helion.pl

Idź do

- Spis treści
- Przykładowy rozdział

Katalog książek

- Katalog online
- Zamów drukowany katalog

Twój koszyk

- Dodaj do koszyka

Cennik i informacje

- Zamów informacje o nowościach
- Zamów cennik

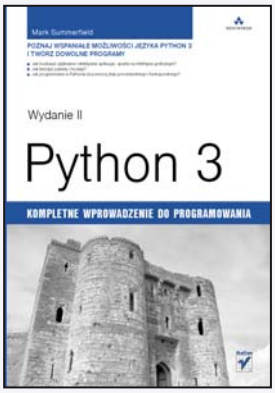
Czytelnia

- Fragmenty książek online

Kontakt

Helion SA
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl
© Helion 1991-2010

Python 3. Kompletne wprowadzenie do programowania. Wydanie II



Autor: Mark Summerfield
Tłumaczenie: Robert Górczyński
ISBN: 978-83-246-2642-7
Tytuł oryginału: Programming in Python 3: A Complete Introduction to the Python Language (2nd Edition)
Format: 170×230, stron: 640

Poznaj wspaniałe możliwości języka Python 3 i twórz dowolne programy

Python 3 uznany został za najlepszą dotychczasową wersję tego języka, ponieważ jego możliwości są dziś znacznie większe niż dawniej. Python 3 jest wygodny, spójny i ekspresyjny, a także niezależny od platformy sprzętowej i – co najważniejsze – dostarczany z pełną biblioteką standardową. Można go wykorzystać do programowania proceduralnego, zorientowanego obiektowo oraz (w mniejszym stopniu) do programowania w stylu funkcjonalnym. Autor książki, Mark Summerfield, ekspert w dziedzinie programowania, przedstawia szczegółowe informacje dotyczące tego języka w bardzo przyjazny sposób, co sprawia, że czytelnik szybko i sprawnie może napisać dowolny program.

Książka „Python 3. Kompletne wprowadzenie do programowania. Wydanie II" została zaprojektowana tak, aby mógł z niej korzystać zarówno ktoś o niewielkim doświadczeniu w programowaniu, jak i profesjonaliści, naukowcy, inżynierowie oraz studenci. Dzięki niej szybko nauczysz się m.in. wykorzystywać zaawansowane rodzaje danych, kolekcje oraz struktury kontrolne. Poznasz techniki analizy składniowej, obejmujące używanie modułów PyParsing i PLY. Dowiesz się, na czym polega rozkładanie obciążenia programu między wiele procesów i wątków, a także zaczniesz używać techniki Test Driven Development, aby uniknąć popełniania błędów. Znajdziesz tu wszelkie niezbędne informacje, dzięki którym będziesz mógł stworzyć solidne i wydajne programy.

- Tworzenie i uruchamianie programów Pythona
- Polecenia kontroli przepływu
- Rodzaje danych
- Funkcje i struktury kontrolne
- Moduły i pakiety
- Programowanie zorientowane obiektowo
- Obsługa plików
- Zaawansowane techniki programowania
- Kontrola dostępu do atrybutów
- Usuwanie błędów, testowanie i profilowanie
- Wyrażenia regularne

Ten podręcznik jest jak Python 3 – spójny, praktyczny i wygodny

Mark Summerfield jest informatykiem z wieloletnim doświadczeniem w dziedzinie programowania. Jest także współautorem książki „C++ GUI Programming with Qt 4" oraz autorem książki „Rapid GUI Programming with Python and Qt: The Definitive Guide to PyQt Programming". Mark założył firmę Qtrac Ltd., http://www.qtrac.eu, w której pracuje jako niezależny publicysta, redaktor, trener i konsultant specjalizujący się w C++, Qt, Pythonie i PyQt.

Spis treści

O autorze **13**

Wprowadzenie **15**

Rozdział 1. Szybkie wprowadzenie do programowania proceduralnego **23**

Tworzenie i uruchamianie programów Pythona 24
„Piękne serce" Pythona 29
Koncepcja 1. — rodzaje danych 29
Koncepcja 2. — odniesienia do obiektów 31
Koncepcja 3. — kolekcje rodzajów danych 33
Koncepcja 4. — operatory logiczne 36
Koncepcja 5. — polecenia kontroli przepływu programu 40
Koncepcja 6. — operatory arytmetyczne 45
Koncepcja 7. — operacje wejścia-wyjścia 48
Koncepcja 8. — tworzenie i wywoływanie funkcji 51
Przykłady 53
bigdigits.py 53
generate_grid.py 56
Podsumowanie 58
Ćwiczenia 61

Rozdział 2. Rodzaje danych **65**

Identyfikatory i słowa kluczowe 65
Całkowite rodzaje danych 69
Liczby całkowite 69
Wartości boolowskie 72
Zmiennoprzecinkowe rodzaje danych 73
Liczby zmiennoprzecinkowe (Float) 74
Liczby zespolone (Complex) 77
Liczby Decimal 78

Ciągi tekstowe 80
Porównywanie ciągów tekstowych 83
Segmentowanie i poruszanie się krokami w ciągu tekstowym 84
Operatory i metody dotyczące ciągu tekstowego 87
Formatowanie ciągu tekstowego za pomocą metody str.format() 95
Kodowanie znaków 107
Przykłady 111
quadratic.py 111
csv2html.py 114
Podsumowanie 118
Ćwiczenia 120

Rozdział 3. Kolekcje rodzajów danych 123
Rodzaje sekwencji 124
Krotki 124
Nazwane krotki 127
Listy 129
Rodzaje danych set 137
Set (zbiór) 138
Rodzaj danych frozenset 142
Rodzaje mapowania 143
Słowniki 143
Słowniki domyślne 152
Słowniki uporządkowane 153
Iteracja i kopiowanie kolekcji 155
Iteratory i operacje oraz funkcje iteracji 155
Kopiowanie kolekcji 164
Przykłady 166
generate_usernames.py 166
statistics.py 169
Podsumowanie 173
Ćwiczenia 175

Rozdział 4. Funkcje i struktury kontrolne 177
Struktury kontrolne 177
Konstrukcje rozgałęziające 178
Pętle 179
Obsługa wyjątków 181
Przechwytywanie i obsługa wyjątków 181
Własne wyjątki 186

Własne funkcje 189
Nazwy i dokumentujące ciągi tekstowe 193
Rozpakowywanie argumentu i parametru 195
Uzyskiwanie dostępu do zmiennych w zasięgu globalnym 197
Funkcja lambda 199
Asercje 201
Przykład: make_html_skeleton.py 202
Podsumowanie 208
Ćwiczenie 209

Rozdział 5. Moduły 213
Moduły i pakiety 214
Pakiety 217
Własne moduły 220
Ogólny opis biblioteki standardowej Pythona 230
Obsługa ciągów tekstowych 230
Programowanie wiersza polecenia 232
Matematyka i liczby 233
Data i godzina 234
Algorytmy i kolekcje rodzajów danych 235
Formaty plików, kodowania znaków i przechowywanie danych 236
Plik, katalog i obsługa przetwarzania 240
Praca w sieci i programowanie internetowe 242
XML 244
Inne moduły 246
Podsumowanie 247
Ćwiczenie 249

Rozdział 6. Programowanie zorientowane obiektowo 251
Podejście zorientowane obiektowo 252
Koncepcje i terminologia programowania zorientowanego obiektowo 253
Własne klasy 256
Atrybuty i metody 257
Dziedziczenie i polimorfizm 262
Używanie właściwości w celu kontrolowania dostępu do atrybutów 264
Tworzenie w pełni zintegrowanych rodzajów danych 266
Własne klasy kolekcji 279
Tworzenie klas agregujących kolekcje 279
Tworzenie klas kolekcji za pomocą agregacji 286
Tworzenie klas kolekcji za pomocą dziedziczenia 292
Podsumowanie 299
Ćwiczenia 301

Rozdział 7. Obsługa plików 303
Zapis i odczyt danych binarnych 308
Peklowanie wraz z opcjonalną konwersją 308
Zwykłe dane binarne wraz z opcjonalną kompresją 312
Zapis i przetwarzanie plików tekstowych 321
Zapis tekstu 321
Przetwarzanie tekstu 322
Przetwarzanie tekstu za pomocą wyrażeń regularnych 325
Zapis i przetwarzanie plików XML 328
Drzewa elementów 329
Model DOM (Document Object Model) 332
Ręczny zapis XML 335
Przetwarzanie XML za pomocą SAX (Simple API dla XML) 336
Swobodny dostęp do plików binarnych 339
Ogólna klasa BinaryRecordFile 339
Przykład: klasy modułu BikeStock 347
Podsumowanie 351
Ćwiczenia 352

Rozdział 8. Zaawansowane techniki programowania 355
Dalsze techniki programowania proceduralnego 356
Rozgałęzianie za pomocą słowników 356
Funkcje i wyrażenia generatora 358
Dynamiczne wykonywanie kodu oraz dynamiczne polecenia import 360
Funkcje lokalne i rekurencyjne 368
Dekoratory funkcji i metod 372
Adnotacje funkcji 376
Dalsze techniki programowania zorientowanego obiektowo 378
Kontrola dostępu do atrybutów 379
Funktor 382
Menedżery kontekstu 384
Deskryptory 388
Dekoratory klas 392
Abstrakcyjne klasy bazowe 395
Dziedziczenie wielokrotne 402
Metaklasy 404
Programowanie funkcjonalne 408
Funkcje częściowe aplikacji 411
Współprogramy 412
Przykład: valid.py 421
Podsumowanie 423
Ćwiczenia 424

Rozdział 9. Usuwanie błędów, testowanie i profilowanie 427
Usuwanie błędów 428
Obsługa błędów składni 429
Obsługa błędów w trakcie działania programu 430
Naukowy sposób usuwania błędów 434
Testy jednostkowe 440
Profilowanie 446
Podsumowanie 451

Rozdział 10. Procesy i wątkowanie 453
Używanie modułu Multiprocessing 454
Używanie modułu Threading 458
Przykład: program wyszukiwania używający wątków 460
Przykład: program wyszukujący powielone pliki używający wątkowania 463
Podsumowanie 468
Ćwiczenia 469

Rozdział 11. Praca w sieci 471
Tworzenie klienta TCP 473
Tworzenie serwera TCP 478
Podsumowanie 485
Ćwiczenia 485

Rozdział 12. Programowanie bazy danych 489
Bazy danych DBM 490
Bazy danych SQL 494
Podsumowanie 501
Ćwiczenie 502

Rozdział 13. Wyrażenia regularne 503
Język wyrażeń regularnych Pythona 504
Znaki i klasy znaków 505
Kwantyfikatory 506
Grupowanie i przechwytywanie 508
Asercje i opcje 511
Moduł wyrażeń regularnych 515
Podsumowanie 526
Ćwiczenia 526

Rozdział 14. Wprowadzenie do analizy składniowej 529
Składnia BNF i terminologia związana z analizą składniową 531
Tworzenie własnych analizatorów składni 535
Prosta analiza składniowa danych klucz – wartość 536
Analiza składniowa listy odtwarzania 539
Analiza składniowa bloków języka specjalizowanego 541

Analiza składniowa za pomocą modułu PyParsing 550
Krótkie wprowadzenie do modułu PyParsing 551
Prosta analiza składniowa danych klucz – wartość 555
Analiza składniowa danych listy odtwarzania 557
Analiza składniowa bloków języka specjalizowanego 559
Analiza składni logiki pierwszego rzędu 564
Analiza składniowa Lex/Yacc za pomocą modułu PLY 569
Prosta analiza składniowa danych klucz – wartość 571
Analiza składniowa danych listy odtwarzania 573
Analiza składniowa bloków języka specjalizowanego 575
Analizator składni logiki pierwszego rzędu 577
Podsumowanie 582
Ćwiczenie 583

Rozdział 15. Wprowadzenie do programowania GUI 585
Programy w stylu okna dialogowego 588
Programy w stylu okna głównego 594
Tworzenie okna głównego 595
Tworzenie własnego okna dialogowego 605
Podsumowanie 608
Ćwiczenia 609

Epilog 611

Wybrana bibliografia 613

Skorowidz 615

1

Szybkie wprowadzenie do programowania proceduralnego

W rozdziale:

- Tworzenie i uruchamianie programów Pythona
- „Piękne serce" Pythona

W tym rozdziale zostanie omówiona taka ilość materiału, że jej lektura pozwoli czytelnikowi na rozpoczęcie tworzenia programów w języku Python. Jeżeli czytelnik jeszcze nie zainstalował Pythona, gorąco zachęcamy do przeprowadzenia instalacji. Pozwoli to na zdobywanie doświadczenia i utrwalanie poznawanego materiału. (Instalacja Pythona na głównych platformach sprzętowych została omówiona we wcześniejszym rozdziale „Wprowadzenie", ◄ 19).

Pierwszy podrozdział zaprezentuje sposób tworzenia i uruchamiania programów Pythona. Kod Pythona można tworzyć w dowolnym edytorze tekstowym. Omówione w rozdziale środowisko programistyczne IDLE oferuje nie tylko sam edytor kodu źródłowego, ale także funkcje dodatkowe pozwalające na między innymi eksperymentowanie z kodem Pythona oraz usuwanie błędów z kodu.

Drugi podrozdział został poświęcony ośmiu kluczowym koncepcjom Pythona, których poznanie wystarczy do tworzenia użytecznych programów. Wszystkie wymienione tutaj konstrukcje będą dokładnie omówione w dalszych rozdziałach książki. W miarę przedstawiania kolejnego materiału koncepcje te zostaną uzupełnione informacjami obejmującymi szerszy zakres języka Python. W ten sposób po przeczytaniu książki czytelnik będzie znał cały język i podczas pracy nad własnymi programami będzie w stanie wykorzystywać wszystkie możliwości języka.

Ostatni podrozdział zawiera dwa krótkie programy, w których użyto niektórych funkcji Pythona omówionych w podrozdziale drugim. Dzięki temu czytelnik natychmiast będzie mógł „wypróbować" Pythona.

Tworzenie i uruchamianie programów Pythona

Kod Pythona można tworzyć w dowolnym edytorze tekstowym, który umożliwia zapis i odczyt plików tekstowych stosujących kodowanie znaków ASCII lub UTF-8 Unicode. Domyślnie pliki Pythona stosują kodowanie znaków UTF-8 obejmujące większy zbiór znaków niż ASCII. Za pomocą kodowania UTF-8 można przedstawiać każdy znak w każdym języku. Pliki Pythona zwykle mają rozszerzenie *.py*, chociaż w niektórych systemach z rodziny Unix (na przykład Linux i Mac OS X) pewne aplikacje Pythona nie mają rozszerzenia pliku. Programy Pythona z graficznym interfejsem użytkownika (GUI, czyli *Graphical User Interface*) zazwyczaj mają rozszerzenie *.pyw*, zwłaszcza w systemach Windows oraz Mac OS X. W niniejszej książce moduły Pythona oraz programy Pythona działające w konsoli zawsze mają rozszerzenie *.py*, natomiast programy GUI są oznaczone rozszerzeniem *.pyw*. Wszystkie przykłady zaprezentowane w książce można w niezmienionej postaci uruchomić na każdej platformie sprzętowej z zainstalowanym Pythonem 3.

Kodowanie znaków ➤ 107.

Aby upewnić się, że wszystko zostało skonfigurowane prawidłowo, oraz przedstawić klasyczny pierwszy przykład, należy w edytorze tekstowym (w systemie Windows można użyć programu Notatnik — wkrótce zaczniemy korzystać z lepszego edytora) utworzyć plik o nazwie *hello.py* zawierający następujące wiersze:

```
#!/usr/bin/env python3
print("Witaj", "Świecie!")
```

Pierwszy wiersz to komentarz. W Pythonie komentarz rozpoczyna się od znaku # i ciągnie się aż do końca wiersza. (Znaczenie powyższego komentarza wyjaśnimy za chwilę). Drugi wiersz jest pusty — poza ciągami tekstowymi ujętymi w cudzysłów, Python ignoruje puste wiersze. Jednak mają one istotne znaczenie dla programistów, gdyż oddzielają duże bloki kodu, co ułatwia ich odczyt. Wiersz trzeci zawiera kod Pythona. W tym wierszu następuje wywołanie funkcji `printf()` wraz z dwoma argumentami, z których każdy jest typu `str` (`string` — ciąg tekstowy, czyli sekwencja znaków).

Każde polecenie napotkane w pliku *.py* zostaje kolejno wykonane, począwszy od pierwszego i dalej wiersz po wierszu. To zupełnie odmienne podejście niż w innych językach, takich jak C++ bądź Java, gdzie mamy określoną funkcję lub metodę posiadającą nazwę specjalną, która stanowi punkt początkowy programu. Oczywiście, istnieje możliwość sterowania przepływem programu, co zostanie zaprezentowane w kolejnym podrozdziale podczas omawiania struktur kontrolnych Pythona.

Przyjmujemy założenie, że użytkownik systemu Windows zapisuje kod Pythona w katalogu *C:\py3eg*, natomiast użytkownik systemu z rodziny Unix (na przykład Unix, Linux lub Mac OS X) przechowuje kod w katalogu *$HOME/py3eg*. Plik *hello.py* należy więc zapisać w katalogu *py3eg* i zamknąć edytor tekstowy.

Mamy więc program, który można uruchomić. Programy w języku Python są wykonywane przez interpreter Pythona, zazwyczaj w oknie konsoli. W systemie Windows konsola nosi nazwę „konsola", „wiersz polecenia DOS", „wiersz polecenia MS-DOS" lub podobnie i zwykle znajduje się w menu *Start/Wszystkie programy/Akcesoria*. W systemie Mac OS X konsola jest dostarczana przez narzędzie *Terminal.app* (domyślnie znajduje się w katalogu *Programy/Narzędzia*) dostępne poprzez Findera. W pozostałych systemach Unix można użyć `xterm` bądź konsoli oferowanej przez menedżera okien, na przykład `konsole` bądź `gnome-terminal`.

Uruchom konsolę, następnie w systemie Windows wprowadź poniższe polecenia (przyjmujemy założenie, że Python został zainstalowany w lokalizacji domyślnej). Dane wyjściowe konsoli są przedstawione `czcionką o stałej szerokości`, natomiast wprowadzane polecenia zostały **pogrubione**:

```
C:\>cd c:\py3eg
C:\py3eg\>c:\python31\python.exe hello.py
```

Ponieważ polecenie `cd` (*change directory*, czyli zmień katalog) używa bezwzględnej ścieżki dostępu, katalog, z poziomu którego zostanie uruchomiony program, nie ma żadnego znaczenia.

Użytkownicy systemu Unix wydają polecenia pokazane poniżej (przyjmujemy założenie, że odniesienie do Pythona 3 znajduje się w systemowej ścieżce `PATH`)[1]:

```
$ cd $HOME/py3eg
$ python3 hello.py
```

W obu przypadkach dane wyjściowe prezentują się tak samo:

```
Witaj świecie!
```

Warto pamiętać, że jeśli nie napisano inaczej, zachowanie języka Python w systemie Mac OS X jest takie samo, jak w każdym innym systemie z rodziny Unix. Możemy więc przyjąć, że kiedy używamy określenia „Unix", oznacza to system Linux, BSD, Mac OS X oraz większość pozostałych systemów Unix i wywodzących się z rodziny Unix.

Chociaż pierwszy program składa się tylko z jednego polecenia wykonywalnego, po jego uruchomieniu możemy wyciągnąć pewne wnioski dotyczące funkcji `print()`. Przede wszystkim funkcja `print()` jest elementem wbudowanym w język Python — w celu jej użycia nie musimy jej „importować" ani „dołączać" z biblioteki. Ponadto każdy element wyświetlany przez funkcję jest oddzielony pojedynczą spacją, a po wyświetleniu ostatniego wstawiany jest znak nowego wiersza. To jest zachowanie domyślne funkcji i jak przekonamy się nieco później, możemy je zmienić. Inną ważną informacją dotyczącą funkcji `print()` jest to, że może pobierać dowolną ilość argumentów.

Funkcja `print()` ➤ 198

Wprowadzanie przedstawionych powyżej poleceń w celu uruchamiania tworzonych programów w Pythonie bardzo szybko może okazać się uciążliwe. Na szczęście w systemach zarówno Windows, jak i Unix można zastosować znacznie wygodniejsze podejście.

[1] Znak zachęty systemu Unix może być odmienny niż przedstawiony w powyższym przykładzie znak dolara ($), to nie ma żadnego znaczenia.

Zakładamy, że znajdujemy się w katalogu *py3eg*. W systemie Windows można więc po prostu wydać polecenie:

```
C:\py3eg\>hello.py
```

Windows używa rejestru przypisań plików w celu automatycznego wywołania interpretera Pythona za każdym razem, gdy w konsoli zostanie wprowadzona nazwa pliku z rozszerzeniem *.py*.

Niestety, takie wygodne podejście nie zawsze działa, ponieważ pewne wersje systemu Windows mają błąd, który czasami negatywnie wpływa na wykonywanie programów interpretowanych wywoływanych jako wynik odczytania zdefiniowanego przypisania pliku. To nie jest błąd w Pythonie — inne interpretery, a także niektóre pliki *.bat* także padają ofiarą tego błędu. Jeżeli czytelnik spotka się z tego rodzaju błędem, wówczas należy bezpośrednio wywołać Pythona, zamiast polegać na mechanizmie rozpoznawania przypisań plików.

Jeśli w systemie Windows dane wyjściowe mają postać:

```
('Witaj', 'Świecie!')
```

oznacza to, że w systemie znajduje się Python 2 i został wywołany zamiast Pythona 3. Jednym z rozwiązań będzie zmiana przypisania pliku *.py* z Pythona 2 na Python 3. Inne rozwiązanie (mniej wygodne, lecz bezpieczniejsze) do dodanie Pythona 3 do systemowej ścieżki dostępu (przyjmujemy założenie, że został zainstalowany w lokalizacji domyślnej) i wyraźne wywoływanie Pythona za każdym razem. (Ten sposób powoduje także rozwiązanie wspomnianego wcześniej błędu w Windows, dotyczącego przypisań plików). Przykładowo:

```
C:\py3eg\>path=c:\python31;%path%
C:\py3eg\>python hello.py
```

Znacznie wygodniejszym rozwiązaniem może być utworzenie pliku *py3.bat* zawierającego pojedynczy wiersz kodu `path=c:\python31;%path%` i zapisanie tego pliku w katalogu *C:\Windows*. Następnie, po każdym włączeniu konsoli w celu uruchamiania programów napisanych w Pythonie, pracę trzeba rozpocząć od wykonania pliku *py3.bat*. Ewentualnie można skonfigurować system tak, aby plik *py3.bat* był wykonywany automatycznie. W tym celu należy zmodyfikować właściwości konsoli (odszukaj konsolę w menu *Start*, a następnie kliknij konsolę prawym przyciskiem myszy i wybierz opcję *Właściwości*). W wyświetlonym oknie dialogowym przejdź na kartę *Skrót*, w polu *Element docelowy* dodaj ciąg tekstowy „ `/u /k c:\windows\py3.bat`” (zwróć uwagę na spację przed, między i po opcjach „/u” oraz „/k” i upewnij się, że ciąg tekstowy znajduje się po poleceniu `cmd.exe`).

W systemie Unix plikowi trzeba w pierwszej kolejności nadać uprawnienia do wykonywania, a dopiero potem można go uruchomić:

```
$ chmod +x hello.py
$ ./hello.py
```

Oczywiście polecenie `chmod` należy wykonać tylko jednokrotnie, później wystarczy po prostu wydać polecenie `./hello.py` i program zostanie uruchomiony.

W systemie Unix podczas wywoływania programu w konsoli następuje odczyt dwóch pierwszych bajtów pliku[2]. Jeżeli odczytane bajty są znakami `#!` zbioru znaków ASCII, wówczas powłoka przyjmuje założenie, że plik powinien być wykonany przez interpreter, który zresztą jest wskazany w pierwszym wierszu pliku. Ten wiersz nosi nazwę *shebang* (*shell execute*, czyli plik wykonywalny powłoki) i jeśli zostaje umieszczony w pliku, to musi być pierwszym wierszem pliku.

Wiersz shebang jest zazwyczaj zapisywany w jednej z dwóch możliwych postaci:

```
#!/usr/bin/python3
```

lub

```
#!/usr/bin/env python3
```

Jeżeli będzie zapisany w pierwszej formie, użyty zostanie wymieniony interpreter. Użycie tej formy może okazać się konieczne w przypadku programów Pythona uruchamianych przez serwer WWW, choć sama ścieżka dostępu może być odmienna niż przedstawiona w powyższym przykładzie. Jeśli zostanie użyta druga forma, użyty zostanie pierwszy interpreter `python3` znaleziony w bieżącym środowisku powłoki. Druga forma jest znacznie elastyczniejsza, ponieważ dopuszcza możliwość, że interpreter Python 3 nie został umieszczony w katalogu */usr/bin* (na przykład może być zainstalowany w katalogu */usr/local/bin* bądź w *$HOME*). W systemie Windows wiersz shebang nie jest wymagany (choć równocześnie pozostaje nieszkodliwy). Wszystkie przykłady zaprezentowane w książce zawierają wiersz shebang w drugiej formie, choć nie umieszczono go w listingach.

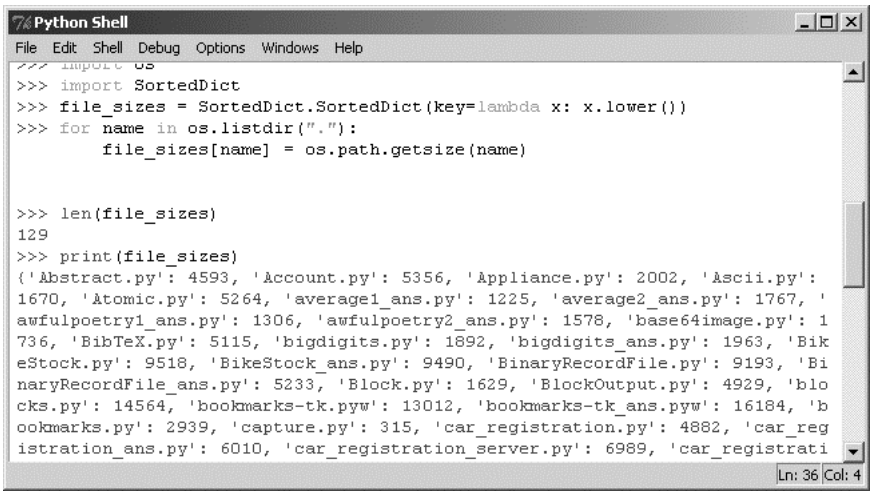
Warto zwrócić uwagę, że dla systemów Unix przyjmujemy założenie, iż nazwa pliku wykonywalnego Pythona 3 (lub dowiązanie symboliczne do niego) w zmiennej systemowej `PATH` to `python3`. Jeżeli tak nie jest, w przykładach należy zmodyfikować wiersz shebang i podać odpowiednią nazwę (prawidłową nazwę i ścieżkę dostępu w przypadku używania pierwszej formy wiersza shebang). Ewentualnie trzeba utworzyć dowiązanie symboliczne z pliku wykonywalnego Python 3 do nazwy `python3` w zmiennej `PATH`.

Pobieranie i instalacja Pythona 19 ➤

Wiele edytorów tekstowych o potężnych możliwościach, na przykład Vim i Ecmacs, jest dostarczanych wraz z wbudowaną obsługą edycji programów w języku Python. Wspominania obsługa zazwyczaj obejmuje kolorowanie składni oraz prawidłowe stosowanie wcięć wierszy. Alternatywnym rozwiązaniem jest używanie środowiska programistycznego dla Pythona, którym jest IDLE. W systemach Windows i Mac OS X środowisko IDLE jest instalowane domyślnie. Z kolei w systemie Unix środowisko IDLE zostaje zbudowane wraz z interpreterem Pythona, o ile jest on budowany z archiwum tarball. W przypadku użycia menedżera pakietów środowisko IDLE zwykle jest dostarczane jako oddzielny pakiet, jak to przedstawiono we wcześniejszym rozdziale „Wprowadzenie".

Na rysunku 1.1 pokazano środowisko IDLE. Jak widać, IDLE charakteryzuje się przestarzałym interfejsem graficznym, który przypomina czasy motywu Motif w systemach Unix bądź systemu Windows 95. Wynika to z faktu używania biblioteki GUI Tkinter

[2] Interakcja między użytkownikiem i konsolą jest obsługiwana przez program „powłoka". Rozróżnienie między programami konsoli i powłoki nie ma tutaj najmniejszego znaczenia, więc oba pojęcia są tu używane zamiennie.



Rysunek 1.1. Konsola Pythona w środowisku IDLE

bazującej na Tk (omówiona w rozdziale 15.), a nie jednej z nowoczesnych i oferujących potężniejsze możliwości bibliotek takich jak PyGtk, PyQt lub wxPython. Powód stosowania biblioteki Tkinter to połączenie historii, liberalnych warunków licencji oraz faktu, że biblioteka Tkinter jest znacznie mniejsza niż inne biblioteki GUI. Na plus można zaliczyć to, że środowisko IDLE jest dostarczane domyślnie wraz z Pythonem i bardzo łatwo je poznać oraz go używać.

Środowisko IDLE zapewnia trzy istotne funkcje: możliwość wprowadzania wyrażeń języka Python oraz kodu źródłowego i obserwowania wyników bezpośrednio w powłoce Pythona, edytor kodu oferujący kolorowanie składni kodu Pythona oraz prawidłowe stosowanie wcięć wierszy, a także moduł usuwania błędów pozwalający na przejście przez kod wiersz po wierszu, znajdywanie i usuwanie błędów. Powłoka Pythona jest szczególnie użyteczna w trakcie wypróbowywania przykładowych algorytmów, fragmentów kodu oraz wyrażeń regularnych. Może być także używana jako oferujący potężne możliwości i bardzo elastyczny kalkulator.

Dostępnych jest kilka innych środowisk programistycznych Pythona, ale zalecamy stosowanie środowiska IDLE, przynajmniej na początku. Alternatywą jest tworzenie programów w zwykłym edytorze tekstowym, a następnie usuwanie błędów poprzez stosowanie wywołań funkcji `print()`.

Możliwe jest wywołanie interpretera Pythona bez wskazywania programu w języku Python. W takim przypadku interpreter zostaje uruchomiony w trybie interaktywnym. Podczas pracy w tym trybie można wprowadzać polecenia Pythona i obserwować wyniki dokładnie w taki sam sposób, jak w trakcie używania okna powłoki Pythona środowiska IDLE — wraz z tymi samymi znakami zachęty w postaci >>>. Jednak środowisko IDLE pozostaje znacznie łatwiejsze w użyciu, więc zalecamy stosowanie IDLE podczas eksperymentowania

z fragmentami kodu. Gdy prezentujemy krótkie przykłady interaktywne, przyjmujemy wówczas założenie, że są one wprowadzane w interaktywnym interpreterze Python bądź oknie powłoki Pythona środowiska IDLE.

Wiemy już, w jaki sposób tworzyć i uruchamiać programy Pythona, ale przecież czytelnicy nie znają jeszcze języka — posiadają jedynie ogólną wiedzę o pojedynczej funkcji. Dzięki lekturze kolejnego rozdziału wyraźnie zwiększy się ich wiedza o języku Python. Przedstawione tu informacje pozwolą na tworzenie krótkich, choć użytecznych programów w Pythonie, co zaprezentujemy w ostatnim podrozdziale niniejszego rozdziału.

„Piękne serce” Pythona

W tym podrozdziale zostanie omówionych osiem kluczowych koncepcji języka Python, natomiast w kolejnym podrozdziale przedstawimy użycie tych koncepcji podczas tworzenia kilku małych, a jednak użytecznych programów. Na temat zagadnień poruszonych w tym podrozdziale można powiedzieć znacznie więcej. Dlatego też jeśli czytelnik poczuje, że czegoś brakuje w Pythonie bądź że pewne zadania są wykonywane w zbyt długi sposób, wtedy może przejść do znajdującego się w dalszej części książki materiału dotyczącego danego zagadnienia. Wystarczy kierować się podanym w tekście odniesieniem lub skorzystać ze spisu treści bądź skorowidza. W ten sposób łatwo przekonać się, że Python jednak ma funkcję poszukiwaną przez czytelnika. Ponadto bardzo często okaże się, że dana funkcja będzie miała nie tylko znacznie czytelniejsze formy wyrażenia niż zastosowane tutaj, ale także i większe możliwości.

Koncepcja 1. — typy danych

Podstawową funkcją każdego języka programowania jest możliwość przedstawienia danych. Python oferuje kilka wbudowanych typów danych, ale w chwili obecnej skoncentrujemy się jedynie na dwóch. Liczby całkowite (dodatnie i ujemne) są przedstawiane w Pythonie za pomocą typu `int`, natomiast ciągi tekstowe (sekwencje znaków Unicode) — za pomocą typu `str`. Poniżej przedstawiono kilka przykładów liczb całkowitych oraz dosłownych ciągów tekstowych:

```
-973
21062458333711437339583605536734086463779019080109822250862195507 2
0
"Nieskończenie wymagający"
'Szymon Kowalski'
'pozytywne    €÷©'
''
```

Nawiasem mówiąc, druga liczba to 2^{217} — wielkość liczb całkowitych w Pythonie jest ograniczona jedynie przez ilość pamięci zainstalowanej w komputerze, a nie przez odgórnie ustaloną liczbę bajtów. Ciągi tekstowe mogą być ujęte w cudzysłów bądź apostrofy, o ile na obu końcach ciągu tekstowego znajduje się ten sam ogranicznik (znak cudzysłowu lub

apostrof). Ponieważ Python używa kodowania znaków Unicode, ciągi tekstowe nie są ograniczone jedynie do zbioru znaków ASCII, co widać wyraźnie w przedostatnim ciągu tekstowym. Pusty ciąg tekstowy składa się po prostu z ograniczników ciągu tekstowego.

W celu uzyskania dostępu do elementu w sekwencji, na przykład w ciągu tekstowym, w Pythonie używa się nawiasów kwadratowych ([]). Przykładowo: pracując w powłoce Pythona (albo w interpreterze interaktywnym lub środowisku IDLE), możemy wprowadzić poniższe polecenia. Dane wyjściowe powłoki Pythona są przedstawione `czcionką o stałej szerokości`, natomiast wprowadzane polecenia zostały **pogrubione**:

```
>>> "Trudne Czasy"[7]
'C'
>>> "żyrafa"[0]
'ż'
```

Tradycyjnie powłoka Pythona jako znaku zachęty używa >>>, choć można to zmienić. Składnia wykorzystująca nawiasy kwadratowe może być stosowana z elementami danych będącymi dowolnego typu sekwencjami, czyli na przykład z ciągami tekstowymi bądź listami. Konsekwencja w składni to jeden z powodów, dla których język Python jest tak piękny. Warto zwrócić uwagę, że wszystkie pozycje indeksu w Pythonie rozpoczynają się od zera.

W Pythonie zarówno `str`, jak i podstawowe typy liczbowe, na przykład `int`, są *niemodyfikowalne* — to znaczy, że raz ustalona wartość nie może być zmieniona. W pierwszej chwili wydaje się, że to dziwne ograniczenie, ale składnia Pythona powoduje, że w praktyce nie stanowi ono żadnego problemu. Jedynym powodem, dla którego wspominamy o nim tutaj, jest to, że chociaż składni nawiasów kwadratowych można użyć do pobrania znaku z pozycji o wskazanym indeksie w ciągu tekstowym, to nie można jej wykorzystać do ustawienia nowego znaku. (Warto zwrócić uwagę, że w Pythonie znak jest po prostu ciągiem tekstowym o długości 1).

Aby przekonwertować element danych z jednego typu na inny, można użyć składni w postaci `typ_danych(element)`, na przykład:

```
>>> int("45")
45
>>> str(912)
'912'
```

Typ konwersji `int()` jest tolerancyjny, jeśli chodzi o znaki odstępu umieszczone przed lub po elemencie, tak więc polecenie `int("  45  ")` również działa doskonale. Typ konwersji `str()` może być zastosowany wobec niemal dowolnego elementu danych. Jak się przekonamy w rozdziale 6., bardzo łatwo możemy utworzyć własny typ danych obsługujący konwersję `str()`, jak również konwersję `int()` i każdą inną, o ile ma to sens. Jeżeli konwersja zakończy się niepowodzeniem, wtedy zostanie zgłoszony wyjątek — obsługa błędów będzie pokrótce omówiona w koncepcji 5. Pełne omówienie wyjątków znajduje się w rozdziale 4.

Ciągi tekstowe i liczby całkowite zostały szczegółowo omówione w rozdziale 2. wraz z innymi wbudowanymi typami danych oraz wybranymi typami danych dostępnymi w bibliotece standardowej Pythona. W rozdziale 2. przedstawiono także operacje, które można wykonać względem niemodyfikowalnych sekwencji, takich jak ciągi tekstowe.

Koncepcja 2. — odniesienia do obiektów

Kiedy posiadamy już pewne typy danych, kolejnym potrzebnym elementem są zmienne pozwalające na przechowywanie danych. Python nie posiada zmiennych jako takich, ale ma *odniesienia do obiektów*. W przypadku niezmiennych obiektów, na przykład typu `str` lub `int`, nie ma żadnej widocznej różnicy między zmienną i odniesieniem do obiektu. Natomiast w przypadku obiektów zmiennych taka różnica występuje, choć w praktyce rzadko ma znaczenie. Pojęć *zmienna* i *odniesienie do obiektu* będziemy więc używać zamiennie.

Kopiowanie płytkie i głębokie ➤ 164

Spójrzmy na kilka prostych przykładów, które następnie będą dokładnie omówione:

```
x = "niebieski"
y = "zielony"
z = x
```

Składnia to po prostu *`odniesienie_do_obiektu = wartość`*. Nie ma konieczności podawania wcześniej jakichkolwiek deklaracji bądź określania rodzaju wartości. Kiedy Python wykonuje pierwsze polecenie, tworzy obiekt `str` wraz z tekstem „niebieski" oraz tworzy odniesienie do obiektu, nazwane `x`, które po prostu odnosi się do obiektu `str`. Ze względów praktycznych możemy powiedzieć: „zmiennej `x` został przypisany ciąg tekstowy `'niebieski'`". Drugie polecenie jest podobne. Trzecie polecenie tworzy nowe odniesienie do obiektu, nazwane `z`, i określa, że będzie odnosiło się do tego samego obiektu, do którego odnosi się odniesienie `x`. W omawianym przykładzie będzie to obiekt `str` zawierający tekst „niebieski".

Operator `=` nie ma takiej samej funkcji jak spotykany w niektórych językach operator przypisania funkcji. Operator `=` łączy w pamięci odniesienie do obiektu wraz z obiektem. Jeżeli odniesienie do obiektu już istnieje, wtedy po prostu nastąpi ponowne dołączenie w celu utworzenia odniesienia do obiektu znajdującego się po prawej stronie operatora `=`. Natomiast jeśli dane odniesienie jeszcze nie istnieje, zostanie utworzone przez operator `=`.

Kontynuujemy pracę z przykładem `x`, `y`, `z` i dokonujemy ponownych dołączeń. Jak już wcześniej wspomniano, komentarz rozpoczyna się znakiem `#` i trwa aż do końca wiersza:

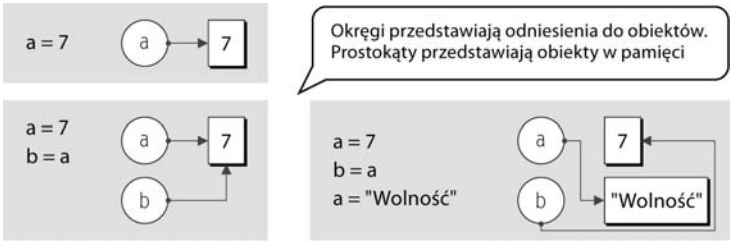
```
print(x, y, z) # dane wyjściowe: niebieski zielony niebieski
z = y
print(x, y, z) # dane wyjściowe: niebieski zielony zielony
x = z
print(x, y, z) # dane wyjściowe: zielony zielony zielony
```

Po czwartym poleceniu (`x = z`) wszystkie trzy odniesienia do obiektów odwołują się do tego samego obiektu `str`. Ponieważ nie ma więcej odniesień do ciągu tekstowego „niebieski", Python może go usunąć (użyć mechanizmu *garbage collection*, czyli zbierania nieużytków).

Na rysunku 1.2 pokazano schematycznie związki zachodzące między obiektami i odniesieniami do obiektów.

Nazwy używane dla odniesień do obiektów (nazywane *identyfikatorami*) mają kilka ograniczeń. W szczególności nie mogą być takie same jak słowa kluczowe języka Python oraz muszą rozpoczynać się literą bądź znakiem podkreślenia, po którym znajduje się zero lub więcej znaków innych niż znaki odstępu (dozwolone się litery, znaki podkreślenia lub cyfry). Nie ma ograniczenia w długości nazwy, a litery i cyfry to te, które są zdefiniowane

Identyfikatory i słowa kluczowe ➤ 65



Rysunek 1.2. Odniesienia do obiektów oraz obiekty

przez Unicode. Zawierają więc zestaw ASCII, choć nie muszą być ograniczone jedynie do liter i cyfr ASCII („a”, „b”, …, „z”, „A”, „B”, …, „Z”, „0”, „1”, …, „9”). Identyfikatory Pythona rozróżniają wielkość liter, więc `LIMIT`, `Limit` i `limit` to trzy różne identyfikatory. Szczegółowe omówienie identyfikatorów oraz pewne nieco egzotyczne przykłady zostały przedstawione w rozdziale 2.

Python stosuje *dynamiczną kontrolę typu*, co oznacza, że w dowolnej chwili odniesienie do obiektu może być dołączone do innego obiektu, który z kolei może być obiektem innego typu danych. Języki stosujące ścisłą kontrolę typu (na przykład C++ i Java) pozwalają na wykonanie jedynie tych operacji, które zostały zdefiniowane w używanym typie danych. Python także stosuje wymienione ograniczenie, ale w przypadku Pythona nie nazywamy tego ścisłą kontrolą typu, ponieważ dopuszczalne operacje mogą ulegać zmianie — na przykład jeśli odniesienie do obiektu zostanie ponownie utworzone i dołączone do obiektu innego typu danych. Przykładowo:

```
droga = 866
print(droga, type(droga)) # dane wyjściowe: 866 <class 'int'>
droga = "Północ"
print(droga, type(droga)) # dane wyjściowe: Północ <class 'str'>
```

W powyższym przykładzie tworzymy nowe odniesienie do obiektu nazwane `droga` i przypisujemy mu nową wartość 866 typu `int`. Na tym etapie względem odniesienia `droga` można użyć operatora /, ponieważ dzielenie jest dopuszczalną i poprawną operacją przeprowadzaną na liczbach całkowitych. Następnie ponownie używamy odniesienia `droga` w celu utworzenia odniesienia do nowej wartości „Północ” typu `str`. Obiekt `int` zostaje skierowany do mechanizmu zbierania nieużytków, ponieważ w chwili obecnej nie ma do niego żadnego odniesienia. Na tym etapie użycie operatora / względem odniesienia `droga` spowoduje zgłoszenie wyjątku `TypeError`, ponieważ dzielenie (/) nie jest operacją dopuszczalną do przeprowadzenia względem ciągu tekstowego.

Wartością zwrotną funkcji `type()` jest typ danych (znany także jako „klasa”) wskazanego elementu danych. Funkcja ta może być więc bardzo użyteczna podczas testowania i usuwania błędów w kodzie, ale zazwyczaj nie będzie stosowana w kodzie produkcyjnym — jak się okaże w rozdziale 6., są tutaj inne, lepsze możliwości.

Funkcja `insin ↪stance()` ➤ 260

Jeżeli czytelnik eksperymentuje z kodem Pythona w interpreterze interaktywnym lub powłoce Pythona, na przykład oferowanej przez środowisko IDLE, wystarczy po prostu podać nazwę odniesienia do obiektu, a Python wyświetli jego wartość. Przykładowo:

```
>>> x = "niebieski"
>>> y = "zielony"
>>> z = x
>>> x
'niebieski'
>>> x, y, z
('niebieski', 'zielony', 'niebieski')
```

To znacznie wygodniejsze rozwiązanie niż ciągłe wywoływanie funkcji `print()`, ale działa jedynie wtedy, gdy Python funkcjonuje w trybie interaktywnym. W celu wyświetlenia danych wyjściowych wszystkie tworzone programy i moduły nadal muszą stosować funkcję `print()` lub podobną. Warto zwrócić uwagę, że ostatnie dane wyjściowe zostały wyświetlone w nawiasach i rozdzielone przecinkami. Oznacza to typ danych `tuple`, czyli krotkę — uporządkowaną, niezmienną sekwencję obiektów. Krotki zostaną przedstawione w kolejnej koncepcji.

Koncepcja 3. — kolekcje typów danych

Bardzo często wygodnym rozwiązaniem jest przechowywanie całej kolekcji elementów danych. Python oferuje kilka kolekcji typów danych, które mogą przechowywać elementy — są to między innymi tablice asocjacyjne i zbiory. W tym miejscu omówimy jednak tylko dwa typy: `tuple` (krotka) i `list` (lista). Krotki i listy Pythona mogą być stosowane w celu przechowywania dowolnej liczby elementów danych dowolnego typu danych. Krotki pozostają *niezmienne*, więc po ich utworzeniu nie ma możliwości wprowadzania zmian. Natomiast listy są *zmienne*, można więc łatwo wstawiać i usuwać elementy listy według własnego uznania.

Jak widać na poniższych przykładach, krotki są tworzone za pomocą przecinków (`,`). Warto zwrócić uwagę, że poniżej i ogólnie od tej chwili wprowadzane przez użytkownika polecenia nie będą przedstawiane pogrubioną czcionką:

```
>>> "Dania", "Finlandia", "Norwegia", "Szwecja"
('Dania', 'Finlandia', 'Norwegia', 'Szwecja')
>>> "jeden"
('jeden',)
```

Kiedy Python wyświetla krotkę, ujmuje ją w nawias. Wielu programistów naśladuje to rozwiązanie i zawsze ujmują dosłowne krotki w nawiasy. W przypadku krotki jednoelementowej i chęci użycia nawiasu nadal trzeba zastosować przecinek, na przykład `(1,)`. Pusta krotka jest tworzona za pomocą pustego nawiasu `()`. Przecinek jest wykorzystywany także do rozdzielania argumentów w wywołaniach funkcji. Dlatego też, jeżeli dosłowna krotka ma zostać przekazana jako argument, musi być ujęta w nawias, aby uniknąć zamieszania.

Typ danych `tuple` ➤ 124

Tworzenie i wywoływanie funkcji ➤ 51

Poniżej przedstawiono kilka przykładów list:

```
[1, 4, 9, 16, 25, 36, 49]
['alpha', 'bravo', 'charlie', 'delta', 'echo']
['zebra', 49, -879, 'mrównik', 200]
[]
```

Jednym ze sposobów utworzenia listy jest użycie nawiasów kwadratowych ([]) — jak pokazano na powyższym przykładzie. W dalszej części książki zostaną przedstawione inne sposoby. Wiersz czwarty pokazuje pustą listę.

Typ danych list ➤ 129

Wewnętrznie listy i krotki w ogóle nie przechowują elementów danych, a raczej odniesienia do obiektów. W trakcie tworzenia listy bądź krotki (oraz podczas wstawiania elementów w przypadku listy) kopiują one odniesienia do podanych obiektów. W przypadku dosłownych elementów, na przykład liczb całkowitych i ciągów tekstowych, w pamięci tworzony jest obiekt odpowiedniego typu danych i później jest on inicjalizowany. Następnie tworzone jest odniesienie do tego obiektu i to utworzone odniesienie zostaje umieszczone w krotce bądź na liście.

Podobnie jak w przypadku pozostałych konstrukcji Pythona, również kolekcje typów danych są obiektami. Istnieje więc możliwość zagnieżdżania jednej kolekcji typu danych w innej, na przykład w celu utworzenia listy zawierającej inne listy. W pewnych sytuacjach fakt, że listy, krotki i większość pozostałych kolekcji typów danych Pythona przechowuje odniesienia do obiektów zamiast obiektów, powoduje różnicę — zostanie to omówione w rozdziale 3.

Kopiowanie płytkie i głębokie ➤ 164

W programowaniu proceduralnym wywołujemy funkcje i często przekazujemy elementy danych jako argumenty. Przykładowo widzieliśmy już w działaniu funkcję `print()`. Inna często używana funkcja Pythona to `len()`, która jako argument pobiera pojedynczy element danych. Wartością zwrotną funkcji jest „długość" elementu podana jako obiekt `int`. Poniżej przedstawiono kilka wywołań funkcji `len()`:

```
>>> len(("jeden",))
1
>>> len([3, 5, 1, 2, "pauza", 5])
6
>>> len("automatycznie")
13
```

Krotki, listy oraz ciągi tekstowe mają „ustaloną wielkość", to znaczy są typami danych, które muszą mieć podaną wielkość. Elementy takiego typu danych mogą być bez problemów przekazane funkcji `len()`. (W przypadku przekazania funkcji `len()` elementu danych bez ustalonej wielkości następuje zgłoszenie wyjątku).

Klasa Sized ➤ 397

Wszystkie elementy danych Pythona są *obiektami* (nazywane są również *egzemplarzami*) określonego typu danych (nazywanego także *klasą*). Pojęć *typ danych* i *klasa* będziemy używać zamiennie. Jedną z istotnych różnic między obiektem i zwykłym elementem danych dostarczanym przez niektóre języki (na przykład C++ lub wbudowane typy liczbowe Javy) pozostaje to, że obiekt może mieć *metody*. Ogólnie rzecz biorąc, metoda to po prostu funkcja, która jest wywoływana dla określonego obiektu. Przykładowo typ `list` ma metodę `append()`, która umożliwia dołączenie obiektu do listy w następujący sposób:

```
>>> x = ['zebra', 49, -879, 'mrównik', 200]
>>> x.append("więcej")
>>> x
['zebra', 49, -879, 'mrównik', 200, 'więcej']
```

Obiekt `x` „wie", że jest typu `list` (w Pythonie wszystkie obiekty znają swój typ danych), tak więc nie ma potrzeby wyraźnego wskazywania typu danych. W implementacji metody `append()` pierwszym argumentem jest sam obiekt `x` — przekazanie tego obiektu jest przeprowadzane automatycznie przez Pythona jako część syntaktycznej obsługi metod.

Metoda `append()` mutuje, czyli zmienia początkową listę. Jest to możliwe, ponieważ listy są elementami zmiennymi. Zwłaszcza w przypadku bardzo długich list jest to także potencjalnie znacznie efektywniejsze rozwiązanie niż tworzenie nowej listy zawierającej początkowe elementy, dodanie nowych elementów, ponowne dołączenie nowej listy do odniesienia do obiektu.

W języku proceduralnym ten sam efekt można uzyskać, stosując metodę `append()` listy, jak przedstawiono poniżej (to zupełnie prawidłowa składnia Pythona):

```
>>> list.append(x, "ekstra")
>>> x
['zebra', 49, -879, 'mrównik', 200, 'więcej', 'ekstra']
```

Tutaj podajemy typ danych oraz metodę typu danych. Ponadto jako pierwszy argument przekazujemy element danych tego typu danych, którego chcemy użyć w metodzie, a następnie dowolną liczbę argumentów dodatkowych. (W przypadku dziedziczenia istnieje subtelna różnica semantyczna między dwiema przedstawionymi składniami. W praktyce najczęściej spotykana jest pierwsza forma. Dziedziczenie zostanie omówione w rozdziale 6.).

Jeżeli czytelnik nie zna programowania zorientowanego obiektowo, w pierwszej chwili powyższe informacje mogą mu wydawać się nieco dziwne. Na chwilę obecną należy zaakceptować fakt, że Python ma funkcje konwencjonalne wywoływane w postaci *`nazwa_funkcji`*`(`*`argumenty`*`)` oraz metody wywoływane w postaci *`nazwa_obiektu.nazwa_`*
`↪metody``(`*`argumenty`*`)`. (Programowanie zorientowane obiektowo zostanie omówione w rozdziale 6.).

Operator kropki („atrybut dostępu") jest używany w celu uzyskania dostępu do atrybutów obiektu. Wspomniany atrybut może być dowolnego typu obiektem, chociaż jak dotąd widzieliśmy jedynie atrybuty metod. Ponieważ atrybut może być obiektem posiadającym atrybuty, które z kolei mogą mieć swoje atrybuty itd., istnieje możliwość użycia tylu operatorów kropki, ilu potrzeba w celu uzyskania dostępu do żądanego atrybutu.

Typ `list` posiada wiele innych metod, między innymi `insert()`, która jest używana w celu wstawiania elementu we wskazanej pozycji indeksu. Z kolei metoda `remove()` usuwa element znajdujący się we wskazanej pozycji indeksu. Jak już wcześniej wspomniano, indeksy Pythona zawsze rozpoczynają się od zera.

Wcześniej widzieliśmy, że istnieje możliwość pobierania znaków z ciągu tekstowego za pomocą operatora w postaci nawiasów kwadratowych. Wspomniano także, że wymieniony operator może być używany w dowolnej sekwencji. Ponieważ listy są sekwencjami, więc bez problemu możemy wykonać przedstawione poniżej polecenia:

```
>>> x
['zebra', 49, -879, 'mrównik', 200, 'więcej', 'ekstra']
>>> x[0]
```

```
'zebra'
>>> x[4]
200
```

Krotki również są sekwencjami, jeśli więc zmienna x byłaby krotką, moglibyśmy pobrać jej elementy, używając składni nawiasów kwadratowych w dokładnie taki sam sposób, jak w przypadku listy x. Jednak ponieważ listy można zmieniać (w przeciwieństwie do niezmiennych ciągów tekstowych i krotek), w celu ustawienia elementów listy też możemy użyć operatora nawiasów kwadratowych. Przykładowo:

```
>>> x[1] = "czterdzieści dziewięć"
>>> x
['zebra', czterdzieści dziewięć, -879, 'mrównik', 200, 'więcej', 'ekstra']
```

W przypadku podania pozycji indeksu wykraczającej poza zakres nastąpi zgłoszenie wyjątku. Obsługa błędów będzie pokrótce omówiona w koncepcji 5. Pełne omówienie wyjątków znajduje się w rozdziale 4.

Jak dotąd kilkakrotnie użyto pojęcia *sekwencja*, polegając jedynie na nieoficjalnym przedstawieniu jego znaczenia. Na tym nieformalnym znaczeniu będziemy polegać jeszcze przez jakiś czas. Jednak język Python precyzyjnie definiuje wymagania, które muszą być spełnione przez sekwencję. Podobnie są zdefiniowane funkcje, które muszą być spełnione przez obiekt o ustalonym rozmiarze. Ponadto zdefiniowane jest także wiele innych kategorii, do których może zaliczać się typ danych, o czym przekonamy się w rozdziale 8.

Listy, krotki i inne wbudowane kolekcje typów danych Pythona zostaną szczegółowo omówione w rozdziale 3.

Koncepcja 4. — operatory logiczne

Jedną z podstawowych funkcji dowolnego języka programowania jest obsługa operacji logicznych. Python oferuje cztery zestawy operacji logicznych, w kolejnych sekcjach omówimy podstawy każdego z nich.

Operator tożsamości

Ponieważ wszystkie zmienne Pythona tak naprawdę są odniesieniami do obiektów, czasami sensowne jest sprawdzenie, czy dwa bądź większa liczba odniesień odwołuje się do tego samego obiektu. Operator `is` to operator dwuargumentowy zwracający wartość `True`, jeśli odniesienie do obiektu znajdujące się po lewej stronie operatora odwołuje się do tego samego obiektu, do którego odwołuje się odniesienie po prawej stronie operatora. Poniżej przedstawiono kilka przykładów:

```
>>> a = ["Retencja", 3, None]
>>> b = ["Retencja", 3, None]
>>> a is b
False
>>> b = a
>>> a is b
True
```

Warto pamiętać, że zazwyczaj nie ma większego sensu używanie operatora `is` w celu porównywania obiektów `int`, `str` oraz większości pozostałych typów danych, ponieważ prawie zawsze chcemy porównywać ich wartości. W rzeczywistości stosowanie operatora `is` w celu porównywania elementów danych może prowadzić do otrzymania niewiarygodnych wyników, jak mogliśmy zobaczyć na powyższym przykładzie. Tutaj, chociaż `a` i `b` miały początkowo ustawione te same wartości listy, same listy były przechowywane jako oddzielne obiekty `list`. Podczas pierwszego użycia operator `is` zwrócił więc wartość `False`.

Zaletą operatorów tożsamości jest ich duża szybkość działania. Wynika to z faktu, że obiekty podawane w operatorze nie muszą być analizowane. Operator `is` porównuje jedynie adresy w pamięci wskazanych obiektów — ten sam adres oznacza ten sam obiekt.

Najczęściej spotykaną sytuacją, w której stosuje się operator `is`, jest porównywanie elementu danych z wbudowanym w język obiektem `null` (`None`). Obiekt ten jest często używany jako oznaczenie wartości „nieznany" bądź „nieistniejący":

```
>>> a = "Coś"
>>> b = None
>>> a is not None, b is None
(True, True)
```

W celu odwrócenia testu tożsamości używamy operatora `is not`.

Celem operatora tożsamości jest sprawdzenie, czy dwa odniesienia do obiektu odwołują się do tego samego obiektu, bądź sprawdzenie, czy którykolwiek z obiektów jest typu `None`. Jeżeli zachodzi potrzeba porównania wartości obiektu, wtedy należy zastosować operator porównania.

Operatory porównania

Python oferuje standardowy zestaw binarnych operatorów porównania charakteryzujących się oczekiwaną semantyką: < mniejszy niż, <= mniejszy niż lub równy, == równy, != nierówny, >= większy niż lub równy oraz > większy niż. Wymienione operatory porównują wartości obiektów, to znaczy obiektów, do których odniesienia zostały wskazane w operatorze. Poniżej przedstawiono kilka przykładów wprowadzonych w powłoce Pythona:

```
>>> a = 2
>>> b = 6
>>> a == b
False
>>> a < b
True
>>> a <= b, a != b, a >= b, a > b
(True, True, False, False)
```

Podczas pracy z liczbami całkowitymi wszystko przebiega zgodnie z oczekiwaniami. Podobnie w przypadku pracy z ciągami tekstowymi wydaje się, że operatory porównania również działają poprawnie:

```
>>> a = "wiele ścieżek"
>>> b = "wiele ścieżek"
>>> a is b
```

```
False
>>> a == b
True
```

Chociaż `a` i `b` to różne obiekty (mają odmienne tożsamości), to posiadają takie same wartości, więc w trakcie porównywania są uznawane za takie same. Trzeba jednak zachować ostrożność, ponieważ w celu przedstawienia ciągu tekstowego Python używa kodowania znaków Unicode. Dlatego też porównywanie ciągów tekstowych zawierających znaki wykraczające poza zbiór znaków ASCII może być nieco odmienne i bardziej skomplikowane, niż się wydaje w pierwszej chwili — to zagadnienie zostanie dokładnie omówione w rozdziale 2.

Porównywanie ciągów tekstowych ➤ 83

W pewnych sytuacjach porównywanie tożsamości dwóch ciągów tekstowych bądź liczb — na przykład użycie polecenia `a is b` — spowoduje wygenerowanie wartości zwrotnej `True`, nawet jeśli każdy element został przypisany oddzielnie jak w omawianym przypadku. Wynika to z faktu, że pewne implementacje Pythona będą ponownie używały tego samego obiektu (ponieważ wartość pozostaje taka sama, a obiekt jest niezmienny) w celu zwiększenia wydajności. Morał jest taki, aby podczas porównywania *wartości* stosować operatory `==` i `!=`. Z kolei operatory `is` i `is not` stosować jedynie w trakcie porównywania z obiektem `None`, ewentualnie jeśli naprawdę zachodzi potrzeba sprawdzenia, czy dwa odniesienia do obiektów — a nie ich wartości — są takie same.

Jedną ze szczególnie użytecznych cech operatorów porównania w Pythonie jest możliwość ich łączenia, na przykład:

```
>>> a = 9
>>> 0 <= a <= 10
True
```

To znacznie ładniejszy sposób sprawdzenia, czy podany element danych mieści się w zakresie, niż stosowanie dwóch oddzielnych operacji porównania połączonych logicznym operatorem `and`, jak ma to miejsce w większości innych języków programowania. Dodatkową zaletą tego rozwiązania jest sprawdzanie elementu danych tylko jednokrotnie (ponieważ w podanym wyrażeniu pojawia się tylko jednokrotnie). Może mieć to ogromne znaczenie w sytuacji, gdy obliczenie wartości elementu danych jest bardzo kosztowne, bądź jeśli uzyskanie dostępu do elementu danych powoduje powstanie efektu ubocznego.

Dzięki „mocnemu" aspektowi Pythona, jakim jest dynamiczna kontrola typu, porównania, które nie mają sensu, będą powodowały zgłoszenie wyjątku. Przykładowo:

```
>>> "trzy" < 4
Traceback (most recent call last):
...
TypeError: unorderable types: str() < int()
```

Kiedy następuje zgłoszenie wyjątku, który nie jest obsłużony, Python wyświetla komunikaty dotyczące ostatnich wywołań (*traceback*) wraz z komunikatem błędu wyjątku. W celu zachowania przejrzystości w powyższym przykładzie usunięto komunikaty dotyczące ostatnich wywołań, zastępując je wielokropkiem[3]. Ten sam wyjątek `TypeError` wystąpi,

Obsługa błędów powstałych w trakcie działania programu ➤ 430

[3] Komunikat dotyczący ostatnich wywołań — *traceback* (czasem nazywany *backtrace*) to lista wszystkich wywołań wykonanych od początku wywołania stosu aż do chwili wystąpienia nieobsłużonego wyjątku.

jeśli wydamy polecenie `"3" < 4`, gdyż Python nie próbuje odgadnąć naszych intencji. Odpowiednim podejściem jest albo przeprowadzenie wyraźniej konwersji — na przykład `int("3") < 4`, albo użycie możliwych do porównania typów danych, to znaczy dwóch liczb całkowitych lub dwóch ciągów tekstowych.

Język Python znacznie ułatwia tworzenie własnych typów danych, które będą doskonale się integrowały z istniejącymi. Przykładowo: możemy utworzyć własny liczbowy typ danych, który następnie będzie można stosować w operacji porównania z wbudowanym typem `int`, a także innymi wbudowanymi bądź utworzonymi liczbowymi typami danych. Nie będzie jednak możliwe porównywanie go z ciągami tekstowymi lub innymi nieliczbowymi typami danych.

Alternatywa w postaci typu danych `FuzzyBool` ➤ 268

Operator przynależności

W przypadku typów danych będących sekwencjami bądź kolekcjami, takimi jak ciągi tekstowe, listy i krotki, przynależność elementu można sprawdzić za pomocą operatora `in`, natomiast brak przynależności — za pomocą operatora `not in`. Przykładowo:

```
>>> p = (4, "żaba", 9, -33, 9, 2)
>>> 2 in p
True
>>> "pies" not in p
True
```

W przypadku list i krotek operator `in` używa wyszukiwania liniowego, które może być bardzo wolne podczas przeszukiwania ogromnych kolekcji (dziesiątki tysięcy elementów bądź więcej). Z drugiej strony, operator `in` jest bardzo szybki podczas stosowania go w słowniku lub zbiorze. Obie wymienione kolekcje typów danych zostaną omówione w rozdziale 3. Poniżej pokazano przykład użycia operatora `in` wraz z ciągiem tekstowym:

```
>>> fraza = "Dzikie Łabędzie autorstwa Jung Chang"
>>> "J" in fraza
True
>>> "han" in fraza
True
```

Na szczęście w przypadku ciągu tekstowego operator przynależności może być stosowany w celu wyszukania podciągu tekstowego o dowolnej długości. (Jak już wcześniej wspomniano, znak jest po prostu ciągiem tekstowym o długości 1).

Operatory logiczne

Język Python oferuje trzy operatory logiczne: `and`, `or` i `not`. Zarówno `and`, jak i `or` używają logiki warunkowej i zwracają w wyniku operand, który determinuje wynik — nie zwracają wartości boolowskiej (o ile operand faktycznie nie będzie typu boolowskiego). Zobaczmy, co to oznacza w praktyce:

```
>>> pięć = 5
>>> dwa = 2
>>> zero = 0
>>> pięć and dwa
```

```
2
>>> dwa and pięć
5
>>> pięć and zero
0
```

Jeżeli wyrażenie występuje w kontekście boolowskim, to wartość zwrotna będzie typu boolowskiego. Dlatego też, gdyby powyższe przykłady znajdowały się na przykład w konstrukcji `if`, wartościami zwrotnymi mogłyby być `True`, `True` i `False`.

```
>>> nic = 0
>>> pięć or dwa
5
>>> dwa or pięć
2
>>> zero or pięć
5
>>> zero or nic
0
```

Operator `or` działa podobnie. Tutaj wartościami zwrotnymi w kontekście boolowskim byłyby `True`, `True`, `True` i `False`.

Jednoargumentowy operator `not` oblicza swój argument w kontekście boolowskim i zawsze zwraca wynik w postaci wartości boolowskiej. Dlatego też — kontynuując wcześniejszy przykład — wartością zwrotną polecenia `not (zero or nic)` będzie `True`, natomiast polecenia `not dwa` — wartość `False`.

Koncepcja 5. — polecenia kontroli przepływu programu

Wcześniej wspomniano, że polecenia napotkane w pliku *.py* zostają wykonane po kolei, począwszy od pierwszego wiersza i dalej wiersz po wierszu. Kontrola nad przepływem programu może być zmieniona przez wywołanie funkcji bądź metody lub strukturę kontrolną, taką jak polecenie warunkowe lub pętla. Kontrola nad przepływem programu jest zmieniana także po zgłoszeniu wyjątku.

W tej sekcji skoncentrujemy się na poleceniu `if` Pythona, podczas gdy pętle `while` i `for` oraz funkcje zostaną omówione w koncepcji 8., natomiast metody w rozdziale 6. Przedstawimy także bardzo prostą obsługę wyjątków. Szczegółowe omówienie tego zagadnienia znajduje się w rozdziale 4. Jednak w pierwszej kolejności należy wyjaśnić kilka pojęć.

Wyrażenie boolowskie to dowolne wyrażenie, którego obliczenie daje w wyniku wartość boolowską (czyli `True` lub `False`). W języku Python takie wyrażenie przyjmuje wartość `False`, jeśli jest predefiniowaną stałą `False`, obiektem specjalnym `None`, pustą sekwencją bądź kolekcją (na przykład pusty ciąg tekstowy, lista lub krotka) lub liczbowym typem danych o wartości 0. Wszystko pozostałe jest uznawane za `True`. Kiedy tworzymy własny typ danych (na przykład jak w rozdziale 6.), istnieje możliwość samodzielnego ustalenia, jaka ma być wartość zwrotna danego typu w kontekście boolowskim.

W języku Python blok kodu, to znaczy sekwencja jednego bądź większej liczby poleceń, nosi nazwę *pakietu*. Ponieważ pewne elementy składni Pythona *wymagają* obecności pakietu, Python dostarcza słowo kluczowe `pass`, które jest poleceniem niewykonującym żadnego działania. Dlatego też może być użyte wszędzie tam, gdzie wymagany będzie pakiet (ewentualnie jeśli chcemy wskazać, że rozważamy dany przypadek), ale nie zachodzi potrzeba przeprowadzania jakiegokolwiek przetwarzania.

Polecenie if

Ogólna składnia polecenia `if` w Pythonie jest następująca[4]:

```
if wyrażenie_boolean1:
    pakiet1
elif wyrażenie_boolean2:
    pakiet2
...
elif wyrażenie_booleanN:
    pakietN
else:
    pakiet_else
```

W bloku instrukcji `if` może być zero lub większa liczba klauzul `elif`, natomiast ostatnia klauzula `else` jest opcjonalna. Jeżeli chcemy uwzględnić określoną sytuację, ale nie trzeba wykonywać żadnych działań w przypadku jej wystąpienia, wtedy można użyć słowa kluczowego `pass` jako pakietu dla tej sytuacji.

Pierwszą rzeczą rzucającą się w oczy programistom C++ lub Javy jest brak nawiasów okrągłych i klamrowych. Kolejną jest obecność dwukropka (`:`). Na początku bardzo łatwo zapomnieć o tej części składni. Dwukropki są używane wraz z klauzulami `else` i `elif` oraz w każdym innym miejscu, do którego przechodzi pakiet.

W przeciwieństwie do większości innych języków programowania, do wskazania struktury bloku Python stosuje wcięcia. Niektórzy programiści nie lubią takiego rozwiązania, zwłaszcza jeśli wcześniej nie próbowali go stosować, i podchodzą do sprawy dość emocjonalnie. Jednak przywyknięcie do tego rozwiązania zabiera kilka dni, a po upływie kilku tygodni bądź miesięcy kod pozbawiony nawiasów klamrowych wydaje się przyjemniejszy i łatwiejszy w odczycie niż kod stosujący takie nawiasy.

Ponieważ pakiety są wskazywane za pomocą wcięć, oczywiste jest, że rodzi się pytanie w stylu „jakiego rodzaju wcięcia?". Dokumentacja Pythona zaleca stosowanie czterech spacji (tylko spacji, żadnych tabulatorów) na każdy poziom wcięcia. Większość nowoczesnych edytorów tekstowych pozwala na ustawienie automatycznej obsługi wcięć. (Edytor środowiska IDLE oczywiście posiada taką funkcję, podobnie jak większość innych edytorów przeznaczonych do tworzenia kodu w języku Python). Python będzie działał bez problemu z dowolną liczbą spacji, tabulatorów lub połączeń obu, zakładając, że użyte wcięcia zachowają spójność. W niniejszej książce stosujemy oficjalne zalecenie dla języka Python.

[4] W niniejszej książce wielokropek (...) oznacza wiersze, które pominięto w listingu.

Poniżej przedstawiono bardzo prosty przykład polecenia `if`:

```
if x:
    print("x jest niezerowe")
```

W tym przypadku, jeżeli wartość wyrażenia (x) będzie wynosiła `True`, nastąpi wykonanie pakietu (wywołanie funkcji `print()`).

```
if wiersze < 1000:
    print("mały")
elif wiersze < 10000:
    print("średni")
else:
    print("duży")
```

Powyżej przedstawiono nieco bardziej złożone polecenie `if`, które wyświetla słowo opisujące wartość zmiennej `wiersze`.

Polecenie while

Polecenie `while` jest używane w celu wykonania pakietu dowolną liczbę razy, choć może wystąpić sytuacja, w której pakiet w ogóle nie zostanie wykonany. Ilość powtórzeń zależy od stanu wyrażenia boolowskiego w pętli `while`. Poniżej przedstawiono składnię polecenia `while`:

```
while wyrażenie_boolean:
    pakiet
```

W rzeczywistości pełna składnia pętli `while` jest nieco bardziej skomplikowana, niż przedstawiono powyżej, ponieważ obsługiwane są słowa kluczowe `break` i `continue`. Ponadto w pętli może znajdować się opcjonalna klauzula `else`, która zostanie omówiona w rozdziale 4. Polecenie `break` powoduje przekazanie kontroli nad programem do pierwszego polecenia w zewnętrznej pętli względem tej, w której wystąpiło polecenie `break` — to znaczy powoduje opuszczenie bieżącej pętli. Z kolei polecenie `continue` przekazuje kontrolę nad programem do początku pętli. Zarówno polecenie `break`, jak i `continue` są zwykle używane wewnątrz poleceń `if` w celu warunkowej zmiany zachowania pętli.

```
while True:
    element = pobierz_następny_element()
    if not element:
        break
    przetwórz_element(element)
```

Powyższa pętla `while` ma bardzo typową strukturę i działa aż do chwili przetworzenia wszystkich elementów przeznaczonych do przetworzenia. (Przyjmujemy założenie, że zarówno `pobierz_następny_element()`, jak i `przetwórz_element()` do własne funkcje zdefiniowane w innym miejscu kodu). W powyższym przykładzie pakiet polecenia `while` zawiera polecenie `if`, które z kolei ma własny pakiet — w tym przypadku składający się z pojedynczego polecenia `break`.

Polecenie for... in

Pętla `for` w Pythonie ponownie używa słowa kluczowego `in` (które w innym kontekście jest operatorem przynależności) i posiada następującą składnię:

```
for zmienna in iteracja:
    pakiet
```

Pętla `for` — podobnie jak pętla `while` — obsługuje polecenia `break` i `continue`, a także opcjonalną klauzulę `else`. Części *zmienna* zostaje po kolei przypisane odniesienie do każdego obiektu w części *iteracja*. Wymieniona *iteracja* to dowolny typ danych, przez który można przejść — obejmuje między innymi ciągi tekstowe (tutaj iteracja następuje znak po znaku), listy, krotki oraz inne kolekcje typów danych Pythona.

```
for kraj in ["Dania", "Finlandia", "Norwegia", "Szwecja"]:
    print(kraj)
```

Powyżej pokazano bardzo proste podejście pozwalające na wyświetlenie listy krajów. W praktyce znacznie częściej spotykane rozwiązanie zakłada użycie zmiennej:

```
kraje = ["Dania", "Finlandia", "Norwegia", "Szwecja"]
for kraj in kraje:
    print(kraj)
```

W rzeczywistości cała lista (lub krotka) może być wyświetlona bezpośrednio za pomocą funkcji `print()`, na przykład `print(kraje)`. Bardzo często do wyświetlenia kolekcji stosuje się pętlę `for` (lub omówione w dalszej części książki listy składane — z ang. *list comprehension*) w celu zachowania pełnej kontroli nad formatowaniem.

Listy składane ➤ 135

```
for litera in "ABCDEFGHIJKLMNOPQRSTUVWXYZ":
    if litera in "AEIOU":
        print(litera, "jest samogłoską")
    else:
        print(litera, "jest spółgłoską")
```

W powyższym fragmencie kodu w pierwszej kolejności używamy słowa kluczowego `in` jako części polecenia `for`. Zmienna `litera` pobiera wartości „A", „B" i tak dalej, aż do „Z", zmieniając pętlę podczas każdej iteracji. W drugim wierszu kodu ponownie używamy słowa kluczowego `in`, ale tym razem jako operatora przynależności. Warto zwrócić uwagę, że powyższy przykład pokazuje zagnieżdżone pakiety. Pakietem pętli `for` jest konstrukcja `if...else`, a zarówno polecenie `if`, jak i `else` mają własne pakiety.

Podstawowa obsługa wyjątków

Wiele funkcji i metod Pythona wskazuje błędy lub inne ważne zdarzenia poprzez zgłoszenie wyjątku. Wspomniany wyjątek jest obiektem, dokładnie takim samym jak dowolny obiekt Pythona. Podczas konwersji na postać ciągu tekstowego (na przykład w chwili wyświetlania) wyjątek generuje komunikat tekstowy. Poniżej przedstawiono prostą formę składni obsługi wyjątku:

```
try:
    pakiet_try
except wyjątek1 as zmienna1:
```

```
    pakiet1_exception
...
except wyjątekN as zmiennaN:
    pakiet_exceptionN
```

Warto zwrócić uwagę, że część *zmienna* jest opcjonalna, możemy być zainteresowani tylko faktem zgłoszenia określonego wyjątku, a nie jego komunikatem tekstowym.

Pełna składnia jest nieco bardziej skomplikowana. Przykładowo klauzula `except` może zawierać obsługę wielu wyjątków, poza tym istnieje opcjonalna klauzula `else`. To wszystko zostanie szczegółowo omówione w rozdziale 4.

Logika wyjątku działa w następujący sposób. Jeżeli wszystkie polecenia w pakiecie bloku `try` zostaną wykonane bez zgłoszenia wyjątku, bloki `except` będą pominięte. W przypadku zgłoszenia wyjątku w bloku `try` kontrola nad programem jest natychmiast przekazywana do pakietu odpowiadającego pierwszemu dopasowanemu wyjątkowi. Oznacza to, że pozostałe polecenia w pakiecie znajdujące się za poleceniem, które spowodowało zgłoszenie wyjątku, nie zostaną wykonane. Jeżeli wystąpi taka sytuacja i jeśli zdefiniowano część *zmienna*, wówczas wewnątrz pakietu obsługi wyjątku *zmienna* odnosi się do obiektu wyjątku.

Jeśli wyjątek zostanie zgłoszony w obsłudze bloku `except` lub jeśli zgłoszony wyjątek nie zostanie dopasowany do żadnego bloku `except`, Python będzie próbował dopasować blok `except` w kolejnym, zewnętrznym zasięgu. Poszukiwanie odpowiedniej procedury obsługi wyjątku będzie prowadzone w kierunku na zewnątrz aż do chwili jej znalezienia i obsłużenia wyjątku lub zakończy się w przypadku nieznalezienia jej. W tym drugim przypadku nastąpi przerwanie wykonywania programu wraz z nieobsłużonym wyjątkiem. Wówczas Python wyświetli komunikat dotyczący ostatnich wywołań oraz komunikat tekstowy wygenerowany przez wyjątek.

Obsługa błędów powstałych w trakcie działania programu ➤ 430

Poniżej przedstawiono przykład:

```
s = input("podaj liczbę całkowitą: ")
try:
    i = int(s)
    print("podano prawidłową liczbę całkowitą:", i)
except ValueError as err:
    print(err)
```

Jeżeli użytkownik wprowadzi wartość „3.5”, wówczas zostaną wyświetlone następujące dane wyjściowe:

```
invalid literal for int() with base 10: '3.5'
```

Natomiast po podaniu wartości „13” będą wyświetlone następujące dane wyjściowe:

```
podano prawidłową liczbę całkowitą: 13
```

W wielu książkach obsługa błędów jest uznawana za temat zaawansowany, którego omówienie jest przekładane na sam koniec. Jednak zgłaszanie i przede wszystkim obsługa wyjątków to bardzo ważny sposób działania Pythona. Dlatego też powinniśmy jak najwcześniej nauczyć się korzystania z tego mechanizmu. Jak się wkrótce będzie można przekonać, używanie procedur obsługi wyjątków może spowodować, że kod stanie się czytelniejszy. Wynika to z faktu oddzielenia przetwarzania od „wyjątkowych” przypadków, którymi jesteśmy rzadko zainteresowani.

Koncepcja 6. — operatory arytmetyczne

Język Python zawiera pełny zestaw operatorów arytmetycznych, między innymi operatory dwuargumentowe dla czterech podstawowych operacji matematycznych: + dodawania, - odejmowania, * mnożenia i / dzielenia. Oprócz tego wiele typów danych Pythona może być używanych z rozszerzonymi operatorami przypisania, takimi jak += i *=. Operatory +, - i * zachowują się zgodnie z oczekiwaniami, kiedy oba operandy są liczbami całkowitymi:

```
>>> 5 + 6
11
>>> 3 - 7
-4
>>> 4 * 8
32
```

Warto zwrócić uwagę, że jak ma to miejsce w większości języków programowania, minus (-) może być używany zarówno jako operator jednoargumentowy (negacja), jak również operator dwuargumentowy (odejmowanie). Podczas przeprowadzania dzielenia można dostrzec różnice między Pythonem i innymi językami:

```
>>> 12 / 3
4.0
>>> 3 / 2
1.5
```

Wynikiem działania operatora dzielenia jest wartość zmiennoprzecinkowa, a nie liczba całkowita. Wiele innych języków programowania stosuje w tym miejscu liczbę całkowitą, usuwając część po przecinku dziesiętnym. Jeżeli w wyniku chcemy otrzymać liczbę całkowitą, zawsze można zastosować konwersję za pomocą funkcji `int()` lub użyć operatora dzielenia usuwającego część po przecinku dziesiętnym (//), który zostanie omówiony w dalszej części książki.

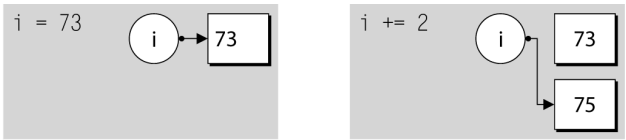
Operatory liczbowe oraz funkcje ➤ 70

```
>>> a = 5
>>> a
5
>>> a += 8
>>> a
13
```

W pierwszej chwili powyższe polecenia nie są zaskakujące, zwłaszcza dla programistów znających język pochodny od języka C. W większości takich języków rozszerzone przypisanie jest skrótem przypisania wyniku operacji — na przykład polecenie `a += 8` daje dokładnie taki sam wynik, jak `a = a + 8`. Jednak istnieją dwie subtelne różnice, jedna charakterystyczna dla Pythona, natomiast druga ogólnie dotyczy operatorów rozszerzonego przypisania stosowanych w dowolnym języku programowania.

Przede wszystkim trzeba pamiętać, że typ danych `int` jest niezmienny, to znaczy raz przypisanej wartości `int` nie można zmienić. Dlatego też podczas używania operatora przypisania względem niezmiennego obiektu następuje wykonanie operacji, utworzenie obiektu przechowującego wynik, a następnie ponownie dołączenie, ale tym razem do obiektu przechowującego wynik, a nie wcześniej dołączonego. Stąd w poprzednim przykładzie

po napotkaniu polecenia `a += 8` Python przeprowadza obliczenie `a + 8`, umieszcza wynik w nowym obiekcie `int`, a następnie ponownie dołącza `a` w taki sposób, aby wskazywał na nowy obiekt `int`. (Jeżeli początkowy obiekt, do którego odnosiła się zmienna `a`, nie ma więcej odniesień od innych obiektów, wtedy zostaje przeznaczony do usunięcia przez mechanizm zbierania nieużytków). Zilustrowano to na rysunku 1.3.



Rysunek 1.3. Rozszerzone przypisanie stosowane względem niezmiennego obiektu

Druga różnica polega na tym, że wykonanie polecenia `a` *`operator`*`= b` nie powoduje takiego samego efektu jak wykonanie polecenia `a = a` *`operator`* `b`. Rozszerzona wersja wyszukuje wartość operandu `a` tylko jednokrotnie, aby działała potencjalnie szybciej. Ponadto, jeżeli `a` będzie skomplikowanym wyrażeniem (takim jak element listy, dla którego następuje obliczenie pozycji, na przykład `elementy[przesunięcie + pozycja]`), wtedy użycie operatora rozszerzonego może się wiązać z mniejszym ryzykiem powstawania błędów. Wynika to z faktu, że jeśli obliczenie będzie wiązało się z koniecznością wprowadzenia zmiany, wówczas zmiana ta będzie musiała być wprowadzona tylko w jednym wyrażeniu zamiast w dwóch.

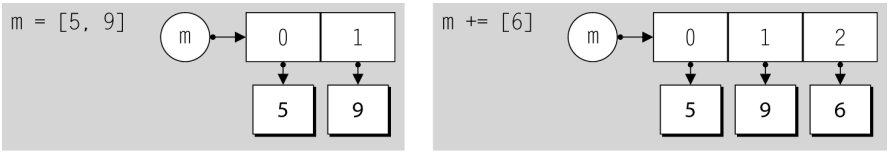
Zarówno w przypadku ciągów tekstowych, jak i list Python przeciąża operatory + i += (na przykład ponownie używa ich z innymi typami danych). W pierwszej sytuacji oznacza to łączenie ciągów tekstowych, natomiast w drugiej — dołączanie do ciągów tekstowych i rozszerzanie (dołączanie innej listy) list:

```
>>> imię = "Jan"
>>> imię + "Kowalski"
'JanKowalski'
>>> imię += " Kowalski"
>>> imię
'Jan Kowalski'
```

Podobnie jak liczby całkowite, tak i ciągi tekstowe są niezmienne, więc w trakcie używania operatora += następuje utworzenie nowego ciągu tekstowego i ponownie dołączenie do niego obiektu znajdującego się po lewej stronie operatora. Przebiega to dokładnie w taki sam sposób, jaki wcześniej omówiono dla obiektów `int`. Listy są obsługiwane przez taką samą składnię, ale w tle zachowują się nieco inaczej:

```
>>> ziarna = ["sezam", "słonecznik"]
>>> ziarna += ["dynia"]
>>> ziarna
['sezam', 'słonecznik', 'dynia']
```

Ponieważ listy można zmieniać, po użyciu operatora += następuje modyfikacja początkowego obiektu listy. Nie występuje więc konieczność ponownego dołączania obiektu do zmiennej `ziarna`. Zilustrowano to na rysunku 1.4.



Rysunek 1.4. Rozszerzone przypisanie stosowane względem obiektu, który można modyfikować

Ponieważ składnia Pythona zręcznie ukrywa różnice między zmiennymi i niezmiennymi typami danych, rodzi się pytanie, dlaczego w ogóle potrzebujemy obu typów? Powody są głównie związane z wydajnością. Niezmienne typy danych są znacznie efektywniejsze w implementacji (ponieważ nigdy nie ulegają zmianie) niż typy zmienne. Ponadto pewne kolekcje typów danych — na przykład zestawy — mogą działać jedynie z niezmiennymi typami danych. Z drugiej strony, zmienne typy danych mogą być znacznie wygodniejsze w użyciu. Kiedy rozróżnienie będzie miało znaczenie, wtedy je omówimy — na przykład w rozdziale 4. podczas przedstawiania sposobu ustawiania argumentów domyślnych we własnych funkcjach, w rozdziale 3. podczas omawiania list, zestawów i kilku innych typów danych oraz w rozdziale 6. podczas prezentacji tworzenia własnych typów danych.

Prawy operand w operatorze += listy musi umożliwiać iterację, gdyż w przeciwnym razie zostanie zgłoszony wyjątek:

Obsługa błędów powstałych w trakcie działania programu ➤ 430

```
>>> ziarna += 5
Traceback (most recent call last):
...
TypeError: 'int' object is not iterable
```

Właściwym sposobem rozbudowy listy jest użycie obiektu pozwalającego na przeprowadzenie iteracji, czyli na przykład listy:

```
>>> ziarna += [5]
>>> ziarna
['sezam', 'słonecznik', 'dynia', 5]
```

Oczywiście umożliwiający iterację obiekt użyty do rozbudowania listy sam może posiadać więcej niż tylko jeden element:

```
>>> ziarna += [9, 1, 5, "mak"]
>>> ziarna
['sezam', 'słonecznik', 'dynia', 5, 9, 1, 5, 'mak']
```

Dołączenie zwykłego ciągu tekstowego — na przykład „durian” — zamiast listy zawierającej ciąg tekstowy `["durian"]` prowadzi do logicznego, choć prawdopodobnie zadziwiającego wyniku:

```
>>> ziarna = ["sezam", "słonecznik ", "dynia"]
>>> ziarna += "durian"
>>> ziarna
['sezam', 'słonecznik', 'dynia', 'd', 'u', 'r', 'i', 'a', 'n']
```

Operator += listy rozbudowuje listę poprzez dołączenie każdego elementu pobranego z dostarczonego mu obiektu pozwalającego na iterację. Ponieważ ciąg tekstowy pozwala na iterację, więc każdy znak ciągu tekstowego zostaje dołączony indywidualnie. Jeżeli użyjemy metody `append()`, jej argument zawsze będzie dodany jako pojedynczy element.

Koncepcja 7. — operacje wejścia-wyjścia

Aby móc tworzyć naprawdę użyteczne programy, musimy mieć możliwość odczytu danych wejściowych — na przykład od użytkownika za pośrednictwem konsoli lub z plików — oraz generowania danych wyjściowych w konsoli bądź do pliku. Jak dotąd używaliśmy już wbudowanej funkcji Pythona o nazwie `print()`, choć będzie ona dokładnie omówiona dopiero w rozdziale 4. W tym miejscu skoncentrujemy się na operacjach wejścia-wyjścia konsoli oraz na stosowaniu przekierowania konsoli podczas odczytu i zapisu plików.

Python oferuje wbudowaną funkcję `input()` przyjmującą dane wejściowe od użytkownika. Funkcja pobiera opcjonalny argument w postaci ciągu tekstowego, który zostanie wyświetlony w konsoli. Następnie odczekuje, aż użytkownik wprowadzi dane wejściowe i naciśnie klawisz *Enter* (lub *Return*). Jeżeli użytkownik nie wprowadzi żadnego tekstu i ograniczy się do naciśnięcia klawisza *Enter*, wartością zwrotną funkcji `input()` będzie pusty ciąg tekstowy. W przeciwnym razie wartością zwrotną będzie ciąg tekstowy zawierający dane wprowadzone przez użytkownika bez żadnego znaku przejścia do nowego wiersza.

Poniżej przedstawiono pierwszy „użyteczny" program — wykorzystuje on wiele z wymienionych dotąd koncepcji, a jedyną nowością jest użycie funkcji `input()`:

```
print("Podaj liczby całkowite, po każdej naciśnij klawisz Enter lub po prostu naciśnij
↪klawisz Enter, aby zakończyć działanie programu.")
total = 0
count = 0

while True:
    line = input("liczba całkowita: ")
    if line:
        try:
            number = int(line)
        except ValueError as err:
            print(err)
            continue
        total += number
        count += 1
    else:
        break

if count:
    print("ilość =", count, "suma =", total, "średnia =", total / count)
```

Przykłady dołączone do książki 17 ➤

Program (w dołączonych przykładach zapisany jako plik *sum1.py*) ma jedynie siedemnaście wykonywanych wierszy. Poniżej przedstawiono typowy sposób działania programu po jego uruchomieniu:

```
Podaj liczby całkowite, po każdej naciśnij klawisz Enter lub po prostu naciśnij
↪klawisz Enter, aby zakończyć działanie programu.
liczba całkowita: 12
```

```
liczba całkowita: 7
liczba całkowita: 1x
invalid literal for int() with base 10: '1x'
liczba całkowita: 15
liczba całkowita: 5
liczba całkowita:
ilość = 4 suma = 39 średnia = 9.75
```

Chociaż program jest bardzo krótki, to pozostaje całkiem solidny. Jeżeli użytkownik wprowadzi ciąg tekstowy, który nie może być skonwertowany na liczbę całkowitą, problem zostanie wychwycony przez procedurę obsługi błędów. Wspomniana procedura wyświetli odpowiedni komunikat błędu, a kontrola nad programem będzie przekazana na początek pętli („kontynuacja wykonywania pętli"). W ostatnim poleceniu `if` upewniamy się, że użytkownik w ogóle nie podał żadnej liczby — wtedy nie zostaną wyświetlone żadne dane wyjściowe w tym kroku i unikniemy operacji dzielenia przez zero.

Pełne omówienie tematu obsługi plików zostanie przedstawione w rozdziale 7. W tej chwili będziemy tworzyć pliki poprzez przekierowanie danych wyjściowych funkcji `print()` z konsoli, na przykład:

```
C:\test.py > wynik.txt
```

Błąd dotyczący powiązań plików w Windows 26 ➤

Powyższe polecenie spowoduje, że dane wyjściowe wywołania funkcji `print()` w przykładowym programie *test.py* zostaną zapisane w pliku *wynik.txt*. Podana składnia działa (zazwyczaj) zarówno w konsoli Windows, jak i w konsolach systemów Unix. Jeżeli domyślną wersją Pythona jest 2 lub występuje błąd dotyczący przypisania plików w konsoli, wtedy w przypadku systemu Windows trzeba wydać polecenie `C:\Python31\python.exe test.py > wynik.txt`. W przeciwnym razie przyjmujemy założenie, że katalog z Pythonem 3 został podany w zmiennej `PATH`, i jeśli polecenie `test.py > wynik.txt` nie zadziała, najczęściej wystarczające będzie polecenie `python test.py > wynik.txt`. W systemach z rodziny Unix program musi mieć uprawnienia wykonywania (`chmod +x test.py`) i uruchamiany jest poleceniem `./test.py`, chyba że katalog, w którym się znajduje, został podany w zmiennej `PATH`. W takim przypadku wystarczające będzie wydanie polecenia `test.py`.

Odczyt danych może być przeprowadzony poprzez przekierowanie danych pliku i użycie ich jako danych wejściowych. Przekierowanie wykonujemy analogicznie jak przekierowanie danych wyjściowych do pliku. Jeśli jednak przekierowanie zastosujemy w programie *sum1.py*, działanie programu zakończy się niepowodzeniem. Wiąże się to z tym, że funkcja `input()` zgłosi wyjątek po otrzymaniu znaku EOF (*End Of File*, czyli koniec pliku). Poniżej przedstawiono solidniejszą wersję programu (plik *sum2.py*), który może przyjmować dane wejściowe albo podane przez użytkownika za pomocą klawiatury albo poprzez przekierowanie danych z pliku:

```
print("Podaj liczby całkowite, po każdej naciśnij klawisz Enter.
↪Aby zakończyć działanie programu, naciśnij klawisze ^D lub ^Z.")
total = 0
count = 0

while True:
    try:
        line = input()
        if line:
```

```
            number = int(line)
            total += number
            count += 1
    except ValueError as err:
        print(err)
        continue
    except EOFError:
        break

if count:
    print("ilość =", licznik, "suma =", suma, "średnia =", suma / licznik)
```

Po wydaniu polecenia `sum2.py < data\sum2.dat` (gdzie *sum2.dat* oznacza plik znajdujący się w podkatalogu *data* i zawierający liczby, po jednej liczbie w każdym wierszu) dane wyjściowe wyświetlone w konsoli przedstawiają się następująco:

```
Podaj liczby całkowite, po każdej naciśnij klawisz Enter. Aby zakończyć działanie programu,
↪naciśnij klawisze ^D lub ^Z.
ilość = 37 suma = 1839 średnia = 49.7027027027
```

W programie wprowadzono wiele drobnych zmian, tak aby działał prawidłowo zarówno w trybie interaktywnym, jak i podczas przekierowania. Przede wszystkim zmieniono sposób przerwania z pustego wiersza na znak EOF (*Ctrl+D* w systemie Unix, *Ctrl+Z*, *Enter* w Windows). Dzięki tej zmianie program działa solidniej podczas obsługi plików z danymi wejściowymi, które zawierają puste wiersze. Zaniechaliśmy także wyświetlania znaku zachęty do wprowadzenia każdej liczby, ponieważ nie ma to sensu podczas stosowania przekierowania. Ponadto użyliśmy pojedynczego bloku `try` wraz z dwiema procedurami obsługi błędów.

Warto zwrócić uwagę, że jeśli zostanie wprowadzona nieprawidłowa liczba całkowita (za pośrednictwem klawiatury albo w wyniku wystąpienia „złego” wiersza danych w pliku dostarczającym danych wejściowych), konwersja `int()` zgłosi wyjątek `ValueError`. Oznacza to, że w takim przypadku nie nastąpi zwiększenie wartości ani zmiennej `suma`, ani zmiennej `licznik`. Takiego zachowania programu oczekujemy.

W powyższym kodzie moglibyśmy zastosować także dwa oddzielne bloki `try` służące do obsługi wyjątków:

```
while True:
    try:
        line = input()
        if line:
            try:
                number = int(line)
            except ValueError as err:
                print(err)
                continue
            total += number
            count += 1
    except EOFError:
        break
```

Preferowanym rozwiązaniem jest grupowanie wyjątków razem i próba zachowania maksymalnej przejrzystości głównego kodu przetwarzającego.

Koncepcja 8. — tworzenie i wywoływanie funkcji

Bez problemów można tworzyć programy, używając w tym celu jedynie typów danych i struktur kontrolnych, które zostały przedstawione we wcześniejszych sekcjach. Jednak bardzo często zachodzi potrzeba wielokrotnego wykonania tej samej operacji, ale z drobną różnicą, na przykład z inną wartością początkową. Python oferuje możliwość hermetyzacji pakietów na postać funkcji, które można parametryzować za pomocą argumentów przekazywanych do funkcji. Poniżej przedstawiono ogólną składnię tworzenia funkcji:

```
def nazwa_funkcji(argumenty):
    pakiet
```

Argumenty są opcjonalne, w przypadku podawania wielu argumentów trzeba je rozdzielić przecinkami. Każda funkcja Pythona posiada wartość zwrotną. Domyślnie wartością zwrotną jest `None`, chyba że zostanie zwrócona z wewnątrz funkcji za pomocą składni `return` *wartość*, gdzie *wartość* oznacza wartość zwrotną danej funkcji. Wartość zwrotna może być pojedynczą wartością bądź krotką wartości. Ponadto wartość zwrotna może zostać zignorowana przez wywołującego funkcję, wówczas będzie po prostu odrzucona.

Polecenie `return` ➤ 191.

Warto zwrócić uwagę, że `def` to polecenie działające podobnie jak operator przypisania. W trakcie wykonywania polecenia `def` następuje utworzenie obiektu funkcji. Poza tym tworzone jest odniesienie do obiektu o wskazanej nazwie, któremu następnie przypisuje się odwołanie do obiektu funkcji. Ponieważ funkcje są obiektami, to mogą być przechowywane w kolekcjach zbiorów danych i przekazywane jako parametry do innych funkcji, o czym przekonamy się w kolejnych rozdziałach.

Jednym z zadań najczęściej wykonywanych w interaktywnych aplikacjach działających w konsoli jest pobieranie liczby całkowitej od użytkownika. Poniżej przedstawiono funkcję realizującą to zadanie:

```
def get_int(msg):
    while True:
        try:
            i = int(line)
            return i
        except ValueError as err:
            print(err)
```

Powyższa funkcja pobiera tylko jeden argument o nazwie `liczba`. Wewnątrz pętli `while` użytkownik jest proszony o podanie liczby całkowitej. Jeżeli wprowadzi inne dane, wówczas zostanie zgłoszony wyjątek `ValueError`, nastąpi wyświetlenie komunikatu błędu i ponowne wykonanie pętli. Po wprowadzeniu liczby całkowitej zostanie ona zwrócona wywołującemu funkcję. Poniżej pokazano przykład wywołania tej funkcji:

```
wiek = pobierz_liczbe("Podaj swój wiek: ")
```

W powyższym przykładzie podanie wartości pojedynczego argumentu jest obowiązkowe, ponieważ nie zdefiniowano wartości domyślnej. W rzeczywistości język Python obsługuje bardzo skomplikowaną i elastyczną składnię parametrów funkcji wraz z obsługą wartości domyślnych argumentów oraz argumenty pozycyjne i w postaci słów kluczowych. Wszystkie zagadnienia związane ze składnią funkcji zostaną omówione w rozdziale 4.

Chociaż tworzenie własnych funkcji może przynosić wiele satysfakcji, w wielu przypadkach nie jest konieczne. Wynika to z faktu, że Python dostarcza wielu wbudowanych funkcji, a jeszcze większa liczba funkcji w modułach znajduje się w bibliotece standardowej języka. Dlatego też wymagana przez programistę funkcja może już istnieć.

Moduł w języku Python to po prostu plik *.py* zawierający kod Pythona, na przykład definicję własnej funkcji lub klasy (własny typ danych) oraz czasami zmienne. W celu uzyskania dostępu do funkcjonalności zawartej w module należy go zaimportować, na przykład:

```
import sys
```

W celu zaimportowania modułu używamy polecenia `import` wraz z nazwą pliku *.py*, ale pomijamy rozszerzenie pliku[5]. Po zaimportowaniu modułu można uzyskać dostęp do dowolnej funkcji, klasy lub zmiennej zawartej w tym module, na przykład:

```
print(sys.argv)
```

Moduł `sys` zawiera zmienną `argv` — lista, gdzie pierwszy element to nazwa, pod jaką zostaje wywołany program, natomiast element drugi i kolejne to argumenty programu uruchamianego z poziomu wiersza poleceń. Dwa powyższe wiersze kodu składają się na cały program *echoargs.py*. Jeżeli program zostanie uruchomiony z poziomu wiersza poleceń `echoargs.py -v`, wówczas w konsoli wyświetli `['echoargs.py', '-v']`. (W systemie z rodziny Unix pierwszy wpis może mieć postać `'./echoargs.py'`).

Operator kropki (.) 35 ➤

Ogólnie rzecz biorąc, składnia używania funkcji pochodzącej z modułu jest następująca: *`nazwa_modułu.nazwa_funkcji`*`(`*`argumenty`*`)`. Wymieniona składnia wykorzystuje operator kropki („atrybut dostępu") przedstawiony w koncepcji 3. Biblioteka standardowa zawiera dużą ilość modułów, wiele z nich będzie wykorzystanych w niniejszej książce. Nazwy wszystkich modułów standardowych są zapisane małymi literami, więc niektórzy programiści stosują nazwy, w których pierwsza litera każdego słowa jest wielka (na przykład `Moj_Modul`), w celu odróżnienia modułów własnych od wbudowanych.

Spójrzmy na jeszcze jeden przykład — moduł `random` (w bibliotece standardowej plik *random.py*), który dostarcza wielu użytecznych funkcji:

```
import random
x = random.randint(1, 6)
y = random.choice(["jabłko", "banan", "wiśnia", "durian"])
```

Po wykonaniu powyższych poleceń zmienna `x` będzie zawierała liczbę całkowitą z przedziału od 1 do 6 włącznie, natomiast zmienna `y` — jeden z ciągów tekstowych z listy przekazanej funkcji `random.choice()`.

Wiersz shebang (#!) 27 ➤

Według konwencji wszystkie polecenia `import` są umieszczane na początku plików *.py* tuż po wierszu shebang oraz dokumentacji modułu. (Dokumentacja modułu będzie omówiona w rozdziale 5.). Zaleca się w pierwszej kolejności importowanie modułów biblioteki standardowej, następnie modułów firm trzecich, a na końcu własnych.

---

[5] Moduły `sys`, podobnie jak inne moduły wbudowane oraz moduły zaimplementowane w języku C, niekoniecznie mają odpowiadające im pliki *.py*. Są jednak używane w taki sam sposób, jak moduły posiadające pliki *.py*.

Przykłady

W poprzednim podrozdziale czytelnicy mogli poznać język Python wystarczająco, aby zacząć tworzyć rzeczywiste programy. W tym podrozdziale można się będzie zapoznać z dwoma pełnymi programami wykorzystującymi jedynie te konstrukcje Pythona, które omówiono do tej chwili. Będzie to zarówno pomocne w utrwaleniu zdobytej dotąd wiedzy, jak również pokaże możliwości języka.

W kolejnych podrozdziałach czytelnicy będą mogli poznawać coraz bardziej język Python i jego bibliotekę. Pozwoli to na tworzenie bardziej treściwych i solidniejszych programów niż przedstawione w tym podrozdziale. Najpierw należy jednak poznać podstawy, na bazie których później będziemy tworzyć programy.

bigdigits.py

Pierwszy z omawianych programów jest dość krótki, ale zawiera pewne ciekawe aspekty, między innymi listę list. Oto sposób działania programu: na podstawie liczby podanej w wierszu polecenia program wyświetla w konsoli tę samą liczbę, używając do tego „dużych" cyfr.

W miejscach, gdzie duża liczba użytkowników korzysta z tej samej drukarki o ogromnej szybkości działania, często spotykaną praktyką jest to, że każdy wydruk danego użytkownika jest poprzedzony stroną tytułową zawierającą nazwę danego użytkownika oraz pewne inne informacje wydrukowane za pomocą omawianej tu techniki.

Kod programu będzie omówiony w trzech częściach: polecenie `import`, tworzenie list przechowujących dane używane przez program oraz właściwe przetwarzanie danych. Jednak w pierwszej kolejności zobaczmy, jak wygląda przykładowe uruchomienie programu:

```
bigdigits.py 41072819
   *     *    ***    *****   ***    ***    *    ****
  **    **   *   *       *  *   *  *   *  **   *   *
 * *     *   *     *     *   *  *  *   *   *   *   *
*  *     *   *    *    *      *    ***    *    ****
******   *   *    *   *       *    *   *   *       *
   *     *   *   *   *       *     *   *   *       *
   *    ***    ***    *     *****   ***   ***      *
```

Nie pokazaliśmy powyżej znaku zachęty konsoli (lub początkowych znaków `./` w przypadku systemu Unix), przyjmujemy za rzecz oczywistą, że są one obecne i stosowane.

```
import sys
```

Ponieważ program musi odczytać argument z wiersza polecenia (liczbę przeznaczoną do wyświetlenia), zachodzi potrzeba uzyskania dostępu do listy `sys.argv`. Program rozpoczynamy więc od zaimportowania modułu `sys`.

Każda liczba będzie przedstawiona jako lista ciągów tekstowych. Na przykład poniżej pokazano zero:

```
Zero = ["  ***  ",
        " *   * ",
        "*     *",
        "*     *",
        "*     *",
        " *   * ",
        "  ***  "]
```

Szczegółem, na który warto zwrócić uwagę, jest to, że lista ciągów tekstowych `Zero` została rozciągnięta w wielu wierszach. Polecenia Pythona zazwyczaj stanowią pojedynczy wiersz, ale mogą rozciągać się na wiele wierszy, jeśli są wyrażeniem umieszczonym w nawiasach, listą, zestawem, dosłownym słownikiem, listą argumentów wywołania funkcji bądź wielowierszowym poleceniem, w którym każdy znak kończący wiersz poza ostatnim jest poprzedzony lewym ukośnikiem (\). We wszystkich wymienionych przypadkach liczba wierszy nie ma znaczenia, podobnie jak wcięcia dla drugiego i kolejnego wiersza.

Typ danych `set` ➤ 138

Typ danych `dict` ➤ 143

Każda lista przedstawiająca cyfrę ma siedem ciągów tekstowych, wszystkie o stałej szerokości, choć ta szerokość jest różna dla każdej z cyfr. Listy pozostałych cyfr stosują ten sam wzorzec, którego użyto dla cyfry 0. Zostały jednak zapisane w sposób zajmujący jak najmniej miejsca, a nie w sposób gwarantujący największą czytelność dla programisty:

```
One = [" * ", "** ", " * ", " * ", " * ", " * ", "***"]
Two = [" *** ", "*   *", "*  * ", "  *  ", " *   ", "*    ", "*****"]
# ...
Nime = [" ****", "*   *", "*   *", " ****", "    *", "    *", "    *"]
```

Ostatni element danych to lista obejmująca wszystkie listy cyfr:

```
Digits = [Zero, One, Two, Three, Four, Five, Six, Seven, Eight, Nine]
```

Istnieje także możliwość bezpośredniego utworzenia listy `Digits` i dzięki temu uniknięcia potrzeby tworzenia dodatkowych zmiennych, na przykład:

```
Digits = [
    ["   ***    ", " *  * ", "* *", "* *", "* *",
     " * * ," *** "],# Zero
    [" * ", "** ", " * ", " * ", " * ", " * ", "***"], # One
    # ...
    [" ****", "*          *", "*   *", " ****", "    *", "    *",
        "    *"] # Nine
]
```

Preferujemy jednak użycie oddzielnej zmiennej dla każdej cyfry — zarówno w celu ułatwienia zrozumienia programu, jak również dlatego, że zastosowanie zmiennych jest bardziej eleganckie.

Pozostała część kodu zostaje przedstawiona w pojedynczym bloku, aby czytelnik mógł spróbować określić sposób jego działania, zanim przejdziemy do omówienia tego kodu.

```
try:
    digits = sys.argv[1]
    row = 0
    while row < 7:
        line = ""
        column = 0
```

```
            while column < len(digits):
                number = int(digits[column])
                digit = Digits[number]
                line += digit[row] + "  "
                column += 1
            print(line)
            row += 1
    except IndexError:
        print("użycie: bigdigits.py <liczba>")
    except ValueError as err:
        print(err, "in", digits)
```

Cały kod został opakowany procedurą obsługi wyjątków, co pozwala na przechwycenie dwóch błędów, jeśli cokolwiek pójdzie niezgodnie z oczekiwaniami. Program rozpoczyna się od pobrania argumentu z wiersza polecenia. Lista `sys.argv` rozpoczyna się od 0, podobnie jak wszystkie listy w Pythonie. Element w pozycji o indeksie 0 jest nazwą, pod jaką program został uruchomiony. Dlatego też w działającym programie lista ta ma przynajmniej jeden element. Jeżeli nie został podany żaden argument, wówczas nastąpi próba uzyskania dostępu do drugiego elementu w jednoelementowej liście, co doprowadzi do zgłoszenia wyjątku `IndexError`. W takim przypadku kontrola nad programem jest natychmiast przekazywana do odpowiedniego bloku odpowiedzialnego za obsługę wyjątku. W omawianym programie będzie to po prostu komunikat informujący o sposobie użycia programu. Wykonywanie programu jest kontynuowane od polecenia po bloku `try`. Ponieważ w tym programie nie ma kodu poza blokiem `try`, następuje zakończenie działania programu.

Jeżeli nie nastąpi zgłoszenie wyjątku `IndexError`, ciąg tekstowy `digits` będzie przechowywał argument wiersza polecenia, którym — mamy nadzieję — będzie sekwencja cyfr. (Jak pamiętamy z sekcji poświęconej koncepcji 2., identyfikatory rozróżniają wielkość liter, więc `digits` i `Digits` to zupełnie inne elementy). Każda duża cyfra jest przedstawiona za pomocą siedmiu ciągów tekstowych. Dlatego też w celu poprawnego wyświetlenia danych wyjściowych musimy wyświetlić najwyższy wiersz każdej cyfry, następnie kolejny itd. aż do wyświetlenia wszystkich wierszy. Do przejścia przez wszystkie wiersze używamy pętli `while`. Równie dobrze można zastosować polecenie `for row in (0, 1, 2, 3, 4, 5, 6):`, a później w znacznie lepszy sposób wykorzystać wbudowaną funkcję `range()`.

Funkcja `range()` ➤ 158

Ciąg tekstowy `line` wykorzystujemy do przechowywania ciągów tekstowych wiersza dla wszystkich cyfr używanych w danej liczbie. Następnie mamy pętlę przechodzącą przez kolumny, to znaczy przez każdy kolejny znak w argumencie wiersza polecenia. Każdy znak jest pobierany za pomocą polecenia `digits[column]`, a następnie konwertowany na liczbę całkowitą o nazwie `number`. Jeżeli konwersja zakończy się niepowodzeniem, wtedy zostanie zgłoszony wyjątek `ValueError`, a kontrola nad program będzie natychmiast przekazana odpowiedniej procedurze obsługi wyjątku. W omawianym przykładzie spowoduje to wyświetlenie komunikatu błędu i działanie programu zostanie wznowione od polecenia znajdującego się tuż po bloku `try`. Jak już wcześniej wspomniano, ponieważ po bloku `try` nie ma więcej kodu, program zakończy działanie.

Gdy konwersja zakończy się powodzeniem, liczba całkowita `number` będzie używana jako wartość indeksu na liście `Digits`, z której wyodrębniamy listę ciągów tekstowych dla danej cyfry (`digit`). Następnie z listy dodajemy ciąg tekstowy odpowiedniego wiersza (`row`) do budowanej linii oraz dodajemy dwie spacje w celu pionowego oddzielenia cyfr od siebie.

Za każdym razem, gdy pętla `while` kończy działanie, wyświetlamy zbudowaną linię. Kluczowy moment dla zrozumienia omawianego programu to ten, w którym dołączamy ciąg tekstowy wiersza każdej cyfry do bieżącego wiersza budowanej linii. Warto uruchomić program i przeanalizować sposób jego działania. Do omówionego w tej sekcji programu powrócimy w ćwiczeniach, aby nieco usprawnić wyświetlane przez program dane wyjściowe.

generate_grid.py

Jedna z często występujących potrzeb to wygenerowanie danych testowych. Nie ma jednego, ogólnego programu służącego do tego celu, ponieważ dane testowe kolosalnie się między sobą różnią. Język Python jest często używany do generowania danych testowych, ponieważ tworzenie i modyfikowanie programów w Pythonie jest bardzo proste. W tej sekcji omówimy program służący do generowania siatki losowych liczb całkowitych. Użytkownik podaje liczbę wierszy i kolumn, które chce otrzymać, oraz zakres, w jakim mają znajdować się wygenerowane liczby. Pracę nad programem rozpoczniemy od spojrzenia na wynik przykładowego uruchomienia programu:

```
generate_grid.py
wiersze: 4x
invalid literal for int() with base 10: '4x'
wiersze: 4
kolumny: 7
minimum (lub naciśnij Enter dla 0): -100
maximum (lub naciśnij Enter dla 1000):
       643       -45       923       252       520       -92       372
       625       342       543       128       302       461       134
        35       954       540       691       676      1000       940
       361       941       233       347       408       319        -7
```

Program działa interaktywnie, na początku popełniamy błąd w trakcie podawania liczby wierszy. Odpowiedzią programu jest komunikat błędu oraz ponowienie prośby o podanie liczby wierszy. W przypadku wartości maksymalnej naciśnięto klawisz *Enter*, akceptując tym samym wartość domyślną.

Kod zostanie omówiony w czterech częściach: polecenie `import`, definicja funkcji `get_int()` — znacznie bardziej złożona wersja tej funkcji niż przedstawiona w koncepcji 8., interakcja z użytkownikiem w celu pobrania używanych wartości oraz samo przetwarzanie danych.

```
import random
```

Funkcja `random.randint()` ➤ 52

Moduł `random` jest potrzebny w celu uzyskania dostępu do funkcji `random.randint()`.

```
def get_int(msg, minimum, default):
    while True:
        try:
            line = input(msg)
```

```
            if not line and default is not None:
                return default
            i = int(line)
            if i < minimum:
                print("musi być >=", minimum)
            else:
                return i
        except ValueError as err:
            print(err)
```

Funkcja wymaga podania trzech argumentów: ciągu tekstowego komunikatu, wartości minimalnej oraz wartości domyślnej. Jeżeli użytkownik po prostu naciśnie klawisz *Enter*, wtedy są dwie możliwości. W pierwszym przypadku, gdy `default` wynosi `None`, to znaczy wartość domyślna nie została podana, kontrola nad programem jest przekazywana do wiersza `int()`. W tym wierszu konwersja zakończy się niepowodzeniem (ponieważ ciągu tekstowego `''` nie można skonwertować na postać liczby całkowitej) i zostanie zgłoszony wyjątek `ValueError`. W drugim przypadku, jeśli `default` jest różna od `None`, wtedy zostanie zwrócona. W przeciwnym razie tekst wprowadzony przez użytkownika funkcja spróbuje przekonwertować na postać liczby całkowitej. Gdy konwersja zakończy się powodzeniem, nastąpi sprawdzenie, czy otrzymana liczba całkowita jest przynajmniej równa zdefiniowanemu minimum.

Widzimy więc, że wartością zwrotną funkcji zawsze będzie albo `default` (jeżeli użytkownik nacisnął jedynie klawisz *Enter*), albo poprawna liczba całkowita równa bądź większa od zdefiniowanego `minimum`.

```
rows = get_int("wierszw: ", 1, None)
columns = get_int("kolumny: ", 1, None)
minimum = get_int("minimum (lub naciśnij Enter dla 0): ", -1000000, 0)

default = 1000
if default < minimum:
    default = 2 * minimum
maximum = get_int("maksimum (lub naciśnij Enter dla  " + str(default) + "): ",
                  minimum, default)
```

Funkcja `get_int()` bardzo ułatwia pobranie liczby wierszy, kolumn oraz wartości minimalnej oczekiwanej przez użytkownika. W przypadku wierszy i kolumn wartość domyślna wynosi `None`, co oznacza, że użytkownik musi podać liczbę całkowitą. W przypadku minimum ustaloną wartością domyślną jest 0, natomiast dla maksimum ustaloną wartością domyślną jest 1000 lub dwukrotna wartość minimum, jeśli minimum jest większe bądź równe 1000.

Jak już wspomniano w poprzednim przykładzie, lista argumentów wywołania funkcji może rozciągać się na dowolną liczbę wierszy, a wcięcia są nieistotne w przypadku drugiego i kolejnych wierszy.

Gdy znana jest już liczba wierszy i kolumn wymaganych przez użytkownika oraz wartości minimalna i maksymalna generowanych liczb, można przystąpić do właściwego przetwarzania danych.

```
row = 0
while row < rows:
    line = ""
```

```
        column = 0
        while column < columns:
            i = random.randint(minimum, maximum)
            s = str(i)
            while len(s) < 10:
                s = " " + s
            line += s
            column += 1
        print(line)
        row += 1
```

W celu wygenerowania siatki używamy pętli `while`, zewnętrzna działa z wierszami, środkowa z kolumnami, a wewnętrzna ze znakami. W środkowej pętli następuje wygenerowanie losowej liczby w ustalonym zakresie, a następnie konwersja liczby na ciąg tekstowy. Wewnętrzna pętla jest używana w celu dopełnienia ciągu tekstowego spacjami, tak aby każda liczba była przedstawiona jako ciąg tekstowy o długości dziesięciu znaków. Ciąg tekstowy `line` służy do zebrania liczb tworzących każdy wiersz i wyświetlenia linii po dodaniu liczb w każdej kolumnie. W ten sposób kończymy omawianie drugiego przykładu.

Język Python oferuje bardzo zaawansowane funkcje pozwalające na formatowanie ciągu tekstowego. Poza tym zapewnia doskonałą obsługę pętli `for...in`, więc znacznie bardziej rzeczywiste wersje obu programów (*bigdigits.py* i *generate_grid.py*) powinny używać pętli `for...in`. Natomiast w programie *generate_grid.py* można zastosować oferowane przez Pythona funkcje formatowania ciągu tekstowego zamiast prymitywnego uzupełniania spacjami. Jednak na obecnym etapie jesteśmy ograniczeni do ośmiu kluczowych koncepcji Pythona, które zostały przedstawione w tym rozdziale. Jak widać, są one wystarczające do tworzenia pełnych i użytecznych programów. W każdym kolejnym programie czytelnicy będą poznawać nowe funkcje Pythona, więc w miarę lektury książki kolejne przedstawiane programy będą coraz bardziej skomplikowane.

Metoda `str.format()` ➤ 95

Podsumowanie

Czytając rozdział, można się było dowiedzieć, jak edytować i uruchamiać programy w języku Python. Przeanalizowaliśmy kilka małych, choć kompletnych programów. Jednak większość miejsca w rozdziale poświęcono na omówienie ośmiu kluczowych koncepcji „Pięknego serca" Pythona — wystarczającej liczby konstrukcji języka, które pozwalają na tworzenie rzeczywistych programów.

Zaczęliśmy od przedstawienia dwóch podstawowych typów danych, czyli `int` i `str`. Dosłowne liczby całkowite są zapisywane w dokładnie taki sam sposób, jak w większości innych języków programowania. Natomiast dosłowne ciągi tekstowe są ujmowane albo w apostrofy, albo w cudzysłów. Nie ma to żadnego znaczenia, o ile po obu stronach ciągu tekstowego zastosowano te same znaki ograniczenia (tzn. apostrofy lub cudzysłów). Istnieje możliwość przeprowadzenia konwersji między liczbami całkowitymi i ciągami tekstowymi, na przykład `int("250")` i `str(125)`. Jeżeli konwersja liczby całkowitej kończy się niepowodzeniem, następuje zgłoszenie wyjątku `ValueError`. Z drugiej strony, warto pamiętać, że niemal wszystko można przekonwertować na postać ciągu tekstowego.

Ciąg tekstowy jest sekwencją, więc funkcje i operacje, które można wykonywać na sekwencjach, mogą także być przeprowadzane na ciągu tekstowym. Przykładowo dostęp do określonego znaku można uzyskać za pomocą operatora dostępu ([]), łączenie ciągów tekstowych umożliwia operator +, natomiast dołączanie jednego ciągu tekstowego do innego — operator +=. Ponieważ ciągi tekstowe są niezmienne, to w tle operacja dołączania powoduje utworzenie nowego ciągu tekstowego łączącego podane ciągi tekstowe, a następnie ponowne dołączenie obiektu ciągu znajdującego się po lewej stronie operatora do nowo utworzonego ciągu tekstowego. Za pomocą pętli `for...in` można przejść przez ciąg tekstowy znak po znaku. Z kolei wbudowana funkcja `len()` informuje o ilości znaków tworzących ciąg tekstowy.

W przypadku obiektów niezmiennych, takich jak ciągi tekstowe, liczby całkowite i krotki, możemy tworzyć kod, jakby odniesienie do obiektu było zmienną, czyli jakby odniesienie było obiektem, do którego następuje odwołanie w tworzonym kodzie. Tę samą technikę można zastosować także w przypadku obiektów pozwalających na modyfikacje, choć wszelkie zmiany takiego obiektu są wprowadzane we wszystkich egzemplarzach tego obiektu (na przykład we wszystkich odniesieniach do obiektu). To zagadnienie będzie omówione w rozdziale 3.

Język Python oferuje wiele wbudowanych kolekcji typów danych, kolejne są dostępne w bibliotece standardowej. W rozdziale omówiliśmy typy `list` i `tuple`, w szczególności sposób tworzenia list i krotek, na przykład `parzyste = [2, 4, 6, 8]`. Listy — podobnie jak wszystkie pozostałe elementy w Pythonie — są obiektami, więc można wywoływać względem nich metody, na przykład polecenie `parzyste.append(10)` spowoduje dodanie kolejnego elementu do listy `parzyste`. Podobnie jak ciągi tekstowe, tak i listy oraz krotki są sekwencjami. Dzięki temu możemy przejść przez nie element po elemencie, używając do tego pętli `for...in`, a także ustalić liczbę elementów za pomocą funkcji `len()`. Istnieje także możliwość pobrania określonego elementu listy bądź krotki przy użyciu operatora dostępu ([]), połączenia dwóch list lub krotek za pomocą operatora + oraz dołączenia jednej do drugiej za pomocą operatora +=. Jeżeli wystąpi potrzeba dołączenia pojedynczego elementu do listy, należy użyć funkcji `lista.append()` lub operatora += wraz z nazwą dołączanego elementu — na przykład parzyste `+= [12]`. Ponieważ listy można modyfikować, zmianę danego elementu można przeprowadzić za pomocą operatora [], na przykład `parzyste[1] = 16`.

Szybki operator tożsamości `is` i `is not` służy do sprawdzenia, czy dwa odniesienia do obiektu odwołują się do tego samego obiektu. Taka możliwość jest szczególnie użyteczna podczas przeprowadzenia sprawdzenia względem wbudowanego obiektu `None`. Dostępne są wszystkie zwykłe operatory porównania (<, <=, ==, !=, >=, >), ale mogą być używane jedynie z porównywalnymi typami danych i tylko wtedy, gdy operatory są obsługiwane. Wszystkie przedstawione dotąd typy danych — `int`, `str`, `list` i `tuple` — w pełni obsługują wymieniony zestaw operatorów porównania. Próba porównania niezgodnych typów danych, na przykład porównanie wartości `int` z `list`, spowoduje zgłoszenie wyjątku `TypeError`.

Python obsługuje standardowe operatory logiczne `and`, `or` i `not`. Zarówno `and`, jak i `or` są operatorami zwracającymi operand, który zdominował wyniki — to nie może być wartość boolowska (choć można ją skonwertować na postać wartości boolowskiej). Natomiast `not` zawsze zwraca wartość `True` lub `False`.

Przynależność typów sekwencji, między innymi ciągów tekstowych, list i krotek, można sprawdzić za pomocą operatorów `in` i `not in`. Sprawdzanie przynależności wykorzystuje powolne, liniowe przeszukiwanie list i krotek, natomiast w przypadku ciągów tekstowych — potencjalnie szybszy algorytm hybrydowy. Jednak wydajność rzadko jest problemem poza bardzo dużymi ciągami tekstowymi, listami i krotkami. W rozdziale 3. zostaną przedstawione tablice asocjacyjne i kolekcje typów danych — obie wymienione konstrukcje zapewniają bardzo szybkie sprawdzenie przynależności. Istnieje także możliwość określenia typu zmiennej obiektu (na przykład typu obiektu, do którego odwołuje się odniesienie do obiektu) przy użyciu funkcji `type()`. Wymieniona funkcja zazwyczaj stosowana jest jedynie w trakcie procesu usuwania błędów oraz testowania.

Język Python oferuje kilka struktur kontrolnych, między innymi polecenia warunkowe `if...elseif...else`, pętle warunkowe `while`, pętle powalające na przejście przez sekwencje `for...in` oraz bloki obsługi wyjątków `try...except`. Zarówno pętla `while`, jak i `for...in` może być wcześniej zakończona za pomocą polecenia `break`. Obie pętle mogą również przekazać kontrolę nad programem do początku pętli, używając w tym celu polecenia `continue`.

Obsługiwane są standardowe operatory matematyczne, między innymi `+`, `-`, `*` i `/`, chociaż w przypadku Pythona nietypowe jest, że wynikiem operatora dzielenia (`/`) zawsze będzie liczba zmiennoprzecinkowa, nawet jeśli oba operandy są liczbami całkowitymi. (Znane z innych języków programowania dzielenie z pominięciem części dziesiętnej także jest dostępne w Pythonie za pomocą operatora `//`). Python oferuje także rozszerzone operatory przypisania, takie jak `+=` i `*=`. Powodują one utworzenie nowych obiektów i ponowne przeprowadzenie operacji dołączenia, jeśli lewy operand jest niezmienny. Operatory arytmetyczne są przeciążane przez typy `str` i `list`, co zostało już wcześniej powiedziane.

Do wykonywania operacji wejścia-wyjścia konsoli stosuje się funkcje `input()` i `print()`. Używając przekierowania pliku w konsoli, można wykorzystać te same funkcje wbudowane do odczytywania i zapisywania plików.

Oprócz wielu możliwości wbudowanych w Pythona istnieje także obszerna biblioteka standardowa, której moduły są dostępne po ich zaimportowaniu za pomocą polecenia `import`. Jednym z najczęściej importowanych modułów jest `sys`, który przechowuje listę argumentów wiersza polecenia `sys.argv`. Kiedy Python nie zawiera funkcji wymaganej przez programistę, bardzo łatwo można utworzyć własną, używając do tego celu polecenia `def`.

Wykorzystując wiedzę przedstawioną w tym rozdziale, można tworzyć krótkie, choć użyteczne programy w języku Python. Z kolejnego rozdziału czytelnik dowie się więcej na temat typów danych Pythona, zagłębimy się w typy `int` i `str` oraz przedstawimy nowe. W rozdziale 3. jeszcze dokładniej przedstawimy krotki i listy, a także opowiemy więcej na temat kolekcji typów danych. Rozdział 4. jest poświęcony na dokładne omówienie struktur kontrolnych w Pythonie. Czytelnik dowie się również, w jaki sposób tworzyć własne funkcje, aby funkcjonalność móc umieścić w pakiecie i uniknąć w ten sposób powielania kodu oraz promować ponowne używanie fragmentów kodu.

Ćwiczenia

Przykłady dołączone do książki 17 ➤

Celem ćwiczeń przedstawionych poniżej oraz w pozostałych rozdziałach książki jest zachęcenie czytelnika do eksperymentowania z Pythonem oraz zdobycia doświadczenia, które pomoże w przyswojeniu materiału przedstawionego w danym rozdziale. Zaprezentowane przykłady i ćwiczenia obejmują przetwarzanie zarówno liczbowe, jak i tekstowe oraz są skierowane do jak najszerszej grupy odbiorców. Ponadto są na tyle małe, aby położyć nacisk na myślenie i naukę, a nie po prostu wprowadzanie kodu. W przykładach dołączonych do książki znajdują się odpowiedzi do każdego ćwiczenia.

1. Jedną z odmian programu *bigdigits.py* jest wersja, w której zamiast gwiazdki (*) wyświetlana jest odpowiednia cyfra, na przykład:

```
bigdigits_ans.py 719428306
77777   1    9999     4     222    888    333     000     666
    7  11   9   9    44    2   2  8   8  3   3   0   0   6
   7    1   9   9   4 4    2  2   8   8      3  0     0  6
  7     1    9999  4  4      2     888     33   0     0  6666
 7      1       9  444444   2     8   8      3  0     0  6   6
7       1       9     4     2     8   8  3   3   0   0  6   6
7      111      9     4     22222   888    333     000     666
```

   W tym celu można zastosować dwa podejścia. Najłatwiejszym jest po prostu zamiana gwiazdek w listach. Jednak to podejście nie jest elastyczne i nie powinno być zastosowane. Zamiast tego należy zmodyfikować kod przetwarzający dane, tak aby zamiast za każdym razem dodawać każdy wiersz do tworzonej linii, dodawać cyfry znak po znaku. Ponadto zamiast gwiazdki trzeba użyć odpowiedniej cyfry.

   Pracę nad rozwiązaniem można zacząć od skopiowania *bigdigits.py* i następnie wprowadzić zmiany w mniej więcej pięciu wierszach. Nie jest to zadanie trudne, raczej kosmetyczne. Rozwiązanie znajduje się w pliku *bigdigits_ans.py*.

2. Środowisko IDLE można wykorzystać jako elastyczny i oferujący potężne możliwości kalkulator. Jednak czasami użyteczne będzie posiadanie kalkulatora przeznaczonego do określonych zadań. Zadaniem jest więc utworzenie programu proszącego użytkownika o podanie liczby (w pętli `while`) i stopniowo budującego listę podanych liczb. Kiedy użytkownik zakończy podawanie liczb (poprzez naciśnięcie klawisza *Enter*), program powinien wyświetlić wprowadzone liczby, ilość podanych liczb, ich sumę, najmniejszą i największą wprowadzoną liczbę oraz wartość średnią (suma / ilość liczb). Poniżej pokazano przykładowe uruchomienie programu:

```
average1_ans.py
podaj liczbę lub naciśniej Enter, aby zakończyć: 5
podaj liczbę lub naciśniej Enter, aby zakończyć: 4
podaj liczbę lub naciśniej Enter, aby zakończyć: 1
podaj liczbę lub naciśniej Enter, aby zakończyć: 8
podaj liczbę lub naciśniej Enter, aby zakończyć: 5
podaj liczbę lub naciśniej Enter, aby zakończyć: 2
podaj liczbę lub naciśniej Enter, aby zakończyć:
liczby: [5, 4, 1, 8, 5, 2]
ilość = 6 suma = 25 najmniejsza = 1 największa = 8 średnia = 4.16666666667
```

Inicjalizacja potrzebnych zmiennych (pusta lista to po prostu `[]`) wymaga około czterech wierszy. Natomiast pętla `while` z prostą obsługą błędów to poniżej piętnastu wierszy kodu. Wyświetlenie danych wyjściowych zabierze kilka dodatkowych wierszy kodu, więc cały program łącznie z pustymi wierszami zwiększającymi czytelność programu powinien się zmieścić w 25 wierszach kodu.

3. W pewnych sytuacjach występuje potrzeba wygenerowania tekstu testowego — na przykład w celu wypełnienia projektu witryny internetowej, zanim dostępna będzie rzeczywista treść witryny. Taki program przydaje się też do zapewnienia treści podczas tworzenia generatora tekstów. Zadaniem jest utworzenie programu generującego wspaniałe wiersze (takie, które spowodują wystąpienie rumieńców na twarzach wieszczy).

   Pracę należy rozpocząć od przygotowania listy słów, na przykład zaimków („ten", „ta"), podmiotu („kot", „pies", „mężczyzna", „kobieta"), czasowników („śpiewał", „uciekał", „skoczył") oraz przymiotników („głośny", „cichy", „doskonały", „zły"). Następnie pętlę trzeba wykonać pięciokrotnie, w trakcie każdej iteracji użyć funkcji `random.choice()` w celu pobrania podmiotu, czasownika i przymiotnika. Za pomocą funkcji `random.randint()` można wybrać jedną z dwóch struktur zdania: przyimek, podmiot, czasownik i przymiotnik lub po prostu przyimek, podmiot i czasownik, a następnie wyświetlić zdanie. Poniżej pokazano przykładowy wynik uruchomienia programu:

Metody `random.int()` i `random.choice()` 52 ➤

```
awfulpoetry1_ans.py
another boy laughed badly
the woman jumped
a boy hoped
a horse jumped
another man laughed rudely
```

   W programie trzeba zaimportować moduł `random`. Listę można zmieścić w czterech – dziesięciu wierszach w zależności od liczby użytych słów. Sama pętla wymaga mniej niż dziesięciu wierszy kodu, więc uwzględniając puste wiersze, cały program może zająć około 20 wierszy kodu. Rozwiązanie zadania znajduje się w pliku *awfulpoetry1_ans.py*.

4. Aby program generujący wiersze był jeszcze bardziej uniwersalny, do programu dodamy kod powodujący, że jeśli użytkownik wprowadzi liczbę w wierszu polecenia (z zakresu od 1 do 10 włącznie), program wygeneruje wiersz o podanej liczbie linii. Jeżeli nie zostanie podany argument wiersza polecenia, program będzie domyślnie generował i wyświetlał wiersz zawierający pięć linii. W tym celu trzeba będzie zmienić główną pętlę programu (na przykład na pętlę `while`). Należy pamiętać, że operatory porównania w Pythonie mogą być grupowane, więc nie zachodzi potrzeba używania operatora logicznego `and` podczas sprawdzania, czy podany przez użytkownika argument mieści się w zdefiniowanym zakresie. Dodatkową funkcjonalność programu można zmieścić w mniej więcej dziesięciu wierszach kodu. Rozwiązanie zadania znajduje się w pliku *awfulpoetry2_ans.py*.

5. Możliwość obliczenia mediany (wartości środkowej) w programie utworzonym w ćwiczeniu 2. byłaby pożądaną funkcją, ale w tym celu trzeba posortować listę. W Pythonie listę można bardzo łatwo posortować za pomocą metody `lista.sort()`. Metoda ta nie została jednak jeszcze omówiona, więc nie zastosujemy jej w tym ćwiczeniu. Zadaniem jest rozbudowa programu o blok kodu, który będzie sortował listę liczb — wydajność nie ma tutaj żadnego znaczenia, należy po prostu zastosować najłatwiejsze podejście. Po posortowaniu listy medianą będzie wartość środkowa, jeśli lista ma nieparzystą liczbę elementów, bądź średnia dwóch środkowych wartości w przypadku listy o parzystej liczbie elementów. Medianę należy obliczyć, a następnie wyświetlić wraz z pozostałymi informacjami.

   To dość trudne zadanie zwłaszcza dla niedoświadczonych programistów. Jeżeli czytelnik ma już pewne doświadczenie w programowaniu w języku Python, to zadanie nadal będzie wyzwaniem, zwłaszcza w przypadku używania Pythona jedynie w zakresie, który dotąd omówiono. Kod odpowiedzialny za sortowanie można zmieścić w mniej więcej dziesięciu wierszach, natomiast obliczenie mediany (w tym fragmencie kodu nie można użyć operatora modulus, ponieważ nie został jeszcze omówiony) w czterech. Rozwiązanie zadania znajduje się w pliku *average2_ans.py*.