazwa *func* jest nazwą funkcji, *pos_args* są zwykłymi i nadmiarowymi argumentami pozycyjnymi (jeśli takie istnieją), a *arg1*, *arg2*,... to nadmiarowe argumenty (wyrażenia) w postaci słów kluczowych, które są umieszczane w słowniku z odpowiadającymi im kluczami *keyword1*, *keyword2*,....

Wskazówki

- Jeśli nie będą przekazywane żadne nadmiarowe argumenty w postaci słów kluczowych, to domyślnie słownik *kwargs* stanie się słownikiem pustym.
- Argumenty, które nie są przekazywane jako słowa kluczowe, muszą występować przed argumentami przekazywanymi w postaci słów kluczowych.
-  Zapis **kwargs wprowadzono w wersji 2.0 języka Python.

```
>>> def arg_demo(a, b = -99, *args,
↳**kwargs):
...    print "Argumenty normalne:"
...    print "\ta =", a, ", b =", b
...    print "Argumenty nadmiarowe
pozycyjne:"
...    print "\t", args
...    print "Argumenty nadmiarowe kluczowe:"
...    print "\t", kwargs
...
>>> arg_demo(1)
Argumenty normalne:
       a = 1 , b = -99
Argumenty nadmiarowe pozycyjne:
       ()
Argumenty nadmiarowe kluczowe:
       {}
>>> arg_demo(b = 1, a = 2)
Argumenty normalne:
       a = 2 , b = 1
Argumenty nadmiarowe pozycyjne:
       ()
Argumenty nadmiarowe kluczowe:
       {}
>>> arg_demo(1, 2, 3, 4)
Argumenty normalne:
       a = 1 , b = 2
Argumenty nadmiarowe pozycyjne:
       (3, 4)
Argumenty nadmiarowe kluczowe:
       {}
>>> arg_demo(1, 2, 3, 4, x = 5, y = 6)
Argumenty normalne:
       a = 1 , b = 2
Argumenty nadmiarowe pozycyjne:
       (3, 4)
Argumenty nadmiarowe kluczowe:
       {'x': 5, 'y': 6}
>>> arg_demo(1, b = 2, x = 5, y = 6)
Argumenty normalne:
       a = 1 , b = 2
Argumenty nadmiarowe pozycyjne:
       ()
Argumenty nadmiarowe kluczowe:
       {'x': 5, 'y': 6}
>>> arg_demo(1, b = 2, 3, 4, x = 5, y = 6)
SyntaxError: non-keyword arg after keyword
↳arg
```

Rysunek 8.17. *Pokazane tu wywołania funkcji ilustrują przypisywanie różnych kombinacji argumentów pozycyjnych (w postaci słów kluczowych i nadmiarowych)*

Mieszane techniki przekazywania argumentów

W jednym wywołaniu funkcji można stosować mieszane sposoby przekazywania argumentów. Czasami jednak trudno rozszyfrować przyporządkowanie argumentów pozycyjnych i nadmiarowych (szczególnie wtedy, gdy funkcja jest skomplikowana). Pokazana na rysunku 8.17 funkcja `arg_demo` pobiera argumenty zwykłe, nadmiarowe pozycyjne i nadmiarowe w postaci słów kluczowych. Po wywołaniu tej funkcji (dotyczy to także każdej innej) można się przekonać, że Python przypisuje argumenty w następującej kolejności:

1. argumenty, które nie są przekazywane jako słowa kluczowe — według ich położenia;
2. argumenty przekazywane w postaci słów kluczowych — według ich nazwy;
3. nadmiarowe argumenty, które nie są przekazywane jako słowa kluczowe — do krotki `args`;
4. nadmiarowe argumenty przekazywane w postaci słów kluczowych — do słownika `kwargs`.

Po tych przypisaniach wszystkie nieprzypisane argumenty uzyskują swoje wartości domyślne.

Wskazówki

- Definicje funkcji i ich wywołania są opisane szczegółowo w kolejnych rozdziałach (7.5 i 5.3.4) podręcznika systemowego *Python Reference Manual*.

Tworzenie funkcji rekurencyjnych

Rekurencja oznacza możliwość wywoływania funkcji przez nią samą. W praktyce jest to często stosowane przy przeglądaniu drzewiastych struktur danych (na przykład hierarchii katalogów) lub przy przetwarzaniu struktur danych zagnieżdżonych na dowolną głębokość (na przykład lista zawierająca listy, zawierające listy itd.). Tak działa na przykład funkcja `copy.deepcopy()` tworząca kopię listy za pomocą rekurencyjnego kopiowania zagnieżdżonych obiektów pochodzących z listy pierwotnej (patrz: podrozdział „Kopiowanie listy lub krotki" w rozdziale 5).

Rekurencja jest także stosowana do obliczania wartości matematycznych, których definicje mają postać rekurencyjną. Na rysunkach 7.16 i 7.15 w rozdziale 7. pokazano obliczenia silni danej liczby z zastosowaniem pętli `while` oraz obliczenia wartości ciągu Fibonacciego. Te same obliczenia można wykonać posługując się rekurencją.

Silnia liczby *n* (zapisywana jako *n*!) jest iloczynem wszystkich kolejnych nieujemnych liczb całkowitych aż do *n*, czyli:

$$n! = n * (n - 1) * (n - 2) * ... * 1.$$

Definicja określa również to, że 0! = 1. Używając zapisu matematycznego rekurencyjną definicję można zapisać następująco:

$$0!=1$$
$$n!=n \times (n-1)!$$

W skrypcie 8.8 pokazano sposób obliczania silni z zastosowaniem rekurencji.

Ciąg Fibonacciego tworzą liczby (1, 1, 2, 3, 5, 8, 13...), z których każda następna jest sumą dwóch poprzednich. Rekurencyjna definicja tego ciągu w zapisie matematycznym ma następującą postać:

$$fibonacci(0) = 1$$
$$fibonacci(1) = 1$$
$$fibonacci(n) = fibonacci(n-1) + fibonacci(n-2)$$

W skrypcie 8.9 pokazano sposób rekurencyjnych obliczeń liczb Fibonacciego.

Skrypt 8.8. *Funkcja factorial oblicza rekurencyjnie silnię liczby n*

Skrypt

```
def factorial(n):
  if n == 0:
    return 1
  else:
    return n * factorial(n - 1)
```

Skrypt 8.9. *Funkcja fibonacci oblicza rekurencyjnie n-ty wyraz ciągu Fibonacciego*

Skrypt

```
def fibonacci(n):
  if n < 2:
    return 1
  else:
    return fibonacci(n - 1) + fibonacci(n - 2)
```

```
>>> for i in range(6):
...   print str(i) + "! =", factorial(i)
...
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
>>>
>>> for i in range(11):
...   print fibonacci(i),
...
1 1 2 3 5 8 13 21 34 55 89
```

Rysunek 8.18. *Przykładowe wywołania funkcji ze skryptów 8.8 i 8.9*

Na rysunku 8.18 pokazano przykładowe wyniki uzyskane po uruchomieniu skryptów 8.8 i 8.9.

Wskazówki

- Funkcja rekurencyjna wywołująca samą siebie działa nieskończenie, zatem wymagane jest wprowadzenie warunku zakończenia obliczeń, po spełnieniu którego funkcja będzie mogła zwrócić wartość bez wywoływania samej siebie. W skrypcie 8.8 takim końcowym warunkiem jest `n == 0`, zaś w skrypcie 8.9 jest to `n < 2`.
- Pomimo że rekurencja umożliwia pisanie niektórych programów w postaci niewielkich i prostych funkcji, nie gwarantuje ona szybkości ani wydajności. Czasem można napisać bardziej czytelne i łatwiejsze w użyciu procedury iteracyjne wykorzystujące pętle `while` lub `for`.
- Błędne użycie rekurencji może doprowadzić do zużycia przez program wszystkich zasobów pamięci i do zawieszenia Pythona. Za pomocą funkcji `sys.getrecursionlimit()` i `sys.setrecursionlimit()` można odpowiednio uzyskać i ustawić maksymalną głębokość rekurencji. Wartością domyślną jest 1000 i po jej przekroczeniu Python zgłasza wyjątek `RuntimeError`. Zagadnienie to jest szczegółowo opisane w podrozdziale 3.1 systemowego podręcznika *Python Library Reference*.
- **J** Języki Perl, C i Java także obsługują rekurencję.

Przekazywanie zmiennych i niezmiennych argumentów do funkcji

Podczas wywołania funkcji Python przekazuje do niej z programu wywołującego odwołania do obiektów jej argumentów. W podrozdziale „Tworzenie zmiennych" w rozdziale 2. pokazano, że odwołania do obiektów są wskaźnikami do obiektów przechowywanych w pamięci.

W przypadku obiektów niezmiennych (liczb, łańcuchów i krotek) takie działanie powoduje w rzeczywistości utworzenie lokalnej kopii argumentu wewnątrz wywoływanej funkcji, ponieważ funkcja nie może zmienić pierwotnego obiektu.

Zupełnie inaczej wygląda to w przypadku obiektów zmiennych (czyli list i słowników). Przekazanie odwołania powoduje utworzenie aliasu pierwotnego obiektu wewnątrz wywoływanej funkcji. W podrozdziale „Tworzenie odwołań do tego samego obiektu" w rozdziale 2. pokazano, że aliasy są zmiennymi, które współużytkują odwołania do tego samego obiektu. Obiekt pierwotny (w programie wywołującym) będzie się zmieniał, jeśli zostanie zmodyfikowany w miejscu obiekt przekazany (w funkcji). Z drugiej strony — jeśli aliasowi zostanie ponownie przypisana wartość, to odwołanie do obiektu pierwotnego zostanie utracone i zmiany w funkcji nie będą wpływały na obiekt w programie wywołującym.

Na rysunku 8.19 pokazano przykłady ilustrujące takie zachowanie przy przekazywaniu argumentów.

```
>>> def change_args(x, seq1, seq2):
...   x = ""
...   seq1[0] = -99
...   seq2 = []
...
>>> s = "mimetyczny"
>>> lista1 = [1, 2, 3]
>>> lista2 = [4, 5, 6]
>>> change_args(s, lista1, lista2)
>>> print s
mimetyczny
>>> print lista1
[-99, 2, 3]
>>> print lista2
[4, 5, 6]
```

Rysunek 8.19. *Ponowne przypisanie x wewnątrz funkcji change_args nie wpływa na wartość s na zewnątrz funkcji, ponieważ s jest niezmienne (więc x jest rzeczywiście kopią s). Zmiana w miejscu seq1 jest widoczna w lista1, ponieważ lista1 jest zmienna (więc lista1 i seq1 są aliasami). Obiekty lista2 i seq2 są początkowo aliasami, lecz przypisanie do seq2 nowego obiektu niszczy jego odwołanie do obiektu współużytkowanego, więc zmiana dokonana wewnątrz funkcji nie będzie widoczna w obiekcie lista2*

Wskazówki

- Aby uniknąć modyfikacji współużytkowanego obiektu zmiennego w funkcji, należy przekazywać do niej kopię tego obiektu. W wywołaniu funkcji przedstawionej na rysunku 8.19 `lista1` nie będzie modyfikowana, gdy wywołanie:

  ```
  change_args(s, lst1, lst2)
  ```

  zmieni się na:

  ```
  change_args(s, lst1[:], lst2)
  ```

  Więcej informacji na temat kopiowania list i słowników podano w podrozdziałach „Kopiowanie listy lub krotki” w rozdziale 5. i „Kopiowanie słownika” w rozdziale 6.

- Modyfikowanie w miejscu obiektów zmiennych dokonywane w funkcji powoduje skutki uboczne. Czasem są one pożądane, lecz zasadą powinno być zwracanie wyników działania funkcji za pomocą instrukcji `return` (patrz: „Zastosowania narzędzi programowania funkcjonalnego” w dalszej części tego rozdziału).

- Do dobrych zwyczajów należy tworzenie lokalnych kopii argumentów zmiennych w funkcji po to, by z całą pewnością można było uniknąć skutków ubocznych. Usuwanie tych skutków ubocznych powinno odbywać się wewnątrz funkcji, a nie w programie wywołującym.

- Informacje o użyciu obiektów zmiennych jako parametrów domyślnych podano w podrozdziale „Określanie domyślnych wartości parametrów”.

- **J** Przekazanie obiektu niezmiennego w Pythonie jest zbliżone do wywołania za pomocą wartości w języku C, natomiast przekazanie obiektu zmiennego jest podobne do przekazania wskaźnika w języku C.

Deklaracje zmiennych globalnych

Jeśli wewnątrz funkcji zostanie utworzona zmienna, to dostęp do niej będzie miał tylko kod z wnętrza samej funkcji. Będzie on mógł odczytywać i modyfikować tę zmienną, ponieważ jego zasięg jest *lokalny* dla funkcji. Wszystkie zmienne z listy parametrów funkcji także są lokalnymi zmiennymi (rysunek 8.20).

Zasięg zmiennej określa obszar, wewnątrz którego w danym bloku można się do niej odwoływać. Zasięg może być lokalny lub globalny. Dostęp i możliwość modyfikacji zmiennej *globalnej* utworzonej poza funkcją można uzyskać z dowolnej funkcji poprzez zadeklarowanie tej zmiennej jako globalnej za pomocą instrukcji `global`.

Jak zadeklarować zmienne jako globalne?

1. W ciele funkcji wpisz:

   `global` *`var1, var2,...`*.

   Nazwy *`var1, var2,...`* są nazwami zmiennych oddzielonymi za pomocą przecinków.

W przykładzie pokazanym na rysunku 8.21 Python zakłada, że `x` jest zmienną lokalną i zgłasza wyjątek przy próbie dostępu do tej zmiennej przed jej określeniem. Na rysunku 8.22 podano rozwiązanie tego problemu za pomocą deklaracji `x` jako zmiennej globalnej.

```
>>> def printsum(a, b):
...   x = a + b
...   print x
...
>>> printsum(3,9)
12
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'x' is not defined
```

Rysunek 8.20. *Zmienna x jest zmienną lokalną w stosunku do funkcji printsum. Gdy printsum zakończy działanie, x zostanie zniszczone. Python zgłaszał więc będzie wyjątek NameError przy próbie użycia x poza funkcją printsum. Obiekty a i b są również niszczone po zakończeniu działania printsum*

```
>>> x = 39
>>> def add1():
...   x = x + 1
...
>>> add1()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 2, in add1
UnboundLocalError: local variable 'x'
↳referenced before assignment
```

Rysunek 8.21. *Funkcja add1 nie rozpoznaje zmiennej x zadeklarowanej na zewnątrz*

```
>>> x = 39
>>> def add1():
...   global x
...   x = x + 1
...
>>> add1()
>>> x
40
```

Rysunek 8.22. *Po zadeklarowaniu x jako zmiennej globalnej funkcja add1 szuka jej poza swoim lokalnym zasięgiem*

```
>>> def accessor():
...   print a
...
>>> def assigner():
...   a = -99
...
>>> a = 10
>>> accessor()
10
>>> assigner()
>>> print a
10
```

Rysunek 8.23. *Python pozwala na dostęp (lecz nie na modyfikację) do zmiennej na zewnątrz funkcji bez jej deklarowania jako globalnej. Zmienna a istnieje poza funkcjami accessor i assigner. Mimo że funkcja accessor może wyświetlić wartość a, funkcja assigner nie jest w stanie zmienić jej wartości, ponieważ przypisanie modyfikuje a jako lokalną zmienną wewnątrz funkcji*

Wskazówki

- Reguły zasięgu obowiązujące w Pythonie podano w tym podrozdziale w dosyć dużym uproszczeniu. Pełny opis znajduje się w podrozdziale „Wyjaśnienie reguł zasięgu" w rozdziale 9.
- Instrukcja `global` może występować w funkcji w dowolnym miejscu, lecz zmienna, której ona dotyczy, musi być umieszczona za nią.
- Nie należy używać zmiennej globalnej ani jako zmiennej sterującej w pętli `for`, ani jako parametru w definicji funkcji, definicji klasy (`class`) lub w instrukcji `import`.
- Dostęp do zmiennej istniejącej poza funkcją (bez możliwości jej modyfikacji) można uzyskać bez użycia instrukcji `global` (patrz: rysunek 8.23). Taka technika jest jedynie przykładem tego, co umożliwia Python dzięki swoim regułom zasięgu. Tego rodzaju sztuczki nie są jednak zalecane, ponieważ zmniejszają czytelność kodu. Należy raczej zadeklarować zmienną jako globalną lub — co jest bardziej wskazane — zbudować funkcję tak, by przyjmowała tę zmienną jako argument.
- Pomimo że zmienne o szerokim zasięgu są przydatne w niektórych sytuacjach, stosowanie ich w nadmiarze prowadzi do uwikłania się w zależności, które utrudniają sesje z debuggerem. Programiści, którzy tak postępują, napotykają często na kłopoty. Należy zatem komunikować się z funkcją za pomocą argumentów i zwracanych wartości.

Przypisanie funkcji do zmiennej

Funkcja — podobnie jak wszystko w Pythonie — jest obiektem i jako obiekt może być przypisana do zmiennej. Przypisanie takie tworzy alias, który odwołuje się do tego samego obiektu funkcji o pierwotnej nazwie (patrz: „Tworzenie odwołań do tego samego obiektu" w rozdziale 2.).

Jak przypisać funkcję do zmiennej?

1. Wpisz *zmienna = func*.

   Nazwa *zmienna* jest nazwą zmiennej, a *func* jest nazwą funkcji. Należy tu wpisać tylko samą nazwę funkcji bez nawiasów czy parametrów.

Oprócz możliwości wywoływania obiekt funkcji może być przypisywany, zachowywany, przekazywany itd. — podobnie jak listy, łańcuchy i inne obiekty (patrz: rysunek 8.24).

```
>>> def square(x): return x*x
...
>>> def cube(x): return x*x*x
...
>>> pow2 = square
>>> pow3 = cube
>>> type(pow2)
<type 'function'>
>>> print pow2
<function square at 007A7FAC>
>>> print pow2(3)
9
>>> print pow3(pow2(2))
64
>>> dict = {
...   "pow2": pow2,
...   "pow3": pow3
...   }
>>> print dict["pow2"](4)
16
>>> print dict["pow3"](3)
27
>>> n = 3
>>> lista = [(pow2, n), (pow3, n)]
>>> for (func, arg) in lista:
...   print func(arg)
...
9
27
>>> def callfunc(func, arg):
...   return func(arg)
...
>>> x = callfunc(pow3, 3)
>>> print x
27
```

Rysunek 8.24. *Obiekty funkcji square i cube są przypisane odpowiednio do aliasów pow2 i pow3. Można także wywołać pow2 i pow3 tak jak square i cube. Można zapisać obiekty funkcji w strukturach danych, takich jak listy (lista) i słowniki (dict), a następnie uzyskać do nich dostęp w zwykły sposób. Można także przekazać obiekt funkcji jako argument, tak jak to zrobiono w callfunc*

```
>>> truncate = int
>>> print truncate(1.5)
1
>>> type(truncate)
<type 'builtin_function_or_method'>
>>> print truncate
<built-in function int>
```

Rysunek 8.25. *Można także utworzyć alias wbudowanych funkcji Pythona, tak jak to zrobiono tutaj w przypadku int*

Wskazówki

- Funkcje wbudowane można także przypisywać do zmiennych (rysunek 8.25).
- Wywołanie funkcji poprzez jej alias jest nazywane *wywołaniem pośrednim*.
- Ponowne przypisanie do aliasu zachowuje łącze z oryginalną funkcją. Na przykład w instrukcjach, takich jak poniżej:

  ```
  def func(): pass
  a = func
  b = a
  ```

  `func`, `a` i `b` mają taką samą wartość identyfikatora `id()`, zatem odwołują się do tego samego obiektu (patrz: podrozdziały „Określanie tożsamości obiektu” i „Tworzenie odwołań do tego samego obiektu” w rozdziale 2.).
- Narzędzia programowania funkcjonalnego używane w Pythonie intensywnie wykorzystują obiekty funkcji (patrz: następny podrozdział zatytułowany „Zastosowania narzędzi programowania funkcjonalnego”).
- J Przypisania funkcji są podobne do wskaźników funkcji w języku C.

Zastosowania narzędzi programowania funkcjonalnego

W podrozdziale „Tworzenie instrukcji z wyrażeń” w rozdziale 2. pokazano, że niektóre funkcje i metody wbudowane powodują działania uboczne. Na przykład metoda `sort()` jako działanie uboczne powoduje sortowanie listy w miejscu. W podrozdziale „Przekazywanie zmiennych i niezmiennych argumentów do funkcji” pokazano, że przekazanie i modyfikacja argumentów zmiennych ma także skutki uboczne.

Skutki uboczne są w pewnym stopniu odrzucane przez tych programistów, którzy wierzą, że programowanie powinno polegać na obliczaniu wyrażeń, a nie na wykonywaniu poleceń. Języki programowania funkcjonalnego zabraniają działań ubocznych. Oznacza to, że funkcje i wyrażenia nie mogą zmieniać żadnych wartości w innych funkcjach, zatem w tych językach nie występują instrukcje przypisania, zmienne globalne, wywołania przez odniesienie, wskaźniki itp.

Zależnie od potrzeb Python może być traktowany jako język proceduralny (dzięki swojej zależności od funkcji) lub język programowania obiektowego (ponieważ obsługuje obiekty). Oprócz tego Python zawiera niektóre narzędzia programowania funkcjonalnego, dzięki którym funkcje mogą działać na pozycjach sekwencji. Narzędzia te są opisane w pozostałej części tego rozdziału (tabela 8.1).

Tabela 8.1. *Narzędzia programowania funkcjonalnego*

Narzędzie	Opis
`lambda`	Tworzy niewielkie, anonimowe funkcje
`apply()`	Niebezpośrednio wywołuje funkcję i przekazuje jej argumenty pozycyjne i słowa kluczowe
`map()`	Umożliwia działanie funkcji na pozycjach sekwencji i zwraca listę zawierającą wyniki
`zip()`	Pobiera zmienną (nieustaloną) liczbę sekwencji i zwraca listę krotek, na której *n*-ta krotka zawiera *n*-tą pozycję każdej sekwencji
`filter()`	Zwraca listę zawierającą pozycje sekwencji, które spełniają dany warunek
`reduce()`	Umożliwia następujące działanie funkcji na pozycjach sekwencji: najpierw funkcja pobiera dwie początkowe pozycje, a potem kolejno uzyskany w poprzednim wywołaniu wynik i następną pozycję sekwencji. Następuje w ten sposób stopniowa redukcja sekwencji do pojedynczej wartości
Skrót listy	Zwarta struktura syntaktyczna, która buduje listę symulując działanie zagnieżdżonych pętli `for` i instrukcji warunkowych `if`

Wskazówki

- Python zawiera opisane tu narzędzia w postaci wygodnych funkcji. Ich używanie nie jest obowiązkowe i zawsze można użyć instrukcji `def` zamiast wyrażenia `lambda` lub odtworzyć działanie innych narzędzi za pomocą instrukcji `if`, `while` i `for`. Narzędzia te są jednak często używane w programach w języku Python i dlatego należy się z nimi zapoznać (a może nawet polubić).
- Narzędzia programowania funkcjonalnego działają na obiektach wywoływalnych, czyli na funkcjach lub metodach zdefiniowanych przez użytkownika i wbudowanych oraz na obiektach klas. Aby określić, czy dany obiekt jest obiektem wywoływalnym, należy posłużyć się funkcją `callable(`*`obiekt`*`)` zwracającą 1 (prawda), jeżeli *`obiekt`* jest wywoływalny, albo 0 (fałsz) w przeciwnym wypadku.
- Więcej informacji na temat programowania funkcjonalnego w Pythonie można znaleźć w następujących materiałach źródłowych:

  Charming Python: Functional programming in Python — dokument autorstwa Davida Mertza, który jest dosępny pod adresem *www.106.ibm.com/developerworks/library/l-prog.html*;

  Python for Lisp Programmers — dokument autorstwa Petera Norviga, który jest dostępny pod adresem *www.norvig.com/python-lisp.html*;

  informacje ogólne o tych zagadnieniach są zawarte w dokumencie *Frequently Asked Questions for comp.lang.functional* dostępnym pod adresem *www.cs.nott.ac.uk/~gmh//faq.html*.
- Przykładami języków programowania funkcjonalnego są Lisp, Scheme i Haskell, lecz nie są one traktowane jako języki „czyste”. Na przykład Lisp zezwala na korzystanie ze zmiennych globalnych.

Zastosowanie wyrażeń lambda do tworzenia funkcji

Obiekty funkcji mogą być tworzone nie tylko za pomocą instrukcji `def`, lecz także za pomocą wyrażenia `lambda`. Funkcje `lambda` są niewielkie, bowiem ich definicje mieszczą się w jednym wierszu. Nie nadaje się im nazw, zatem bywają określane jako funkcje anonimowe.

Jak tworzyć funkcje za pomocą wyrażeń *lambda*?

1. Wpisz `lambda` *`param_list`*`:` *`expr`*,

   gdzie *`param_list`* jest listą zawierającą zero lub więcej parametrów oddzielonych przecinkami, a *`expr`* jest wyrażeniem, które korzysta z tych parametrów. Podobnie jak funkcje określone za pomocą instrukcji `def` funkcje `lambda` mogą mieć domyślne wartości parametrów i pobierać argumenty pozycyjne w postaci słów kluczowych i nadmiarowe.

Obiekt funkcji zwrócony przez wyrażenie `lambda` zachowuje się prawie tak jak odpowiednia funkcja określona za pomocą `def` (rysunek 8.26).

Definicja funkcji:

```
def name(param_list):
  return expr
```

jest odpowiednikiem:

```
name = lambda param_list: expr
```

Między `def` i `lambda` występują następujące różnice:

- `lambda` jest wyrażeniem, a `def` jest instrukcją (wyrażenie `lambda` można wstawiać wszędzie tam, gdzie dozwolone jest użycie zwykłego wyrażenia: jako argument, pozycja listy, wartość w słowniku itp.);
- blok `def` umożliwia użycie wielu instrukcji, zaś ciało `lambda` musi zawierać pojedyncze wyrażenie;

```
>>> def iloczyn(x,y): return x*y
...
>>> iloczyn(3,4)
12
>>> razy = lambda x,y: x*y
>>> razy(3,4)
12
>>> type(iloczyn)
<type 'function'>
>>> type(razy)
<type 'function'>
>>> print (lambda x,y: x*y)(3,4)
12
```

Rysunek 8.26. *Funkcja lambda o nazwie razy wykonuje tę samą czynność, co zdefiniowana za pomocą instrukcji def funkcja iloczyn. Zarówno lambda, jak i def zwracają obiekty funkcji. Wyrażenie lambda w instrukcji print jest funkcją wbudowaną w wierszu. Nie można tego uczynić za pomocą instrukcji def*

```
>>> dict = {
...    "pow2": (lambda x: x*x),
...    "pow3": (lambda x: x*x*x)
...    }
>>> print dict["pow2"](4)
16
>>> print dict["pow3"](3)
27
>>> n = 3
>>> lista = [lambda x: x*x, lambda x: x*x*x]
>>> for func in lista: print func(n),
...
9 27
>>> print lista[-1](n)
27
>>> draw_line = lambda x="-", n=10: x*n
>>> draw_line()
'----------'
>>> draw_line("*", n = 20)
'********************'
```

Rysunek 8.27. *Wyrażenia lambda można użyć wszędzie tam, gdzie można użyć zwykłego wyrażenia, czyli w listach lub w postaci wartości słownika — jak pokazano to tutaj. Funkcje lambda zostały wywołane poprzez zwyczajne odniesienia: za pomocą indeksu listy lub klucza słownika. Funkcja draw_line pokazuje, że w funkcjach lambda (podobnie jak w funkcji def) można definiować domyślne wartości parametrów oraz przyjmować argumenty pozycyjne lub w postaci słów kluczowych*

```
>>> a = "Travis trips trapshooting
↳Trappist traders".split()
>>> print a
['Travis', 'trips', 'trapshooting',
↳'Trappist', 'traders']
>>>
>>> a.sort(lambda x, y: cmp(x.lower(),
↳y.lower()))
>>> print a
['traders', 'Trappist', 'trapshooting',
↳'Travis', 'trips']
```

Rysunek 8.28. *W pierwotnym sortowaniu korzystającym z instrukcji def w celu określenia funkcji definiującej kolejność sortowania występowała zależność od wielkości liter (patrz: rysunek 5.45 w rozdziale 5.). Tutaj usunięto funkcję def i logika sortowania została wbudowana bezpośrednio do metody sortowania jako wyrażenie lambda*

- instrukcje, które nie są wyrażeniami (na przykład `if`, `while`, `for` i `print`), nie mogą wystąpić w ciele wyrażenia `lambda`, co ogranicza zakres operacji, które można zawrzeć w funkcji `lambda`;
- wyrażenie `lambda` nie wymaga przypisywania nazwy.

Na rysunku 8.27 pokazano kilka przykładów wyrażeń `lambda`. Wyrażenie `lambda` pokazane na rysunku 8.28 służy do modyfikacji sortowania pokazanego na rysunku 5.45 w taki sposób, by było ono niewrażliwe na wielkość liter (patrz: podrozdział „Samodzielne określanie kolejności sortowania listy” w rozdziale 5.).

Wskazówki

- Wyrażenia `lambda` są wygodnym sposobem tworzenia skróconych alternatyw dla funkcji `def`, chociaż ich użycie jest opcjonalne i wiąże się ze stylem programowania. Funkcje `lambda` są przydatne wtedy, gdy programista nie chce zaśmiecać tworzonego programu niewielkimi zwykłymi funkcjami, które nie będą wielokrotnie używane.
- Wyrażenia `lambda` podlegają tym samym regułom zasięgu, co funkcje (patrz: wcześniejszy podrozdział „Deklarowanie zmiennych globalnych” oraz podrozdział „Wyjaśnienie reguł zasięgu” w rozdziale 9.).
- **J** Nazwa `lambda` pochodzi od podobnych wyrażeń znanych z języka Lisp.

Zastosowanie apply w celu wywołania funkcji

Funkcja `apply()` pobiera funkcję, krotkę lub słownik jako argument. Wywołuje ona funkcję używając pozycji krotki jako argumentów pozycyjnych i pozycji słownika jako argumentów w postaci słów kluczowych, a następnie zwraca wynik wywołania funkcji.

Jak użyć apply w celu wywołania funkcji?

1. Wpisz `apply(`*`func`* `[,`*`args`*`[,`*`kwargs`*`]])`,

   gdzie *func* jest funkcją lub innym wywoływalnym obiektem, *args* jest krotką zawierającą argumenty pozycyjne, które są przekazywane do *func*, a *kwargs* jest słownikiem zawierającym argumenty w postaci słów kluczowych przekazywane do *func*. Klucze *kwargs* muszą być łańcuchami. Jeśli *args* lub *kwargs* są pominięte, to ich wartości nie są przekazywane do *func* (rysunek 8.29).

```
>>> apply(lambda x,y: x*y, (2,3))
6
>>> def get_color(red = 0, green = 0,
↳blue = 0):
...   kolory = {
...     (0,   0,   0)  : "black",
...     (0,   0,   255): "blue",
...     (0,   255, 0)  : "green",
...     (0,   255, 255): "cyan",
...     (255, 0,   0)  : "red",
...     (255, 0,   255): "magenta",
...     (255, 255, 0)  : "yellow",
...     (255, 255, 255): "white"
...     }
...   rgb = (red, green, blue)
...   if kolory.has_key(rgb):
...     return kolory[rgb]
...   else:
...     return rgb
...
>>> apply(get_color)
'black'
>>> apply(get_color, (255, 0 , 0))
'red'
>>> apply(get_color, (255, 0),
↳{"blue": 255})
'magenta'
>>> apply(get_color, (), {"blue": 255,
↳"green": 255, "red": 255,})
'white'
>>>
```

Rysunek 8.29. — *ciąg dalszy na następnej stronie*

```
>>> def circle(radius):
...     return 3.14159 * radius * radius
...
>>> def square(side):
...     return side * side
...
>>> def rectangle(base, height):
...     return base * height
...
>>> shape = "circle"
>>> radius = 2
>>> if shape == "circle":
...     func, args = circle, (radius,)
... elif shape == "square":
...     func, args == square, (side,)
... elif shape == "rectangle":
...     func, args = rectangle, (base, height)
...
>>> print shape, "area is", apply(func, args)
circle area is 12.56636
```

Rysunek 8.29. *Funkcja apply() dopasowuje przekazane argumenty do przekazanych parametrów funkcji, wywołuje funkcję i zwraca wynik wywołania*

Wskazówki

- Wywołanie funkcji w następującej postaci:

  `apply(`*`func, args, kwargs`*`)`

  jest odpowiednikiem wywołania:

  `func``(*`*`args`*`, **`*`kwargs`*`)`

  Więcej informacji o argumentach pozycyjnych i w postaci słów kluczowych podano w podrozdziałach „Określanie dowolnej liczby argumentów pozycyjnych” i „Określanie dowolnej liczby argumentów jako słów kluczowych”.

- Wywołania funkcji `apply(`*`func, args`*`)` i *`func`*`(`*`args`*`)` nie są sobie równoważne. W pierwszym wywołaniu liczba pozycji w krotce *`args`* jest równa liczbie argumentów przekazanych do funkcji *`func`*, natomiast w drugim — do funkcji *`func`* jest przekazany jeden argument: krotka *`args`*.

- Argument *`args`* może być dowolną sekwencją (łańcuchem, listą lub krotką). Jeśli nie jest on krotką, to zostanie na nią przekształcony.

-  W przedpotopowych wersjach języka Python użycie `apply()` było jedynym sposobem przekazania do funkcji argumentów pozycyjnych i argumentów w postaci słów kluczowych.

Użycie map w celu zastosowania funkcji do pozycji sekwencji

Argumentami funkcji `map()` są: funkcja i jedna lub więcej sekwencji. Funkcja będąca argumentem `map()` jest wielokrotnie wywoływana z argumentami będącymi pozycjami sekwencji i na zakończenie zwracana jest lista zawierająca wartości uzyskane w każdym z tych wywołań.

Jak użyć map, by funkcja oddziaływała na pozycje sekwencji?

1. Wpisz `map(`*`func, seq1 [,seq2,...]`*`)`,

   gdzie *`func`* jest funkcją lub innym obiektem wywoływalnym, a *`seq1`*, *`seq2,...`* to jedna lub więcej sekwencji oddzielonych za pomocą przecinków (łańcuchów, list lub krotek), których pozycje są przekazywane do *`func`*. Funkcja *`func`* musi pobierać tyle argumentów, ile jest sekwencji. Funkcja `map()` zawsze zwraca listę (bez względu na typy przekazywanych do niej sekwencji).

Jeśli zostanie podany tylko jeden argument w postaci sekwencji *`seq1`*, to *i*-ta pozycja w zwróconej liście będzie równa *`func(seq1[i])`* (patrz: rysunek 8.30).

Jeśli jako argumenty poda się więcej niż jedną sekwencję, to *i*-ta pozycja w zwróconej liście będzie równa *`func(seq1[i], seq2[i],...)`*. Jeżeli sekwencje mają nierówne długości, to funkcja `map()` rozszerza krótsze sekwencje wypełniając je pozycjami `None` tak, by pasowały do najdłuższej sekwencji. Zwracana lista ma taką samą długość, co najdłuższa sekwencja (patrz: rysunek 8.31).

```
>>> map(lambda x: x*x, range(5))
[0, 1, 4, 9, 16]
>>> map(ord, "sekwoja")
[115, 101, 107, 119, 111, 106, 97]
>>> def frac(x): return x - int(x)
...
>>> map(frac, [2.0, 0.5, -2.5])
[0.0, 0.5, -0.5]
>>> map(int, map(lambda x:2*x,
↳[2.9, 3.1, 4.0]))
[5, 6, 8]
>>> dict = {1: "jeden", 2: "dwa",
↳3: "trzy"}
>>> map(lambda s: s.upper(),
↳dict.values())
['JEDEN', 'DWA', 'TRZY']
```

Rysunek 8.30. *Funkcja map() powtarza wywołania funkcji (za każdym razem używając następującej pozycji sekwencji jako argumentu) i zwraca listę zawierającą wszystkie wyniki wywołań*

```
>>> map(max, [1, 5, 9], [0, 6, 4],
↳[8, 3, 5])
[8, 6, 9]
>>> map(min, [1, 5, 9], [0, 6, 4],
↳[8, 3, 5, 99])
[0, 3, 4, None]
>>> map(lambda x,y: x*y, [1, 2, 3],
↳[4, 5, 6])
[4, 10, 18]
>>> def sumprod(x, y):
...   return (x+y, x*y)
...
>>> map(sumprod, [1, 2, 3], [4, 5, 6])
[(5, 4), (7, 10), (9, 18)]
>>> m = [[1, 2, 3],
...      [4, 5, 6]]
>>> map(*[lambda *args: list(args)] + m)
[[1, 4], [2, 5], [3, 6]]
```

Rysunek 8.31. *W przykładzie z funkcją max największa z i-tych pozycji we wszystkich sekwencjach jest umieszczana na i-tym miejscu zwróconej listy. W przykładzie z funkcją min Python wypełnia krótsze sekwencje pozycjami None, by zrównać je z najdłuższą sekwencją. Funkcja sumprod zwraca krotkę, zatem zwrócona lista zawiera krotki. Końcowy przykład dokonuje transponowania macierzy m (listy list)*

```
>>> map(None, [1, 2, 3])
[1, 2, 3]
>>> map(None, "abc")
['a', 'b', 'c']
>>> map(None, [1, 2, 3], "abc")
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> map(None, [1, 2, 3], [4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
>>> map(None, [1, 2], [3, 4], [5, 6])
[(1, 3, 5), (2, 4, 6)]
>>> map(None, [1, 2, 3], [])
[(1, None), (2, None), (3, None)]
>>> map(None, [1, 2, 3], [4, 5])
[(1, 4), (2, 5), (3, None)]
>>> seq = range(1, 6)
>>> map(None, seq, map(lambda x: x*x, seq))
[(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
```

Rysunek 8.32. *Jeśli None jest odwzorowywane do pojedynczej sekwencji, to funkcja map() zwraca pierwotną sekwencję jako listę. Jeśli None jest odwzorowywane do wielu sekwencji, to map() zwraca listę krotek zawierających pozycję z każdej sekwencji*

Jeśli jako *func* zostanie użyte `None`, to funkcja `map()` działać będzie jak funkcja identyczności `lambda *x:x`. Jeśli zostanie dostarczony jeden argument w postaci sekwencji, to `map()` po prostu zwróci tę sekwencję przekształconą (jeśli będzie trzeba) na listę (czyli zwróci `list(`*`seq1`*`)`). Jeśli dostarczy się kilka sekwencji, to `map()` zwróci listę krotek w których *i*-ta krotka zawierać będzie *i*-tą pozycję każdej sekwencji. Jeśli sekwencje są nierównej długości, to `map()` dopełnia krótsze z nich pozycjami `None` zgodnie z wcześniejszym opisem (patrz: rysunek 8.32).

Wskazówki

- Wyrażenie:

  `map(`*`func`*`, `*`seq`*`)`

  jest podobne do tego kodu:

  ```
  if func is None:
    return list(seq)
  result = []
  for i in seq:
    result.append(func(i))
  return result
  ```

- Jeśli trzeba odwzorować `None` w sekwencji, warto zastanowić się nad użyciem funkcji `zip()` zamiast `map()` (patrz: następny podrozdział).

- Zamiast stosowania funkcji `map()` można rozważyć użycie skrótu listy. Wyrażenie:

  `map(`*`func`*`, `*`seq`*`)`

  jest równoważne następującemu skrótowi listy:

  `[`*`func`*`(x) for x in `*`seq`*`]`

  Zagadnienie to opisano w podrozdziale „Zastosowanie skrótów listy do tworzenia list”.

Zastosowanie zip do grupowania pozycji sekwencji

Funkcja `zip()` pobiera jedną lub więcej sekwencji jako argumenty. Zwraca ona listę krotek, w której *i*-ta krotka zawiera *i*-tą pozycję każdej sekwencji. Zwracana lista jest obcinana w taki sposób, aby jej długość była równa długości najkrótszej sekwencji pobranej jako argument.

Jak grupować pozycje sekwencji za pomocą zip?

1. Wpisz `zip(`*seq1* `[,`*seq2*`,...])`,

   gdzie *seq1, seq2,...* są jedną lub większą ilością sekwencji oddzielonych za pomocą przecinków (czyli są łańcuchami, listami lub krotkami). Funkcja `zip()` zawsze zwraca listę krotek bez względu na to, jakie są typy przekazywanych do niej sekwencji (rysunek 8.33).

Wskazówki

- Jeśli do funkcji `zip()` zostanie przekazany jeden argument, to zwróci ona listę jednopozycyjnych krotek.
- Jeśli do funkcji `zip()` zostaną przekazane dwa (lub więcej) argumenty o jednakowej długości, to `zip(`*seq1, seq2,...*`)` będzie równoważna `map(None,` *seq1, seq2,...*`)`.
-  Funkcja `zip()` została wprowadzona wersji 2.0 języka Python.

```
>>> zip([1, 2, 3])
[(1,), (2,), (3,)]
>>> zip("abc")
[('a',), ('b',), ('c',)]
>>> zip([1, 2, 3], "abc")
[(1, 'a'), (2, 'b'), (3, 'c')]
>>> zip([1, 2, 3], [4, 5, 6])
[(1, 4), (2, 5), (3, 6)]
>>> zip([1, 2], [3, 4], [5, 6])
[(1, 3, 5), (2, 4, 6)]
>>> zip([1, 2, 3], [])
[]
>>> zip([1, 2, 3], [4, 5])
[(1, 4), (2, 5)]
```

Rysunek 8.33. *Funkcja zip() zwraca listę krotek, w której i-ta krotka jest równa (seq1[i], seq2[i],...). Zwrócona lista ma długość zmniejszoną do długości najkrótszej sekwencji*

```
>>> def isvowel(c):
...   return c in "aeiouAEIOU"
...
>>> filter(isvowel, "sekwoja")
'eoa'
>>> filter((lambda x: x % 2), range(10))
[1, 3, 5, 7, 9]
>>>
>>> a = [1, 2, 3, 1, 5]
>>> b = [5, 6, 4, 4, 2]
>>> zip(a,b)
[(1, 5), (2, 6), (3, 4), (1, 4), (5, 2)]
>>> filter(lambda (x,y): x+y > 6, zip(a,b))
[(2, 6), (3, 4), (5, 2)]
>>>
>>> d1 = {"a": 1, "b": 2, "c": 3}
>>> d2 = {"b": 20, "c": 30, "d": 40}
>>> intersect = filter(d2.has_key,
↳d1.keys())
>>> print intersect
['b', 'c']
>>>
>>> def primes(n):
...   nums = range(2, n)
...   i = 0
...   while i < len(nums):
...     nums = filter(
...       lambda x, y=nums[i]:
...       x%y or x<=y, nums)
...     i += 1
...   return nums
...
>>> primes(20)
[2, 3, 5, 7, 11, 13, 17, 19]
```

Rysunek 8.34. *Kod z pierwszego przykładu zwraca samogłoski, z drugiego — usuwa liczby parzyste z listy range(), a z trzeciego — zwraca listę krotek, których pozycje dają w sumie liczbę większą niż 6. Następny kod zwraca listę kluczy wspólnych dla dwóch słowników, a ostatni zwraca listę liczb pierwszych, które nie są większe od danej liczby n*

- Zamiast funkcji `filter()` można rozważać zastosowanie skrótu listy. Wyrażenie

  `filter(`*`func`*`, `*`seq`*`)`

  jest równoważne następującemu skrótowi listy:

  `[x for x in `*`seq`*` if `*`func`*`(x)]`

  Zagadnienie to opisano w podrozdziale „Zastosowanie skrótów listy do tworzenia list”.

Zastosowanie filter do warunkowego usuwania pozycji sekwencji

Funkcja `filter()`, której argumentami są: funkcja i sekwencja, oblicza każdą pozycję sekwencji korzystając z funkcji przekazanej jako argument, a następnie zwraca sekwencję zawierającą te pozycje, dla których obliczana funkcja zwróciła wartość prawdziwą. Funkcja `filter()` nie modyfikuje pierwotnej sekwencji.

Jak warunkowo usuwać pozycje sekwencji za pomocą filter?

1. Wpisz `filter(`*`func`*`, `*`seq`*`)`,

   gdzie *`func`* jest funkcją lub innym wywoływalnym obiektem, który zwraca prawdę lub fałsz, a *`seq`* jest sekwencją (łańcuchem, listą lub krotką), której pozycje są przetwarzane przez *`func`*. Funkcja `filter()` zwraca sekwencję tego samego typu, co typ sekwencji przekazanej jako argument. Jeśli *`func`* jest równe `None`, to `filter()` zwraca sekwencję zawierającą wszystkie pozycje z *`seq`*, których wartość jest prawdą (rysunek 8.34).

Wskazówki

- Następujące wyrażenie:

  `filter(`*`func`*`, `*`seq`*`)`

  działa podobnie jak poniższy kod:

  ```
  result = []
  for i in seq:
    if (func(i) or
        (i and (func is None))):
      result.append(i)
  return result
  ```

- Informacje o wartościach prawdziwości podano w podrozdziale „Użycie operatorów logicznych” w rozdziale 2.

Zastosowanie reduce do redukcji sekwencji

Funkcja `reduce()` pobiera funkcję i sekwencję jako argumenty. Wywołuje ona pobraną funkcję używając dwóch początkowych pozycji sekwencji jako argumentów wywoływanej funkcji, a następnie wywołuje tę funkcję po raz drugi używając wyniku poprzedniego wywołania i trzeciej pozycji z sekwencji itd. (aż do wyczerpania wszystkich pozycji sekwencji). Funkcja `reduce()` zwraca wynik ostatniego wywołania funkcji pobranej jako argument. Wyrażenie `reduce(`*`func`*`, [a, b, c])` jest zatem równoważne wyrażeniu *`func`*`(`*`func`*`(a, b), c)`.

Jak używać reduce do redukcji sekwencji?

1. Wpisz `reduce(`*`func`*`, `*`seq`*` [,`*`init`*`])`,

   gdzie *`func`* jest funkcją lub innym wywoływalnym obiektem, a *`seq`* jest sekwencją (łańcuchem, listą lub krotką), której pozycje są kolejno przekazywane do *`func`*. Funkcja *`func`* musi przyjmować dwa argumenty i zwracać jedną wartość. Wielkość *`init`* jest opcjonalną wartością początkową (wyrażeniem), która jest używana w pierwszym obliczeniu (lub jako wartość zwracana jest domyślnie, jeśli *`seq`* okaże się sekwencją pustą). Python zgłasza wyjątek `TypeError`, jeśli *`seq`* jest sekwencją pustą i pominięte jest wyrażenie *`init`* (rysunek 8.35).

Wskazówka

- Poniższe wyrażenie:

  `reduce(`*`func`*`, `*`seq`*`, `*`init`*`)`

  działa podobnie jak następujący kod (zakładając, że domyślna wartość *`init`* wynosi `None`):

```
if list(seq) == []:
  return init
if init != None:
  seq.insert(0, init)
result = seq[0]
for next in seq[1:]:
  result = func(result, next)
return result
```

```
>>> nums = range(1,6)
>>> print nums
[1, 2, 3, 4, 5]
>>> sum = reduce(lambda x,y: x+y, nums)
>>> mean = reduce(lambda x,y: x+y,
↳nums)/float(len(nums))
>>> sumsq = reduce(lambda x,y: x+y,
↳map(lambda x: x*x, nums))
>>> print sum, mean, sumsq
15 3.0 55
>>>
>>> n = 5
>>> factorial = reduce((lambda a,
↳b: a*b), range(1, n+1))
>>> print factorial
120
>>>
>>> lst = [1, 2, 0, 4]
>>> first_false = reduce(lambda x,
↳y: x and y, lst)
>>> print first_false
0
>>>
>>> m = [[1, 2, 3],
...      [4, 5, 6],
...      [7, 8, 9]]
>>> n = 1
>>> def col_sum(m, n):
...   return reduce(lambda x,y: x+y,
↳map(lambda x, n=n: x[n], m))
...
>>> for i in range(len(m)):
...   print col_sum(m, i)
...
12
15
18
```

Rysunek 8.35. *Funkcje sum, mean i sumsq obliczają odpowiednio: sumę, wartość średnią i sumę kwadratów listy liczb. Funkcja factorial oblicza silnię liczby n (patrz: rysunek 7.16 w rozdziale 7.). Funkcja first_false zwraca pierwszą fałszywą wartość z listy (należy zmienić and na or, by zwracać pierwszą prawdziwą wartość). Funkcja col_sum zwraca sumę pozycji w n-tej kolumnie macierzy m (listy list)*

Zastosowanie skrótów listy do tworzenia list

Skrót listy jest zwartą i często bardziej czytelną alternatywą przy tworzeniu listy niż użycie funkcji `lambda`, `map()` lub `filter()`. Na rysunku 8.36 pokazano kilka przykładów ilustrujących różne zastosowania skrótów.

Użycie prostego skrótu listy

1. Wpisz `[`*`expr`* `for` *`var`* `in` *`seq`*`]`,

   gdzie *expr* jest wyrażeniem, *var* jest zmienną, a *seq* jest sekwencją (łańcuchem, listą lub krotką). Python przeszukuje w pętli sekwencję *seq* i przypisuje kolejno jej każdą pozycję do zmiennej *var* (obliczając dla każdej pozycji wartość wyrażenia *expr*). Zwrócona lista zawiera kolejne wartości wyrażenia *expr*.

Na końcu skrótu listy można umieścić warunek `if`, który umożliwia wyłączenie niektórych pozycji ze zwracanej listy.

Użycie prostego skrótu listy z warunkiem if

1. Wpisz `[`*`expr`* `for` *`var`* `in` *`seq`* `if` *`cond`*`]`.

   Ten skrót listy zachowuje się podobnie jak opisany wyżej (z tym wyjątkiem, że wartość wyrażenia *expr* jest obliczana tylko wtedy, gdy *cond* jest prawdą).

Skrót listy w rozszerzonej postaci może zawierać dowolną liczbę zagnieżdżonych pętli `for`, z których każda może mieć własny warunek `if`. W praktyce stosuje się nie więcej niż trzy poziomy zagnieżdżenia pętli `for`.

Skrót listy jest pojedynczym wyrażeniem. W podanych niżej przykładach wyrażenie to zostało podzielone na wiersze w celu zwiększenia czytelności.

Użycie rozszerzonego skrótu listy

1. Wpisz:

```
[expr for var1 in seq1 if cond1
      for var2 in seq2 if cond2
      ...
      for varN in seqN if condN]
```

Inna często spotykana postać zawiera pojedynczy warunek `if`:

```
[expr for var1 in seq1
      for var2 in seq2
      ...
      for varN in seqN
      if cond]
```

Odpowiada to w przybliżeniu następującemu kodowi:

```
result = []
for var1 in seq1:
  for var2 in seq2:
    ...
      for varN in seqN:
        if cond:
          result.append(expr)
```

```
>>> s = range(8)
>>> print s
[0, 1, 2, 3, 4, 5, 6, 7]
>>> [2*x for x in s]
[0, 2, 4, 6, 8, 10, 12, 14]
>>> [x*x for x in s]
[0, 1, 4, 9, 16, 25, 36, 49]
>>> [x*x for x in s if x % 2]
[1, 9, 25, 49]
>>> [(x, x*x) for x in s if x % 2]
[(1, 1), (3, 9), (5, 25), (7, 49)]
>>>
>>> s1 = [1, 2, 3]
>>> s2 = [4, 5, 6]
>>> s3 = [7, 8, 9]
>>> [x+y+z for (x,y,z) in zip(s1,s2,s3)]
[12, 15, 18]
>>>
>>> s1 = "aeiou"
>>> s2 = "sequoia"
>>> ords = [(c, ord(c)) for c in s1]
>>> print ords
[('a', 97), ('e', 101), ('i', 105),
↳('o', 111), ('u', 117)]
>>> all_in = 0 not in [c in s2 for c in s1]
>>> any_in = 1 in [c in s2 for c in s1]
>>> print all_in, any_in
1 1
>>>
>>> d1 = {"a": 1, "b": 2, "c": 3}
>>> values = [d1[key] for key in d1.keys()]
>>> print values
[2, 3, 1]
>>> ["%s=%s" % (k, v) for (k, v) in
↳d1.items()]
['b=2', 'c=3', 'a=1']
>>> d2 = {"b": 20, "c": 30, "d": 40}
```

Rysunek 8.36. — *ciąg dalszy na następnej stronie*

```
>>> intersection = [key for key in
↪d1.keys() if d2.has_key(key)]
>>> print intersection
['b', 'c']
>>>
>>> s = [[1, 2], [], [3, 4, 5], [6]]
>>> flatten = [x for nested in s for
↪x in nested]
>>> print flatten
[1, 2, 3, 4, 5, 6]
>>>
>>> s1 = [1, 2, 3]
>>> s2 = [5, 4, 3]
>>> pairs = [(x,y) for x in s1 for y in s2]
>>> bigpairs = [(x,y) for x in s1
...              for y in s2
...              if x*y > 6]
>>> print pairs
[(1, 5), (1, 4), (1, 3), (2, 5), (2, 4),
↪(2, 3), (3, 5), (3, 4), (3, 3)]
>>> print bigpairs
[(2, 5), (2, 4), (3, 5), (3, 4), (3, 3)]
>>> lst1 = [0, 1, 2, 3, 4]
>>> lst2 = [5, 6, 7, 8, 9]
>>> [(x,y) for x in lst1 if x > 2
↪for y in lst2 if y < 7]
[(3, 5), (3, 6), (4, 5), (4, 6)]
```

Rysunek 8.36. *Kilka początkowych przykładów pokazuje często stosowane skróty list liczbowych. W następnym przykładzie pokazano użycie zip() do sumowania pozycji w sekwencjach. Lista ords jest listą krotek, w których każda krotka zawiera znak i jego kod ASCII. Zmienna all_in określa to, czy wszystkie znaki z s1 znajdują się w s2; zmienna any_in określa natomiast, czy jakiekolwiek znaki z s1 znajdują się w s2. Lista values zawiera wartości słownika d1. Lista intersection jest listą kluczy, które występują w obydwu słownikach d1 i d2. Lista flatten powstaje w wyniku często spotykanej operacji „odgnieżdżania" zagnieżdżonych list. Lista pairs zawiera wszystkie dwupozycyjne połączenia dwóch list, zaś bigpairs zawiera dwupozycyjne połączenia tych pozycji, których iloczyn jest większy niż 6*

Wskazówki

- Zmienne zdefiniowane w skrócie listy (*var1*, *var2*,...) zastępują tak samo nazwane zmienne w bieżącym zasięgu i pozostają zdefiniowane także po utworzeniu listy. Na przykład w skrócie `[x for x in seq]` zmienna `x` zachowuje wartość równą wartości ostatniej pozycji w sekwencji `seq`, gdy skrót zakończy swoje działanie.
- Sekwencje mogą mieć różną długość, ponieważ każda z nich jest związana ze swoją własną zagnieżdżoną pętlą `for`.
- Jeśli skrót tworzy listę krotek, to wyrażenie musi być ujęte w nawiasy. Na przykład zapis `[(x, x*x) for x in seq]` jest poprawny, ale zapis `[x, x*x for x in seq]` — nie jest.
-  Skróty list zostały wprowadzone w wersji 2.0 języka Python.