Poznaj język wykorzystywany przez Google i YouTube!

Wydanie IV

Wprowadzenie

Python

HELION

O'REILLY®

Mark Lutz

Python. Wprowadzenie. Wydanie IV

Autor: Mark Lutz
Tłumaczenie: Anna Trojan, Marek Pętlicki
ISBN: 978-83-246-2694-6
Tytuł oryginału: Learning Python, 4th edition
Format: 172×245, stron: 1184

Poznaj język, który wykorzystuje Google i YouTube!

- Jak tworzyć i przetwarzać obiekty za pomocą instrukcji Pythona?
- Jak stworzyć strukturę kodu i wykorzystać go ponownie?
- Jak programować obiektowo w Pythonie?

Czy wiesz, dlaczego ponad milion programistów na całym świecie używa właśnie tego języka skryptowego? Jego atuty to niezwykła czytelność, spójność i wydajność – pewnie dlatego także i Ty chcesz opanować słynnego Pythona. Kod napisany w tym języku można z łatwością utrzymywać, przenosić i używać go ponownie. Pozostaje on zrozumiały nawet wówczas, jeśli analizuje go ktoś, kto nie jest jego autorem. Co więcej, taki kod ma rozmiary średnio o dwie trzecie do czterech piątych mniejsze od kodu w językach C++ czy Java, co wielokrotnie zwiększa wydajność pracy używających go programistów. Python obsługuje także zaawansowane mechanizmy pozwalające na ponowne wykorzystanie kodu, takie jak programowanie zorientowane obiektowo, a programy w nim napisane działają natychmiast, bez konieczności przeprowadzania długiej kompilacji, niezależnie od wykorzystywanej platformy.

Naukę rozpoczniesz od najważniejszych wbudowanych typów danych – liczb, list czy słowników. Przyjrzysz się również typom dynamicznym oraz ich interfejsom. Później poznasz instrukcje oraz ogólny model składni Pythona. Poszerzysz wiedzę na temat powiązanych z nim narzędzi, takich jak system PyDoc, a także alternatywnych możliwości tworzenia kodu. Dowiesz się wszystkiego na temat modułów: jak się je tworzy, przeładowuje i jak się ich używa. W końcu poznasz klasy oraz zagadnienia związane z programowaniem zorientowanym obiektowo i nauczysz się obsługiwać wyjątki. Czwarte wydanie tej książki zostało wzbogacone o wiele nowych, ciekawych i bardzo zaawansowanych zagadnień, dzięki czemu stanowi doskonałą lekturę także dla zawodowców, na co dzień piszących kod w tym języku. Dzięki tej książce:

- zapoznasz się z podstawowymi typami wbudowanymi Pythona,
- nauczysz się tworzyć i przetwarzać obiekty za pomocą instrukcji Pythona, a także opanujesz ogólny model składni tego języka
- stworzysz strukturę kodu i wykorzystasz kod ponownie dzięki podstawowym narzędziom proceduralnym Pythona
- dowiesz się wszystkiego o modułach Pythona – pakietach instrukcji i funkcji oraz innych narzędziach zorganizowanych w większe komponenty
- odkryjesz narzędzie programowania zorientowanego obiektowo, umożliwiające strukturyzację kodu
- opanujesz model obsługi wyjątków i narzędzia programistyczne służące do pisania większych programów
- zapoznasz się z zaawansowanymi narzędziami Pythona, w tym dekoratorami, deskryptorami, metaklasami i przetwarzaniem tekstu Unicode

Opanuj Pythona z Markiem Lutzem – najbardziej znanym ekspertem w tej dziedzinie!

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

Spis treści

ROZDZIAŁ 4.

Wprowadzenie do typów obiektów Pythona

Niniejszy rozdział rozpoczyna naszą wycieczkę po języku Python. W pewnym sensie w Pythonie „robi się coś z różnymi rzeczami". To „coś" ma postać operacji (działań), takich jak dodawanie czy konkatenacja, natomiast „różne rzeczy" to obiekty, na których wykonuje się owe operacje. W tej części książki skupimy się właśnie na owych „różnych rzeczach", jak również na tym, co mogą z nimi robić nasze programy.

Mówiąc bardziej formalnym językiem, w Pythonie dane przybierają postać *obiektów* — albo wbudowanych obiektów udostępnianych przez Pythona, albo obiektów tworzonych za pomocą Pythona lub innych narzędzi zewnętrznych, takich jak biblioteki rozszerzeń języka C. Choć definicję tę nieco później rozbudujemy, obiekty są generalnie fragmentami pamięci z wartościami i zbiorami powiązanych operacji.

Ponieważ obiekty są najbardziej podstawowym elementem Pythona, ten rozdział rozpoczniemy od przeglądu obiektów wbudowanych w sam język.

Tytułem wstępu warto jednak najpierw ustalić, jak niniejszy rozdział wpisuje się w całość Pythona. Programy napisane w Pythonie można rozbić na moduły, instrukcje, wyrażenia i obiekty — w następujący sposób:

1. Programy składają się z modułów.
2. Moduły zawierają instrukcje.
3. Instrukcje zawierają wyrażenia.
4. *Wyrażenia tworzą i przetwarzają obiekty*.

Omówienie modułów zamieszczone w rozdziale 3. uwzględnia najwyższy poziom w tej hierarchii. Rozdziały tej części książki odwołują się do najniższego jej poziomu, czyli wbudowanych obiektów oraz wyrażeń, które tworzy się w celu korzystania z tych obiektów.

Po co korzysta się z typów wbudowanych?

Osoby używające języków niższego poziomu, takich jak C czy C++, wiedzą, że większość ich pracy polega na implementowaniu *obiektów* — znanych również jako *struktury danych* — tak by reprezentowały one komponenty w dziedzinie naszej aplikacji. Konieczne jest rozplanowanie struktur pamięci, zarządzanie przydzielaniem pamięci czy zaimplementowanie procedur wyszukiwania i dostępu. Te zadania są tak żmudne (i podatne na błędy), na jakie wyglądają, i zazwyczaj odciągają programistę od prawdziwych celów programu.

W typowych programach napisanych w Pythonie większość tej przyziemnej pracy nie jest konieczna. Ponieważ Python udostępnia typy obiektów jako nieodłączną część samego języka, zazwyczaj nie istnieje konieczność kodowania implementacji obiektów przed rozpoczęciem rozwiązywania prawdziwych problemów. Tak naprawdę, o ile oczywiście nie mamy potrzeby korzystania ze specjalnych metod przetwarzania, które nie są dostępne w obiektach wbudowanych, prawie zawsze lepiej będzie skorzystać z gotowego typu obiektu, zamiast tworzyć własny. Poniżej znajduje się kilka przyczyn takiego stanu rzeczy.

- **Obiekty wbudowane sprawiają, że programy łatwo się pisze.** W przypadku prostych zadań obiekty wbudowane często wystarczą nam do stworzenia struktur właściwych dla określonych problemów. Od ręki dostępne są narzędzia o sporych możliwościach, jak zbiory (listy) i tabele, które można przeszukiwać (słowniki). Wiele zadań można wykonać, korzystając z samych obiektów wbudowanych.
- **Obiekty wbudowane są komponentami rozszerzeń.** W przypadku bardziej zaawansowanych zadań być może nadal konieczne będzie udostępnianie własnych obiektów, wykorzystywanie klas Pythona czy interfejsów języka C. Jednak jak okaże się w dalszej części książki, obiekty implementowane ręcznie są często zbudowane na bazie typów wbudowanych, takich jak listy czy słowniki. Strukturę danych stosu można na przykład zaimplementować jako klasę zarządzającą wbudowaną listą lub dostosowującą tę listę do własnych potrzeb.
- **Obiekty wbudowane często są bardziej wydajne od własnych struktur danych.** Wbudowane obiekty Pythona wykorzystują już zoptymalizowane algorytmy struktur danych, które zostały zaimplementowane w języku C w celu zwiększenia szybkości ich działania. Choć możemy samodzielnie napisać podobne typy obiektów, zazwyczaj trudno nam będzie osiągnąć ten sam poziom wydajności, jaki udostępniają obiekty wbudowane.
- **Obiekty wbudowane są standardową częścią języka.** W pewien sposób Python zapożycza zarówno od języków opierających się na obiektach wbudowanych (jak na przykład LISP), jak i języków, w których to programista udostępnia implementacje narzędzi czy własnych platform (jak C++). Choć można w Pythonie implementować własne, unikalne typy obiektów, nie trzeba tego robić, by zacząć programować w tym języku. Co więcej, ponieważ obiekty wbudowane są standardem, zawsze pozostaną one takie same. Rozwiązania własnościowe zazwyczaj mają tendencję do zmian ze strony na stronę.

Innymi słowy, obiekty wbudowane nie tylko ułatwiają programowanie, ale mają także większe możliwości i są bardziej wydajne od większości tego, co tworzy się od podstaw. Bez względu na to, czy zdecydujemy się implementować nowe typy obiektów, obiekty wbudowane stanowią podstawę każdego programu napisanego w Pythonie.

Najważniejsze typy danych w Pythonie

W tabeli 4.1 zaprezentowano przegląd wbudowanych obiektów Pythona wraz ze składnią wykorzystywaną do kodowania ich *literałów* — czyli wyrażeń generujących te obiekty.[1] Niektóre z typów powinny dla osób znających inne języki programowania wyglądać znajomo. Liczby i łańcuchy znaków reprezentują, odpowiednio, wartości liczbowe i tekstowe. Pliki udostępniają natomiast interfejsy służące do przetwarzania plików przechowywanych na komputerze.

Tabela 4.1. Przegląd obiektów wbudowanych Pythona

Typ obiektu	Przykładowy literał (tworzenie)
Liczby	`1234`, `3.1415`, `3+4j`, `Decimal`, `Fraction`
Łańcuchy znaków	`'mielonka'`, `"Brian"`, `b'a\x01c'`
Listy	`[1, [2, 'trzy'], 4]`
Słowniki	`{'jedzenie': 'mielonka', 'smak': 'mniam'}`
Krotki	`(1,'mielonka', 4, 'U')`
Pliki	`myfile = open('jajka', 'r')`
Zbiory	`set('abc'),{'a', 'b', 'c'}`
Inne typy podstawowe	Wartości `Boolean`, typy, `None`
Typy jednostek programu	Funkcje, moduły, klasy (część IV, V i VI książki)
Typy powiązane z implementacją	Kod skompilowany, ślady stosu (część IV i VII książki)

Tabela 4.1 nie jest kompletna, ponieważ *wszystko*, co przetwarzamy w programie napisanym w Pythonie, jest tak naprawdę rodzajem obiektu. Kiedy na przykład wykonujemy w Pythonie dopasowanie tekstu do wzorca, tworzymy obiekty wzorców, natomiast kiedy tworzymy skrypty sieciowe, wykorzystujemy obiekty gniazd. Te pozostałe typy obiektów tworzy się przede wszystkim za pomocą importowania i wykorzystywania modułów; każdy z nich wiąże się z pewnym typem zachowania.

Jak zobaczymy w dalszych częściach książki, *jednostki programów*, takie jak funkcje, moduły i klasy, także są w Pythonie obiektami — są one tworzone za pomocą instrukcji oraz wyrażeń, takich jak `def`, `class`, `import` czy `lambda`, i można je swobodnie przekazywać w skryptach bądź przechowywać w innych obiektach. Python udostępnia również zbiór *typów powiązanych z implementacją*, takich jak obiekty skompilowanego kodu, które są zazwyczaj bardziej przedmiotem zainteresowania osób tworzących narzędzia niż twórców aplikacji. Zostaną one omówione w późniejszych częściach książki.

Pozostałe typy obiektów z tabeli 4.1 nazywane są zazwyczaj *typami podstawowymi*, ponieważ są one tak naprawdę wbudowane w sam język. Oznacza to, że istnieje określona składnia wyrażeń służąca do generowania większości z nich. Na przykład kiedy wykonamy poniższy kod:

```
>>> 'mielonka'
```

[1] W niniejszej książce pojęcie *literał* oznacza po prostu wyrażenie, którego składnia generuje obiekt — czasami nazywane również *stałą*. Warto zauważyć, że „stała" nie oznacza wcale obiektów czy zmiennych, które nigdy nie mogą być zmienione (czyli pojęcie to nie ma związku z `const` z języka C++ czy określeniem „niezmienny" [ang. *immutable*] z Pythona — zagadnienie to omówione zostanie w dalszej części rozdziału).

z technicznego punktu widzenia wykonujemy właśnie wyrażenie z literałem, które generuje i zwraca nowy obiekt łańcucha znaków. Istnieje specyficzna składnia Pythona, która tworzy ten obiekt. Podobnie wyrażenie umieszczone w nawiasach kwadratowych tworzy listę, a w nawiasach klamrowych — słownik. Choć — jak się niedługo okaże — w Pythonie nie istnieje deklarowanie typu, składnia wykonywanego wyrażenia określa typy tworzonych i wykorzystywanych obiektów. Wyrażenia generujące obiekty, jak te z tabeli 4.1, to właśnie miejsca, z których pochodzą typy obiektów.

Co równie ważne: kiedy tworzymy jakiś obiekt, wiążemy go z określonym zbiorem operacji. Na łańcuchach znaków można wykonywać tylko operacje dostępne dla łańcuchów znaków, natomiast na listach — tylko te dla list. Jak się za chwilę okaże, Python jest językiem z *typami dynamicznymi* (to znaczy automatycznie przechowuje za nas informacje o typach, zamiast wymagać kodu z deklaracją), jednak jego typy są *silne* (to znaczy na obiekcie można wykonać tylko te operacje, które są poprawne dla określonego typu).

Z funkcjonalnego punktu widzenia typy obiektów z tabeli 4.1 są bardziej ogólne i mają większe możliwości, niż bywa to w innych językach. Jak się okaże, już same listy i słowniki mają wystarczająco duże możliwości, by zlikwidować większość pracy związanej z obsługą zbiorów i wyszukiwania w językach niższego poziomu. Listy udostępniają uporządkowane zbiory innych obiektów, natomiast słowniki przechowują obiekty i ich klucze. Oba typy danych mogą być zagnieżdżane, mogą rosnąć i kurczyć się na życzenie oraz mogą zawierać obiekty dowolnego typu.

W kolejnych rozdziałach szczegółowo omówimy poszczególne typy obiektów zaprezentowane w tabeli 4.1. Zanim jednak zagłębimy się w szczegóły, najpierw przyjrzyjmy się podstawowym obiektom Pythona w działaniu. Pozostała część rozdziału zawiera przegląd operacji, które bardziej dokładnie omówimy w kolejnych rozdziałach. Nie należy oczekiwać, że zostanie tutaj zaprezentowane wszystko — celem niniejszego rozdziału jest tylko zaostrzenie apetytu i wprowadzenie pewnych kluczowych koncepcji. Najlepszym sposobem na rozpoczęcie czegoś jest… samo rozpoczęcie, zatem czas zabrać się za prawdziwy kod.

Liczby

Dla osób, które zajmowały się już programowaniem czy tworzeniem skryptów, niektóre typy danych z tabeli 4.1 będą wyglądały znajomo. Nawet dla osób niemających nic wspólnego z programowaniem liczby wyglądają dość prosto. Zbiór podstawowych obiektów Pythona obejmuje typowe rodzaje liczb: całkowite (liczby bez części ułamkowej), zmiennoprzecinkowe (w przybliżeniu liczby z przecinkiem), a także bardziej egzotyczne typy liczbowe (liczby zespolone z liczbami urojonymi, liczby stałoprzecinkowe, liczby wymierne z mianownikiem i licznikiem, a także pełne zbiory).

Choć oferują kilka bardziej zaawansowanych opcji, podstawowe typy liczbowe Pythona są… właśnie podstawowe. Liczby w Pythonie obsługują normalne działania matematyczne. Znak + wykonuje dodawanie, znak * mnożenie, natomiast ** potęgowanie.

```
>>> 123 + 222                          # Dodawanie liczb całkowitych
345
>>> 1.5 * 4                            # Mnożenie liczb zmiennoprzecinkowych
6.0
>>> 2 ** 100                           # 2 do potęgi 100
1267650600228229401496703205376
```

Warto zwrócić uwagę na wynik ostatniego działania. Typ liczby całkowitej Pythona 3.0 automatycznie udostępnia dodatkową precyzję dla tak dużych liczb, kiedy jest to potrzebne (w Pythonie 2.6 osobny typ długiej liczby całkowitej w podobny sposób obsługiwał liczby zbyt duże dla zwykłego typu liczby całkowitej). Można na przykład w Pythonie obliczyć 2 do potęgi 1000000 jako liczbę całkowitą (choć pewnie lepiej byłoby nie wyświetlać wyniku tego działania — z ponad trzystoma tysiącami cyfr będziemy musieli trochę poczekać!).

```
>>> len(str(2 ** 1000000))                    # Ile cyfr będzie w naprawdę DUŻEJ liczbie?
301030
```

Kiedy zaczniemy eksperymentować z liczbami zmiennoprzecinkowymi, z pewnością natkniemy się na coś, co na pierwszy rzut oka może wyglądać nieco dziwnie:

```
>>> 3.1415 * 2                                # repr: jako kod
6.2830000000000004
>>> print(3.1415 * 2)                         # str: w postaci przyjaznej dla użytkownika
6.283
```

Pierwszy wynik nie jest błędem — to kwestia sposobu wyświetlania. Okazuje się, że każdy obiekt można wyświetlić na dwa sposoby — z pełną precyzją (jak w pierwszym wyniku powyżej) oraz w formie przyjaznej dla użytkownika (jak w drugim wyniku). Pierwsza postać znana jest jako `repr` obiektu (jak w kodzie), natomiast druga jest przyjazną dla użytkownika `str`. Różnica ta zacznie mieć znaczenie, kiedy przejdziemy do używania klas. Na razie, kiedy coś będzie dziwnie wyglądało, należy wyświetlić to za pomocą wywołania wbudowanej instrukcji `print`.

Poza wyrażeniami w Pythonie znajduje się kilka przydatnych modułów liczbowych. *Moduły* są po prostu pakietami dodatkowych narzędzi, które musimy zaimportować, by móc z nich skorzystać.

```
>>> import math
>>> math.pi
3.1415926535897931
>>> math.sqrt(85)
9.2195444572928871
```

Moduł `math` zawiera bardziej zaawansowane narzędzia liczbowe w postaci funkcji, natomiast moduł `random` wykonuje generowanie liczb losowych, a także losowe wybieranie (tutaj z listy Pythona omówionej w dalszej części rozdziału):

```
>>> import random
>>> random.random()
0.59268735266273953
>>> random.choice([1, 2, 3, 4])
1
```

Python zawiera również bardziej egzotyczne obiekty liczb, takie jak liczby zespolone, liczby stałoprzecinkowe, liczby wymierne, a także zbiory i wartości Boolean (logiczne). Można również znaleźć różne dodatkowe rozszerzenia na licencji open source (na przykład dla matryc czy wektorów). Szczegółowe omówienie tych typów zostawimy sobie na później.

Jak na razie omawialiśmy Pythona w funkcji prostego kalkulatora. Żeby jednak oddać sprawiedliwość jego typom wbudowanym, warto przejść do omówienia łańcuchów znaków.

Łańcuchy znaków

Łańcuchy znaków (ang. *strings*) wykorzystywane są do przechowywania informacji tekstowych, a także dowolnych zbiorów bajtów. Są pierwszym przykładem tego, co w Pythonie znane jest pod nazwą *sekwencji* — czyli uporządkowanych zbiorów innych obiektów. Sekwencje zachowują porządek zawieranych elementów od lewej do prawej strony. Elementy te są przechowywane i pobierane zgodnie z ich pozycją względną. Mówiąc dokładnie, łańcuchy znaków to sekwencje łańcuchów składających się z pojedynczych znaków. Pozostałe typy sekwencji to omówione później listy oraz krotki.

Operacje na sekwencjach

Jako sekwencje łańcuchy znaków obsługują operacje zakładające pozycyjne uporządkowanie elementów. Jeśli na przykład mamy łańcuch czteroznakowy, możemy zweryfikować jego długość za pomocą wbudowanej funkcji `len` i pobrać jego elementy za pomocą wyrażeń *indeksujących*:

```
>>> S = 'Mielonka'
>>> len(S)                          # Długość
8
>>> S[0]                            # Pierwszy element w S, indeksowanie rozpoczyna się od zera
'M'
>>> S[1]                            # Drugi element od lewej
'i'
```

W Pythonie indeksy zakodowane są jako wartość przesunięcia od początku łańcucha, dlatego rozpoczynają się od zera. Pierwszy element znajduje się pod indeksem 0, kolejny pod indeksem 1 i tak dalej.

Warto tutaj zwrócić uwagę na przypisanie łańcucha znaków do *zmiennej* o nazwie `S`. Szczegółowe omówienie tego, jak to działa, odłożymy na później (zwłaszcza do rozdziału 6.), natomiast warto wiedzieć, że zmiennych Pythona nigdy nie trzeba deklarować z wyprzedzeniem. Zmienna tworzona jest, kiedy przypisujemy do niej wartość; można do niej przypisać dowolny typ obiektu i zostanie zastąpiona swoją wartością, kiedy pojawi się w wyrażeniu. Przed użyciem jej wartości musi najpierw zostać przypisana. Na cele niniejszego rozdziału wystarczy nam wiedza, że by móc zapisać obiekt w celu późniejszego użycia, musimy przypisać ten obiekt do zmiennej.

W Pythonie można również indeksować od końca — indeksy dodatnie odliczane są od lewej strony do prawej, natomiast ujemne od prawej do lewej:

```
>>> S[-1]                           # Pierwszy element od końca S
'a'
>>> S[-2]                           # Drugi element od końca S
'k'
```

Z formalnego punktu widzenia indeks ujemny jest po prostu dodawany do rozmiaru łańcucha znaków, dzięki czemu dwie poniższe operacje są równoważne (choć pierwszą łatwiej jest zapisać, a trudniej się w niej pomylić):

```
>>> S[-1]                           # Ostatni element w S
'a'
>>> S[len(S)-1]                     # Indeks ujemny zapisany w trudniejszy sposób
'a'
```

Warto zauważyć, że w nawiasach kwadratowych możemy wykorzystać dowolne wyrażenie, a nie tylko zakodowany na stałe literał liczbowy. W każdym miejscu, w którym Python oczekuje wartości, można użyć literału, zmiennej lub dowolnego wyrażenia. Składnia Pythona jest w tym zakresie bardzo ogólna.

Poza prostym indeksowaniem zgodnie z pozycją sekwencje obsługują również bardziej ogólną formę indeksowania znaną jako *wycinki* (ang. *slice*). Polega ona na ekstrakcji całej części (wycinka) za jednym razem, jak na przykład:

```
>>> S                                  # Łańcuch z ośmioma znakami
'Mielonka'
>>> S[1:3]                             # Wycinek z S z indeksami od 1 do 2 (bez 3)
'ie'
```

Wycinki najłatwiej jest sobie wyobrazić jako sposób pozwalający na ekstrakcję całej *kolumny* z łańcucha znaków za jednym razem. Ich ogólna forma, `X[I:J]`, oznacza: „Zwróć wszystko z `X` od przesunięcia `I` aż do przesunięcia `J`, ale bez niego". Wynik zwracany jest w nowym obiekcie. Druga operacja z listingu wyżej zwraca na przykład wszystkie znaki w łańcuchu znaków `S` od przesunięcia 1 do przesunięcia 2 (czyli 3–1) jako nowy łańcuch znaków. Wynikiem jest wycinek składający się z dwóch znaków znajdujących się w środku łańcucha `S`.

W wycinku lewą granicą jest domyślnie zero, natomiast prawą — długość sekwencji, z której coś wycinamy. Dzięki temu można spotkać różne warianty użycia wycinków:

```
>>> S[1:]                              # Wszystko poza pierwszym znakiem (1:len(S))
'ielonka'
>>> S                                  # Łańcuch S się nie zmienił
'Mielonka'
>>> S[0:7]                             # Wszystkie elementy bez ostatniego
'Mielonk'
>>> S[:7]                              # To samo co S[0:7]
'Mielonk'
>>> S[:-1]                             # Wszystkie elementy bez ostatniego w łatwiejszej postaci
'Mielonk'
>>> S[:]                               # Całość S jako kopia najwyższego poziomu (0:len(S))
'Mielonka'
```

Warto zwrócić uwagę na to, że do ustalenia granic wycinków mogą również zostać wykorzystane ujemne wartości przesunięcia. Widać również, że ostatnia operacja w rezultacie kopiuje cały łańcuch znaków. Jak się okaże później, kopiowanie łańcucha znaków nie ma sensu, jednak to polecenie może się przydać w przypadku sekwencji takich, jak listy.

Wreszcie tak samo jak pozostałe sekwencje, łańcuchy znaków obsługują również *konkatenację* (ang. *concatenation*) z użyciem znaku + (łączącą dwa łańcuchy znaków w jeden), a także *powtórzenie* (ang. *repetition*), czyli zbudowanie nowego łańcucha znaków poprzez powtórzenie innego:

```
>>> S
'Mielonka'
>>> S + 'xyz'                          # Konkatenacja
'Mielonkaxyz'
>>> S                                  # S pozostaje bez zmian
'Mielonka'
>>> S * 8                              # Powtórzenie
'MielonkaMielonkaMielonkaMielonkaMielonkaMielonkaMielonkaMielonka'
```

Warto zwrócić uwagę na to, że znak plusa (+) oznacza coś innego w przypadku różnych obiektów — dodawanie dla liczb, a konkatenację dla łańcuchów znaków. Jest to właściwość Pythona, którą w dalszej części książki będziemy nazywali *polimorfizmem*. Oznacza to, że każda

operacja uzależniona jest od typu obiektów, na jakich się ją wykonuje. Jak się okaże przy okazji omawiania typów dynamicznych, polimorfizm odpowiada za zwięzłość i elastyczność kodu napisanego w Pythonie. Ponieważ typy nie są ograniczone, operacja napisana w Pythonie może zazwyczaj automatycznie działać na wielu różnych typach obiektów, o ile obsługują one zgodny z nią interfejs (jak operacja+powyżej). W Pythonie jest to bardzo ważna koncepcja, do której jeszcze powrócimy.

Niezmienność

Warto zauważyć, że w poprzednich przykładach żadna operacja wykonana na łańcuchu znaków nie zmieniła oryginalnego łańcucha. Każda operacja na łańcuchach znaków zdefiniowana jest w taki sposób, by w rezultacie zwracać nowy łańcuch znaków, ponieważ łańcuchy są w Pythonie *niezmienne* (ang. *immutable*). Nie mogą być zmienione w miejscu już po utworzeniu. Nie można na przykład zmienić łańcucha znaków, przypisując go do jednej z jego pozycji. Można jednak zawsze stworzyć nowy łańcuch znaków i przypisać go do tej samej nazwy (zmiennej). Ponieważ Python czyści stare obiekty w miarę przechodzenia dalej (o czym przekonamy się nieco później), nie jest to aż tak niewydajne, jak mogłoby się wydawać.

```
>>> S
'Mielonka'
>>> S[0] = 'z'                              # Niezmienne obiekty nie mogą być modyfikowane
...pominięto tekst błędu...
TypeError: 'str' object does not support item assignment

>>> S = 'z' + S[1:]                         # Można jednak tworzyć wyrażenia budujące nowe obiekty
>>> S
'zielonka'
```

Każdy obiekt Pythona klasyfikowany jest jako niezmienny (niemodyfikowalny) bądź zmienny (modyfikowalny). Wśród typów podstawowych niezmienne są liczby, łańcuchy znaków oraz krotki. Listy i słowniki można dowolnie zmieniać w miejscu. Niezmienność można wykorzystać do zagwarantowania, że obiekt pozostanie stały w czasie całego cyklu działania programu.

Metody specyficzne dla typu

Każda z omówionych dotychczas operacji na łańcuchach znaków jest tak naprawdę operacją na sekwencjach, która będzie działała na innych typach sekwencji Pythona, w tym na listach i krotkach. Poza tymi wspólnymi dla sekwencji operacjami łańcuchy znaków obsługują również własne operacje dostępne w formie *metod* (funkcji dołączanych do obiektu i wywoływanych za pomocą odpowiedniego wyrażenia).

Metoda `find` jest na przykład podstawową operacją działającą na podłańcuchach znaków (zwraca ona wartość przesunięcia przekazanego podłańcucha znaków lub `-1`, jeśli podłańcuch ten nie jest obecny). Metoda `replace` służy z kolei do globalnego odszukiwania i zastępowania.

```
>>> S.find('ie')                            # Odnalezienie przesunięcia podłańcucha
1
>>> S
'Mielonka'

>>> S.replace('ie', 'XYZ')                  # Zastąpienie wystąpień podłańcucha innym
'MXYZlonka'
>>> S
'Mielonka'
```

I znów, mimo że w metodach łańcucha znaków występuje nazwa zmiennej, nie zmieniamy oryginalnych łańcuchów znaków, lecz tworzymy nowe. Ponieważ łańcuchy znaków są niezmienne, trzeba to robić właśnie w taki sposób. Metody łańcuchów znaków to pierwsza pomoc w przetwarzaniu tekstu w Pythonie. Inne dostępne metody pozwalają dzielić łańcuch znaków na podłańcuchy w miejscu wystąpienia ogranicznika (co przydaje się w analizie składniowej), zmieniać wielkość liter, sprawdzać zawartość łańcuchów znaków (cyfry, litery i inne znaki) lub usuwać białe znaki (ang. *whitespace*) z końca łańcucha.

```
>>> line = 'aaa,bbb,ccccc,dd'
>>> line.split(',')                    # Podzielenie na ograniczniku na listę podłańcuchów
['aaa', 'bbb', 'ccccc', 'dd']
>>> S = 'mielonka'
>>> S.upper()                          # Konwersja na wielkie litery
'MIELONKA'

>>> S.isalpha()                        # Sprawdzenie zawartości: isalpha, isdigit
True

>>> line = 'aaa,bbb,ccccc,dd\n'
>>> line = line.rstrip()               # Usunięcie białych znaków po prawej stronie
>>> line
'aaa,bbb,ccccc,dd'
```

Łańcuchy znaków obsługują również zaawansowane operacje zastępowania znane jako *formatowanie*, dostępne zarówno w postaci wyrażeń (oryginalne rozwiązanie), jak i wywołania metody łańcuchów znaków (nowość w wersjach 2.6 oraz 3.0):

```
>>> '%s, jajka i %s' % ('mielonka', 'MIELONKA!')        # Wyrażenie formatujące (wszystkie wersje)
'mielonka, jajka i MIELONKA!'
>>> '{0}, jajka i {1}'.format('mielonka', 'MIELONKA!') # Metoda formatująca (2.6, 3.0)
'mielonka, jajka i MIELONKA!'
```

Jedna uwaga: choć operacje na sekwencjach są uniwersalne, metody takie nie są — choć niektóre typy współdzielą pewne nazwy metod. Metody łańcuchów znaków działają wyłącznie na łańcuchach znaków i na niczym innym. Generalnie zbiór narzędzi Pythona składa się z warstw — uniwersalne operacje, które dostępne są dla większej liczby typów danych, występują jako wbudowane funkcje lub wyrażenia (na przykład `len(X)`, `X[0]`), natomiast operacje specyficzne dla określonego typu są wywołaniami metod (jak `aString.upper()`). Odnalezienie odpowiednich narzędzi w tych kategoriach stanie się bardziej naturalne z czasem, natomiast w kolejnym podrozdziale znajduje się kilka wskazówek przydatnych już teraz.

Otrzymanie pomocy

Metody wprowadzone w poprzednim podrozdziale są reprezentatywną, choć dość niewielką próbką tego, co dostępne jest dla łańcuchów znaków. Generalnie w niniejszej książce omówienie metod obiektów nie jest kompletne. W celu uzyskania większej liczby szczegółów zawsze można wywołać wbudowaną funkcję `dir`, która zwraca listę wszystkich atrybutów dostępnych dla danego obiektu. Ponieważ metody są atrybutami obiektów, zostaną na tej liście wyświetlone. Zakładając, że zmienna `S` nadal jest łańcuchem znaków, oto jej atrybuty w Pythonie 3.0 (w wersji 2.6 nieznacznie się one różnią):

```
>>> dir(S)
['__add__', '__class__', '__contains__', '__delattr__', '__doc__', '__eq__',
↪'__format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__',
↪'__gt__', '__hash__', '__init__', '__iter__', '__le__', '__len__', '__lt__',
↪'__mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',
```

```
↪'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
↪'__subclasshook__', '_formatter_field_name_split', '_formatter_parser',
↪'capitalize', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
↪'format', 'index', 'isalnum','isalpha', 'isdecimal', 'isdigit', 'isidentifier',
↪'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',
↪'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
↪'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
↪'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

Nazwy z dwoma znakami _ nie będą nas raczej interesować aż do późniejszej części książki, kiedy będziemy omawiać przeciążanie operatorów w klasach. Reprezentują one implementację obiektu łańcucha znaków i są dostępne w celu ułatwienia dostosowania obiektu do własnych potrzeb. Ogólna reguła jest taka, że podwójne znaki _ na początku i końcu to konwencja wykorzystywana w Pythonie do szczegółów implementacyjnych. Nazwy bez tych znaków są metodami, które można wywoływać na łańcuchach znaków.

Funkcja `dir` podaje same nazwy metod. Żeby dowiedzieć się, co te metody robią, można przekazać je do funkcji `help`.

```
>>> help(S.replace)
Help on built-in function replace:

replace(...)
   S.replace (old, new[, count]) -> str
   Return a copy of S with all occurrences of substring old replaced by new. If the
   ↪optional argument count is given, only the first count occurrences are replaced.
```

Funkcja `help` jest jednym z kilku interfejsów do systemu kodu udostępnianego wraz z Pythonem i znanego pod nazwą *PyDoc* — narzędzia służącego do pobierania dokumentacji z obiektów. W dalszej części książki pokażemy, że PyDoc może również zwracać swoje raporty w formacie HTML.

Można również próbować uzyskać pomoc dla całego łańcucha znaków (na przykład `help(S)`), jednak w rezultacie otrzymamy więcej, niż chcielibyśmy zobaczyć, to znaczy informacje o każdej metodzie łańcuchów znaków. Lepiej jest pytać o określoną metodę, jak na listingu powyżej.

Więcej szczegółowych informacji można również znaleźć w dokumentacji biblioteki standardowej Pythona lub publikowanych komercyjnie książkach, jednak to `dir` oraz `help` są w Pythonie pierwszym źródłem pomocy.

Inne sposoby kodowania łańcuchów znaków

Dotychczas przyglądaliśmy się operacjom na sekwencjach na przykładzie łańcuchów znaków, a także metodom przeznaczonym dla tego typu danych. Python udostępnia również różne sposoby kodowania łańcuchów znaków, które bardziej szczegółowo omówione zostaną nieco później. Przykładowo znaki specjalne można reprezentować jako sekwencje ucieczki z ukośnikiem lewym.

```
>>> S = 'A\nB\tC'                    # \n to koniec wiersza, \t to tabulator
>>> len(S)                          # Każde jest jednym znakiem
5

>>> ord('\n')                       # \n jest bajtem z wartością binarną 10 w ASCII
10
```

```
>>> S = 'A\OB\OC'                          # \0, binarny bajt zerowy, nie kończy łańcucha znaków
>>> len(S)
5
```

Python pozwala na umieszczanie łańcuchów znaków zarówno w cudzysłowach, jak i apostrofach (oznaczają one to samo). Zawiera również wielowierszową postać literału łańcucha znaków umieszczaną w trzech cudzysłowach lub trzech apostrofach. Kiedy użyta zostanie ta forma, wszystkie wiersze są ze sobą łączone, a w miejscu złamania wiersza dodawane są odpowiednie znaki złamania wiersza. Jest to pewna konwencja syntaktyczna, która przydaje się do osadzania kodu HTML czy XML w skrypcie Pythona.

```
>>> msg = """ aaaaaaaaaaaaa
bbb'''bbbbbbbbbb""bbbbbbb'bbbb
cccccccccccccc"""
>>> msg
'\naaaaaaaaaaaaa\nbbb\'\'\'bbbbbbbbbb""bbbbbbb\'bbbb\ncccccccccccccc'
```

Python obsługuje również „surowy" literał łańcucha znaków wyłączający mechanizm ucieczki z ukośnikiem lewym (rozpoczyna się on od litery `r`), a także format Unicode obsługujący znaki międzynarodowe. W wersji 3.0 podstawowy typ łańcucha znaków `str` obsługuje także znaki Unicode (co ma sens, biorąc pod uwagę, że tekst ASCII jest po prostu częścią Unicode), natomiast typ `bytes` reprezentuje surowe łańcuchy bajtowe. W Pythonie 2.6 Unicode jest odrębnym typem, natomiast `str` obsługuje zarówno łańcuchy ośmiobitowe, jak i dane binarne. W wersji 3.0 zmieniły się także pliki zwracające i przyjmujące `str` dla tekstu i `bytes` dla danych binarnych. Ze wszystkimi specjalnymi formami łańcuchów znaków spotkamy się w kolejnych rozdziałach.

Dopasowywanie wzorców

Zanim przejdziemy dalej, warto odnotować, że żadna z metod łańcuchów znaków nie obsługuje przetwarzania tekstu opartego na wzorcach. Dopasowywanie wzorców tekstowych jest zaawansowanym narzędziem pozostającym poza zakresem niniejszej książki, jednak osoby znające inne skryptowe języki programowania pewnie będą zainteresowane tym, że dopasowywanie wzorców w Pythonie odbywa się z wykorzystaniem modułu o nazwie `re`. Moduł ten zawiera analogiczne wywołania dla wyszukiwania, dzielenia czy zastępowania, jednak dzięki użyciu wzorców do określania podłańcuchów znaków możemy być o wiele bardziej ogólni.

```
>>> import re
>>> match = re.match('Witaj,[ \t]*(.*)Robinie', 'Witaj, sir   Robinie')
>>> match.group(1)
'sir '
```

Powyższy przykład szuka podłańcucha znaków zaczynającego się od słowa `Witaj,`, po którym następuje od zera do większej liczby tabulatorów lub spacji, po nich dowolne znaki, które będą zapisane jako dopasowana grupa, a na końcu słowo `Robinie`. Kiedy taki podłańcuch zostanie odnaleziony, jego części dopasowane przez części wzorca umieszczone w nawiasach dostępne są jako grupy. Poniższy wzorzec wybiera na przykład trzy grupy rozdzielone ukośnikami prawymi:

```
>>> match = re.match('/(.*)/(.*)/(.*)', '/usr/home/drwal')
>>> match.groups()
('usr', 'home', 'drwal')
```

Dopasowywanie wzorców samo w sobie jest dość zaawansowanym narzędziem do przetwarzania tekstu, natomiast w Pythonie udostępniana jest nawet obsługa jeszcze bardziej

zaawansowanych technik przetwarzania tekstu i języka, w tym analizy składniowej XML i analizy języka naturalnego. Dość jednak informacji o łańcuchach znaków — czas przejść do kolejnego typu danych.

Listy

Obiekt listy jest w Pythonie najbardziej ogólnym rodzajem sekwencji dostępnym w tym języku. Listy to uporządkowane pod względem pozycji zbiory obiektów dowolnego typu; nie mają one ustalonej wielkości. Są również *zmienne* (ang. *mutable*) — w przeciwieństwie do łańcuchów znaków listy można modyfikować w miejscu, przypisując coś do odpowiednich wartości przesunięcia, a także za pomocą różnych metod.

Operacje na sekwencjach

Ponieważ listy są sekwencjami, obsługują wszystkie operacje na sekwencjach przedstawione przy okazji omawiania łańcuchów znaków. Jedyną różnicą jest to, że wynikiem zazwyczaj są listy, a nie łańcuchy znaków. Mając na przykład listę trzyelementową:

```
>>> L = [123, 'mielonka', 1.23]          # Trzyelementowa lista z elementami różnego typu
>>> len(L)                               # Liczba elementów listy
3
```

możemy ją zindeksować, sporządzić z niej wycinki i wykonać pozostałe operacje zaprezentowane na przykładzie łańcuchów znaków:

```
>>> L[O]                                 # Indeksowanie po pozycji
123

>>> L[:-1]                               # Wycinkiem listy jest nowa lista
[123, 'mielonka']

>>> L + [4, 5, 6]                        # Konkatenacja także zwraca nową listę
[123, 'mielonka', 1.23, 4, 5, 6]

>>> L                                    # Oryginalna lista pozostaje bez zmian
[123, 'mielonka', 1.23]
```

Operacje specyficzne dla typu

Listy Pythona powiązane są z tablicami obecnymi w innych językach programowania, jednak zazwyczaj mają większe możliwości. Przede wszystkim nie mają żadnych ograniczeń odnośnie typów danych — przykładowo lista z listingu powyżej zawiera trzy obiekty o zupełnie różnych typach (liczbę całkowitą, łańcuch znaków i liczbę zmiennoprzecinkową). Co więcej, wielkość listy nie jest określona. Oznacza to, że może ona rosnąć i kurczyć się zgodnie z potrzebami i w odpowiedzi na operacje specyficzne dla list.

```
>>> L.append('NI')                       # Rośnięcie: dodanie obiektu na końcu listy
>>> L
[123, 'mielonka', 1.23, 'NI']

>>> L.pop(2)                             # Kurczenie się: usunięcie obiektu ze środka listy
1.23

>>> L                                    # "del L[2]" również usuwa element z listy
[123, 'mielonka', 'NI']
```

Na powyższym listingu metoda `append` rozszerza rozmiar listy i wstawia na jej końcu element. Metoda `pop` (lub jej odpowiednik, instrukcja `del`) usuwa następnie element znajdujący się na pozycji o podanej wartości przesunięcia, co sprawia, że lista się kurczy. Pozostałe metody list pozwalają na wstawienie elementu w dowolnym miejscu listy (`insert`) czy usunięcie elementu o podanej wartości (`remove`). Ponieważ listy są zmienne, większość ich metod zmienia obiekt listy w miejscu, a nie tworzy nowy obiekt.

```
>>> M = ['bb', 'aa', 'cc']
>>> M.sort()
>>> M
['aa', 'bb', 'cc']

>>> M.reverse()
>>> M
['cc', 'bb', 'aa']
```

Zaprezentowana wyżej metoda listy o nazwie `sort` porządkuje listę w kolejności rosnącej, natomiast metoda `reverse` odwraca ją. W obu przypadkach lista modyfikowana jest w sposób bezpośredni.

Sprawdzanie granic

Choć listy nie mają ustalonej wielkości, Python nie pozwala na odnoszenie się do elementów, które nie istnieją. Indeksowanie poza końcem listy zawsze jest błędem, podobnie jak przypisywanie elementu poza jej końcem.

```
>>> L
[123, 'mielonka', 'NI']

>>> L[99]
...pominięto tekst błędu...
IndexError: list index out of range

>>> L[99] = 1
...pominięto tekst błędu...
IndexError: list assignment index out of range
```

Takie rozwiązanie jest celowe, ponieważ zazwyczaj błędem jest próba przypisania elementu poza końcem listy (szczególnie paskudne jest to w języku C, który nie sprawdza błędów w tak dużym stopniu jak Python). Zamiast w odpowiedzi po cichu powiększać listę, Python zgłasza błąd. By powiększyć listę, należy zamiast tego wywołać odpowiednią metodę, taką jak `append`.

Zagnieżdżanie

Jedną z miłych właściwości podstawowych typów danych w Pythonie jest obsługa dowolnego zagnieżdżania. Możliwe jest zagnieżdżanie tych obiektów w dowolnej kombinacji i na dowolną głębokość (czyli można na przykład utworzyć listę zawierającą słownik, który z kolei zawiera kolejną listę). Bezpośrednim zastosowaniem tej właściwości jest reprezentowanie w Pythonie macierzy czy, inaczej, tablic wielowymiarowych. Lista mieszcząca zagnieżdżone listy doskonale się sprawdzi w prostych aplikacjach:

```
>>> M = [[1, 2, 3],                 # Macierz 3x3 w postaci list zagnieżdżonych
         [4, 5, 6],                 # Kod może się rozciągać na kilka wierszy, jeśli znajduje się w nawiasach
         [7, 8, 9]]
>>> M
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Powyżej zakodowaliśmy listę zawierającą trzy inne listy. W rezultacie otrzymujemy reprezentację macierzy liczb 3×3. Dostęp do takiej struktury można uzyskać na wiele sposobów:

```
>>> M[1]                          # Pobranie drugiego wiersza
[4, 5, 6]

>>> M[1][2]                       # Pobranie drugiego wiersza, a z niego trzeciego elementu
6
```

Pierwsza operacja pobiera cały drugi wiersz, natomiast druga pobiera trzeci element z tego wiersza. Łączenie ze sobą indeksów wciąga nas coraz głębiej i głębiej w zagnieżdżoną strukturę.[2]

Listy składane

Poza operacjami dla sekwencji i metodami list Python zawiera bardziej zaawansowaną operację znaną pod nazwą *list składanych* (ang. *list comprehension*), która świetnie się przydaje do przetwarzania struktur takich, jak nasza macierz. Przypuśćmy na przykład, że potrzebujemy pobrać drugą kolumnę naszej prostej macierzy. Łatwo jest za pomocą zwykłego indeksowania pobierać wiersze, ponieważ macierz przechowywana jest wierszami. Podobnie łatwo jest pobrać kolumnę za pomocą wyrażenia listy składanej:

```
>>> col2 = [row[1] for row in M]  # Zebranie elementów z drugiej kolumny
>>> col2
[2, 5, 8]

>>> M                             # Macierz pozostaje bez zmian
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Listy składane pochodzą z notacji zbiorów. Służą do budowania nowej listy poprzez wykonanie wyrażenia na każdym elemencie po kolei, jeden po drugim, od lewej strony do prawej. Listy składane kodowane są w nawiasach kwadratowych (by podkreślić fakt, że tworzą listę) i składają się z wyrażenia i konstrukcji pętli, które dzielą ze sobą nazwę zmiennej (tutaj `row`). Powyższa lista składana oznacza mniej więcej: „Zwróć `row[1]` dla każdego wiersza w macierzy `M` w nowej liście". Wynikiem jest nowa lista zawierająca drugą kolumnę macierzy.

Listy składane mogą w praktyce być bardziej skomplikowane:

```
>>> [row[1] + 1 for row in M]     # Dodanie 1 do każdego elementu w drugiej kolumnie
[3, 6, 9]

>>> [row[1] for row in M if row[1] % 2 == 0] # Odfiltrowanie elementów nieparzystych
[2, 8]
```

Pierwsza operacja z listingu powyżej dodaje 1 do każdego pobranego elementu, natomiast druga wykorzystuje wyrażenie `if` do odfiltrowania liczb nieparzystych z wyniku, używając do tego reszty z dzielenia (wyrażenia z `%`). Listy składane tworzą nowe listy wyników, jednak mogą również zostać wykorzystane do iteracji po dowolnym obiekcie, na którym jest to możliwe. Listing poniżej prezentuje użycie list składanych do przejścia zakodowanej listy współrzędnych oraz łańcucha znaków:

[2] Taka struktura macierzy jest dobra dla zadań na małą skalę, jednak w przypadku poważniejszego przetwarzania liczb lepiej będzie skorzystać z jednego z rozszerzeń liczbowych do Pythona, takiego jak system *NumPy* na licencji open source. Takie narzędzia są w stanie przechowywać i przetwarzać duże macierze o wiele bardziej wydajnie od zagnieżdżonej struktury list. O NumPy mówi się, że zmienia Pythona w darmowy i bardziej zaawansowany odpowiednik systemu MatLab, a organizacje takie, jak NASA, Los Alamos czy JPMorgan Chase wykorzystują to narzędzie w swojej pracy naukowej i finansowej. Więcej informacji na ten temat można znaleźć w Internecie.

```
>>> diag = [M[i][i] for i in [0, 1, 2]]       # Pobranie przekątnej z macierzy
>>> diag
[1, 5, 9]

>>> doubles = [c * 2 for c in 'mielonka']     # Powtórzenie znaków w łańcuchu
>>> doubles
['mm', 'ii', 'ee', 'll', 'oo', 'nn', 'kk', 'aa']
```

Listy składane i pokrewne konstrukcje, takie jak funkcje wbudowane `map` oraz `filter`, są trochę zbyt zaawansowane, by omawiać je teraz bardziej szczegółowo. Najważniejsze w tym krótkim wprowadzeniu jest to, że Python zawiera zarówno narzędzia podstawowe, jak i bardziej zaawansowane. Listy składane są opcjonalne, jednak w praktyce często okazują się przydatne i nierzadko pozwalają zyskać na szybkości przetwarzania. Działają na każdym typie danych będącym sekwencją, a także na pewnych typach, które sekwencjami nie są. Wrócimy do nich w dalszej części książki.

Jako zapowiedź tego, co będzie dalej, możemy napisać, że w nowszych wersjach Pythona składnia złożeń w nawiasach może być wykorzystywana do tworzenia *generatorów* produkujących wyniki na żądanie (wbudowana funkcja `sum` sumuje na przykład elementy w sekwencji):

```
>>> G = (sum(row) for row in M)               # Utworzenie generatora sum wierszy
>>> next(G)                                   # iter(G) nie jest tu wymagane
6
>>> next(G)                                   # Wykonanie protokołu iteracji
15
```

Funkcja wbudowana `map` wykonuje podobne zadanie, generując wyniki wykonania funkcji dla elementów. Opakowanie jej w wywołanie `list` wymusza zwrócenie wszystkich wartości w Pythonie 3.0.

```
>>> list(map(sum, M))                         # Wykonanie funkcji sum dla elementów M
[6, 15, 24]
```

W Pythonie 3.0 składnię złożeń można także wykorzystać do tworzenia *zbiorów* oraz *słowników*:

```
>>> {sum(row) for row in M}                   # Utworzenie zbioru sum wierszy
{24, 6, 15}
>>> {i : sum(M[i]) for i in range(3)}         # Utworzenie tabeli klucz-wartość sum wierszy
{0: 6, 1: 15, 2: 24}
```

Tak naprawdę listy, zbiory oraz słowniki można w wersji 3.0 tworzyć za pomocą złożeń:

```
>>> [ord(x) for x in 'mieloonka']             # Lista numerów porządkowych znaków
[109, 105, 101, 108, 111, 111, 110, 107, 97]
>>> {ord(x) for x in 'mieloonka'}             # Zbiory usuwają duplikaty
{97, 101, 105, 107, 108, 109, 110, 111}
>>> {x: ord(x) for x in 'mieloonka'}          # Klucze słownika są unikalne
{'a': 97, 'e': 101, 'i': 105, 'k': 107, 'm': 109, 'l': 108, 'o': 111, 'n': 110}
```

By jednak zrozumieć obiekty, takie jak generatory, zbiory oraz słowniki, musimy przejść nieco dalej.

Słowniki

Słowniki są w Pythonie czymś zupełnie innym — nie są zupełnie sekwencjami, są za to znane jako *odwzorowania* (ang. *mapping*). Odwzorowania są również zbiorami innych obiektów, jednak obiekty te przechowują po kluczu, a nie ich pozycji względnej. Odwzorowania nie zachowują

żadnej niezawodnej kolejności od lewej do prawej strony, po prostu łączą klucze z powiązanymi z nimi wartościami. Słowniki, jedyny typ odwzorowania w zbiorze typów podstawowych Pythona, są również zmienne — mogą być modyfikowane w miejscu i podobnie do list, mogą rosnąć i kurczyć się na życzenie.

Operacje na odwzorowaniach

Kiedy słowniki zapisze się w formie literałów, znajdują się w nawiasach klamrowych i składają się z serii par *klucz*: *wartość*. Słowniki są przydatne wtedy, gdy chcemy powiązać zbiór wartości z kluczami — na przykład opisać właściwości czegoś. Rozważmy na przykład poniższy słownik składający się z trzech elementów (których kluczami będą `jedzenie`, `ilość` i `kolor`):

```
>>> D = {'jedzenie': 'Mielonka', 'ilość': 4, 'kolor': 'różowy'}
```

W celu zmiany wartości powiązanych z kluczami słownik można zindeksować po kluczu. Operacja indeksowania słownika wykorzystuje tę samą składnię co wykorzystywana w sekwencjach, jednak elementem znajdującym się w nawiasach kwadratowych jest klucz, a nie pozycja względna.

```
>>> D['jedzenie']                          # Pobranie wartości klucza "jedzenie"
'Mielonka'

>>> D['ilość'] += 1                        # Dodanie 1 do wartości klucza "ilość"
>>> D
{'jedzenie': 'Mielonka', 'kolor': 'różowy', 'ilość': 5}
```

Choć forma literału z nawiasami klamrowymi jest czasami używana, częściej widzi się słowniki tworzone w inny sposób. Poniższy przykład rozpoczyna się od pustego słownika, który jest następnie wypełniany po jednym kluczu na raz. W przeciwieństwie do przypisania poza granicami listy, co jest zakazane, przypisania do nowych kluczy słownika tworzą te klucze.

```
>>> D = {}
>>> D['imię'] = 'Robert'                   # Tworzenie kluczy przez przypisanie
>>> D['zawód'] = 'programista'
>>> D['wiek'] = 40

>>> D
{'wiek': 40, 'zawód': 'programista', 'imię': 'Robert'}

>>> print(D['imię'])
Robert
```

Powyżej wykorzystaliśmy klucze jako nazwy pól w spisie opisującym kogoś. W innych zastosowaniach słowniki mogą być również wykorzystywane do zastępowania operacji wyszukiwania — zindeksowanie słownika po kluczu jest często najszybszą metodą zakodowania wyszukiwania w Pythonie. Jak dowiemy się później, słowniki można także tworzyć przez przekazywanie argumentów będących słowami kluczowymi do nazwy typu, jak w przykładzie `dict(imię='Robert', zawód='programista', wiek=40)` tworzącym ten sam słownik.

Zagnieżdżanie raz jeszcze

W poprzednim przykładzie słownik wykorzystaliśmy do opisania hipotetycznej osoby za pomocą trzech kluczy. Załóżmy jednak, że informacje są bardziej złożone. Być może konieczne będzie zanotowanie imienia i nazwiska, a także kilku zawodów czy tytułów. Prowadzi to do

innego zastosowania zagnieżdżania obiektów w praktyce. Poniższy słownik — zakodowany od razu jako literał — zawiera bardziej ustrukturyzowane informacje:

```
>>> rec = {'dane osobowe': {'imię': 'Robert', 'nazwisko': 'Zielony'},
           'zawód': ['programista', 'inżynier'],
           'wiek': 40.5}
```

W powyższym przykładzie znowu mamy na górze słownik z trzema kluczami (o kluczach `dane osobowe`, `zawód` oraz `wiek`), natomiast wartości stają się nieco bardziej skomplikowane. Zagnieżdżony słownik z danymi osobowymi może pomieścić kilka informacji, natomiast zagnieżdżona lista z zawodem mieści kilka ról i można ją w przyszłości rozszerzyć. Dostęp do elementów tej struktury można uzyskać w podobny sposób jak w przypadku pokazanej wcześniej macierzy, jednak tym razem indeksami będą klucze słownika, a nie wartości przesunięcia listy.

```
>>> rec['dane osobowe']                    # 'dane osobowe' to zagnieżdżony słownik
{'nazwisko': 'Zielony', 'imię': 'Robert'}

>>> rec['dane osobowe']['nazwisko']        # Indeksowanie zagnieżdżonego słownika
'Zielony'

>>> rec['zawód']                           # 'zawód' to zagnieżdżona lista
['programista', 'inżynier']
>>> rec['zawód'][-1]                       # Indeksowanie zagnieżdżonej listy
'inżynier'

>>> rec['zawód'].append('leśniczy')        # Rozszerzenie listy zawodów Roberta
>>> rec
{'wiek': 40.5, 'zawód': ['programista', 'inżynier', 'leśniczy'], 'dane osobowe':
↪{'nazwisko': 'Zielony', 'imię': 'Robert'}}
```

Warto zwrócić uwagę na to, jak ostatnia operacja rozszerza osadzoną listę z zawodami. Ponieważ lista zawodów jest fragmentem pamięci oddzielnym od zawierającego ją słownika, może dowolnie rosnąć i kurczyć się (rozkład pamięci obiektów omówiony zostanie w dalszej części książki).

Prawdziwym powodem pokazania tego przykładu jest chęć zademonstrowania *elastyczności* podstawowych typów obiektów Pythona. Jak widać, zagnieżdżanie pozwala na budowanie skomplikowanych struktur informacji w sposób łatwy i bezpośredni. Zbudowanie podobnej struktury w języku niskiego poziomu, takim jak C, byłoby żmudne i wymagałoby o wiele większej ilości kodu. Musielibyśmy zaprojektować i zadeklarować układ struktury i tablic, wypełnić je wartościami i wreszcie połączyć wszystko ze sobą. W Pythonie wszystko to dzieje się automatycznie — jedno wyrażenie tworzy za nas całą zagnieżdżoną strukturę obiektów. Tak naprawdę jest to jedna z najważniejszych zalet języków skryptowych, takich jak Python.

Co równie ważne, w języku niższego poziomu musielibyśmy uważnie czyścić przestrzeń zajmowaną przez obiekty, które jej już dłużej nie potrzebują. W Pythonie, kiedy znika ostatnia referencja do obiektu (na przykład gdy do zmiennej przypisze się coś innego), całe miejsce w pamięci zajmowane przez tę strukturę obiektu jest automatycznie zwalniane.

```
>>> rec = 0                                # Miejsce zajmowane przez obiekt zostaje odzyskane
```

Z technicznego punktu widzenia Python korzysta z czegoś, co znane jest pod nazwą *czyszczenia pamięci* (ang. *garbage collection*). Nieużywana pamięć jest czyszczona w miarę wykonywania programu, co zwalnia nas z odpowiedzialności za zarządzanie takimi szczegółami w naszym kodzie. W Pythonie miejsce to jest odzyskiwane natychmiast, kiedy tylko zniknie ostatnia referencja do

obiektu. W dalszej części książki omówimy tę kwestię bardziej szczegółowo. Na razie powinna nam wystarczyć wiedza, że możemy swobodnie korzystać z obiektów, bez konieczności troszczenia się o tworzenie dla nich miejsca w pamięci i zwalnianie go.[3]

Sortowanie kluczy — pętle for

Słowniki, będąc odwzorowaniami, obsługują jedynie dostęp do elementów po ich kluczu. Obsługują jednak również operacje specyficzne dla tego typu z wywołaniami metod przydatnymi w wielu często spotykanych sytuacjach.

Jak wspomniano wcześniej, ponieważ słowniki nie są sekwencjami, nie zachowują żadnej kolejności elementów od lewej do prawej strony. Oznacza to, że jeśli utworzymy słownik i wyświetlimy jego zawartość, klucze mogą zostać zwrócone w innej kolejności, niż je wpisaliśmy.

```
>>> D = {'a': 1, 'b': 2, 'c': 3}
>>> D
{'a': 1, 'c': 3, 'b': 2}
```

Co jednak można zrobić, kiedy potrzebne nam jest wymuszenie określonej kolejności elementów słownika? Jednym z często spotykanych rozwiązań jest tu pobranie listy kluczy za pomocą metody słownika `keys`, posortowanie tej listy za pomocą metody listy `sort`, a następnie przejście wyników za pomocą pętli `for`. Należy pamiętać o dwukrotnym naciśnięciu przycisku *Enter* po utworzeniu poniższego kodu pętli — zgodnie z informacjami z rozdziału 3. pusty wiersz oznacza w sesji interaktywnej „dalej”, natomiast w niektórych interfejsach znak zachęty się zmienia.

```
>>> Ks = list(D.keys())                    # Nieuporządkowana lista kluczy
>>> Ks
['a', 'c', 'b']

>>> Ks.sort()                              # Posortowana lista kluczy
>>> Ks
['a', 'b', 'c']

>>> for key in Ks:                         # Iteracja przez posortowane klucze
        print(key, '=>', D[key])           # <== Tutaj należy dwukrotnie nacisnąć Enter

a => 1
b => 2
c => 3
```

Ten proces składa się jednak z trzech etapów, natomiast jak zobaczymy w kolejnych rozdziałach, w nowszych wersjach Pythona można go wykonać w jednym etapie — dzięki nowszej funkcji wbudowanej o nazwie `sorted`. Wywołanie `sorted` zwraca wynik i sortuje różne typy obiektów; w tym przypadku automatycznie sortuje klucze słownika.

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> for key in sorted(D):
```

[3] Jedna uwaga: należy pamiętać, że utworzony przed chwilą spis `rec` mógłby być prawdziwym wpisem do bazy danych, gdybyśmy użyli systemu *trwałości obiektów* (ang. *object persistence*) Pythona — łatwego sposobu przechowywania obiektów Pythona w plikach lub bazach danych dostępnych za pomocą klucza. Nie będziemy teraz omawiać szczegółów tego rozwiązania; po więcej informacji na ten temat należy sięgnąć do omówionych w dalszej części książki modułów `pickle` oraz `shelve`.

```
        print(key, '=>', D[key])

a => 1
b => 2
c => 3
```

Poza zaprezentowaniem słowników powyższy przypadek posłużył nam jako pretekst do wprowadzenia pętli `for` dostępnej w Pythonie. Pętla `for` jest prostym i wydajnym sposobem przejścia wszystkich elementów sekwencji i wykonania bloku kodu dla każdego elementu po kolei. Zdefiniowana przez użytkownika zmienna pętli (tutaj: `key`) wykorzystana jest do referencji do aktualnie przechodzonego elementu. Wynikiem tego kodu jest wyświetlenie kluczy i wartości słownika w kolejności posortowanej po kluczach.

Pętla `for`, a także jej bardziej ogólny kuzyn — pętla `while`, są podstawowymi sposobami kodowania powtarzalnych zadań w skryptach jako instrukcji. Tak naprawdę jednak pętla `for`, a także jej krewniak — listy składane, to operacja na sekwencjach. Zadziała na każdym obiekcie będącym sekwencją, podobnie do list składanych — a nawet na pewnych obiektach, które sekwencjami nie są. Poniżej widać przykład przechodzenia znaków w łańcuchu i wyświetlenia wersji każdego z tych znaków napisanego wielką literą.

```
>>> for c in 'mielonka':
        print(c.upper())

M
I
E
L
O
N
K
A
```

Pętla `while` Pythona jest narzędziem bardziej ogólnym i nie jest ograniczona do przechodzenia sekwencji:

```
>>> x = 4
>>> while x > 0:
        print('mielonka!' * x)
        x -= 1
mielonka!mielonka!mielonka!mielonka!
mielonka!mielonka!mielonka!
mielonka!mielonka!
mielonka!
```

Instrukcje, składnia i narzędzia powiązane z pętlami omówione zostaną bardziej szczegółowo w dalszej części książki.

Iteracja i optymalizacja

Jeśli pętla `for` z poprzedniego podrozdziału przypomina nam omówione wcześniej wyrażenie z listami składanymi, tak właśnie powinno być — oba są ogólnymi narzędziami iteracyjnymi. Tak naprawdę oba rozwiązania będą działały na każdym obiekcie zgodnym z *protokołem iteracji* — rozpowszechnioną w Pythonie koncepcją, która oznacza fizycznie przechowywaną sekwencję w pamięci czy obiekt generujący jeden element na raz w kontekście operacji iteracji. Obiekt mieści się w tej drugiej kategorii, jeśli odpowiada na wbudowane wywołanie `iter` obiektem, który posuwa się naprzód w odpowiedzi na `next`. Wyrażenie składane *generatora*, które widzieliśmy wcześniej, jest takim właśnie typem obiektu.

Na temat protokołu iteracji będę miał więcej do powiedzenia w dalszej części książki. Na razie należy pamiętać, że każde narzędzie Pythona, które przegląda obiekt od lewej do prawej strony, wykorzystuje protokół iteracji. Dlatego właśnie wywołanie `sorted` wykorzystane wyżej działa bezpośrednio na słowniku. Nie musimy wywoływać metody `keys` w celu otrzymania sekwencji, ponieważ słowniki poddają się iteracji, a `next` zwraca w ich przypadku kolejne klucze.

Oznacza to także, że każde wyrażenie list składanych — jak poniższe, obliczające kwadrat z listy liczb:

```
>>> squares = [x ** 2 for x in [1, 2, 3, 4, 5]]
>>> squares
[1, 4, 9, 16, 25]
```

zawsze może zostać zapisane w kodzie jako odpowiednik pętli `for` tworzącej listę ręcznie — poprzez dodanie do niej kolejnych przechodzonych elementów:

```
>>> squares = []
>>> for x in [1, 2, 3, 4, 5]:                # To robi lista składana
        squares.append(x ** 2)                # Wewnętrznie wykonują protokół iteracji

>>> squares
[1, 4, 9, 16, 25]
```

Lista składana, a także powiązane z nią narzędzia programowania funkcjonalnego, takie jak `map` oraz `filter`, zazwyczaj będą obecnie działały szybciej od pętli `for` (być może nawet dwa razy szybciej) — ta cecha może mieć znaczenie w przypadku większych zbiorów danych. Mimo to warto podkreślić, że pomiar wydajności może być w Pythonie dość podchwytliwy, ponieważ język ten tak dużo optymalizuje, a wydajność zmienia się z wydania na wydanie.

Najważniejszą regułą jest w Pythonie kodowanie w sposób prosty oraz czytelny i martwienie się o wydajność dopiero później, kiedy sam program już działa i przekonamy się, że wydajność jest dla nas rzeczywistym problemem. Najczęściej kod będzie wystarczająco szybki. Jeśli musimy w nim trochę pomajstrować pod kątem wydajności, Python zawiera narzędzia, które mogą nam w tym pomóc, w tym moduły `time`, `timeit` i `profile`. Więcej informacji na ten temat znajduje się w dalszej części książki, a także w dokumentacji Pythona.

Brakujące klucze — testowanie za pomocą if

Zanim przejdziemy dalej, warto odnotować jeszcze jedną kwestię dotyczącą słowników. Choć możemy przypisać wartość do nowego klucza w celu rozszerzenia słownika, próba pobrania nieistniejącego klucza jest błędem.

```
>>> D
{'a': 1, 'c': 3, 'b': 2}

>>> D['e'] = 99                               # Przypisanie nowego klucza rozszerza słownik
>>> D
{'a': 1, 'c': 3, 'b': 2, 'e': 99}

>>> D['f']                                    # Referencja do nieistniejącego klucza jest błędem
...pominięto tekst błędu...
KeyError: 'f'
```

Tego właśnie oczekujemy — zazwyczaj pobieranie czegoś, co nie istnieje, jest błędem programistycznym. Jednak w pewnych ogólnych programach nie zawsze będziemy w momencie pisania kodu wiedzieć, jakie klucze będą obecne. W jaki sposób można sobie poradzić w takim przypadku i uniknąć błędów? Jednym z rozwiązań jest sprawdzenie tego zawczasu. Wyrażenie

słownika `in` pozwala na sprawdzenie istnienia klucza i odpowiednie zachowanie w zależności od wyniku tego sprawdzenia — dzięki instrukcji `if`. Tak jak w przypadku `for`, należy pamiętać o dwukrotnym naciśnięciu przycisku *Enter* w celu wykonania instrukcji `if` w sesji interaktywnej.

```
>>> 'f' in D
False

>>> if not 'f' in D:
       print('nie ma')

nie ma
```

Na temat instrukcji `if` i ogólnej składni instrukcji powiemy więcej nieco później, jednak forma wykorzystywana w kodzie powyżej jest dość prosta. Instrukcja składa się ze słowa `if`, po nim następuje wyrażenie, którego wynik interpretowany jest jako prawdziwy lub fałszywy, a następnie blok kodu do wykonania, jeśli wyrażenie będzie prawdziwe. W pełnej formie instrukcja `if` może również zawierać instrukcję `else` z przypadkiem domyślnym, a także jedną lub większą liczbę instrukcji `elif` (od „else if") sprawdzających inne testy. Jest to podstawowe narzędzie wyboru w Pythonie, a także sposób kodowania logiki w skryptach.

Istnieją inne sposoby tworzenia słowników, które pozwalają na uniknięcie prób uzyskania dostępu do nieistniejących kluczy, w tym metoda `get` (warunkowe indeksowanie z wartością domyślną), w Pythonie 2.X metoda `has_key` (w wersji 3.0 już niedostępna), instrukcja `try` (narzędzie, z którym spotkamy się w rozdziale 10. przy okazji przechwytywania wyjątków i radzenia sobie z nimi), a także wyrażenie `if/else` (instrukcja `if` zmieszczona w jednym wierszu). Poniżej znajduje się kilka przykładów:

```
>>> value = D.get('x', 0)                  # Indeks, ale z wartością domyślną
>>> value
0
>>> value = D['x'] if 'x' in D else 0      # Forma wyrażenia if/else
>>> value
0
```

Szczegółowe informacje na temat tych alternatywnych rozwiązań odłożymy jednak do jednego z kolejnych rozdziałów. Teraz czas przejść do omawiania krotek.

Krotki

Obiekt *krotki* (ang. *tuple*) jest w przybliżeniu listą, której nie można zmodyfikować. Krotki są sekwencjami, podobnie do list, jednak są też niezmienne — tak jak łańcuchy znaków. Z punktu widzenia składni kodowane są w zwykłych nawiasach, a nie w nawiasach kwadratowych. Krotki obsługują dowolne typy danych, zagnieżdżanie i zwykłe operacje na sekwencjach.

```
>>> T = (1, 2, 3, 4)                       # Krotka z 4 elementami
>>> len(T)                                 # Długość krotki
4

>> T + (5, 6)                              # Konkatenacja
(1, 2, 3, 4, 5, 6)

>>> T[0]                                   # Indeksowanie i wycinki
1
```

Krotki mają w Pythonie 3.0 dwie metody wywoływalne specyficzne dla tego typu — nie jest ich zatem tak wiele jak w przypadku list.

```
>>> T.index(4)                    # Metody krotek — 4 znajduje się na wartości przesunięcia 3
3
>>> T.count(4)                    # 4 pojawia się raz
1
```

Podstawową wyróżniającą cechą krotek jest to, że po utworzeniu nie można ich zmodyfikować. Oznacza to, że są sekwencjami niezmiennymi.

```
>>> T[0] = 2                      # Krotki są niezmienne
...pominięto tekst błędu...
TypeError: 'tuple' object does not support item assignment
```

Tak jak listy i słowniki, krotki obsługują obiekty o mieszanych typach i zagnieżdżanie, jednak nie rosną i nie kurczą się, ponieważ są niezmienne.

```
>>> T = ('mielonka', 3.0, [11, 22, 33])
>>> T[1]
3.0
>>> T[2][1]
22
>>> T.append(4)
AttributeError: 'tuple' object has no attribute 'append'
```

Czemu służą krotki?

Po co nam zatem typ podobny do listy, który obsługuje mniejszą liczbę operacji? Szczerze mówiąc, w praktyce krotki nie są używane tak często jak listy, jednak ich niezmienność jest ich zaletą. Kiedy w programie przekazujemy zbiór obiektów w postaci listy, obiekty te mogą się w dowolnym momencie zmienić. W przypadku użycia krotki zmienić się nie mogą. Krotki nakładają pewne ograniczenia w zakresie integralności, które przydają się w programach większych od pisanych tutaj. O krotkach powiemy jeszcze w dalszej części książki. Najpierw jednak omówmy nasz ostatni główny typ podstawowy, czyli pliki.

Pliki

Obiekty plików są w Pythonie głównym interfejsem do plików zewnętrznych znajdujących się na komputerze. Są typem podstawowym, jednak nieco innym od pozostałych. Nie istnieje żadna składnia literału służąca do ich tworzenia. Zamiast tego w celu utworzenia obiektu wywołuje się wbudowaną funkcję `open`, przekazując nazwę pliku zewnętrznego jako łańcuch znaków wraz z łańcuchem określającym tryb przetwarzania. By na przykład utworzyć plik wyjściowy, należy przekazać jego nazwę wraz z łańcuchem znaków `'w'` określającym tryb przetwarzania umożliwiający zapis danych:

```
>>> f = open('data.txt', 'w')          # Utworzenie nowego pliku w trybie do zapisu
>>> f.write('Witaj,\n')                # Zapisanie do niego łańcuchów bajtów
6
>>> f.write('Brian\n')                 # Zwraca liczbę zapisanych bajtów w Pythonie 3.0
6
>>> f.close()                          # Zamknięcie pliku i wyczyszczenie bufora wyjściowego na dysku
```

Powyższy kod tworzy plik w katalogu bieżącym i zapisuje do niego tekst (nazwa pliku może być pełną ścieżką do katalogu, jeśli potrzebny jest nam dostęp do pliku znajdującego się w innym miejscu komputera). By wczytać z powrotem to, co napisaliśmy, należy ponownie

otworzyć plik, tym razem w trybie przetwarzania `'r'`, w celu odczytania danych tekstowych (jest to wartość domyślna, jeśli pominiemy tryb w wywołaniu). Następnie należy wczytać zawartość pliku do łańcucha znaków i wyświetlić go. Zawartość pliku jest zawsze dla naszego skryptu łańcuchem znaków, bez względu na typ danych umieszczonych w pliku.

```
>>> f = open('data.txt')                    # 'r' jest domyślnym trybem przetwarzania
>>> text = f.read()                         # Wczytanie całego pliku do łańcucha znaków
>>> text
'Witaj,\nBrian\n'

>>> print(text)                             # Instrukcja print interpretuje znaki sterujące
Witaj,
Brian

>>> text.split()                            # Zawartość pliku jest zawsze łańcuchem znaków
['Witaj,', 'Brian']
```

Inne metody plików obsługują dodatkowe właściwości, których nie mamy teraz czasu omawiać. Obiekty plików udostępniają na przykład większą liczbę sposobów odczytywania i zapisywania (`read` przyjmuje opcjonalny rozmiar w bajtach, `readLine` wczytuje po jednym wierszu naraz), a także inne narzędzia (`seek` przechodzi do nowej pozycji pliku). Jak jednak zobaczymy później, najlepszym obecnie sposobem na wczytanie pliku jest *niewczytywanie go* — pliki udostępniają *iterator*, który automatycznie wczytuje wiersz po wierszu w pętlach `for` oraz innych kontekstach.

Z pełnym zbiorem metod plików spotkamy się w dalszej części książki, a osoby już teraz chcące przyjrzeć się tym metodom mogą wywołać funkcję `dir` dla dowolnego otwartego pliku oraz funkcję `help` dla dowolnej ze zwróconych nazw metod:

```
>>> dir(f)
[ ...pominięto wiele nazw...
'buffer', 'close', 'closed', 'encoding', 'errors', 'fileno', 'flush', 'isatty',
↪'line_buffering', 'mode', 'name', 'newlines', 'read', 'readable', 'readline',
↪'readlines', 'seek', 'seekable', 'tell', 'truncate', 'writable', 'write',
↪'writelines']

>>> help(f.seek)
...przekonaj się sam!...
```

W dalszej części książki zobaczymy także, że pliki w Pythonie 3.0 zachowują rozróżnienie pomiędzy danymi tekstowymi a binarnymi. *Pliki tekstowe* reprezentują zawartość w postaci łańcuchów znaków i automatycznie wykonują kodowanie Unicode, natomiast *pliki binarne* reprezentują zawartość w postaci specjalnego typu łańcucha bajtowego `bytes` i pozwalają na dostęp do niezmienionej zawartości. Poniższy fragment kodu zakłada, że w katalogu bieżącym znajduje się już plik binarny.

```
>>> data = open('data.bin', 'rb').read()  # Otwarcie pliku binarnego
>>> data                                  # Łańcuch bajtowy przechowuje dane binarne
b'\x00\x00\x00\x07mielonka\x00\x08'
>>> data[4:12]
b'spam'
```

Osoby, które będą miały do czynienia jedynie z tekstem w formacie ASCII, nie będą się musiały przejmować tym rozróżnieniem. Łańcuchy znaków i pliki Pythona 3.0 są jednak bardzo cenne, jeśli mamy do czynienia z aplikacjami międzynarodowymi lub danymi bajtowymi.

Inne narzędzia podobne do plików

Funkcja `open` jest koniem pociągowym dla większości czynności związanych z przetwarzaniem plików w Pythonie. Dla bardziej zaawansowanych zadań Python udostępnia jednak dodatkowe narzędzia podobne do plików: potoki, kolejki FIFO, gniazda, pliki dostępne po kluczu, trwałość obiektów, pliki oparte na deskryptorze czy interfejsy do relacyjnych i zorientowanych obiektowo baz danych. Pliki deskryptorów obsługują na przykład blokowanie pliku i inne narzędzia niskiego poziomu, natomiast gniazda udostępniają interfejs do zadań sieciowych i komunikacji międzyprocesowej. Niewiele z tych zagadnień poruszymy w niniejszej książce, jednak osobom, które zajmą się programowaniem w Pythonie na poważnie, z pewnością wiele z nich się przyda.

Inne typy podstawowe

Poza omówionymi dotychczas typami podstawowymi istnieją inne, które mogą się zaliczać do tej kategorii — w zależności od tego, jak szeroko ją zdefiniujemy. *Zbiory* są na przykład nowym dodatkiem do języka, który nie jest ani odwzorowaniem, ani sekwencją. Zbiory to raczej nieuporządkowane kolekcje unikalnych i niezmiennych obiektów. Tworzy się je, wywołując wbudowaną funkcję `set` lub za pomocą nowych literałów i wyrażeń zbiorów z Pythona 3.0. Obsługują one zwykłe operacje matematyczne na zbiorach. Wybór nowej składni `{...}` dla literałów zbiorów w wersji 3.0 ma sens, ponieważ zbiory przypominają klucze słownika bez wartości.

```
>>> X = set('mielonka')                    # Utworzenie zbioru z sekwencji w wersjach 2.6 oraz 3.0
>>> Y = {'s', 'z', 'y', 'n', 'k', 'a'}     # Utworzenie zbioru za pomocą nowego literału z wersji 3.0
>>> X, Y
({'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n'}, {'a', 'k', 'n', 's', 'y', 'z'})

>>> X & Y                                  # Część wspólna zbiorów
{'a', 'k', 'n'}

>>> X | Y                                  # Suma zbiorów
{'a', 'e', 'i', 'k', 'm', 'l', 'o', 'n', 's', 'y', 'z'}

>>> X - Y                                  # Różnica
{'i', 'm', 'e', 'l', 'o'}

>>> {x ** 2 for x in [1, 2, 3, 4]}         # Zbiory składane z wersji 3.0
{16, 1, 4, 9}
```

Dodatkowo Python od niedawna urósł o kilka nowych typów liczbowych: liczby *dziesiętne* (liczby zmiennoprzecinkowe o stałej precyzji) oraz liczby *ułamkowe* (liczby wymierne zawierające licznik i mianownik). Oba typy można wykorzystać do obejścia ograniczeń i niedokładności będących nierozerwalną częścią arytmetyki liczb zmiennoprzecinkowych.

```
>>> 1 / 3                                  # Liczby zmiennoprzecinkowe (w Pythonie 2.6 należy użyć .0)
0.33333333333333331
>>> (2/3) + (1/2)
1.1666666666666665

>>> import decimal                         # Liczby dziesiętne — stała precyzja
>>> d = decimal.Decimal('3.141')
>>> d + 1
Decimal('4.141')
>>> decimal.getcontext().prec = 2
```

```
>>> decimal.Decimal('1.00') / decimal.Decimal('3.00')
Decimal('0.33')

>>> from fractions import Fraction              # Ułamki — licznik i mianownik
>>> f = Fraction(2, 3)
>>> f + 1
Fraction(5, 3)
>>> f + Fraction(1, 2)
Fraction(7, 6)
```

Python zawiera także wartości typu *Boolean* (ze zdefiniowanymi obiektami `True` i `False`, które są tak naprawdę liczbami całkowitymi `1` i `0` z własną logiką wyświetlania). Od dawna obsługuje również specjalny obiekt pojemnika o nazwie `None`, wykorzystywany najczęściej do inicjalizowania zmiennych i obiektów.

```
>>> 1 > 2, 1 < 2                                # Wartości typu Boolean
(False, True)

>>> bool('mielonka')
True

>>> X = None                                    # Pojemnik None
>>> print(X)
None
>>> L = [None] * 100                            # Inicjalizacja listy stu None
>>> L
[None, None, None, None, None, None, None, None, None, None, None, None, None,
↪...lista stu None...]
```

Jak zepsuć elastyczność kodu

Więcej o tych typach obiektów Pythona powiemy w dalszej części książki, jednak jeden z nich zasługuje na kilka słów już teraz. Obiekt *typu*, zwracany przez funkcję wbudowaną `type`, to obiekt dający typ innego obiektu. Jego wynik różni się nieco w Pythonie 3.0 z uwagi na całkowite zlanie się typów i klas (coś, co omówimy w szóstej części książki w kontekście klas w nowym stylu). Zakładając, że `L` nadal jest listą z poprzedniego podrozdziału:

```
# W Pythonie 2.6:

>>> type(L)                                     # Typy: L jest obiektem typu lista
<type 'list'>
>>> type(type(L))                               # Typy też są obiektami
<type 'type'>

# W Pythonie 3.0:

>>> type(L)                                     # 3.0: typy są klasami i odwrotnie
<class 'list'>
>>> type(type(L))                               # Więcej na temat typów klas w rozdziale 31.
<class 'type'>
```

Poza umożliwieniem interaktywnego zbadania obiektów w praktyce pozwala to kodowi sprawdzać typy przetwarzanych obiektów. W skryptach Pythona można to zrobić na przynajmniej trzy sposoby.

```
>>> if type(L) == type([]):                     # Sprawdzanie typów, skoro już musimy…
...     print('tak')
...
tak
>>> if type(L) == list:                         # Użycie nazwy typu
```

```
        print('tak')

tak
>>> if isinstance(L, list):                    # Sprawdzanie zorientowane obiektowo
        print('tak')

tak
```

Pokazałem wszystkie te sposoby sprawdzania typów, jednak moim obowiązkiem jest dodać, że korzystanie z nich jest prawie zawsze złym pomysłem w programie napisanym w Pythonie (i często jest też znakiem rozpoznawczym byłego programisty języka C, który zaczyna przygodę z Pythonem). Przyczyna takiego stanu rzeczy stanie się całkowicie zrozumiała dopiero później, kiedy zaczniemy pisać większe jednostki kodu, takie jak funkcje, jednak jest to koncepcja kluczowa dla Pythona (być może wręcz najważniejsza ze wszystkich). Sprawdzając określone typy w kodzie, efektywnie niszczymy jego elastyczność, ograniczając go do tylko jednego typu danych. Bez takiego sprawdzania kod może działać na szerokiej gamie typów danych.

Jest to powiązane ze wspomnianą wcześniej koncepcją polimorfizmu i ma swoje źródło w braku deklaracji typu w Pythonie. W Pythonie, czego nauczymy się już niedługo, koduje się pod kątem *interfejsów* obiektów (obsługiwanych operacji), a nie typów. Nieprzejmowanie się konkretnymi typami sprawia, że kod automatycznie można zastosować do wielu typów danych. Każdy obiekt ze zgodnym interfejsem będzie działał bez względu na typ. Choć sprawdzanie typów jest obsługiwane — a nawet w niektórych przypadkach wymagane — to nie jest to „pythonowy" sposób myślenia. Jak się okaże, sam polimorfizm jest jedną z kluczowych koncepcji stojących za Pythonem.

Klasy zdefiniowane przez użytkownika

W dalszej części książki będziemy bardziej szczegółowo omawiali *programowanie zorientowane obiektowo* w Pythonie — opcjonalną, lecz mającą duże możliwości właściwość tego języka pozwalającą na skrócenie czasu programowania dzięki dostosowaniu obiektów do własnych potrzeb. Z abstrakcyjnego punktu widzenia klasy definiują nowe typy obiektów, które rozszerzają zbiór typów podstawowych, dlatego zasługują na kilka słów w tym miejscu. Powiedzmy na przykład, że potrzebny jest nam typ obiektu będący modelem pracownika. Choć taki właśnie typ nie istnieje w Pythonie, poniższa zdefiniowana przez użytkownika klasa powinna się przydać.

```
>>> class Worker:
        def __init__(self, name, pay):          # Inicjalizacja przy utworzeniu
            self.name = name                    # self jest nowym obiektem; name to nazwisko, a pay — płaca
            self.pay = pay
        def lastName(self):
            return self.name.split()[-1]        # Podział łańcucha znaków na znakach pustych
        def giveRaise(self, percent):
            self.pay *= (1.0 + percent)         # Uaktualnienie płacy w miejscu
```

Powyższa klasa definiuje nowy rodzaj obiektu, który będzie miał atrybuty `name` (nazwisko) i `pay` (płaca) — czasami nazywane *informacjami o stanie* (ang. *state information*) — a także dwa rodzaje zachowania zakodowane w postaci funkcji (normalnie nazywanych *metodami*). Wywołanie klasy w sposób podobny do funkcji generuje obiekty naszego nowego typu, a metody klasy automatycznie otrzymują obiekt przetwarzany przez dane wywołanie metody (w argumencie `self`).

```
>>> bob = Worker('Robert Zielony', 50000)     # Utworzenie dwóch obiektów
>>> anna = Worker('Anna Czerwona', 60000)     # Każdy ma atrybut imienia i nazwiska oraz płacy
>>> bob.lastName()                            # Wywołanie metody — self to bob
'Zielony'
>>> anna.lastName()                           # Teraz self to anna
'Czerwona'
>>> anna.giveRaise(.10)                       # Uaktualnienie płacy Anny
>>> anna.pay
66000.0
```

Domniemany obiekt `self` jest przyczyną nazwania tego modelu zorientowanym obiektowo — w funkcjach klasy zawsze istnieje domniemany podmiot. W pewnym sensie typ oparty na klasie po prostu tworzy coś na bazie typów podstawowych. Zdefiniowany wyżej obiekt `Worker` jest na przykład zbiorem łańcucha znaków oraz liczby (odpowiednio `name` i `pay`) wraz z funkcjami odpowiedzialnymi za przetwarzanie tych wbudowanych obiektów.

Bardziej rozbudowane wyjaśnienie kwestii klas mówi, że to ich mechanizm dziedziczenia obsługuje hierarchię oprogramowania, która pozwala na dostosowywanie programów do własnych potrzeb poprzez rozszerzanie ich. Programy rozszerza się poprzez pisanie nowych klas, a nie modyfikowanie tego, co już działa. Warto również wiedzieć, że klasy są w Pythonie opcjonalne i w wielu przypadkach proste obiekty wbudowane, takie jak listy i słowniki, są często lepszymi narzędziami od klas zakodowanych przez użytkowników. Wszystko to jednak wykracza poza zakres wstępnego omówienia typów, dlatego należy uznać to tylko za zapowiedź tego, co znajduje się dalej. Pełne informacje na temat typów zdefiniowanych przez użytkownika i tworzonych za pomocą klas będzie można znaleźć w szóstej części książki.

I wszystko inne

Jak wspomniano wcześniej, wszystko, co możemy przetworzyć w skrypcie Pythona, jest typem obiektu, przez co nasze omówienie nie może być kompletne. Jednak mimo że wszystko w Pythonie jest obiektem, jedynie obiekty omówione wyżej stanowią zbiór obiektów podstawowych tego języka. Pozostałe typy są albo obiektami powiązanymi z omówionym później wykonywaniem programu (jak funkcje, moduły, klasy i skompilowany kod), albo są implementowane przez funkcje importowanych modułów, a nie składnię języka. Ta druga kategoria pełni zwykle role specyficzne dla określonego zastosowania — wzorców tekstowych, interfejsów do baz danych czy połączeń sieciowych.

Należy również pamiętać, że obiekty omówione wyżej są *obiektami*, ale niekoniecznie są *zorientowane obiektowo*. Zorientowanie obiektowe zazwyczaj wymaga w Pythonie dziedziczenia i instrukcji `class`, z którą spotkamy się w dalszej części książki. Obiekty podstawowe Pythona są siłą napędową prawie każdego skryptu napisanego w tym języku, z jakim można się spotkać, i zazwyczaj są też podstawą większych typów niemieszczących się w tej kategorii.

Podsumowanie rozdziału

I to by było na tyle, jeśli chodzi o naszą krótką wycieczkę po typach danych. W niniejszym rozdziale zawarłem zwięzłe wprowadzenie do podstawowych typów obiektów w Pythonie wraz z omówieniem rodzajów operacji, jakie można do nich zastosować. Przyjrzeliśmy się uniwersalnym operacjom, które działają na wielu typach obiektów (na przykład operacjom na sekwencjach, takim jak indeksowanie czy wycinki), a także operacjom specyficznym dla

określonego typu, dostępnym w postaci wywołania metody (na przykład dzielenie łańcuchów znaków czy dodawanie elementów do listy). Zdefiniowaliśmy również pewne kluczowe pojęcia, takie jak niezmienność, sekwencje i polimorfizm.

Po drodze widzieliśmy również, że podstawowe typy obiektów Pythona są bardziej elastyczne i mają większe możliwości od tego, co dostępne jest w językach niższego poziomu, takich jak C. Listy i słowniki w Pythonie usuwają na przykład większość pracy koniecznej do obsługiwania kolekcji oraz wyszukiwania w językach niższego poziomu. Listy to uporządkowane kolekcje innych obiektów, natomiast słowniki to kolekcje innych obiektów indeksowane po kluczu, a nie pozycji. Zarówno słowniki, jak i listy mogą być zagnieżdżane, mogą się rozszerzać i kurczyć na życzenie i mogą zawierać obiekty dowolnego typu. Co więcej, po porzuceniu ich automatycznie odzyskiwane jest miejsce zajmowane przez nie w pamięci.

Pominąłem większość szczegółów w celu skrócenia naszej wycieczki, dlatego nie należy oczekiwać, że całość tego rozdziału będzie miała sens. W kolejnych rozdziałach zaczniemy kopać coraz głębiej i głębiej, odkrywając szczegóły podstawowych typów obiektów Pythona, które tutaj pominęliśmy w celu lepszego zrozumienia całości. Zaczniemy już w następnym rozdziale od pogłębionego omówienia liczb w Pythonie. Najpierw jednak — kolejny quiz.

Sprawdź swoją wiedzę — quiz

Koncepcje przedstawione w niniejszym rozdziale omówimy bardziej szczegółowo w kolejnych rozdziałach, dlatego teraz czas na szerszą perspektywę.

1. Należy podać cztery podstawowe typy danych Pythona.
2. Dlaczego typy te nazywane są podstawowymi?
3. Co oznacza pojęcie „niezmienny” i które trzy z podstawowych typów danych Pythona są uznawane za niezmienne?
4. Czym jest sekwencja i które trzy typy danych należą do tej kategorii?
5. Czym jest odwzorowanie i który typ podstawowy zalicza się do tej kategorii?
6. Czym jest polimorfizm i dlaczego powinno to być dla nas ważne?

Sprawdź swoją wiedzę — odpowiedzi

1. Liczby, łańcuchy znaków, listy, słowniki, krotki, pliki i zbiory są uważane za podstawowe typy danych w Pythonie. Czasami w ten sam sposób klasyfikowane są również typy, `None` i wartości typu Boolean. Istnieje kilka typów liczbowych (liczby całkowite, zmiennoprzecinkowe, zespolone, ułamkowe i dziesiętne), a także różne typy łańcuchów znaków (zwykłe łańcuchy znaków, łańcuchy znaków Unicode z Pythona 2.X, łańcuchy tekstowe oraz bajtowe z Pythona 3.X).
2. Typy te nazywane są podstawowymi, ponieważ są częścią samego języka i są zawsze dostępne. By natomiast utworzyć inne obiekty, konieczne jest wywołanie funkcji z importowanych modułów. Większość typów podstawowych ma określoną składnię służącą do ich generowania — na przykład `'mielonka'` to wyrażenie tworzące łańcuch znaków i ustalające

zbiór operacji, które mogą być do niego zastosowane. Z tego powodu typy podstawowe są ściśle połączone ze składnią Pythona. Żeby natomiast utworzyć obiekt pliku, trzeba wywołać wbudowaną funkcję `open`.

3. Obiekt niezmienny to taki obiekt, który nie może być zmodyfikowany po utworzeniu. W Pythonie do tej kategorii zaliczamy liczby, łańcuchy znaków i krotki. Choć nie można zmodyfikować niezmiennego obiektu w miejscu, zawsze można utworzyć nowy obiekt, wykonując odpowiednie wyrażenie.
4. Sekwencja to uporządkowana pod względem pozycji kolekcja obiektów. W Pythonie sekwencjami są łańcuchy znaków, listy oraz krotki. Dzielą one wspólne operacje na sekwencjach, takie jak indeksowanie, konkatenacja czy wycinki, jednak każdy ma również specyficzne dla danego typu metody.
5. Pojęcie „odwzorowanie" oznacza obiekt, który odwzorowuje klucze na powiązane wartości. Jedynym takim typem w zbiorze podstawowych typów obiektów Pythona jest słownik. Odwzorowania nie zachowują żadnej stałej kolejności elementów od lewej do prawej strony. Obsługują dostęp do danych po kluczu oraz metody specyficzne dla danego typu.
6. Polimorfizm oznacza, że znaczenie operacji (takiej jak +) uzależnione jest od obiektów, na których się ona odbywa. Okazuje się to kluczową koncepcją stanowiącą podstawę Pythona (być może nawet tą najważniejszą) — nieograniczanie kodu do określonych typów sprawia, że kod ten można automatycznie zastosować do wielu typów.