Przewodnik po języku Python

Wydanie 2.3

Guido van Rossum
Fred L. Drake, Jr., editor

27 października 2004

PythonLabs
Email: python-docs@python.org

Streszczenie

Python jest łatwym do nauczenia, pełnym mocy i siły językiem programowania. Posiada struktury danych wysokiego poziomu i w prosty, acz efektywny sposób umożliwia programowanie obiektowe. Składnia Pythona jest elegancka, a dynamiczny system typów oraz fakt, iż Python jest interpreterem, powoduje, że jest idealnym językiem do pisania skryptów oraz tzw. błyskawicznego rozwijania aplikacji w wielu dziedzinach, oraz na wielu platformach sprzętowo-programowych.

Interpreter Pythona oraz bogata biblioteka standardowa są dostępne w sposób wolny i za darmo, zarówno w postaci źródłowej jak i binarnej (dowolnie dystrybuowanych) na większość platform systemowych. Zainteresowany użytkownik znajdzie więcej informacji na stronie internetowej Pythona, http://www.python.org. Ta sama witryna zawiera również pakiety instalacyjne i odnośniki do wielu pythonowych modułów (wolnych od opłat), programów oraz narzędzi jak i dodatkowej dokumentacji.

Python daje się łatwo rozszerzać o nowe funkcje i typy danych, które mogą zostać zaimplementowane w C lub C++ (lub innych językach, które mogą być skonsolidowane z modułami C). Python nadaje się również do wykorzystania jako dodatkowy język w aplikacjach, jako dodatkowy język ułatwiający dopasowanie ich do potrzeb użytkownika.

Przewodnik ten wprowadza czytelnika w podstawowe założenia i cechy Pythona jako języka i systemu. Pomocne będzie posiadanie interpretera Pythona „pod ręką” do ćwiczeń „na gorąco”, aczkolwiek przykłady są na tyle czytelne, że materiał ten może być czytany również przy kawie.

Opis standardowych obiektów i modułów znajduje się w *Opisie biblioteki Pythona*. Formalna definicja języka przedstawiona jest w *Podręczniku języka Python*. Aby pisać rozszerzenia w języku C lub C++ przeczytaj *Rozszerzanie i wbudowywanie interpretera języka Python* oraz *Opis Python/C API*. Ponadto istnieje kilka książek wgłębiających się w sam język Python.

Przewodnik ten nie usiłuje opisać Pythona w sposób wszechstronny, poruszając każdą cechę języka, nawet najbardziej używaną. Zamiast tego informuje czytelnika o wielu wartych zauważenia cechach i daje dobre wyobrażenie o stylu pisania programów Pythona„ jak i o jego „smaku” i „zapachu”. Po przeczytaniu tego przewodnika, czytelnik będzie w stanie dowiedzieć się i zrozumieć wiele innych modułów Pythona, które zostały omówione w *Opisie biblioteki Pythona*.

SPIS TREŚCI

ROZDZIAŁ

PIERWSZY

Wzniecając apetyt. . .

Jeżeli kiedykolwiek pisałeś pokaźny skrypt, znasz prawdopodobnie to uczucie: z wielką chęcią chciałbyś dodać jeszcze jedną cechę do programu, ale i tak jest już zbyt wolny i zbyt duży oraz wystarczająco skomplikowany; albo ta nowa właściwość wymaga zaangażowania funkcji systemowej, która jest dostępna tylko z poziomu C . . . Zwykle problem ten nie jest na tyle ważny, aby wymagało to przepisania wszystkiego w C, być może wymagana będzie obecność łańcuchów znakowych zmiennej długości lub innych typów danych (jak uporządkowane listy lub nazwy plików), które łatwo wprowadzić w skrypcie, ale wymagają mnóstwa pracy w C, lub też po prostu nie znasz tak dobrze C.

Inna sytuacja: być może zaistniała potrzeba pracy z paroma bibliotekami C i zwyczajowy cykl pisanie/kompilacja-/testowanie/rekompilacja jest zbyt wolny. Potrzebujesz czegoś szybszego do rozwijania programów. Być może napisałeś program, który mógłby używać jakiegoś języka rozszerzającego jego możliwości, ale nie chciałbyś projektować jeszcze jednego języka, pisać i sprawdzać jego interpreter i dopiero wpleść go w swoją aplikację.

W powyższych przypadkach Python może być właśnie tym, czego szukasz. Python jest prosty w użyciu„ lecz jest prawdziwym językiem programowania, który oferuje o wiele więcej struktur i pomocnych właściwości dla dużych programów w porównaniu z językiem powłoki. Z drugiej strony, posiada o wiele więcej sposobów wyłapania błędów niż C i będąc *językiem bardzo wysokiego poziomu*, posiada wbudowane typy danych wysokiego poziomu„ jak rozszerzalne tablice i słowniki, których efektywne zaimplementowanie w C kosztowałyby cię wiele dni pracy. Z powodu tego, iż typy danych Pythona są bardzo „ogólne”, nadają się do zastosowania w o wiele większych obszarach problemowych, niż robi to *Perl*, aczkolwiek dużo rzeczy jest przynajmniej tak łatwych w Pythonie jak w powyższych językach.

Python pozwala na podział twego programu na moduły, które mogą być obiektem ponownego użycia w innych pythonowych programach. Język dostarczany jest z obszernym zbiorem standardowych modułów, które mogą być podstawowymi składnikami twoich programów, lub służyć jako przykłady przy rozpoczęciu nauki programowania w Pythonie. Istnieją również moduły wbudowane w interpreter, mające na celu obsługę I/O, wywołań systemowych, gniazdek lub nawet moduły interfejsu użytkownika (GUI), jak np. Tk.

Python jest językiem interpretowanym, co oznacza, że zaoszczędza ci zauważalnie czas w procesie rozwijania oprogramowania, ponieważ nie ma potrzeby kompilacji i łączenia modułów. Interpreter może być użyty w sposób interaktywny, co pomaga w eksperymentowaniu z właściwościami języka, pisaniu podręcznych programów lub testowaniu funkcji podczas rozwijania programu (*ang. bottom-up developement*). Jest też podręcznym biurkowym kalkulatorem.

Python umożliwia pisanie bardzo zwartych i czytelnych programów. Programy napisane w nim, są zwykle o wiele krótsze, niż odpowiedniki napisane w C lub C++, a to dlatego że:

- typy danych wysokiego poziomu pozwalają wyrazić złożone operacje w pojedynczej instrukcji;
- grupowanie poleceń uzyskuje się poprzez indentację (wcinanie wierszy) zamiast używania słów kluczowych `begin/end` lub nawiasów klamrowych;
- nie potrzeba deklarować zmiennych lub argumentów wywołań;

Python jest *rozszerzalny*: jeśli znasz język C, to łatwo będzie ci dodać nową funkcję wbudowaną lub moduł do interpretera, zarówno aby zrobić jakąś krytyczną pod względem czasu wykonania operację„ jak i włączyć do Pythona bibliotekę, która być może jest dostępna tylko w formie binarnej (np. jakaś specjalistyczna biblioteka

graficzna). Jak tylko poczujesz pewny grunt pod nogami, będziesz mógł włączyć interpreter Pythona w aplikację napisaną w C i używać go jako rozszerzenia lub języka komend w tej aplikacji.

A'propos, język został tak nazwany w ślad za programem telewizyjnym BBC „Latający cyrk Monty Pythona" („Monty Python's Flying Circus") i nie ma nic wspólnego z tymi okropnymi wężami. Nie tylko pozwalamy na wprowadzanie odniesień do Monty Pythona w dokumentacji, lecz również zachęcamy do tego!

1.1 Dokąd dalej?

Mam nadzieję, że jesteś w tym momencie mocno podekscytowany Pythonem i chcesz szczegółowo go popróbować. Ponieważ najlepszym sposobem, aby nauczyć się języka, jest jego używanie... więc zapraszamy cię i zachęcamy abyś to zaraz zrobił.

W następnym rozdziale wyjaśnione są sposoby używania interpretera. Rewelacje te są dość nużące, lecz konieczne do poznania w celu przejścia do przykładów pokazanych w następnych częściach przewodnika.

Reszta przewodnika wprowadzi cię w różnorakie właściwości Pythona jako języka i systemu poprzez przykłady, zaczynając od najprostszych wyrażeń, poleceń i typów danych, poprzez funkcje i moduły i kończąc na nauce zaawansowanych pojęć, takich jak wyjątki i definiowane przez użytkownika klasy.

ROZDZIAŁ

DRUGI

Używanie interpretera Pythona

2.1 Wywołanie interpretera

Interpreter Pythona jest zwykle zainstalowany jako plik „/usr/local/bin/python”;[1] wstawienie „/usr/local/bin” do ścieżki wyszukiwań powłoki na twojej odmianie UNIXA, umożliwia rozpoczęcie pracy interpretera poprzez polecenie

```
python
```

Ponieważ wybór katalogu, w którym umieszczono interpreter Pythona jest opcją instalacji języka, również inne miejsca są możliwe — skonsultuj się ze swoim lokalnym pythonowym guru lub administratorem systemu (innymi słowy, „/usr/local/bin” jest jednym z bardziej popularnych katalogów).

Wpisanie znaku końca pliku EOF (Control-D w UNIKSIE, Control-Z w DOS-ie lub Windows) za początkowym znakiem zachęty (*ang. prompt*) spowoduje zakończenie pracy interpretera z kodem zakończenia równym zero. Jeżeli ten sposób nie zadziała można w dalszym ciągu bezboleśnie zakończyć pracę poprzez wpisanie następujących poleceń: „`import sys; sys.exit()`”.

Właściwości edycyjne linii poleceń interpretera nie są bardzo wysublimowane. Na UNIKSIE być może ktoś, kto zainstalował interpreter włączył również wspomaganie edycji linii poleceń za pomocą biblioteki GNU `readline`, co dodaje bardziej wyszukanych właściwości interaktywnej edycji i historii poleceń. Prawdopodobnie najszybszym sposobem sprawdzenia czy posiadasz rozszerzone właściwości linii poleceń, jest naciśnięcie Control-P za pierwszym znakiem zachęty Pythona, który zobaczysz po jego uruchomieniu. Jeżeli coś zabrzęczy— jest wzbogacona edycja linii poleceń; dodatek A zwiera wprowadzenie do klawiszologii. Jeśli nic się nie zdarzy lub pojawi się echo `^P`, to edycja linii poleceń nie jest możliwa—będzie można tylko używać klawisza `backspace`, aby usuwać znaki z bieżącego wiersza.

Interpreter przypomina nieco unixową powłokę: kiedy jest wywołany ze standardowym wejściem połączonym z jakimś urządzeniem „tty”, czyta i wykonuje komendy interaktywnie. Gdy zostanie wywołany z argumentem w postaci nazwy pliku lub ze standardowym wejściem jako plik, wtedy czyta go i wykonuje jak *skrypt*.

Trzecim sposobem na wywołanie interpretera jest „**python -c** *polecenia* `[arg] ...`”, co powoduje wykonanie poleceń zawartych w *polecenia*, analogicznie do opcji **-c** w powłoce. Z powodu tego, iż polecenia Pythona często zawierają spacje lub inne znaki, które są najpierw interpretowane przez powłokę, najlepiej jest umieścić *polecenia* w podwójnych cudzysłowach.

Koniecznym jest zauważenie różnicy pomiędzy „`python plik`” i „`python <plik`”. W ostatnim przypadku polecenie pobrania wejścia„ jak np. wywołanie `input()` i `raw_input()` dotyczy zawartości *plik*. Ponieważ plik ten zostanie przeczytany w całości na początku działania interpretera, zanim polecenia w nim zawarte zostaną wykonane, program natychmiast napotka znak EOF. W przypadku pierwszym (który jest tym co chce się zwykle uzyskać) polecenia pobrania wejścia dotyczą zarówno pliku, jak i urządzenia dołączonego do standardowego wejścia interpretera.

Kiedy używa się skryptu, czasami jest potrzebne uruchomić tryb interaktywny zaraz po zakończeniu jego działania. Można to uzyskać poprzez opcję **-i**, którą przekazuje się przy wywołaniu interpretera (ten sposób nie zadziała,

[1] „C:\Program Files\Python\python.exe” lub „C:\Python\python.exe” w systemie Windows

jeśli skrypt czytany jest ze standardowego wejścia—powód: ten sam, który został opisany w poprzednim paragrafie).

2.1.1 Przekazywanie argumentów

Nazwa skryptu i dodatkowe parametry wywołania są przechowywane w zmiennej `sys.argv`, która jest listą łańcuchów znaków. Jej długość jest zawsze przynajmniej równa jeden—nawet wtedy, gdy nie podano żadnej nazwy skryptu i żadnych argumentów wywołania, `sys.argv[0]`, jest pustym ciągiem. Gdy nazwa skryptu przekazana jest w postaci `'-'` (co oznacza standardowe wejście), `sys.argv[0]` przyjmuje wartość `'-'`. Gdy zostanie użyta opcja **-c** i przekazane zostaną polecenia w *polecenia*, `sys.argv[0]`, przyjmuje wartość `'-c'`. Opcje za **-c** *polecenia* nie są połykane przez interpreter Pythona, lecz pozostawiane w `sys.argv` dla pozostałych poleceń.

2.1.2 Tryb interaktywny

Jeżeli instrukcje pochodzą z urządzenia „tty", mówi się wtedy, że interpreter jest w *trybie interaktywnym*. Interpreter zachęca wtedy do podania kolejnej instrukcji za pomocą wyświetlenia tzw. znaku zachęty, zwykle w postaci trzech znaków większości („>>>"). Gdy wymaga kontynuacji instrukcji, w następnej linii wyświetla drugi znak zachęty, domyślnie w postaci trzech kropek („`...` "). Interpreter zaraz po uruchomieniu drukuje swoją wersję i notkę o prawach autorskich przed pierwszym znakiem zachęty, tzn.:

```
python
Python 1.5.2b2 (#1, Feb 28 1999, 00:02:06)  [GCC 2.8.1] on sunos5
Copyright 1991-1995 Stichting Mathematisch Centrum, Amsterdam
>>>
```

Linie kontynuacji instrukcji są wymagane podczas wejścia w składanie wielowierszowej instrukcji. Jako przykład weźmy instrukcję `if`:

```
>>> swiat_jest_plaski = 1
>>> if swiat_jest_plaski:
...     print "Bądź ostrożnym: nie spadnij!"
...
Bądź ostrożnym: nie spadnij!
```

2.2 Interpreter i jego środowisko

2.2.1 Obsługa (wyłapywanie) wyjątków

Jeżeli pojawi się błąd, interpreter drukuje komunikat o błędzie i ślad stosu wywołań. W trybie interaktywnym powraca natychmiast do pierwszej zachęty, a jeśli wejście pochodziło z pliku, kończy swoją pracę z niezerowym kodem wyjścia zaraz po wydrukowaniu ślady stosu wywołań. (Wyjątki obsługiwane przez klauzulę `except` w instrukcji `try` nie są błędami w tym kontekście). Niektóre błędy są bezwzględnie fatalne i powodują zakończenie pracy z niezerowym kodem wyjścia—odnosi się to do wewnętrznych błędów i przypadków wyczerpania pamięci. Wszystkie komunikaty o błędach są zapisywane na standardowym wyjściu diagnostycznym, a zwykłe komunikaty pochodzące od wykonywanych poleceń zapisywane są na standardowym wyjściu.

Naciśnięcie klawisza przerwania (zwykle Control-C lub DEL) w czasie pierwszej lub drugiej zachęty usuwa znaki z wejścia i powoduje powrót do pierwszej zachęty.[2] Naciśnięcie wyżej wspomnianego klawisza w czasie wykonywania instrukcji powoduje zgłoszenie wyjątku `KeyboardInterrupt`, który może być wyłapany za pomocą instrukcji `try`.

[2] Pewien błąd w bibliotece GNU `readline` może to uniemożliwić.

2.2.2 Wykonywalne skrypty Pythona

[3]

W systemach UNIX pochodzących z gałęzi BSD, skrypty Pythona mogą być wykonywane bezpośrednio przez powłokę, za pomocą wstawienia wiersza

```
#! /usr/bin/env python
```

na początku pliku skryptu (zakładając, że interpreter jest widziany z zbiorze katalogów zawartych w zmiennej $PATH) oraz ustawienia dla tego pliki atrybutu wykonywalności. Znaki „#!” muszą być pierwszymi znakami pliku. Zauważcie, iż znak „#”, jest znakiem rozpoczęcia komentarza w Pythonie.

2.2.3 Plik rozpoczęcia pracy interaktywnej

Kiedy używa się Pythona interaktywnie, często pomocne jest wywoływanie pewnych standardowych poleceń za każdym razem, gdy interpreter jest uruchamiany. Można to uzyskać poprzez umieszczenie w zmiennej systemowej $PYTHONSTARTUP nazwy pliku zawierającego twoje specyficzne instrukcje. Jest to podobne do mechanizmu pliku „.profile” w powłoce UNIXA.

Plik ten jest czytany tylko podczas rozpoczęcia sesji interaktywnej, nigdy w przypadku czytania poleceń ze skryptu i w przypadku, gdy plik „/dev/tty” jest podany *explicite* jako źródło poleceń (co skądinąd zachowuje się jak sesja interaktywna). Plik ten wykonywany jest w tej samej przestrzeni nazw, w której wykonywane są instrukcje, tak więc obiekty, które definiuje lub importuje mogą być użyte bez kwalifikowania podczas sesji interaktywnej.[4] Można w nim zmienić również znaki zachęty, które umieszczone są w zmiennych `sys.ps1` i `sys.ps2`.

Jeśli chce się dołączyć dodatkowy plik rozpoczęcia z bieżącego katalogu, można dokonać tego z poziomu globalnego pliku rozpoczęcia „`execfile('.pythonrc.py')`”. Jeśli chce się użyć pliku rozpoczęcia w skrypcie, trzeba zrobić to *explicite* w skrypcie:

```
import os
if os.environ.get('PYTHONSTARTUP') \
   and os.path.isfile(os.environ['PYTHONSTARTUP']):
    execfile(os.environ['PYTHONSTARTUP'])
```

[3] XXX Jak to się robi w Windows?

[4] Całe to zawracanie głowy o kwalifikowaniu zostanie zrozumiane po zapoznaniu się z pracą z modułami (*przyp. tłum.*)

ROZDZIAŁ

TRZECI

Nieformalne wprowadzenie do Pythona

W poniższych przykładach wejście i wyjście rozróżnione zostało poprzez obecność znaków zachęty („>>> ” i „... ”): aby powtórzyć przykład, trzeba przepisać wszystko za znakiem zachęty; linie, które nie zaczynają się nim, są wyjściem (odpowiedziami) interpretera.

Uwaga! Pojawienie się po raz drugi znaku zachęty oznacza, że trzeba wprowadzić pusty wiersz. Wtedy następuje koniec instrukcji wielowierszowej.

Wiele przykładów w tym podręczniku, nawet te wprowadzone w linii poleceń, zawierają komentarze. W Pythonie komentarz rozpoczyna się od znaku „#” i ciągnie się aż do końca fizycznego wiersza. Komentarz może pojawić się na początku linii lub kończyć instrukcję„ lecz nie może być zawarty w literale ciągu znaków. Znak «hash» („#”) w ciągu znaków jest po prostu zwykłym znakiem, jednym z wielu tego ciągu.

Trochę przykładów:

```
# to jest pierwszy komentarz
POMYJE = 1                 # a to jest drugi komentarz
                         # ... uch, a to trzeci!
LANCUCH = "# To nie jest komentarz. To jest łańcuch znaków."
```

3.1 Używanie Pythona jako kalkulatora

Wypróbujmy parę prostych poleceń Pythona. Uruchom interpreter i poczekaj na pojawienie się pierwszego znaku zachęty „>>>”. (Nie powinno to zająć dużo czasu).

3.1.1 Liczby

Interpreter działa jak prosty kalkulator: można wpisać wyrażenie do niego, a on wypisze jego wartość. Składnia wyrażenia jest prosta: operator `+`, `-`, `*` i `/` działają jak w wielu innych językach programowania (np. Pascal lub C); nawiasy można użyć do grupowania. Na przykład:

```
>>> 2+2
4
>>> # To jest komentarz
... 2+2
4
>>> 2+2  # a to jest komentarz w tej samej linii co kod instrukcji
4
>>> (50-5*6)/4
5
>>> # Dzielenie całkowite zwraca liczbę zaokrągloną w dół
... 7/3
2
>>> 7/-3
-3
```

Podobnie jak w C, znak równości („=”) jest używany do przypisania wartości do zmiennej. Przypisanie do zmiennej nie jest wypisywane przez interpreter:

```
>>> szerokosc = 20
>>> wysokosc = 5*9
>>> szerokosc * wysokosc
900
```

Wartość może być przypisana jednocześnie paru zmiennym:

```
>>> x = y = z = 0  # Zero x, y i z
>>> x
0
>>> y
0
>>> z
0
```

Python implementuje oczywiście arytmetykę zmiennoprzecinkową, a operatory z operandami typów mieszanych przekształcają operandy całkowite w zmiennoprzecinkowe:

```
>>> 4 * 2.5 / 3.3
3.0303030303
>>> 7.0 / 2
3.5
```

Liczby zespolone są również wbudowane—części urojone zapisywane są z przyrostkiem „`j`” lub „`J`”. Liczby zespolone z niezerową częścią rzeczywistą zapisywane są jako „`(`*real*`+`*imag*`j)`” lub mogą być stworzone za pomocą funkcji „`complex(`*real*`, `*imag*`)`”.

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> 3+1j*3
(3+3j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Liczby zespolone są zawsze reprezentowane jako dwie zmiennoprzecinkowe liczby, odpowiednio część rzeczywista i urojona. Aby wydobyć je z liczby urojonej *z*, należy użyć `z.real` i `z.imag`.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```

Funkcje konwersji z liczby zmiennoprzecinkowej do całkowitej (`float()`, `int()` i `long()`) nie działają dla liczb urojonych — nie ma poprawnego sposobu na przekształcenie liczby zespolonej w rzeczywistą. Użyj `abs(z)`, aby uzyskać jej moduł (jako liczbę zmienno przecinkową) lub `z.real`, aby uzyskać część rzeczywistą.

```
>>> a=1.5+0.5j
>>> float(a)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use e.g. abs(z)
>>> a.real
1.5
>>> abs(a)
1.58113883008
```

(**TypeError***: nie można przekształcić zespolonej do zmiennoprzecinkowej; użyj np. abs(z)*)

W trybie interaktywnym, ostatnio wydrukowane wyrażenie przypisywane jest do zmiennej _. Oznacza to, że jeśli używa się Pythona jako biurkowego kalkulatora, niejako łatwym staje się kontynuowanie obliczeń„ jak w poniższym przykładzie:

```
>>> podatek = 17.5 / 100
>>> cena = 3.50
>>> cena * podatek
0.6125
>>> cena + _
4.1125
>>> round(_, 2) # zaokrągla do dwóch miejsc po przecinku
4.11
```

Zmienna ta powinna być traktowana przez użytkownika jak tylko do odczytu. Nie warto wprost zapisywać (przypisywać) coś do niej—stworzy się inną, niezależną lokalną zmienną z tą samą nazwą przykrywającą wbudowaną zmienną o tych magicznych właściwościach.

3.1.2 Ciągi znaków czyli napisy

Oprócz liczb Python może również manipulować ciągami znaków, które można wyrazić na parę sposobów. Mogą one być objęte pojedynczym lub podwójnym cudzysłowem:

```
>>> 'zepsute jaja'
'zepsute jaja'
>>> 'A\'propos'
"A'propos"
>>> "A'propos"
"A'propos"
>>> '"Tak," powiedział.'
'"Tak," powiedział.'
>>> "\"Tak,\" powiedział."
'"Tak," powiedział.'
>>> '"A\'propos," powiedziała.'
'"A\'propos," powiedziała.'
```

Napisy mogą rozciągać się na wiele wierszy. Znaki nowej linii mogą zostać zacytowane za pomocą znaków ukośnika:

```
hello = "Jest to raczej długi ciąg znaków zawierający\n\
kilka linii tekstu, tak jak robisz to w C.\n\
    Zauważ, że znaki spacji i znaki białe na początku linii\
 są znaczące.\n"
print hello
```

spowoduje pojawienie się zaraz za tym:

```
Jest to raczej długi ciąg znaków zawierający
kilka linii tekstu, tak jak robisz to w C.
    Zauważ, że znaki spacji i znaki białe na początku linii są znaczące.
```

Równie dobrze ciągi znaków mogą być ujęte w pary potrójnych cudzysłowów: `"""` lub `'''`. Końce linii nie muszą być w takim przypadku cytowane„ lecz zostaną dosłownie włączone do ciągu.

```
print """
Użytkowanie: cienias [OPCJE]
     -h                        Pokaż ten opis poleceń
     -H nazwaserwera      Nazwa serwera, z którym chcesz się połączyć
"""
```

powoduje następującą odpowiedź interpretera:

```
Użytkowanie: cienias [OPCJE]
     -h                        Pokaż ten opis poleceń
     -H nazwaserwera      Nazwa serwera, z którym chcesz się połączyć
```

Interpreter drukuje wynik działań na napisach w ten sam sposób, w jaki są one wprowadzane: objęte cudzysłowami i wraz z cudzysłowami i innymi zabawnymi znakami poprzedzonymi znakiem ukośnika (\), tak aby w sposób dokładny pokazać zawartość ciągu. Łańcuch objęty jest podwójnym cudzysłowem, jeśli zawiera tylko pojedyncze cudzysłowie, w przeciwnym wypadku objęty jest pojedynczym. (Instrukcja `print`, które zostanie opisane później, może zostać użyta do wypisywania ciągów znaków bez okalających je cudzysłowów lub znaków cytowania[1]).

Łańcuchy mogą być sklejane za pomocą operatora `+` i powielane za pomocą `*`:

```
>>> slowo = 'Pomoc' + 'A'
>>> slowo
'PomocA'
>>> '<' + slowo*5 + '>'
'<PomocAPomocAPomocAPomocAPomocA>'
```

Dwa literały napisu występujące jeden obok drugiego są automatycznie sklejane; np. pierwszy wiersz w przykładzie powyżej mógłby być równie dobrze zapisany jako „`slowo='Pomoc' 'A'`” — działa to tylko z dwoma literałami, a nie z dowolnym wyrażeniem typu znakowego:

[1] Czyli znaków specjalnych poprzedzonych znakiem ukośnika

```
>>> import string # znaczenie słowa 'import' zostanie wyjaśnione później ...
>>> 'str' 'ing'                   #  <-  Tak jest w porządalu
'string'
>>> string.strip('str') + 'ing'   #  <-  to też
'string'
>>> string.strip('str') 'ing'     #  <-  to nie zadziała!
  File "<stdin>", line 1
    string.strip('str') 'ing'
                        ^
SyntaxError: invalid syntax
```

(**SyntaxError***: zła składnia*)

Łańcuchy znaków mogą być indeksowane. Podobnie jak w C, pierwszy znak w ciągu ma indeks (numer porządkowy) 0. Nie istnieje osobny typ oznaczający znak — znak jest po prostu napisem o długości jeden. Podobnie jak w języku Icon[2] podciągi znaków mogą zostać wyspecyfikowane za pomocą notacji tzw. *wykrawania*: dwóch indeksów przedzielonych dwukropkiem.

```
>>> slowo[5]
'A'
>>> slowo[0:2]
'Po'
>>> word[2:5]
'moc'
```

Odmiennie niż w C, łańcuchy znaków w Pythonie nie mogą być modyfikowane. Przypisanie do zaindeksowanej pozycji w ciągu powoduje powstanie błędu:

```
>>> slowo[0] = 'x'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> slowo[:-1] = 'Splat'
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

(**TypeError***: na obiekcie nie można wykonać operacji przypisania do elementu*)

(**TypeError***: na obiekcie nie można wykonać opercji przypisania do wycinka*)

Stworzenie, jednakże nowego ciągu znaków z połączenia innych jest łatwe i wydajne:

```
>>> 'x' + słowo[1:]
'xomocA'
>>> 'Splat' + slowo[-1:]
'SplatA'
```

Indeksy wykrawania posiadają użyteczne parametry domyślne: pominięty pierwszy indeks posiada domyślną wartość zero, a pominięty drugi domyślnie równy jest długości łańcucha znaków, którego dotyczy wykrawanie.

[2] . . . pierwszy raz słyszę o takim języku (*przyp. tłum.*)

```
>>> slowo[:2]    # Dwa pierwsze znaki
'Po'
>>> slowo[2:]    # Wszystkie oprócz dwóch pierwszych znaków
'mocA'
```

Oto użyteczny niezmiennik operacji wykrawania: `s[:i] + s[i:]` jest równe `s`.

```
>>> slowo[:2] + slowo[2:]
'PomocA'
>>> slowo[:3] + slowo[3:]
'PomocA'
```

Zdegenerowane indeksy wykrawania obsługiwane są dość ostrożnie: indeks, który jest zbyt duży, zastępowany jest długością łańcucha, a ograniczenie górne, które okaże się mniejsze od ograniczenia dolnego, powoduje powstanie pustego napisu.

```
>>> slowo[1:100]
'omocA'
>>> slowo[10:]
''
>>> slowo[2:1]
''
```

Aby wyznaczyć znaki, licząc od strony prawej łańcucha znaków, używa się indeksów będących liczbami ujemnymi . Oto przykład:

```
>>> slowo[-1]     # Ostatni znak
'A'
>>> slowo[-2]     # Przedostatni znak
'c'
>>> slowo[-2:]    # Dwa ostatnie znaki
'cA'
>>> slowo[:-2]    # Wszystkie, oprócz dwóch ostatnich znaków
'Pomo'
```

Proszę zauważyć, że -0 oznacza to samo co 0, tak więc nie oznacza liczenia od prawej!

```
>>> slowo[-0]     # (ponieważ -0 jest równe 0)
'P'
```

Ujemne wykrojenia, które są przekraczają ograniczenia łańcucha są skracane, ale nie próbuj tego na indeksach jednoelementowych (nie oznaczających wykrawania):

```
>>> slowo[-100:]
'PomocA'
>>> word[-10]    # błąd
Traceback (innermost last):
  File "<stdin>", line 1
IndexError: string index out of range
```

(**IndexError***: indeks napisu poza jego granicami*)

Najlepszą formą zapamiętania sposobu działania wykrawania jest wyobrażenie sobie indeksów jako wskazówek odnoszących się do miejsc *pomiędzy* znakami, gdzie lewa krawędź pierwszego znaku nosi numer 0. Tak więc, prawa krawędź ostatniego znaku w łańcuchu *n* znaków posiada indeks *n*, np.:

```
+---+---+---+---+---+---+
| P | o | m | o | c | A |
+---+---+---+---+---+---+
0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

W pierwszym rzędzie liczb możemy zobaczyć pozycje wyznaczone przez indeksy 0...6. Drugi rząd zawiera odpowiednie indeksy ujemne. Wycinek od *i* do *j* składa się ze wszystkich znaków pomiędzy krawędziami oznaczonymi odpowiednio *i* i *j*.

Długość wycinka można obliczyć z róznicy nieujemnych indeksów, pod warunkiem, że oba mieszczą się w granicach ciągu, np. długość `slowo[1:3]` równa jest 2.

Wbudowana w interpreter funkcja `len()` zwraca długość łańcucha:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

3.1.3 Napisy Unicode

Od wersji 1.6 wprowadzono nowy typ danych: łańcuchy znaków Unicode. Mogą one zostać użyte do przechowywania i manipulacji danymi typu Unicode (zobacz http://www.unicode.org). Zostały one w niezły sposób zintegrowane z istniejącym w Pythonie systemem napisów i dostarczają niezbędnych konwersji tam, gdzie trzeba.

Unicode posiada przewagę w oznaczeniu każdego znaku używanego w starożytnych i nowoczesnych systemach piśmienniczych. Wcześniejsze rozwiązania umożliwiają przechowywanie tylko 256 znaków, których pozycje w tym zestawie zdefiniowane były przez strony kodowe. Prowadzi to do dużego zamieszania, zwłaszcza w odniesieniu do internalizacji oprogramowania (zwykle nazywa się to „`i18n`" — „`i`" + 18 znaków + „`n`" = `internalization`). Unicode rozwiązuje te problemy poprzez zdefiniowanie jednej strony kodowej dla wszystkich systemów piśmienniczych.

Stworzenie łańcucha znaków Unicode w Pythonie jest tak samo proste jak w przypadku zwykłego:

```
>>> u'Hello World !'
u'Hello World !'
```

Mała litera „`u`" z przodu łańcucha oznacza, że zostanie stworzony łańcuch Unicode. Jeśli chce się umieścić jakieś znaki specjalne w tym łańcuchu, można zrobić to poprzez kodowanie *Unicode-Escape*. Poniższy przykład pokazuje jak to uczynić:

```
>>> u'Hello\\u0020World !'
u'Hello World !'
```

Sekwencja cytowania (*ang. escape sequence*) `\\u0020` oznacza wstawienie znaku Unicode na podanej pozycji z kodem określonym szesnastkowo 0x0020 (znak spacji).

Inne znaki interpretowane są poprzez użycie ich odpowiedniego numeru porządkowego wprost jako porządku wyrażonego w Unicode. Fakt, iż pierwsze 256 znaków Unicode są takie same jak standardowego zestawu Latin-1 używanego w wielu krajach zachodnich, proces kodowania ich w Unicode bardzo się upraszcza.

Kąsek dla ekspertów: istnieje także tryb „surowy" wprowadzania znaków, podobnie jak dla zwykłych łańcuchów. Trzeba dołączyć z przodu łańcucha znak 'r', aby Python traktował wszystko w tym łańcuchu jako znaki kodowane w trybie *Raw-Unicode-Escape*. Będzie się to stosowało tylko do konwersji `\\uXXXX`, tak jak w przykładzie powyżej, gdy pojawia się nieparzysta liczba ukośników z przodu litery 'u'.

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\\\u0020World !'
```

Tryb surowy jest najbardziej użyteczny w przypadku, gdy trzeba wprowadzić mnóstwo ukośników, np. jakieś wyrażenie regularne.[3]

Oprócz wspomnianych przed chwilą standardowych sposobów kodowania znaków, Python dostarcza wiele innych sposobów tworzenia łańcuchów Unicode opartych na znanych już kodowaniach.

Funkcja wbudowana `unicode()` dostarcza środków dostępu do wszystkich zarejestrowanych kodeków Unicode (tzw. COders i DECoders). Niektóre z bardziej znanych kodowań, dla których istniejące kodeki mogą przeprowadzić konwersje to *Latin-1*, *ASCII*, *UTF-8* i *UTF-16*. Dwa ostatnie są systemami kodowania zmiennej długości, które pozwalają przechowywać znaki Unicode w 8 lub 16 bitach. Python używa UTF-8 jako domyślnego kodowania. Staje się to ważne, gdy decydujemy się umieszczać takie znaki w plikach lub drukować.

```
>>> u"äöü"
u'\344\366\374'
>>> str(u"äöü")
'\303\244\303\266\303\274'
```

Jeśli przechowujesz dane kodowane w specyficzny sposób i chce się wyprodukować odpowiadający im łańcuch Unicode, można użyć `unicode()` z podaną nazwą kodowania jako drugi parametr wywołania.

```
>>> unicode('\303\244\303\266\303\274','UTF-8')
u'\344\366\374'
```

Metoda łańcucha znakowego `encode()` pozwala dokonać odwrotnej konwersji.

```
>>> u"äöü".encode('UTF-8')
'\303\244\303\266\303\274'
```

3.1.4 Listy

Istnieje w Pythonie pewna liczba złożonych typów danych, używanych do grupowania innych wartości. Najbardziej użytecznym typem jest *lista*, którą można zapisać jako listę elementów poprzedzielanych przecinkiem, umieszczoną w kwadratowych nawiasach. Elementy listy nie muszą być tego samego typu.[4]

```
>>> a = ['wędzonka', 'jaja', 100, 1234]
>>> a
['wędzonka', 'jaja', 100, 1234]
```

Podobnie jak indeksy łańcuchów znaków, indeksy listy rozpoczynają się od wartości 0. Listy mogą być przedmiotem operacji wykrawania, sklejania itd.:

[3] Moduł „`re`” do obsługi wyrażeń regularnych opisany został w «Opisie biblioteki» (*przyp. tłum.*)
[4] I to jest to, co najbardziej rajcuje tygryski… (*przyp. tłum.*)

```
>>> a[0]
'wędzonka'
>>> a[3]
1234
>>> a[-2]
100
>>> a[1:-1]
['jaja', 100]
>>> a[:2] + ['bekon', 2*2]
['wędzonka', 'jaja', 'bekon', 4]
>>> 3*a[:3] + ['Srutututu!']
['wędzonka', 'jaja', 100, 'wędzonka', 'jaja', 100, 'wędzonka', 'jaja', 100,
'Srutututu!']
```

Odmiennie niż napisy, które są *niemutowalne*, można zmieniać poszczególne elementy listy:

```
>>> a
['wędzonka', 'jaja', 100, 1234]
>>> a[2] = a[2] + 23
>>> a
['wędzonka', 'jaja', 123, 1234]
```

Możliwe jest także przypisanie do wycinka, co może także zmienić długość listy:

```
>>> # Zastąp pewne elementy:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Inne usuń:
... a[0:2] = []
>>> a
[123, 1234]
>>> # Włóż parę:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a     # Wstaw kopię siebie na początek
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

Wbudowana funkcja `len()` również dotyczy list:

```
>>> len(a)
8
```

Możliwe jest zagnieżdżanie list (tworzenie list, której elementami są inne listy), np:

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
>>> p[1].append('xtra')     # Zobacz podpunkt 5.1
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Zauważcie, że w ostatnim przykładzie `p[1]` i `q` tak naprawdę odnoszą się do tego samego egzemplarza obiektu! Powrócimy do *obiektowej semantyki* później.

3.2 Pierwsze kroki w programowaniu

Dodawanie 2 do 2 jest oczywiście nie wszystkim, co można zrobić w Pythonie. Możemy na przykład napisać początkowe elementy ciągu *Fibonacciego*: [5]

```
>>> # Ciąg Fibonacciego:
... # suma dwóch elementów definiuje następny element
... a, b = 0, 1
>>> while b < 10:
...       print b
...       a, b = b, a+b
...
1
1
2
3
5
8
```

Powyższy przykład ukazuje nam parę nowych cech Pythona.

- Pierwsza linia zawiera tzw. *wielokrotne przypisanie*: zmiennym `a` i `b` przypisuje się jednocześnie wartości 0 i 1. W ostatniej linii użyto jeszcze raz tego mechanizmu, gdzie możemy się przekonać, że wyrażenia po prawej stronie przypisania są obliczane w pierwszej kolejności, zanim jakiekolwiek przypisanie ma miejsce. Wyrażenia te obliczane są od lewej do prawej.

- Pętla `while` wykonuje się do chwili, gdy warunek (w tym przypadku: `b < 10`) jest prawdziwy. W Pythonie, podobnie jak w C, każda niezerowa wartość całkowita oznacza prawdę; zero oznacza fałsz. Warunek może być określony przez łańcuch znaków lub listę wartości — tak naprawdę, każda sekwencja o długości niezerowej oznacza prawdę, a puste ciągi oznaczają fałsz. Test użyty w powyższym przykładzie jest prostym porównaniem. Standardowe operatory porównania są takie same jak w C: `<` (mniejszy niż), `>` (większy niż), `==` (równy), `<=` (mniejszy równy),

- *Ciało* pętli jest *wcięte*: indentacja (wcinanie) jest sposobem na grupowanie instrukcji. Obecnie Python nie dostarcza żadnych udogodnień związanych z inteligentną edycją wierszy wejściowych, tak więc trzeba wprowadzić znak spacji lub tabulacji, aby wciąć wiersz. W praktyce programy w Pythonie będzie się pisać w jakimś edytorze tekstów, a większość z nich posiada coś na kształt mechanizmu auto-indentacji. W chwili, gdy wprowadza się jakąś instrukcję złożoną w czasie sesji interpretera Pythona, trzeba zakończyć

[5] Kiedy się wreszcie skończy katowanie Fibbonaciego?! (*przyp. tłum.*)

ją pustym wierszem (bowiem interpreter nie wie, czy ostatni wprowadzony wiersz jest ostatnim z tej instrukcji). Ważne jest, aby każdy wiersz należący do tej samej grupy instrukcji, był wcięty o taką samą liczbę spacji lub znaków tabulacji.

- Instrukcja `print` zapisuje na standardowym wyjściu wyrażenie, które stoi za nim. Różnica pomiędzy tą instrukcją, a zwykłym zapisem wyrażenia, które chce się wypisać (tak jak robiliśmy to w przykładzie z kalkulatorem) występuje w sposobie obsługi wielu wyrażeń i napisów. Łańcuchy znaków wypisywane są bez cudzysłowów, a pomiędzy nimi zapisywane są spacje, tak aby można było ładnie sformatować pojawiający się napis, np:

```
>>> i = 256*256
>>> print 'Wartością i jest', i
Wartością i jest 65536
```

Przecinek postawiony na końcu instrukcji `print` powoduje pozostanie w tym samym wierszu po zakończeniu wypisywania:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print b,
...     a, b = b, a+b
...
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Zauważcie, że interpreter wstawia znak nowej linii, zanim wydrukuje znak zachęty, jeżeli poprzedni wiersz nie był zakończony.

ROZDZIAŁ

CZWARTY

Jeszcze więcej sposobów na kontrolowanie programu

Python wyposażony jest w zestaw wielu instrukcji, nie tylko w `while`, którą już poznaliśmy. Są one dobrze znane z innych języków programowania, choć posiadają swój lokalny koloryt.

4.1 Instrukcje `if`

Zdaje mi się, że najbardziej znaną instrukcją jest instrukcja `if`. Np.:

```
>>> x = int(raw_input("Proszę podać liczbę: "))
>>> if x < 0:
...      x = 0
...      print 'Ujemna zmieniła się na zero'
... elif x == 0:
...      print 'Zero'
... elif x == 1:
...      print 'Jeden'
... else:
...      print 'Więcej'
...
```

Można wstawić zero lub więcej części `elif`, a część `else` jest opcjonalna. Słowo kluczowe `elif` jest skrótem instrukcji 'else if' i przydaje się, gdy chce się uniknąć kolejnej indentacji. Sekwencja `if` … `elif` … `elif` … zastępuje instrukcje *switch* lub *case* spotykane w innych językach.[1]

4.2 Instrukcje `for`

Instrukcja `for` różni się troszeczkę w Pythonie od tego, co używasz w C lub Pascalu. Nie prowadzi się iteracji od liczby do liczby (jak w Pascalu) lub daje się użytkownikowi możliwość definiowania kroku iteracji i warunki zakończenia iteracji (jak w C). Instrukcja `for` w Pythonie powoduje iterację po elementach jakiejkolwiek sekwencji (np. listy lub łańcucha znaków), w takim porządku, w jakim są one umieszczone w danej sekwencji. Na przykład:

[1] Czyli instrukcje wyboru. (*przyp. tłum.*)

```
>>> # Mierzy pewne napisy:
... a = [’kot’, ’okno’, ’wypróżnić’]
>>> for x in a:
...     print x, len(x)
...
kot 3
okno 4
wypróżnić 9
```

Nie jest bezpiecznym modyfikacja sekwencji, która właśnie jest przedmiotem iteracji (można to zrobić tylko dla mutowalnego typu sekwencji, tzn. listy). Jeśli chce się ją modyfikować, np. duplikować wybrane jej elementy, to przeprowadź iterację na jej kopii. Notacja wykrawania jest tu szczególnie użyteczna:

```
>>> for x in a[:]: # wykrój całą listę (zrób jej kopię)
...    if len(x) > 6: a.insert(0, x)
...
>>> a
[’wypróżnić’, ’kot’, ’okno’, ’wypróżnić’]
```

4.3 Funkcja `range()`

Jeśli zaszła potrzeba iteracji określonej zakresem liczbowym (czyli iteracji na sekwencji liczb w Pythonie), można użyć wbudowanej w interpreter funkcji `range()`. Wynikiem jej działania jest lista zawierająca ciąg arytmetyczny, tzn.:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Podany na jej wejściu punkt końcowy nigdy nie zostanie zawarty w wynikowej liście. `range(10)` tworzy listę 10 wartości, a tak naprawdę dokładnie zbiór dopuszczalnych wartości indeksów dla listy o długości 10. Jeśli jest taka potrzeba, można określić liczbę początkową tego ciągu albo krok (nawet liczbę ujemną):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Aby przeglądnąć wszystkie elementy listy łączy się funkcje `range()` i `len()`, tak jak poniżej:

```
>>> a = [’Marysia’, ’miała’, ’małego’, ’baranka’]
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Marysia
1 miała
2 małego
3 baranka
```

4.4 Instrukcja `break` i `continue` oraz klauzule `else` w pętlach

Instrukcja `break`, podobnie jak w C, powoduje wyjście z najbliżej zagnieżdżonej pętli `for` lub `while`.

Instrukcja `continue` została również zapożyczona z C, powoduje przejście do następnego kroku iteracji w pętli.

Instrukcje pętli posiadają klauzulę `else`: jest ona wykonywana w momencie zakończenia działania pętli przy wyczerpaniu się (dojścia za ostatni element) listy (pętla `for`) lub gdy warunek pętli zwraca wartość fałszu (pętla `while`), ale nie w momencie opuszczenia pętli w wyniku zadziałania instrukcji `break`. Pokazane to zostało na przykładzie algorytmu poszukiwania liczby pierwszej:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...            print n, 'równe', x, '*', n/x
...            break
...     else:
...          print n, 'jest liczbą pierwszą'
...
2 jest liczbą pierwszą
3 jest liczbą pierwszą
4 równe 2 * 2
5 jest liczbą pierwszą
6 równe 2 * 3
7 jest liczbą pierwszą
8 równe 2 * 4
9 równe 3 * 3
```

4.5 Instrukcje `pass`

Instrukcja `pass` nic nie robi. Może być użyta wszędzie tam, gdzie wymagana jest jakaś instrukcja z powodów składniowych, ale program nie przewiduje w tym miejscu żadnego działania. Na przykład:

```
>>> while 1:
...       pass # Zajęty-poczekaj na naciśnięcie klawisza
...
```

4.6 Definiowanie funkcji

Możemy stworzyć funkcję, która wypisuje ciąg Fibonaciego o wybranych granicach:

```
>>> def fib(n):    # wypisz ciąg Fibonacciego aż do n
...     "Wypisuje ciąg Fibonacciego aż do n"
...     a, b = 0, 1
...     while b < n:
...         print b,
...         a, b = b, a+b
...
>>> # Teraz, wywołajmy funkcję, którą przed chwilą zdefiniowaliśmy:
... fib(2000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Słowo kluczowe `def` wprowadza *definicję* funkcji. Musi po nim następować nazwa funkcji i lista jej parametrów formalnych umieszczonych w nawiasach okrągłych. Instrukcje, które tworzą ciało funkcji, są oczywiście wsunięte

w stosunku do wiersza zawierającego nazwę funkcji i muszą zaczynać się w nowym wierszu. Opcjonalnie, pierwszy wiersz ciała funkcji może być gołym napisem (literałem): jest to tzw. napis dokumentujący lub (inna nazwa tego zjawiska) *docstring*.

Istnieją pewne narzędzia, które używają napisów dokumentacyjnych (docstringów) do automatycznego tworzenia drukowanej dokumentacji albo pozwalają użytkownikowi na interaktywne przeglądanie kodu. Dobrym zwyczajem jest pisane napisów dokumentacyjnych w czasie pisania programu: spróbuj się do tego przyzwyczaić.

Wykonanie funkcji powoduje stworzenie nowej tablicy symboli lokalnych używanych w tej funkcji. Mówiąc precyzyjniej: wszystkie przypisania do zmiennych lokalnych funkcji powodują umieszczenie tych wartości w lokalnej tablicy symboli, z czego wynika, że odniesienia do zmiennych najpierw szukają swych wartości w lokalnej tablicy symboli, a potem w globalnej, a dopiero na końcu w tablicy nazw wbudowanych w interpreter. Tak więc, zmiennym globalnym nie można wprost przypisać wartości w ciele funkcji (chyba, że zostaną wymienione w niej za pomocą instrukcji `global`), aczkolwiek mogą w niej być używane (czytane).

Parametry wywołania funkcyjnego (argumenty) wprowadzane są do lokalnej tablicy symboli w momencie wywołania funkcji. Tak więc, argumenty przekazywane są jej *przez wartość* (gdzie *wartość* jest zawsze *odniesieniem* do obiektu, a nie samym obiektem).[2] Nowa tablica symboli tworzona jest również w przypadku, gdy funkcja wywołuje inną funkcję.

Definicja funkcji wprowadza do aktualnej tablicy symboli nazwę tej funkcji. Nazwa ta identyfikuje wartość, której typ rozpoznawany jest przez interpreter jako funkcja zdefiniowana przez użytkownika. Wartość ta (a właściwie obiekt (*przyp. tłum.*)) może być przypisana innej nazwie, która potem może zostać użyta jak funkcja. Ta właściwość może posłużyć jako ogólny mechanizm zmiany nazw:

```
>>> fib
<function object at 10042ed0>
>>> f = fib
>>> f(100)
1 1 2 3 5 8 13 21 34 55 89
```

Możnaby tutaj naprostować moje opowieści, że `fib` nie jest funkcją, ale procedurą. W Pythonie, podobnie jak w C, procedury są specyficznymi funkcjami, które nie zwracają wartości[3] Tak naprawdę, mówiąc językiem technicznym, procedury naprawdę zwracają wartość, aczkolwiek raczej nudną.[4] Wartość ta nazywana jest `None` (jest to nazwa wbudowana). Napisanie wartości `None` jest w normalnych warunkach pomijane przez interpreter, jeżeli jest to jedyna wartość, która miała być wypisana. Można ją zobaczyć, jeżeli naprawdę tego się chce:

```
>>> print fib(0)
None
```

Bardzo proste jest napisanie funkcji, która zwraca listę liczb ciągu Fibonacciego, zamiast wypisywać je:

[2] Właściwie *wywołanie przez odniesienie*, byłoby lepszym określeniem, ponieważ jeżeli przekazywane jest odniesienie do obiektu mutowalnego, to wywołujący funkcję zobaczy wszystkie zmiany dokonane na takim obiekcie (np. elementy wstawione do listy).

[3] Więc nie są to funkcje. W C lub C++procedury-funkcje zwracają wartość `void` (przynajmniej składniowo)! I tam na pewno są to funkcje, specyficzne, ale zawsze funkcje! (*przyp. tłum.*)

[4] Czyli jednak są to funkcje! (*przyp. tłum.*)

```
>>> def fib2(n): # zwraca wartości ciągu Fibonacciego aż do n
...     "Zwraca wartości ciągu Fibonacciego, aż do n"
...     wynik = []
...     a, b = 0, 1
...     while b < n:
...         wynik.append(b)    # zobacz poniżej
...         a, b = b, a+b
...     return wynik
...
>>> f100 = fib2(100)    # wywołaj ją
>>> f100                # wypisz wynik
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Ten przykład„ jak zwykle, demonstruje parę nowych pythonowatych właściwości:

- Instrukcja `return` powoduje powrót z funkcji z pewną wartością. `return` bez wyrażenia za nim używane jest do powrotu ze środka procedury (dojście do końca procedury, również to powoduje), przy czym w takim wypadku do wywołującego powraca wartość `None`.
- Instrukcja `wynik.append(b)` wywołuje *metodę* obiektu listy `wynik`. Metoda jest funkcją „należącą" do obiektu i nazywa się `obiekt.nazwametody`, gdzie `obiekt` jest jakimś tam obiektem (równie dobrze może to być wyrażenie), a `nazwametody` jest nazwą metody zdefiniowanej przez typ obiektu `obiekt`. Różne typy definiują różne metody. Metody różnych typów mogą mieć te same nazwy bez powodowania niejednoznaczności. (Możliwe jest zdefiniowane swoich własnych typów i metod przy użyciu *klas*, tak jak pokazane to będzie później). Metoda `append()` użyta w przykładzie, zdefiniowana jest dla listy obiektów: dodaje nowe elementy do końca listy. W tym przykładzie jest to odpowiednik „`wynik = wynik + [b]`", ale jest bardziej wydajne.

4.7 Jeszcze więcej o definiowaniu funkcji

Możliwe jest definiowanie funkcji ze zmienną liczbą argumentów. Istnieją trzy formy takiej definicji, które mogą być ze sobą splatane.

4.7.1 Domyślne wartości argumentów

Najbardziej użyteczną formą jest określenie domyślnej wartości dla jednego lub większej liczby argumentów. W ten sposób funkcja może zostać wywołana z mniejszą liczbą argumentów, niż była zdefiniowana, tzn.:

```
def zapytaj_ok(zacheta, liczba_prob=4, zazalenie='Tak lub nie, bardzo
proszę!'):       while 1:
        ok = raw_input(zacheta)
        if ok in ('t', 'ta', 'tak'): return 1
        if ok in ('n', 'nie', 'ee', 'gdzie tam'): return 0
        liczba_prob = liczba_prob - 1
        if liczba_prob < 0: raise IOError, 'użytkownik niekumaty'
        print zazalenie
```

Funkcja ta może zostać wywołana w taki sposób: `zapytaj_ok('Naprawdę chcesz zakończyć?')` lub w taki: `zapytaj_ok('Zgadzasz się nadpisać ten plik?', 2)`.

Wartości domyślne określane są w punkcie definicji funkcji, w przestrzeni *definiowania* nazw, tak więc np.:

```
i = 5
def f(arg = i): print arg
i = 6
f()
```

wypisze 5.

Ważne ostrzeżenie: wartość domyślna określana jest tylko raz. Ma to znaczenie w przypadku, gdy wartością tą jest obiekt mutowalny, jak np. lista czy słownik. W przykładzie poniżej, funkcja akumuluje argumenty przekazane jej w kolejnych wywołaniach:

```
def f(a, l = []):
    l.append(a)
    return l
print f(1)
print f(2)
print f(3)
```

Instrukcje `print` spowodują:

```
[1]
[1, 2]
[1, 2, 3]
```

jeśli nie chce się, aby wartości domyślne były współdzielone pomiędzy kolejnymi wywołaniami funkcji, można ją napisać w taki sposób:

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

4.7.2 Argumenty kluczowe

Funkcja może być wywołana z użyciem argumentów kluczowych, tzn. w formie „*klucz* = *wartość*”. Na przykład, poniższa funkcja:

```
def papuga(napiecie, stan='racja', akcja='voom', typ='Norwegian Blue'):
    print "-- Ta papuga nie zrobiłaby", akcja
    print "jeśli przyłożysz", napiecie, "woltów do niej."
    print "-- Śliczne upierzenie, ten", typ
    print "-- Tak,", stan, "!"
```

mogłaby być wywołana na parę różnych sposobów:

```
papuga(1000)
papuga(akcja = 'VOOOOOM', napiecie = 1000000)
papuga('tysiąc', stan = 'już wącha kwiatki od spodu')
parrot('milion', 'bereft of life', 'skoku')
```

lecz poniższe wywołania byłyby nieprawidłowe:

```
papuga()                      # brakuje wymaganego argumentu
papuga(napiecie=5.0, 'trup')  # niekluczowy argument za kluczowym
papuga(110, napiecie=220)     # zduplikowana wartość argumentu
papuga(aktor='John Cleese')  # nieznana nazwa argumentu
```

W ogólności mówiąc,[5] lista argumentów wywołania, musi mieć jakiś argument pozycyjny, po którym następuje jakikolwiek argument kluczowy, gdzie klucze wybrane są z listy parametrów formalnych. Nie jest ważne, czy parametr formalny ma wartość domyślną, czy też nie. Żaden z argumentów nie może otrzymać wartości więcej niż jeden raz — nazwy parametrów formalnych odpowiadające argumentom pozycyjnym w wywołaniu nie mogą być w nim użyte jako kluczowe. Oto przykład wywołania, które się nie powiedzie z uwagi na wymienione przed chwilą ograniczenia:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: keyword parameter redefined
```

Gdy na liście parametrów formalnych funkcji widnieje ***nazwa*, to przy wywołaniu funkcji przypisywany jest mu słownik zawierający wszystkie klucze, które nie odpowiadają nazwom parametrów formalnych. Mechanizm ten może być połączony z wystąpieniem parametru formalnego o nazwie **nazwa* (co zostanie opisane w następnym podrozdziale), który w momencie wywołania staje się krotką (ang. tuple), która zawiera wszystkie argumenty pozycyjne (niekluczowe) wymienione w wywołaniu funkcji za parametrami formalnymi. (**nazwa* musi pojawić się przed ***nazwa* na liście parametrów formalnych). Jeżeli, na przykład, zdefiniujemy funkcję w ten sposób:

```
def sklep_z_serami(rodzaj, *argumenty, **klucze):
    print "-- Czy macie", rodzaj, '?'
    print "-- Przykro mi,", rodzaj, "właśnie się skończył."
    for arg in argumenty: print arg
    print '-'*40
    for kl in klucze.keys(): print kl, ':', klucze[kl]
```

która mogłaby być wywołana o tak[6]:

```
sklep_z_serami('Limburger', "Jest bardzo dojrzały, proszę pana.",
           "Jest naprawdę bardzo, BARDZO dojrzały, proszę pana.",
           klient='John Cleese',
           wlasciciel='Michael Palin',
           skecz='Skecz ze sklepem z serami')
```

co oczywiście spowoduje wypisanie:

[5] jak mawiał Nikuś Dyzma (*przyp. tłum.*)

[6] To jest bardzo nieudolne tłumaczenie tego prześmiesznego skeczu. Ś.p. Beksiński zrobił to wiele śmieszniej: niestety, nie miałem tego nagrania przy sobie (*przyp. tłum.*)

```
-- Czy macie Limburger ?
-- Przykro mi, Limburger właśnie się skończył.
Jest bardzo dojrzały, proszę pana.
Jest naprawdę bardzo, BARDZO dojrzały, proszę pana.
----------------------------------------
klient : John Cleese
wlasciciel : Michael Palin
skecz : Skecz ze sklepem z serami
```

4.7.3 Lista arbitralnych argumentów

Ostatnim sposobem na to, aby funkcja została wywołana w dowolną liczbą argumentów jest wprost określenie tego w definicji. Wszystkie argumenty zostaną zapakowane w krotkę.[7] Przedtem na liście argumentów może pojawić się zero lub więcej normalnych argumentów.

```
def fprintf(plik, format, *args):
    plik.write(format % args)
```

4.7.4 Formy lambda

Ze względu na wzrastającą liczbę głosów użytkowników, dodano do Pythona parę nowych właściwości spotykanych zwykle w językach funkcjonalnych. Za pomocą słowa kluczowego `lambda` możesz tworzyć małe, anonimowe (czyli nienazwane) funkcje. Oto funkcja, która zwraca sumę jej dwóch argumentów: „`lambda a, b: a+b`”. Formy lambda mogą zostać użyte we wszystkich miejscach, gdzie wymagane są obiekty funkcji. Składniowo ograniczone są do pojedynczego wyrażenia. Semantycznie, są właściwie szczyptą cukru na składnię zwykłych definicji funkcji. Podobnie jak zagnieżdżone definicje funkcji, formy lambda nie mogą używać nazw zmiennych z zakresu zewnętrznego„ lecz może to być ominięte poprzez sprytne użycie argumentów domyślnych:

```
def stworz_powiekszacza(n):
    return lambda x, incr=n: x+incr
```

4.7.5 Napisy dokumentujące

Istnieje pewna konwencja dotycząca zawartości i formatowania napisów dokumentujących.

Pierwszy wiersz powinien być krótki, zwięźle podsumowujący działanie obiektu. Dla zwięzłości nie powinien wprost określać nazwy obiektu lub jego typu albowiem atrybuty te są dostępne dla czytelnika za pomocą innych środków (z wyjątkiem czasownika opisującego działanie funkcji). wiersz ten powien zaczynać się od dużej litery i kończyć kropką.

Jeśli napis zawiera więcej linii, drugi wiersz powinien być pusty, wizualnie oddzielając resztę opisu. Następne wiersze powinny obejmować jeden lub więcej akapitów, opisując sposób wywołania obiektu, efekty uboczne itd.

Parser Pythona nie wycina znaków tabulacji i wcięcia z wielowierszowych literałów napisów, dlatego też narzędzia do produkcji dokumentacji muszą to robić same, jeśli jest to wymagane. Robi się to wg następującej konwencji. Pierwszy niepusty wiersz *po* pierwszym wierszu napisu określa wielkość wsunięcia całego napisu dokumentującego. (Nie można użyć pierwszego wiersza, gdyż zazwyczaj położony jest bliżej otwierającego napis cydzysłowia, tak, że indentacja nie jest widoczna). Ekwiwalent tego wcięcia złożony z „białych znaków” (czyli spacji, tabulacji) jest wycinany z początków wierszy całego napisu. Wiersze, które są wcięte mniej niż ta ilość, nie powinny się w napisie pojawić, ale jeśli to się stanie, to wszystkie znaki białe z przodu wiersza zostaną

[7] listę niemutowalną (*przyp. tłum.*)

usunięte. Równowartość wcięcia liczona w spacjach powinna być sprawdzona po rozwinięciu znaków tabulacji (zazwyczaj do 8 spacji).

Oto przykład wielowierszowego docstring'u:

```
>>> def moja_funkcja():
...     """Nie rób nic, ale udokumentuj to.
...
...     Naprawdę, tutaj niczego się nie robi.
...     """
...     pass
...
>>> print moja_funkcja.__doc__
Nie rób nic, ale udokumentuj to.

     Naprawdę, tutaj niczego się nie robi.
```

ROZDZIAŁ

PIĄTY

Struktury danych

Rozdział ten opisuje pewne rzeczy, o których powiedziano już szczegółowo oraz dodaje parę nowinek.

5.1 Więcej o listach

Typ listy posiada pewien zbiór metod. Oto wszystkie jego metody:

`append(x)` Dodaje element do kończ listy, odpowiednik `a[len(a):] = [x]`.

`extend(L)` Rozszerza listę poprzez dołączenie wszystkich elementów podanej listy *L*, odpowiednik `a[len(a):] = L`.

`insert(i, x)` Wstawia element na podaną pozycję listy. Pierwszym argumentem wywołania jest indeks elementu, przed którym nowy element ma zostać wstawiony: tak więc `a.insert(0,x)` wstawia na początek listy, a `a.insert(len(a),x)` jest odpowiednikiem `a.append(x)`

`remove(x)` Usuwa pierwszy napotkany element z listy, którego wartością jest `x`. Jeżeli nie ma na liście takiego elementu, zgłaszany jest błąd.

`pop(`[**`i`**]**`)`** Usuwa element z podanej pozycji na liście i zwraca go jako wynik. Jeżeli nie podano żadnego indeksu `a.pop()`, zwraca ostatni element na liście. Oczywiście, jest on z niej usuwany.

`index(x)` Zwraca indeks pierwszego elementu listy, którego wartością jest `x`. Jeżeli nie ma takiego elementu zgłaszany jest błąd.

`count(x)` Zwraca liczbę wystąpień elementu `x` na liście.

`sort()` Sortuje elementy na liście, w niej samej. (W wyniku nie jest tworzona nowa, posortowana lista (*przyp. tłum.*))

`reverse()` Odwraca porządek elementów listy, również w niej samej.

Przykład, w którym użyto większość z podanych powyżej metod:

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

5.1.1 Używanie listy jako stosu

Użycie listy w roli stosu typu „last-in, first-out" (ostatni dodany element jest pierwszym pobranym) jest bardzo łatwe gdy spróbujemy wykorzystać jej metody. Aby dodać element na szczyt stosu, należy użyć `append()`. `pop()`, bez podawania wprost indeksu, używane jest do pobrania elementu ze szczytu stosu. Na przykład:

```
>>> stos = [3, 4, 5]
>>> stos.append(6)
>>> stos.append(7)
>>> stos
[3, 4, 5, 6, 7]
>>> stos.pop()
7
>>> stos
[3, 4, 5, 6]
>>> stos.pop()
6
>>> stos.pop()
5
>>> stos
[3, 4]
```

5.1.2 Użycie listy jako kolejki

Można używać listy równie wygodnie w roli kolejki „first-in, first-out" (pierwszy dodany element jest pierwszym pobieranym). Aby dodać element na koniec kolejki, użyj `append()`. Aby pobrać element z przodu kolejki, użyj `pop(0)`. Na przykład:

```
>>> kolejka = ["Eric", "John", "Michael"]
>>> kolejka.append("Terry")           # przybywa Terry
>>> kolejka.append("Graham")          # przybywa Graham
>>> kolejka.pop(0)
'Eric'
>>> kolejka.pop(0)
'John'
>>> kolejka
['Michael', 'Terry', 'Graham']
```

5.1.3 Mechanizmy programowania funkcjonalnego

Istnieją trzy, bardzo użyteczne przy pracy z listami, funkcje: `filter()`, `map()`, i `reduce()`.

„`filter(`*funkcja*`,` *sekwencja*`)`” zwraca sekwencje (tego samego typu, gdy to możliwe) zawierającą te elementy z listy wejściowej, dla których wywołanie *funkcja*`(`*element*`)` zwróci wartość prawdziwą. Oto przykład obliczania liczb pierwszych:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

„`map(`*funkcja*`,` *sekwencja*`)`” wywołuje *funkcja*`(`*element*`)` dla każdego elementu listy wejściowej i zwraca listę wartości zwróconych przez *funkcja*. Na przykład, aby obliczyć sześcian dla każdego elementu z ciągu liczb:

```
>>> def sześcian(x): return x*x*x
...
>>> map(szescian, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

Przekazana może zostać więcej, niż jedna sekwencja — funkcja *funkcja* musi mieć wtedy tyle argumentów, ile zostało podanych sekwencji i jest wywoływana z poszczególnym elementem z każdej sekwencji wejściowej (lub z `None` jeśli któraś z nich jest krótsza od innej). Jeżeli `None` został przekazany zamiast pierwszego argumentu `map`, funkcja zwracająca swoje argumenty jest zastępowana.[1]

Składając te dwa przypadki zauważmy, iż „`map(None,` *lista1*`,` *lista2*`)`” jest wygodnym sposobem przekształcenia pary list w listę par. Na przykład:

```
>>> sekw = range(8)
>>> def kwadrat(x): return x*x
...
>>> map(None, sekw, map(kwadrat, sekw))
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49)]
```

„`reduce(`*funkcja*`,` *sekwencja*`)`” zwraca pojedynczą wartość, która powstała w wyniku: wywołania dwuparametrowej funkcji *funkcja* dla dwóch pierwszych elementów sekwencji, potem dla wyniku tego działania i następnego elementu sekwencji itd. Na przykład, aby obliczyć sumę liczb od 1 do 10:

```
>>> def dodaj(x,y): return x+y
...
>>> reduce(dodaj, range(1, 11))
55
```

Jeśli istnieje tylko jeden element w sekwencji, zwracana jest jego wartość. Jeżeli sekwencja jest pusta, zgłaszany jest wyjątek.

Można przekazać początkową wartość jako trzeci argument wywołania. W tym przypadku wartość ta jest zwracana, gdy sekwencja jest pusta. Funkcja jest stosowana dla wartości początkowej i pierwszego elementu sekwencji, a następnie wynik tej operacji stanowi argument wejściowy wraz z następnym elementem itd. Oto przykład:

[1] Nic z tego nie rozumiem. Taki ze mnie tłumacz... (*przyp. tłum.*)

```
>>> def suma(sekw):
...     def dodaj(x,y): return x+y
...     return reduce(dodaj, sekw, 0)
...
>>> suma(range(1, 11))
55
>>> suma([])
0
```

5.1.4 Rozszerzenia składni list

Są one spójnym sposobem na tworzenie list bez odwoływania się do `map()`, `filter()` i/lub `lambda`. Przedstawione poniżej definicje list często są bardziej przejrzyste niż listy tworzone za pomocą w.w. konstrukcji. Każda z rozszerzonych konstrukcji składa się z wyrażenia, po którym następuje klauzula `for`, potem zero lub więcej klauzuli `for` lub `if`. W rezultacie otrzymujemy listę powstałą w wyniku wyliczenia wyrażenia w kontekście klauzul `for` i `if`, które po nim występują. Jeśli wyrażenie ma typ krotki, musi zostać objęte nawiasami okrągłymi.

```
>>> owoce = [’  banan’, ’  jerzyna ’, ’owoc pasji  ’]
>>> [uzbrojenie.strip() for bron in owoce]
[’banan’, ’jerzyna’, ’owoc pasji’]
>>> wek = [2, 4, 6]
>>> [3*x for x in wek]
[6, 12, 18]
>>> [3*x for x in wek if x > 3]
[12, 18]
>>> [3*x for x in wek if x < 2]
[]
>>> [{x: x**2} for x in wek]
[{2: 4}, {4: 16}, {6: 36}]
>>> [[x,x**2] for x in wek]
[[2, 4], [4, 16], [6, 36]]
>>> [x, x**2 for x in wek] # błąd - dla krotek wymaga się nawiasów okrągłych
  File "<stdin>", line 1
    [x, x**2 for x in wek]
               ^
SyntaxError: invalid syntax
>>> [(x, x**2) for x in wek]
[(2, 4), (4, 16), (6, 36)]
>>> wek1 = [2, 4, 6]
>>> wek2 = [4, 3, -9]
>>> [x*y for x in wek1 for y in wek2]
[8, 6, -18, 16, 12, -36, 24, 18, -54]
>>> [x+y for x in wek1 for y in wek2]
[6, 5, -7, 8, 7, -5, 10, 9, -3]
```

(**SyntaxError**: *zła składnia*)

5.2 Instrukcja `del`

Jednym ze sposobów usunięcia elementu z listy za pomocą podania jego indeksu zamiast jego wartości jest instrukcja `del`. Można ją również użyć do usunięcia wielu elementów poprzez usunięcie wycinka (robiliśmy to wcześniej poprzez przypisanie pustej listy do wycinka). Na przykład:

```
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.6, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.6, 1234.5]
```

`del` może zostać użyte do usuwania zmiennych:

```
>>> del a
```

Próba użycia zmiennej `a` po takiej operacji, powoduje zgłoszenie błędu (chyba, że będzie to instrukcja przypisania wartości do tej zmiennej). Inne zastosowania `del` zostaną przedstawione później.

5.3 Listy niemutowalne i sekwencje

Przekonaliśmy się już, że listy i napisy mają wiele wspólnych właściwości, np. indeksacja o operacje wycinania. Są to dwa przykłady typów *sekwencyjnych*. Ze względu na to, że Python jest językiem rozszerzalnym[2], można dodawać inne typy danych. Istnieje jeszcze jeden typ danych sekwencyjnych: *krotka*[3].

Taka lista składa się z pewnej liczby elementów oddzielonych przecinkami, np.:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Krotki można zagnieżdżać:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Jak widać, krotka pokazywana jest zawsze na wyjściu w okrągłych nawiasach, aby jeśli zagnieżdżone, były interpretowane poprawnie. Mogą być wprowadzane z lub bez nawiasów, aczkolwiek nawiasy są niekiedy potrzebne (np. jeżeli krotka jest częścią jakiegoś długiego wyrażenia).

Krotki mają dużo zastosowań, np. jako para współrzędnych (x,y), rekord pracowników z bazy danych itd. Nie jest możliwe przypisanie poszczególnym elementom takiej listy wartości, aczkolwiek można to zasymulować poprzez wycinanie i sklejanie.

Pewien problem z takimi listami pojawia się jeżeli mają zawierać zero lub jeden element: aby osiągnąć taki efekt trzeba użyć lekko pokręconej składni. Puste krotki tworzone są przez pustą parę nawiasów okrągłych; lista z jednym elementem poprzez wartość, po której następuje przecinek (nie wystarcza otoczyć pojedynczą wartość nawiasami okrągłymi). Brzydkie, ale efektywne. Na przykład:

[2] ang. evolving
[3] ang. tuple

```
>>> pusta = ()
>>> jednoelementowa = 'hello',    # <-- zauważcie przecinek na końcu!
>>> len(pusta)
0
>>> len(jednoelementowa)
1
>>> jednoelementowa
('hello',)
```

Instrukcja `t = 12345, 54321, 'hello!'` jest przykładem *pakowania krotki*: wartości `12345`, `54321` i `'hello!'` pakowane są do krotki. Operacja odwrotna jest również możliwa, np.:

```
>>> x, y, z = t
```

Takie coś nazywane jest, odpowiednio, *krotki*. Takie rozpakowywanie wymaga, aby lista zmiennych po lewej stronie miała tyle samo elementów, ile długość listy niemutowalnej. Zauważcie, że takie przypisanie jest kombinacją pakowania i rozpakowywania listy niemutowalnej!

Przez przypadek taką samą operację można powtórzyć dla zwykłej listy. Osiągamy to przez otoczenie listy zmiennych nawiasami kwadratowymi:

```
>>> a = ['wędzonka', 'jaja', 100, 1234]
>>> [a1, a2, a3, a4] = a
```

LISTY NIEMUTOWALNE–KROTKI *MOGĄ* ZAWIERAĆ OBIEKTY MUTOWALNE!

5.4 Słowniki

Innym użytecznym typem danych w Pythonie jest *słownik*. Słowniki spotykane są czasami w innych językach programowania jako „pamięć asocjacyjna” lub „tablice asocjacyjne”. W odróżnieniu od sekwencji, które są indeksowane liczbami, słowniki indeksowane są *kluczami*, które mogą być obiektami dowolnego, niemutowalnego typu, np. napisy i liczby zawsze mogą być kluczami. Listy niemutowalne również mogą zostać użyte jako klucze, jeżeli zawierają napisy, liczby lub listy niemutowalne. Nie można użyć zwykłych list jako kluczy, ponieważ można je modyfikować za pomocą metody `append()`.

Najlepiej wyobrazić sobie słownik jako nieuporządkowany zbiór par *klucz:wartość*, z założeniem, że klucze są unikalne (w jednym słowniku). Para nawiasów klamrowych tworzy pusty słownik: `{}`. Umieszczenie listy par klucz:wartość, oddzielonych przecinkami w tych nawiasach dodaje początkowe pary klucz:wartość do słownika. w ten sposób słowniki są wyświetlane na standardowym wyjściu.

Głównymi operacjami na słownikach są wkładanie wartości z jakimś kluczem i wyciąganie wartości opatrzonej podanym kluczem. Możliwe jest usuwanie pary klucz:wartość za pomocą `del`. Jeżeli próbuje się przechować klucz, który istnieje już w słowniku, poprzednia wartość związana z tym kluczem jest zapominana. Błąd powstaje w wyniku próby pozyskania wartości spod klucza, który nie istnieje w słowniku.

Metoda obiektu słownika `keys()` zwraca listę wszystkich kluczy używanych w słowniku, w porządku losowym (jeżeli chce się uzyskać posortowaną listę kluczy, zastosuj po prostu metodę `sort()` na tej liście). Do sprawdzenia obecności klucza w słowniku służy metoda `has_key()`.[4]

Oto mały przykład użycia słownika:

[4] „posiada klucz?” (*przyp. tłum.*)

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
1
```

5.5 Jeszcze trochę o warunkach

Warunki używane w instrukcjach `while` i `if` mogą zawierać inne operatory, niż poznane dotychczas operatory porównania.

Operatory porównania `in` oraz `not in` sprawdzają czy jakaś wartość pojawia się lub nie w sekwencji. Operatory `is` i `is not` porównują, czy dwa obiekty są w rzeczywistości tymi samymi obiektami: ma to znaczenie tylko dla przypadków obiektów mutowalnych, takich jak listy. Wszystkie operatory porównania mają taki sam priorytet, który jest niższy niż priorytet wszystkich operatorów numerycznych.

Porównania mogą być sklejane w jeden łańcuch, np. `a < b == c` sprawdza czy `a` jest mniejsze niż `b` i ponadto czy `b` jest równe `c`.

Porównania mogą być łączone operatorami logicznymi: `and` i `or`, a wynik porównania (lub każdego innego wyrażenia logicznego) może być zanegowany poprzez `not`. Wszystkie, wyżej wymienione operatory mają niższy priorytet od priorytetu operatorów porównania, aczkolwiek `not` ma wyższy priorytet od `or`. Tak więc `A and not B or C` są odpowiednikiem `(A and (not B) )or C`. Oczywiście, zawsze można użyć nawiasów, aby wyrazić pożądaną kolejność.

Operatory logiczne `and` i `or` są tzw. operatorami *skrótu*[5]. Ich argumenty ewaluowane są od lewej do prawej, a ewaluacja jest przerywane w momencie określenia ostatecznego wyniku. Np.: jeżeli `A` i `C` są prawdziwe ale `B` jest fałszywe, `A and B and C` nie ewaluuje wyrażenia C. W ogólności, wartość jaką zwraca operator skrótu jest ostatnim ewaluowanym argumentem, gdy używana jest jako wartość ogólna a nie logiczna[6]

Możliwe jest przypisanie wyniku porównania lub innego wyrażenia logicznego do zmiennej. Na przykład:

```
>>> napis1, napis2, napis3 = '', 'Trondheim', 'Hammer Dance'
>>> nie_pusty = napis1 or napis2 or napis3
>>> nie_pusty
'Trondheim'
```

Zauważcie, że w Pythonie, odmiennie niż w C, przypisanie nie może pojawić się w środku wyrażenia. Programiści C mogą się na to zżymać, ale w ten sposób można uniknąć powszechnego problemu spotykanego w C: pisząc `=` w wyrażeniu, kiedy zamierzało się napisać `==`.

5.6 Porównanie sekwencji i innych typów

Sekwencje mogą być porównane z innymi obiektami tego samego typu sekwencyjnego. Porównanie stosuje porządek *leksykograficzny*: na początku porównywane są pierwsze dwa elementy, a jeżeli się różnią to wynik

[5] ang. shortcut operators

[6] Czy ktoś może to wyjaśnić lepiej? (*przyp. tłum.*)

porównania jest już określony. Jeżeli są równe, do porównania brane są następne dwa elementy itd., aż do wyczerpania sekwencji. Jeżeli porównywane elementy są tego samego typu sekwencyjnego, porównanie leksykograficzne przeprowadzane jest na nich rekursywnie. Jeżeli wszystkie elementy okażą się równe, sekwencje uważane są za równe. Jeżeli jedna sekwencja jest początkowym podzbiorem drugiej, to krótsza sekwencja jest mniejsza od dłuższej. Leksykograficzny porządek napisów ustanowiony jest za pomocą porządku ASCII dla poszczególnych znaków. Oto parę przykładów relacji porównania pomiędzy sekwencjami tego samego typu:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Zauważcie, że można porównywać obiekty różnych typów. Wynik takiego porównania jest powtarzalny„ lecz arbitralny: typy są porządkowane wg swoich nazw. Tak więc lista jest zawsze mniejsza niż napis (`'list'<'string'`), a napis jest zawsze mniejszy niż lista niemutowalna (`'string'<'tuple'`), itd. Typy liczbowe mieszane porównywane są wg ich wartości numerycznej, tak więc 0 równe jest 0.0, etc.[7]

[7] Nie powinno się polegać na powyższej zasadzie porównywania różnych typów: może się ona zmienić w przyszłych wersjach języka.

ROZDZIAŁ

SZÓSTY

Moduły

W chwili gdy zakończy się pracę w interpreterze Pythona i ponownie rozpocznie, wszystkie definicje, które wprowadzono (funkcje i zmienne) zostają stracone. Dlatego też, jeśli chce się napisać ździebko dłuższy program, lepiej będzie gdy użyje się edytora tekstów do przygotowania poleceń dla interpretera i uruchomi go z przygotowanym plikiem na wejściu. Nazywa się to tworzeniem *skryptu*[1] W miarę„ jak twój program staje się dłuższy, zajdzie konieczność podzielenia go na kilka plików w celu łatwiejszej pielęgnacji [2] całości. Będziesz chciał również użyć funkcji, które właśnie napisałeś w paru innych programach bez potrzeby wklejania ich w każdy program z osobna.

Python wspomoże te działania poprzez pewien sprytny mechanizm umieszczania definicji w pliku i używania ich w skrypcie lub w interaktywnej postaci interpretera. Taki plik nazywany jest *modułem*: definicje z modułu mogą być *importowane* do innych modułów lub do *głównego* modułu (zestaw zmiennych, których używałeś w skrypcie wykonywanym na najwyższym poziomie i w trybie kalkulatora).

Moduł jest plikiem zawierającym definicje Pythona i jego instrukcje. Nazwa pliku jest nazwą modułu pozbawionego rozszerzenia „.py”. W module, nazwa modułu dostępna jest jako wartość zmiennej globalnej `__name__`. Na przykład, użyj swojego ulubionego edytora tekstów[3] i stwórz plik o nazwie „fibo.py”. Umieść go w bieżącym katalogu z następującą zawartością:

```
# Moduł liczb Fibonacciego

def fib(n):    # wypisz ciąg Fibonacciego aż do n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n): # zwróć ciąg Fibonacciego aż do n
    wybik = []
    a, b = 0, 1
    while b < n:
        wynik.append(b)
        a, b = b, a+b
    return wynik
```

Teraz, po uruchomieniu interpretera Pythona można zaimportować ten moduł za pomocą następującego polecenia:

```
>>> import fibo
```

W ten sposób nie zaimportuje się nazw funkcji zdefiniowanych w module `fibo` wprost do bieżącej tablicy symboli: to polecenie wprowadza tylko nazwę `fibo` do tej tablicy. Aby dostać się do owych funkcji, trzeba użyć nazwy modułu:

[1] To czy nazwa *skrypt* brzmi mniej poważnie od nazwy *program* jest sprawą gustu... (*przyp. tłum.*)
[2] ang. maintenance (*przyp. tłum.*)
[3] Nieśmiertelne «favorite text editor»... (*przyp. tłum.*)

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Jeżeli chce się używać funkcji często, można przypisać jej lokalną nazwę:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1 Ciąg dalszy o modułach

Moduł może zawierać instrukcje wykonywalne obok definicji funkcji. Instrukcje te mają na celu inicjalizację modułu. Wykonywane są tylko w chwili importowania modułu *po raz pierwszy*, gdzieś w programie.[4]

Każdy z modułów posiada swoją prywatną tablicę symboli, która używana jest przez wszystkie funkcje zdefiniowane w module jako globalna tablica symboli. W ten sposób, autor modułu może używać w nim zmiennych globalnych bez konieczności martwienia się o przypadkowy konflikt nazwy występującej w module z nazwą globalną zdefiniowaną w programie użytkownika. Z drugiej jednak strony, jeżeli wiesz co robisz, można wpłynąć na globalną zmienną modułu za pomocą takiej samej notacji, jakiej użyliśmy poprzednio, aby użyć wprost nazwy funkcji z modułu: `nazwa_modulu.nazwa_elementu`.

Moduły mogą importować inne moduły. Zazwyczaj, acz nie jest to wymagane, wszystkie instrukcje `import` umieszczane są na początku modułu (lub skryptu). Nazwy zaimportowanych modułów umieszczane są w globalnej tablicy symboli importujących modułów.

Istnieje wariant instrukcji `import`, która importuje nazwy z modułu wprost do tablicy symboli modułów importujących. Na przykład:

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ta konstrukcja nie wprowadza nazwy modułu, z którego importujemy, do lokalnej tablicy symboli (tak więc, w tym przykładzie `fibo` nie jest zdefiniowane).

Jest też pewien wariant, który importuje wszystkie nazwy z modułu:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ten mechanizm pozwala zaimportować wszystkie nazwy z wyjątkiem tych, które zaczynają się od znaku podkreślenia (_).

6.1.1 Ścieżka poszukiwań modułów

Gdy moduł o nazwie `pomyje` jest importowany, interpreter poszukuje pliku o nazwie „pomyje.py” w bieżącym katalogu, następnie w katalogach określonych przez zmienną systemową $PYTHONPATH. Zmienna ta ma

[4] Tak naprawdę, definicja funkcji jest także „instrukcją”, która jest „wykonywana”. Wykonanie to powoduje wprowadzenie nazwy funkcji do globalnej tablicy symboli modułu.

taką samą składnię co zmienna $PATH, tzn. jest listą katalogów. W przypadku, gdy $PYTHONPATH nie jest określona, lub gdy plik nie jest znaleziony w katalogach tam wymienionych, poszukiwanie kontynuowane jest na ścieżkach ustawionych w momencie instalacji: na UNIKSIE jest to zazwyczaj „.:/usr/local/lib/python”.

Na samym końcu, moduły poszukiwane są na liście katalogów umieszczonych w zmiennej pythonowej `sys.path`, która inicjalizowana jest nazwami katalogu zawierającego skrypt wejściowy (lub z bieżącego katalogu), zawartością zmiennej $PYTHONPATH i domyślnymi katalogami instalacyjnymi. W ten sposób zmyślne programy w Pythonie mogą modyfikować a nawet zastępować ścieżkę poszukiwań modułów. Zobacz później podrozdział na temat standardowych modułów.

6.1.2 Skompilowane pliki Pythona

Ważnym czynnikiem przyspieszenia rozruchu dla małych programów, które używają mnóstwo standardowych modułów jest obecność pliku „pomyje.pyc”. W pliku tym zawarta jest skompilowana już „bajt-kodowa”[5] wersja modułu `spam`. Czas modyfikacji wersji pliku „pomyje.py”, z którego powstał „pomyje.pyc”, jest zarejestrowany w tymże ostatnim. Plik „.pyc” jest ignorowany, jeżeli oba te czasy nie pasują do siebie[6]

W normalnych warunkach, nie trzeba zrobić nic szczególnego, aby utworzyć plik „pomyje.pyc”. Kiedykolwiek „pomyje.py” zostało pomyślnie skompilowane, interpreter usiłuje zapisać skompilowana wersję do „pomyje.pyc”. Nie ma błędu, jeśli zapis się nie powiedzie. Jeżeli z jakiegokolwiek powodu plik ten nie został zapisany, to „pomyje.pyc” zostanie rozpoznane jako niepoprawny i zignorowany. Zawartość „pomyje.pyc” jest niezależna od platformy, tak więc katalog modułów może być dzielony pomiędzy maszynami o różnej architekturze.[7]

Parę wskazówek dla ekspertów:

- Gdy interpreter Pythona wywołany został z flagą **-O**, wygenerowany zostanie zoptymalizowany kod i umieszczony w plikach „.pyo”. Obecnie, optymalizator nie robi żadnych rewelacyjnych rzeczy: usuwa tylko instrukcje `assert` i instrukcje `SET_LINENO`. Gdy używasz **-O**, *cały* kod pośredni jest optymalizowany, pliki „.pyc” są ignorowane, a pliki „.py” kompilowane do zoptymalizowanego kodu pośredniego.
- Wywołanie interpretera z dwoma flagami **-O** (**-OO**) spowoduje optymalizację kodu, która w pewnych przypadkach objawi się w wadliwym działaniu programu. Obecnie tylko napisy `__doc__` usuwane są z kodu pośredniego, co objawia się mniejszymi plikami „.pyo”. Ponieważ działanie niektórych programów może zależeć od obecności tych zmiennych, powinno się używać tej opcji tylko wtedy gdy jest się pewnym, co się robi.
- Program nie działa szybciej, gdy czytany jest z pliku „.pyc” lub „.pyo”, w porównaniu gdy czytany jest z pliku „.py”: jedyne co jest szybsze, to prędkość ładowania plików.
- W przypadku gdy nazwa skryptu podana jest w linii poleceń, kod pośredni dla niego nigdy nie zostanie zapisany w pliku „.pyo”. Tak więc, czas rozruchu skryptu może być zredukowany poprzez przesunięcie większości kodu do modułów, pozostawiając mały skrypt rozruchowy importujący te moduły.
- Możliwe jest posiadanie pliku „pomyje.pyc” (lub „pomyje.pyo” gdy używasz **-O**) bez modułu „pomyje.py”. Ma to zastosowanie w przypadku dystrybucji bibliotek Pythona w formie, która ma sprawić trudności w procesie «reverse engineering».
- Moduł `compileall` może zostać użyty do stworzenia plików „.pyc” (lub „.pyo” gdy użyto **-O**) dla wszystkich modułów z podanego katalogu.

6.2 Moduły standardowe

Python dostarczany jest z biblioteką standardowych modułów, które opisane są w osobnym dokumencie: *Opis biblioteki Pythona* (inaczej «Opis biblioteki»). Niektóre moduły wbudowane są w interpreter: są one źródłem tych operacji, które nie są częścią jądra[8] języka, lecz pomimo tego zostały wbudowane albo z powodu wydajności, lub

[5] ang. byte-coded
[6] Innymi słowy: używany jest „.pyc” jeśli data modyfikacji „.py” jest wcześniejsza od daty modyfikacji „.pyc”(*przyp. tłum.*)
[7] Zazwyczaj wersje interpretera muszą być tes same (*przyp. tłum.*)
[8] ang. core (*przyp. tłum.*)

aby wprowadzić dostęp do podstawowych operacji systemu operacyjnego, np. funkcje systemowe. To co zostało wbudowane w interpreter, jest kwestią opcji instalacji, tzn. moduł `ameba` dostarczany jest tylko na te systemy, które w pewien sposób wspomagają podstawowe operacje Ameby. Jeden z modułów wbudowanych zasługuje na szczególną uwagę: `sys`, który jest wbudowany w niemal każdy interpreter Pythona. Zmienne `sys.ps1` i `sys.ps2` definiują napisy używane jako pierwszy i drugi znak zachęty:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Zmienne te dostępne są tylko w trybie interakcyjnym interpretera.

Zmienna `sys.path` jest listą napisów, które decydują o ścieżce poszukiwań modułów przez interpreter. Domyślnie inicjowane są zawartością zmiennej systemowej $PYTHONPATH lub wbudowanymi domyślnymi katalogami poszukiwań, jeśli $PYTHONPATH nie istnieje. Można modyfikować `sys.path` poprzez standardowe operacje na listach:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 Funkcja `dir()`

Funkcja wbudowana `dir()` służy do znajdywania wszystkich nazw, które są zdefiniowane w module. Zwraca ona posortowaną listę napisów:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__name__', 'argv', 'builtin_module_names', 'copyright', 'exit',
'maxint', 'modules', 'path', 'ps1', 'ps2', 'setprofile', 'settrace',
'stderr', 'stdin', 'stdout', 'version']
```

Wywołana bez argumentów `dir()` zwróci listę nazw, które właśnie zdefiniowałeś:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo, sys
>>> fib = fibo.fib
>>> dir()
['__name__', 'a', 'fib', 'fibo', 'sys']
```

Zauważ, że zwrócona została lista wszystkich typów nazw: zmiennych, modułów, funkcji, itd.

`dir()` nie poda listy nazw funkcji i zmiennych wbudowanych. Jeśli chce się uzyskać taką listę, to posłuż się standardowym modułem `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['AccessError', 'AttributeError', 'ConflictError', 'EOFError', 'IOError',
'ImportError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'MemoryError', 'NameError', 'None', 'OverflowError', 'RuntimeError',
'SyntaxError', 'SystemError', 'SystemExit', 'TypeError', 'ValueError',
'ZeroDivisionError', '__name__', 'abs', 'apply', 'chr', 'cmp', 'coerce',
'compile', 'dir', 'divmod', 'eval', 'execfile', 'filter', 'float',
'getattr', 'hasattr', 'hash', 'hex', 'id', 'input', 'int', 'len', 'long',
'map', 'max', 'min', 'oct', 'open', 'ord', 'pow', 'range', 'raw_input',
'reduce', 'reload', 'repr', 'round', 'setattr', 'str', 'type', 'xrange']
```

6.4 Pakiety

Pakiety są sposobem na „ustrukturalnienie" przestrzeni nazw modułów Pythona poprzez używanie „kropkowanych nazw modułów". Na przykład, nazwa modułu `A.B` oznacza moduł składowy „B" pakietu „A". Tak samo jak używanie modułów zaoszczędza autorom różnych modułów martwienia się o konflikt nazw globalnych, tak samo używanie kropkowanych nazw modułów zaoszczędza autorom wielomodułowych pakietów jak NumPy lub Python Imaging Library martwienia się nazwy swoich modułów.

Załóżmy, że chce się zaprojektować zbiór modułów („pakiet") służący do jednolitej obsługi plików i danych dźwiękowych. Istnieje mnóstwo formatów zapisu dźwięku (zwykle rozpoznawane po rozszerzeniu plików, np. „.wav", „.aiff", „.au"), co spowoduje, że może będziesz musiał stworzyć i utrzymywać zwiększającą się kolekcję modułów konwersji pomiędzy różnymi formatami plików. Jest również dużo różnych operacji, które można przeprowadzić na danych dźwiękowych (np. miksowanie, dodawanie pogłosu, zastosowanie funkcji equalizera, tworzenie sztucznego efektu stereo), tak więc, kończąc już, będziesz pisał niekończący się strumień modułów do wykonywania tych operacji. Poniżej przedstawiono jedną z możliwych struktur takiego pakietu (wyrażona ona jest w postaci hierarchicznego drzewa, na podobieństwo systemu plików):

```
Sound/                          Pakiet szczytowy
      __init__.py               Inicjalizacja pakietu do obsługi dźwięku
      Formats/                  Moduły składowe do konwersji plików
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      Effects/                  Pakiet składowy z efektami dźwiękowymi
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      Filters/                  Pakiet składowy z operacjami filtrowania
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Plik „__init__.py" są wymagane, aby zmusić Pythona do traktowania katalogów jako zawierające pakiety. Jest to konieczne, aby uchronić katalogi z powszechnie spotykanymi nazwami, takie jak „`string`", od niezamierzonego ukrycia poprawnych modułów, które pojawią się później na ścieżce poszukiwań modułów. W najprostszym przypadku, plik „__init__.py" może być pustym plikiem, ale może też zawierać pewien kod do wykonania, lub ustawiać

wartość zmiennej `__all__`, opisaną później.

Użytkownicy tego pakietu mogą importować poszczególne moduły, np.:

```
import Sound.Effects.echo
```

To spowoduje załadowanie modułu składowego `Sound.Effects.echo`. Później, trzeba używać go za pomocą pełnej nazwy, tzn.

```
Sound.Effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Alternatywnym sposobem importowania modułu składowego jest:

```
from Sound.Effects import echo
```

W ten sposób również załadujemy moduł składowy `echo` ale sprawimy, że dostępny jest z nazwą niezawierającą nazwy pakietu rodzicielskiego:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Inną wariacją na ten temat jest zaimportowanie wprost żądanej funkcji:

```
from Sound.Effects.echo import echofilter
```

W ten sposób ponownie ładujemy moduł składowy `echo`, lecz jego funkcja `echofilter()` jest dostępna wprost:

```
echofilter(input, output, delay=0.7, atten=4)
```

Zauważ, że kiedy używa się `from` *pakiet* `import` *element*, element może być modułem składowym (lub pakietem składowym) lub inną nazwą zdefiniowaną dla tego pakietu„ jak funkcja, klasa czy zmienna. Instrukcja `import` najpierw sprawdza, czy element jest zdefiniowany w pakiecie; jeśli nie, zakłada się, że jest to moduł i poczynione są starania o jego załadowanie. Jeśli nie zostanie on znaleziony, zgłaszany jest wyjątek `ImportError`.

W przeciwieństwie do tego, jeśli używa się składni `import` *element.element_skladowy.kolejny_element_skladowy*, każdy element z wyjątkiem ostatniego musi być pakietem: ostatni element może być modułem lub pakietem ale nie klasą, funkcją lub zmienną zdefiniowaną w poprzednim elemencie.

6.4.1 Importowanie * z pakietu

A teraz, co się stanie, kiedy użytkownicy napiszą `from Sound.Effects import *`? Ktoś mógłby pomyśleć naiwnie, że ta instrukcja w jakiś sposób wychodzi do systemu plików, znajduje wszystkie pliki z modułami składowymi, które są obecne w pakiecie-katalogu i wszystkie je importuje. Niestety, taka operacja nie działa zbyt dobrze na macintosh'ach i w Windows, gdzie system plików nie zawsze wyposażony jest w dokładną informację na temat wielkości liter w nazwach![9] Na tych platformach nie ma sposobu dowiedzenia się, czy plik „ECHO.PY" powien być zaimportowany jako moduł `echo`, `Echo` czy `ECHO` (np. Windows 95 ma denerwujący zwyczaj pokazywania nazw plików z pierwszą dużą literą). Ograniczenie DOS-a (8+3 znaki na nazwę pliku) dodaje jeszcze jeden interesujący problem do tej listy jeżeli chodzi o długie nazwy modułów.

Jedynym wyjściem dla autora takiego pakietu jest dostarczenie wprost indeksu jego części składowych. Instrukcja importu posługuje się następującą konwencją: jeżeli kod pakietu „__init__.py" zdefiniuje listę o nazwie `__all__`, to przyjmuje się, że zawiera ona nazwy wszystkich modułów importowanych za pomocą instrukcji `from` *pakiet* `import *`. Zadaniem autora pakietu jest utrzymywanie stanu tej listy na bieżąco, w chwili gdy wypuszczane są

[9] Ten wykrzyknik powinien się tu pojawić! (*przyp. tłum.*)

nowe wersje pakietu. Autorzy pakietów mogą również nie dostarczać tej listy, jeśli nie widzą sensu z importowania * z ich pakietu. Na przykład plik „Sound/Effects/__init__.py” mógłby zawierać następujący kod:

```
__all__ = ["echo", "surround", "reverse"]
```

Znaczyłoby to, że `from Sound.Effects import *` zaimportuje trzy moduły składowe z pakietu `Sound`.

Jeśli `__all__` nie jest zdefinowana, instrukcja `from Sound.Effects import *` *nie* zaimportuje wszystkich modułów składowych z pakietu `Sound.Effects` do bieżącej przestrzeni nazw: upewni się tylko, czy pakiet `Sound.Effects` został zaimportowany (z możliwym wykonaniem kodu z pliku „__init__.py”) a potem zaimportuje wszystkie nazwy zdefiniowane w tym pakiecie. Wchodzą w nie nazwy zdefiniowane (i moduły składowe załadowane wprost) w pliku „__init__.py” oraz moduły składowe pakietu załadowane wprost przez poprzednie instrukcje importu, tzn.

```
import Sound.Effects.echo
import Sound.Effects.surround
from Sound.Effects import *
```

W tym przykładzie moduły `echo` i `sorround` importowane są do bieżącej przestrzeni nazw ponieważ są zdefiniowane w pakiecie `Sound.Effects`, gdy wykonywany jest `from...import` (działa to również, gdy zdefiniowano `__all__`).

Trzeba tu zauważyć, że stosowanie importowania * z modułu lub pakietu idzie powoli w odstawkę, ponieważ często powoduje nieczytelny kod. Jest oczywiście w porządku, gdy stosuje się tę formułę, aby zaoszczędzić trochę na klepaniu klawiszami w trybie interaktywnym i gdy pewne moduły są zaprojektowane tak, że eksportują tylko nazwy odpowiadające pewnym schematom.

Trzeba pamiętać, że nie ma nic złego w używaniu formuły `from Pakiet import pewien_modul`! Właściwie, ta notacja jest zalecana, chyba że moduł importujący używa modułów składowych o takiej samej nazwie z paru różnych pakietów.

6.4.2 Odniesienia pomiędzy pakietami

Często zachodzi wymóg odniesienia się jednego modułu do drugiego. Na przykład, moduł `sorround` może używać modułu `echo`. W rzeczywistości, jest to tak często spotykane, że instrukcja `import` poszukuje najpierw w zawierającym ją pakiecie, zanim przejdzie na standardową ścieżkę poszukiwań. Tak więc, moduł `sorround` może użyć zwykłego `import echo` lub `from echo import echofilter`. Jeśli importowany moduł nie zostanie znaleziony w bieżącym pakiecie (w którym bieżący moduł jest modułem składowym), instrukcja `import` szuka w szczytowym module o podanej nazwie.

Kiedy pakiety posiadają składowe pakiety (tak jak `Sound` w naszym przykładzie), nie istnieje skrót, aby odnieść się do modułu składowego pakietów składowych — trzeba użyć pełnej nazwy pakietu składowego. Na przykład, jeśli moduł `Sound.Filters.vocoder` musi użyć modułu `echo` z pakietu `Sound.Effects`, może zastosować `from Sound.Effects import echo`.

ROZDZIAŁ

SIÓDMY

Wejście i wyjście

Istnieje parę sposobów na prezentacje wyników działania programu: dane mogą zostać wydrukowane w czytelny sposób, lub zapisane do pliku w celu ponownego użycia. Ten rozdział omawia parę z tych możliwości.

7.1 Ładniejsze formatowanie wyjścia

Jak dotąd spotkaliśmy się z dwoma sposobami wypisywania wartości: *instrukcje wyrażenia* i instrukcja `print` (trzecim sposobem jest metoda `write()` należąca do obiektu pliku. Standardowym obiektem standardowego pliku wyjściowego jest `sys.stdout`. Poszukaj w «Opisie biblioteki» więcej informacji na ten temat).

Często chce się mieć większy wpływ na format drukowania wyniku, niż proste wiersze wartości oddzielonych spacjami. Istnieją dwa sposoby formatowania wyniku. Pierwszym jest przejęcie na własną odpowiedzialność wszelkich czynności związanych z napisami: używa się wycinania napisów i sklejania w ten sposób, że tworzy się taki format wydruku, jaki przyjdzie do głowy. Standardowy moduł `string` zawiera parę użytecznych operacji na napisach, jak rozszerzanie napisów do żądanej szerokości kolumny itd. Opiszemy to po krótce. Drugim sposobem jest użycie operatora % z napisem jako lewy argument. Operator % interpretuje lewy argument w podobny sposób jak funkcja C `sprintf()`:, jak format stylu, który zostanie zastosowany do argumentów po prawej stronie tego operatora, a wynik zwracany jest w postaci napisu.

Oczywiście, pozostaje jedno pytanie:, jak przekształca się wartości w napisy? Na szczęście, Python zna sposób, aby przekształcić dowolną wartość w napis: funkcja `repr()` lub umieszczenie wartośći pomiędzy odwrotnymi apostrofami (“). Oto parę przykładów:

```
>>> x = 10 * 3.14
>>> y = 200*200
>>> s = 'Wartością x jest ' + 'x' + ', a y jest ' + 'y' + '...'
>>> print s
Wartością x jest 31.4, a y jest 40000...
>>> # Odwrotne apostrofy działają także na inne wartości niż liczby:
... p = [x, y]
>>> ps = repr(p)
>>> ps
'[31.4, 40000]'
>>> # Przekształcenie napisu powoduje dodanie apostrofów i ukośników:
... hello = 'hello, world\n'
>>> hellos = 'hello'
>>> print hellos
'hello, world\012'
>>> # Argumentem odwrotnych apostrofów może być lista niemutowalna:
... 'x, y, ('wędzonka', 'jaja')'
"(31.4, 40000, ('wędzonka', 'jaja'))"
```

Oto dwa sposoby na wypisanie kwadratów i sześcianów liczb:

```
>>> import string
>>> for x in range(1, 11):
...     print string.rjust(‘x‘, 2), string.rjust(‘x*x‘, 3),
...     # Zauważ przyklejony na końcu przecinek w poprzednim wierszu
...     print string.rjust(‘x*x*x‘, 4)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
>>> for x in range(1,11):
...     print ’%2d %3d %4d’ % (x, x*x, x*x*x)
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

(Spacje pomiędzy kolumnami zostały dodane przez instrukcje `print`: zawsze dodaje spację pomiędzy argumentami.)

Ten przykład demonstruje funkcję `string.rjust()`, która justuje do prawej napis w polu o podanej szerokości poprzez dodanie spacji z lewej strony. Podobna do niej są funkcje `string.ljust()` i `string.center()`. Funkcje te niczego nie wypisują, po prostu zwracają nowy napis. Jeżeli napis wejściowy jest zbyt duży, nie skracają go, lecz zwracają nietknięty. Pomiesza to trochę w wyglądzie kolumn ale zwykle jest to lepsza alternatywa od pokazywania fałszywej wartości (jeśli naprawdę chce się skrócić napis można zawsze dodać operację wycinania, np. „`string.ljust(x, n)[0:n]`”).

Istnieje też inna funkcja, mianowicie `string.zfill()`, która uzupełnia z lewej strony napis numeryczny zerami. Rozumie ona znaki plus i minus:

```
>>> import string
>>> string.zfill(’12’, 5)
’00012’
>>> string.zfill(’-3.14’, 7)
’-003.14’
>>> string.zfill(’3.14159265359’, 5)
’3.14159265359’
```

Użycie operatora % wygląda następująco:

```
>>> import math
>>> print ’Wartością PI jest w przybliżeniu %5.3f.’ % math.pi
Wartością PI jest w przybliżeniu 3.142.
```

Jeżeli podano więcej niż jeden format w napisie, jako prawy operand podaje się krotkę wartości, np.:

```
>>> tabela = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for nazwa, telefon in tabela.items():
...     print '%-10s ==> %10d' % (nazwa, telefon)
...
Jack       ==>       4098
Dcab       ==>       7678
Sjoerd     ==>       4127
```

Większość formatów działa dokładnie tak samo jak w C i wymaga podania właściwego typu, aczkolwiek jeżeli nie zastosuje się do tego, to w wyniku dostaje się wyjątek, a nie zrzut obrazu pamięci[1] i przerwanie działania programu. Format `%s` jest bardziej tolerancyjny: jeżeli odpowiadający mu argument nie jest napisem, przekształcany jest w napis za pomocą wbudowanej funkcji `str()`. Można użyć `*` do podania szerokości lub precyzji w oddzielnym (liczba całkowita) argumencie. Formaty C `%n` i `%p` nie są obsługiwane.

Jeśli posiadasz już naprawdę długi napis formatujący, którego nie chce się dzielić, miło by było odnosić się wprost do zmiennych będących formatowanymi, zamiast do ich pozycji na liście argumentów. Można to osiągnąć poprzez użycie rozszerzenia w stosunku do formatów C w formie `%(nazwa)format`, tzn.:

```
>>> tabela = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: %(Jack)d; Sjoerd: %(Sjoerd)d; Dcab: %(Dcab)d' % tabela
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Jest to szczególnie użyteczne w połączeniu z nową funkcją wbudowaną `vars()`, która zwraca słownik zawierający wszystkie zmienne lokalne.

7.2 Czytanie z i pisanie do plików

`open()` zwraca obiekt pliku i powszechnie używana jest z dwoma argumentami: „`open(`*nazw_pliku*`, `*tryb*`)`”.

```
>>> f=open('/tmp/workfile', 'w')
>>> print f
<open file '/tmp/workfile', mode 'w' at 80a0960>
```

Pierwszym argumentem jest napis zawierający nazwę pliku. Drugim jest następny napis zawierający parę znaków opisujących sposób użycia pliku. *tryb* zawiera `'r'`, kiedy plik będzie tylko czytany[2], `'w'`, gdy będzie odbywało się wyłącznie pisanie do pliku (istniejący plik o tej nazwie zostanie zmazany), a `'a'` otwiera plik, do którego można dodawać dane: dowolna dana zapisana do pliku będzie dodana na jego koniec. `'r+'` otwiera plik zarówno do czytania jak i do pisania. Argument *tryb* jest opcjonalny: w przypadku jego braku plik zostanie otwarty w trybie `'r'`.

W systemach Windows i na macintoshach dodanie `'b'` do *tryb* powoduje otwarcie pliku w trybie binarnym, tak więc możemy podać tryby `'rb'`, `'wb'` i `'r+b'`. Windows rozróżnia pliki tekstowe i binarne: znaki końca linii w plikach tekstowych są automatycznie zmieniane, gdy dane są czytane lub pisane. Taka modyfikacja „w białych rękawiczkach” pliku wydaje się być wygodna dla plików tekstowych zapisanych w kodzie ASCII, ale zepsuje zawartość plików z danymi binarnymi jak np. pliki „.EXE”. Trzeba być ostrożnym stosując tryb binarny przy czytaniu lub zapisie (trzeba jeszcze dodać, że dokładne znaczenie trybu tekstowego na macintoshu, zależy od zastosowanej biblioteki C).

7.2.1 Metody obiektów pliku

W reszcie przykładów tego podrozdziału zakłada się, że stworzono już obiekt pliku o nazwie `f`.

[1] ang. core dump (*przyp. tłum.*)
[2] ang. read-only (*przyp. tłum.*)

Użyj `f.read(`*ile*`)`, aby przeczytać jakąś liczbę danych i dostać je w postaci napisu. *ile* jest opcjonalnym argumentem liczbowym. Kiedy *ile* jest pominięte lub ujemne, całą zawartość pliku zostanie przeczytana i zwrócona: to twój problem, jeśli wielkość czytanego pliku dwukrotnie przewyższa wielkość pamięci twojego komputera. W przypadku przeciwnym, co najwyżej *ile* bajtów zostanie przeczytanych i zwróconych. Jeśli osiągnięto koniec pliku, `f.read()` zwróci pusty napis ().

```
>>> f.read()
'To jest cały plik.\012'
>>> f.read()
''
```

`f.readline()` czyta z pliku pojedynczy wiersz. Znak nowej linii (`\n`) umiejscowiony jest na lewym końcu napisu i zostaje pominięty tylko w wypadku ostatniego wiersza pod warunkiem, że plik nie kończy się znakiem nowego wiersza. Powoduje to powstanie niejednoznaczności w zwracanym napisie: jeżeli `f.readline()` zwróci pusty napis osiągnęliśmy koniec pliku, podczas gdy pusta linia reprezentowana przez `'\n'`, tzn. napis zawierający tylko znak nowej linii.

```
>>> f.readline()
'To jest pierwsza linia pliku.\012'
>>> f.readline()
'Druga linia pliku.\012'
>>> f.readline()
''
```

`f.readlines()` stosuje `f.readline()` w sposób ciągły i zwraca listę napisów reprezentujących wszystkie wiersze z pliku.

```
>>> f.readlines()
['To jest pierwsza linia pliku.\012', 'Druga linia pliku.\012']
```

`f.write(`*napis*`)` zapisuje zawartość *napis*u do pliku zawracając `None`.

```
>>> f.write('To jest test\n')
```

`f.tell()` zwraca liczbę całkowitą oznaczającą bieżącą pozycję w pliku mierzoną w bajtach, licząc od początku pliku. Aby zmienić tę pozycję użyj „`f.seek(`*przesuniecie*`, `*od_czego*`)`”. Nowa pozycja obliczana jest poprzez dodanie *przesuniecie* do punktu odniesienia, a ten z kolei wyznaczony jest przez wartość argumentu *od_czego*. Wartość *od_czego* równa 0 oznacza początek pliku, 1 oznacza bieżącą pozycję, a 2 to koniec pliku. *od_czego* może zostać pominięte i domyślnie przyjmowane jest jako 0, używając jako punktu odniesienia początek pliku.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)     # Idź do 5 bajtu w pliku
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Idź do 3 bajtu od końca pliku
>>> f.read(1)
'd'
```

Jeżeli kończy się pracę z plikiem, trzeba wywołać `f.close()`[3], aby go zamknąć i zwolnić wszystkie zasoby systemowe związane z otwarciem i obsługą tego pliku. Po wywołaniu `f.close()` wszelkie próby użycia obiektu pliku `f` automatycznie spalą na panewce.

[3]Metoda ta zostanie wywołana przez interpreter przy ostatecznym niszczeniu obiektu (*przyp. tłum.*)

```
>>> f.close()
>>> f.read()
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Obiekty pliku posiadają dodatkowe metody, takie jak `isatty()` i `truncate()`, które są żadziej używane. Aby uzyskać kompletny opis obiektów pliku, sięgnij po «Opis biblioteki».

7.2.2 Moduł `pickle`

Napisy mogą być w łatwy sposób zapisywane i czytane z pliku. Liczby wymagają trochę więcej zachodu, bowiem metoda `read()` zwraca tylko napis, który musi być poddany działaniu takiej funkcji„ jak `string.atoi()`, która pobiera napis w rodzaju `'123'` i zwraca wartość liczbową 123. W przypadku jednak, gdy chce się przechowywać w pliku bardziej złożone typy danych jak listy, słowniki lub instancje klas, sprawy się nieco komplikują.

Python dostarcza standardowy moduł `pickle`, który zaoszczędza użytkownikom pisania i śledzenia kodu służącego do zachowywania skomplikowanych typów danych. Ten zadziwiający[4] moduł potrafi wziąć na wejściu prawie każdy obiekt Pythona (nawet pewne formy kodu!) i przekształcić go w napis. Proces ten zwie się *marynowaniem*[5]. Rekonstruowanie obiektu z formy napisowej zwie się *odmarynowaniem*[6]. Pomiędzy marynowaniem a odmarynowaniem, napis reprezentujący obiekt może zostać zapisany w pliku lub w innej danej, lub przesłany połączeniem sieciowym do jakiegoś oddalonego komputera.[7]

Jeżeli istnieje obiekt `x` i obiekt pliku `f`, który został otwarty do pisania, to najprostszy sposób zamarynowania obiektu zajmuje jeden wiersz kodu:

```
pickle.dump(x, f)
```

Zakładając, że `f` jest obiektem pliku, który został otwarty do czytania, odmarynowanie przebiega następująco:

```
x = pickle.load(f)
```

(Istnieją warianty tego mechanizmu użyteczne w przypadku marynowania wielu obiektów, lub gdy nie chce się zapisać danych marynaty w pliku — skonsultuj się z pełną dokumentacją dla modułu `pickle`, którą znajdziesz w «Opisie biblioteki»).

`pickle` jest standardowym sposobem na uczynienie obiektów Pythona trwałymi i ponownie użytymi przez inne programy lub przyszłe wywołania tego samego programu: technicznym określeniem tego mechanizmu jest *trwałość* obiektu. Z powodu powszechnego użycia modułu `pickle`, wielu autorów piszących rozszerzenia do Pythona, dba o to, aby nowe typy danych, takie jak macierze, mogły być poprawnie zamarynowane i odmarynowane.

[4] Skromność ze wszech miar godna podziwu... (*przyp. tłum.*)
[5] ang. pickling (*przyp. tłum.*)
[6] ang. unpickling (*przyp. tłum.*)
[7] Wybrałem te dwa określenia ze względu na niesamowity efekt komiczny (*przyp. tłum.*)

ROZDZIAŁ

ÓSMY

Błędy i wyjątki

Aż do teraz, komunikaty o błędach nie były często wspominane, ale jeśli próbowałeś uruchamiać wszystkie przykłady prawdopodobnie zobaczyłeś parę. Istnieją (przy najmniej) dwa rozróżnialne rodzaje błędów: *błędy składni* i *wyjątki*.

8.1 Błędy składni

Błędy składniowe, znane również jako błędy parsingu, są być może najbardziej powszechnym rodzajem zażaleń, które otrzymuje się podczas nauki Pythona:

```
>>> while 1 print 'Cześć świecie'
  File "<stdin>", line 1
    while 1 print 'Cześć świecie'
                  ^
SyntaxError: invalid syntax
```

Parser powtarza na wyjściu obraźliwy dla niego wiersz i wyświetla małą „strzałkę”, wskazującą na najwcześniejszy punkt w wierszu gdzie znaleziono błąd. Błąd spowodowany został (lub przynajmniej wykryty w) przez składnik *poprzedzający* strzałkę: w tym przykładzie, błąd wykryty został na słowie kluczowym `print`, z powodu braku znaku dwukropka („`:`”) przed tą instrukcją. Nazwa pliku i numer linii są również wyświetlane, aby wiedzieć, gdzie szukać błędu, w przypadku gdy na wejściu znajdował się skrypt.

8.2 Wyjątki

Nawet gdy wyrażenie jest składniowo poprawne, może spowodować błąd podczas próby wykonania go. Błędy wykryte podczas wykonania nazywane są *wyjątkami* i niekoniecznie muszą zakończyć program: za chwilę nauczysz się jak radzić sobie z nimi w programach Pythona. Większość wyjątków nie jest obsługiwana przez programy i objawiają się w komunikatach o błędzie, jak poniżej:

```
>>> 10 * (1/0)
Traceback (innermost last):
  File "<stdin>", line 1
ZeroDivisionError: integer division or modulo
>>> 4 + szynka*3
Traceback (innermost last):
  File "<stdin>", line 1
NameError: szynka
>>> '2' + 2
Traceback (innermost last):
  File "<stdin>", line 1
TypeError: illegal argument type for built-in operation
```

(**ZeroDivisionError***: dzielenie całkowite lub operacja modulo*)

(**TypeError***: nieprawidłowy typ argumentu operacji wbudowanej*)

Ostatnia linia komunikatu o błędzie pokazuje co się stało. Wyjątki są różnego typu, i ten typ wyświetlany jest jako część komunikatu. W powyższym przykładzie tymi typami są `ZeroDivisionError`, `NameError` i `TypeError`. Napis wyświetlany jako typ wyjątku jest nazwą wbudowanego w interpreter wyjątku, który się właśnie pojawił. Zachodzi to dla wszystkich wyjątków wbudowanych„ lecz niekoniecznie jest to prawdą dla wyjątków zdefiniowanych przez użytkownika (jakkolwiek, jest to użyteczna konwencja). Nazwy standardowych wyjątków są wbudowanymi identyfikatorami, a nie zastrzeżonymi słowami kluczowymi.

Reszta wiersza komunikatu w szczegółach opisuje to, co się stało. Interpretacja i znaczenie tego opisu zależy od typu wyjątku.

Pierwsza część komunikatu pokazuje kontekst, w jakim wyjątek się zdarzył, w postaci śladu ze stosu wywołań. Ogólnie mówiąc, kontekst zawiera ślady wywołań w postaci wierszy kodu, aczkolwiek nie pokaże wierszy podchodzących ze standardowego wejścia.

Opis biblioteki Pythona wyszczególnia wszystkie wbudowane wyjątki i przedstawia ich znaczenie.

8.3 Obsługa wyjątków

Możliwe jest napisanie programu, który obsługuje wybrane wyjątki. Spójrzmy na przykład poniżej, w którym użytkownik podaje dane na wejściu aż do momentu, kiedy zostanie wprowadzona poprawna liczba całkowita. Użytkownik może przerwać pracę (poprzez naciśnięcie `Control-C` lub dowolny sposobem, jaki jest możliwy poprzez klawiaturę w danym systemie operacyjnym). Przerwanie użytkownika sygnalizowane jest poprzez zgłoszenie wyjątku `KeyboardInterrupt`.

```
>>> while 1:
...     try:
...         x = int(raw_input("Proszę wprowadzić liczbę: "))
...         break
...     except ValueError:
...         print "Uch! to nie jest poprawna liczba! Spróbuj jeszcze raz..."
...
```

Oto jak działa instrukcja `try`:

- Na początku wykonywana jest *klauzula try* (czyli instrukcje pomiędzy `try` a `except`).
- Jeżeli nie pojawi się żaden wyjątek *klauzula except* jest pomijana. Wykonanie instrukcji `try` uważa się za zakończone.
- Jeżeli podczas wykonywania klauzuli `try` pojawi się wyjątek, reszta niewykonanych instrukcji jest pomijana. Następnie, w zależności od tego, czy jego typ pasuje do typów wyjątków wymienionych w części `except`, wykonywany jest kod następujący w tym bloku, a potem interpreter przechodzi do wykonywania instrukcji umieszczonych po całym bloku `try...except`.

- W przypadku pojawienia się wyjątku, który nie zostanie dopasowany do żadnego z wyjątków wymienionych w klauzuli `except`, zostaje on przekazany do do następnych, zewnętrznych instrukcji `try`. Jeżeli również tam nie zostanie znaleziony odpowiadający mu blok `except`, wyjątek ten nie zostanie wyłapany, stanie *nieobsłużonym wyjątkiem*, a wykonywanie programu zostanie wstrzymane wraz z pojawieniem się komunikatu podobnego do pokazanego powyżej.

Aby umożliwić obsługę wielu wyjątków, instrukcja `try` może posiadać więcej niż jedną klauzulę `except`. W takim przypadku, kod dla co najwyżej jednego wyjątku zostanie wykonany. Kody obsługi wyjątków wykonywane są tylko dla wyjątków, które zostały zgłoszone w odpowiadającej im części `try`, a nie w innych, sąsiednich częściach `except`. Klauzula `except` może zawierać nazwy wielu wyjątków, podanych w formie listy otoczonej nawiasami okrągłymi[1]:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

W ostatniej podanej klauzuli `except` można pominąć nazwę lub nazwy wyjątków w celu obsłużenia dowolnego wyjątku. Należy posługiwać się tą konstrukcją bardzo ostrożnie, ponieważ jest to sposób do łatwego zamaskowania prawdziwego wystąpienia błędu w programie! Można jej również użyć do wydrukowania komunikatu o błędzie i ponownie zgłosić wyłapany wyjątek (umożliwiając w ten sposób funkcji wywołującej wyłapanie zgłoszonego wyjątku):

```
import string, sys

try:
    f = open('mójplik.txt')
    s = f.readline()
    i = int(string.strip(s))
except IOError, (errno, strerror):
    print "Błąd I/O (%s): %s" % (errno, strerror)
except ValueError:
    print "Nie mogę przekształcić danej w liczbę całkowitą."
except:
    print "Nieobsługiwany błąd:", sys.exc_info()[0]
    raise
```

Instrukcja `try` ... `except` wyposażona jest w opcjonalną klauzulę *else*, która musi pojawić się za wszystkimi podanymi blokami `except`. Można po niej umieścić kod, który zostanie wykonany, jeżeli *nie* zostanie zgłoszony wyjątek:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'nie mogę otworzyć  pliku', arg
    else:
        print arg, 'ma', len(f.readlines()), 'linię/linii'
        f.close()
```

Wyjątek może pojawić się z przypisaną sobie wartością, która nazywana jest *argumentem* wyjątku. Obecność tej wartości i jej typ zależy od rodzaju wyjątku. Jeżeli chce się poznać tę wartość, należy podać nazwę zmiennej (lub listę zamiennych) za nazwą typu wyjątku w klauzuli `except`:

[1] Czyli w formie krotki (*przyp. tłum.*).

```
>>> try:
...     szynka()
... except NameError, x:
...     print 'nazwa', x, 'nie została zdefiniowana'
...
nazwa szynka nie została zdefiniowana
```

Argument ten (jeśli istnieje) pokazywany jest w ostatniej części («detail») komunikatu o niewyłapanym wyjątku.

Kody obsługujące wyjątek uruchamiane są nie tylko po zgłoszeniu wyjątku w ich klauzuli `try`, ale także w przypadku, gdy pojawią się one w instrukcjach `try` umieszczonych w wywoływanych funkcjach (nawet pośrednio):

```
>>> def to_sie_nie_uda():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError, detale:
...     print 'Obsługa błędu wykonania programu:', detale
...
Obsługa błędu wykonania programu: integer division or modulo
```

8.4 Zgłaszanie wyjątków

Instrukcja `raise` służy do wymuszania pojawienia się podanego wyjątku. Na przykład:

```
>>> raise NameError, 'HejTam'
Traceback (innermost last):
  File "<stdin>", line 1
NameError: HejTam
```

Pierwszy argument instrukcji `raise` służy do podania nazwy wyjątku. Opcjonalny drugi argument jest jego wartością (argumentem wyjątku).

8.5 Wyjątki definiowane przez użytkownika

W programach Pythona można tworzyć swoje wyjątki poprzez przypisanie napisu do zmiennej lub stworzenie nowej klasy wyjątku. [2] Na przykład:

[2] O klasach, już w następnym rozdziale… (*przyp. tłum.*)

```
>>> class NowyWyjatek:
...     def __init__(self, wartosc):
...         self.wartosc = wartosc
...     def __str__(self):
...         return ‘self.wartosc‘
...
>>> try:
...     raise NowyWyjatek(2*2)
... except NowyWyjatek, w:
...     print ’Zgłoszono nowy wyjątek o wartości:’, w.wartosc
...
Zgłoszono nowy wyjątek o wartości: 4
>>> raise NowyWyjatek, 1
Traceback (innermost last):
  File "<stdin>", line 1
__main__.NowyWyjatek: 1
```

W wielu standardowych modułach używa się tego sposobu opisania błędów, które mogą pojawić się w zdefiniowanych w nich funkcjach.

Więcej informacji o klasach można znaleźć w rozdziale 9 «Klasy».

8.6 Jak posprzątać po bałaganiarzu?

Instrukcja `try` posiada jeszcze jedną, opcjonalną klauzulę, która służy do definiowania działań, mających na celu dokonanie koniecznych pod wszelkimi względami porządków. Na przykład:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print ’Żegnaj, świecie!’
...
Żegnaj, świecie!
Traceback (innermost last):
  File "<stdin>", line 2
KeyboardInterrupt
```

Klauzula *finally* jest wykonywana niezależnie od tego, czy pojawił sie wyjątek, czy też nie. Kod zawarty w tym bloku jest również wykonywany, gdy blok `try` zostanie „opuszczony” za pomocą instrukcji `break` lub `return`.

Instrukcja `try` musi posiadać co najmniej jeden blok `except` lub jeden blok `finally`, ale nigdy oba równocześnie.

ROZDZIAŁ

DZIEWIĄTY

Klasy

System klas w Pythonie wyrażony jest minimalnym zestawem nowych środków składniowych i semantycznych. Jest to mieszanka mechanizmów znanych w C++ i Moduli-3. Tak jak w przypadku modułów, klasy w Pythonie nie stawiają żadnych barier pomiędzy swoją definicją a programistą. Polega się tutaj raczej na „grzeczności" programisty, którego dobre wychowanie nie pozwala na „włamywanie się do definicji" klasy. Najważniejsze cechy systemu klas zachowane są tu w całej swej mocy: mechanizm dziedziczenia klas pozwala na posiadanie wielu klas bazowych, klasa pochodna jest w stanie przesłonić definicje metod klas bazowych, jej metoda może wywoływać metodę klasy bazowej o tej samej nazwie. Obiekty mogą posiadać dowolną liczbę danych prywatnych.

W terminologii przejętej w C++, wszystkie składowe klasy pythonowej (włącznie z atrybutami) są *publiczne*, a wszystkie funkcje składowe są *wirtualne*. Nie istnieją specjalne funkcje destruktora i konstruktora. Podobnie jak w Moduli-3 nie istnieją skrótowe sposoby zapisu odniesienia do składowej obiektu we wnętrzu metody: metoda deklarowana jest z wymienionym wprost pierwszym argumentem reprezentującym obiekt, na rzecz którego zostanie wywołana. Podobnie jak w Smalltalku, klasy same są obiektami, aczkolwiek w szerszym tego słowa znaczeniu: w Pythonie wszystkie typy danych to obiekty. Ta cecha określa semantykę mechanizmu importowania i zmiany nazw. Jednak, podobnie jak w C++ i Moduli-3, typy wbudowane nie mogą być użyte jako klasy bazowe klas zdefiniowanych przez użytkownika. Tak jak w C++, ale w odróżnieniu od Moduli-3, większość wbudowanych operatorów mogą zostać za pomocą specjalnej składni przedefiniowane dla różnych typów klasowych.

9.1 Słowo na temat terminologii

Z powodu braku ogólnie przyjętej terminologii, w odniesieniu do klas używać będę terminów z języka Smalltalk i C++. (Chętnie używałbym terminologii zaczerpniętej z Moduli-3, ponieważ obiektowa semantyka tego języka jest bliższa temu, co reprezentuje w tym względzie Python, ale sądzę, że o Moduli-3 słyszało niewielu czytelników).

Muszę także ostrzec, że istnieje pewna terminologiczna pułapka dla czytelników obeznanych z technologią obiektową: słowo «obiekt» w Pythonie nie oznacza koniecznie konkretu (instancji) klasy. Podobnie jak w C++ i Moduli-3, w odróżnieniu od Smalltalka, nie wszystkie typy w Pythonie są klasowe. Podstawowe, wbudowane typy danych„ jak liczby całkowite i listy nie są klasowe (tzn. nie posiadają klasy), nawet tak egzotyczne typy jak pliki też nie są. Jednakże, *wszystkie* typy Pythona przejawiają w swym zachowaniu to, co najlepiej można by wyrazić słowem «obiekt».

Obiekty posiadają swoją indywidualność, mogąc posiadać jednocześnie wiele nazw (w wielu zasięgach nazw). W innych językach programowania nazywa się to aliasami. Zazwyczaj nie jest to docenianie przy pierwszym spotkaniu z Pythonem i w bezpieczny sposób jest ignorowane przy pracy z niemutowalnymi typami danych (liczbami, napisami, niemutowalnymi listami). Jednakże, aliasy wpływają (celowo) na semantykę kodu Pythona, gdy w grę wchodzą mutowalne typy danych, takie jak listy, słowniki i większość typów reprezentujących byty jednostkowe na zewnątrz programu (pliki, okna itp.) Zwykle aliasy przynoszą korzyści, ponieważ zachowują się pod pewnymi względami jak wskaźniki. Na przykład: przekazanie obiektu jako argumentu wywołania nie kosztuje dużo tylko w przypadku przekazania przez implementację wskaźnika do niego. Jeżeli funkcja modyfikuje przekazany w ten sposób obiekt, wywołujący funkcje widzi tę zmianę — jest to powodem występowania dwóch różnych mechanizmów przekazywania parametrów np. w Pascalu.

9.2 Przestrzenie i zasięgi nazw w Pythonie

Zanim przejdziemy do omawiania klas, muszę wspomnieć co nieco o regułach zasięgu nazw w Pythonie. Definicja klasy wprowadza do koncepcji przestrzeni nazw trochę zamieszania i powinno się wiedzieć jak działają zasięgi przestrzeni nazw, aby w pełni zrozumieć, co się w klasach dzieje. Przez przypadek, wiedza o tym mechanizmie jest bardzo użyteczna dla każdego zaawansowanego programisty Pythona.

Zacznijmy od definicji.

Przestrzeń nazw jest wskazaniem obiektu poprzez nazwę. Większość przestrzeni nazw w Pythonie zaimplementowanych jest obecnie w postaci pythonowych słowników, ale zwykle nie jest to zauważalne (z wyjątkiem wydajności) i może się w przyszłości zmienić. Przykładami przestrzeni nazw są: zbiór nazw wbudowanych (funkcje takie jak `abs()` i nazwy wyjątków wbudowanych), nazwy globalne modułu i nazwy lokalne podczas wywołania funkcji. W pewnym sensie, zbiór atrybutów obiektów także jest przestrzenią nazw. Ważne jest, aby pamiętać, że nie istnieje absolutnie żaden związek pomiędzy nazwami z różnych przestrzeni nazw. Dwa różne moduły mogą definiować funkcję „maksymalizuj” — użytkownik tych modułów musi poprzedzić jej nazwę nazwą modułu.

Przy okazji: używam słowa *atrybut* dla każdej nazwy poprzedzonej kropką — np. w wyrażeniu `z.real`, `real` jest atrybutem obiektu `z`. Mówiąc ściśle, odniesienia do nazw w modułach są odniesieniami do atrybutów: w wyrażeniu `nazwa_modulu.nazwa_funkcji`, `nazwa_modulu` jest obiektem modułu, a `nazwa_funkcji` jest jego atrybutem. W tym przypadku istnieje bezpośrednia relacja pomiędzy atrybutami modułu i nazwami globalnymi w nim zdefiniowanymi: oba rodzaje nazw zdefiniowane są w tej samej przestrzeni nazw! [1]

Atrybuty mogą być tylko do odczytu lub zapisywalne. W tym ostatnim przypadku można dokonać operacji przypisania wartości do atrybutu. Atrybuty modułu są zapisywalne: można np. napisać „`modul.wynik = 42`”. Tego rodzaju atrybuty mogą być usuwane za pomocą operatora `del`, np. „`del modul.wynik`”.

Przestrzenie nazw tworzone są w różnych chwilach i są aktywne przez różny czas. Przestrzeń nazw zawierająca nazwy wbudowane tworzona jest podczas rozpoczęcia pracy Pythona i nigdy nie jest usuwana. Przestrzeń nazw globalnych modułu tworzona jest podczas wczytywania jego definicji i jest aktywna również do chwili zakończenia pracy interpretera. Instrukcje wykonywane przez szczytowe wywołania interpretera, zarówno czytane z pliku jak i wprowadzane interaktywnie, są częścią modułu o nazwie `__main__` — tak więc posiadają swoją własną przestrzeń nazw globalnych. (Nazwy wbudowane również przechowywane są w module o nazwie `__builtin__`.)

Przestrzeń nazw lokalnych funkcji tworzona jest w momencie jej wywołania i niszczona, gdy następuje powrót z funkcji lub zgłoszono został w niej wyjątek, który nie został tam obsłużony („zapomniany” byłoby właściwym słowem, które dokładnie oddałoby to, co naprawdę się wtedy dzieje). Oczywiście, wywołanie rekurencyjne powoduje tworzenie za każdym razem nowej przestrzeni nazw lokalnych.

Zasięg jest tekstowym obszarem programu Pythona, w którym przestrzeń nazw jest wprost osiągalna. „Wprost osiągalna” oznacza tutaj, że niekwalifikowane[2] odniesienia do nazwy znajdują tę nazwę w obowiązującej przestrzeni nazw.

Pomimo iż zasięgi określane są w sposób statyczny, używane są w sposób dynamiczny. W każdym momencie wykonania programu, używa się dokładnie trzech zagnieżdżonych zasięgów nazw (tzn. wprost osiągalne są trzy przestrzenie nazw): (i) najbardziej zagnieżdżony, w którym najpierw poszukuje się nazwy, zawiera on nazwy lokalne; (ii) środkowy, przeszukiwany w następnej kolejności, który zawiera aktualne nazwy globalne modułu oraz (iii) zewnętrzny (przeszukiwany na końcu) jest zasięgiem nazw wbudowanych.

Lokalny zasięg nazw umożliwia zwykle odniesienia do nazw występujących (tekstowo) w bieżącej funkcji. Poza obrębem funkcji, lokalna nazwa jest nazwą z przestrzeni nazw globalnych modułu. Definicja klasy stanowi jeszcze jedną przestrzeń nazw w rodzinie przestrzeni nazw lokalnych.

Ważne jest zdawać sobie sprawę z tego, że zasięgi nazw określane są statycznie: zasięg globalny funkcji zdefiniowanej w module jest jego przestrzenią nazw globalnych, bez względu na to skąd i za pomocą jakiego aliasu funkcja została wywołana. Z drugiej strony, właściwe poszukiwanie nazw zachodzi w sposób dynamiczny, w czasie wykonywania programu — jakkolwiek, definicja języka ewoluuje w stronę statycznego sposobu rozstrzy-

[1] Z wyjątkiem jednej rzeczy. Obiekty modułów posiadają tajemniczy tylko atrybut do odczytu: `__dict__`, któremu przypisany jest słownik użyty do zaimplementowania przestrzeni nazw. Nazwa `__dict__` jest atrybutem, ale nie nazwą globalną. To oczywiste, że używanie tej nazwy jest pogwałceniem zasad implementacji przestrzeni nazw i powinno być ograniczone tylko do przypadków śledzenia programu typu „post-mortem”.

[2] tzn. nie trzeba podawać nazwy modułu, klasy, obiektu wraz z kropką (*przyp. tłum.*).

gania nazw, w czasie kompilacji. Nie można więc polegać w programach na dynamicznym rozstrzyganiu nazw (w rzeczywistości, zmienne lokalne są już w obecnej wersji określane statycznie).

Jedną z cech szczególnych Pythona jest to, że przypisanie zawsze zachodzi w najbardziej zagnieżdżonym zasięgu. Przypisania nie powodują kopiowania danych — przywiązują jedynie nazwy do obiektów. To samo zachodzi w przypadku usuwania: instrukcja „`del x`” usuwa związek obiektu identyfikowanego przez nazwę `x` z ta nazwą w przestrzeni nazw lokalnych. W rzeczywistości, wszystkie operacje, które wprowadzają nowe nazwy do przestrzeni nazw lokalnych, to robią. Dotyczy to zawłaszcza instrukcji importu i definicji funkcji (instrukcja `global` może zostać użyta do oznaczenia, że wyszczególniona z nazwa należy do przestrzeni nazw globalnych).

9.3 Pierwszy wgląd w klasy

Klasy wymagają użycia pewnej nowej składni, trzech nowych typów obiektowych i trochę nowej semantyki.

9.3.1 Składnia definicji klasy

Najprostszą formą definicji klasy jest:

```
class NazwaKlasy:
    <instrukcja-1>
    .
    .
    .
    <instrukcja-N>
```

Definicje klas, podobnie jak definicje funkcji (instrukcja `def`) muszą zostać wykonane, zanim zostaną użyte (można umieścić definicję klasy w rozgałęzieniu instrukcji `if` lub wewnątrz funkcji).

W praktyce, instrukcje wewnątrz definicji klasy będą definicjami funkcji, ale również dozwolone są inne, czasem bardzo użyteczne — powrócimy do tego później. Definicja funkcji w ciele klasy posiada zwykle szczególną formę listy argumentów formalnych, dedykowanych szczególnej konwencji wywoływania metod [3] — to również zostanie wyjaśnione później.

Kiedy wprowadza się definicję klasy do programu, tworzona jest nowa przestrzeń nazw używana jako zasięg lokalny nazw — w ten sposób, wszystkie przypisania do zmiennych lokalnych dotyczą nazw z tej właśnie przestrzeni. W szczególności, definicja funkcji powoduje powiązanie jej nazwy z obiektem tej funkcji w lokalnej przestrzeni nazw klasy.

Kiedy definicja klasy się kończy, tworzony jest *obiekt klasy*. Mówiąc wprost, jest rodzaj opakowania dla przestrzeni nazw stworzonej przez definicję klasy — o obiektach klasy dowiemy się nieco więcej w dalszej części rozdziału. Zasięg pierwotny (ten, który obowiązywał przed rozpoczęciem definicji klasy) jest przywracany, a nowo utworzony obiekt klasy powiązany zostaje z jej nazwą, która została podana w nagłówku definicji klasy (w tym przypadku `NazwaKlasy`).

9.3.2 Obiekty klas

Na obiektach klasy można przeprowadzić dwa rodzaje operacji: odniesienia do atrybutów i konkretyzację.

Odniesienie do atrybutu da się wyrazić za pomocą standardowej składni używanej w przypadku odniesień dla wszystkich atrybutów w Pythonie: `obiekt.nazwa`. Prawidłowymi nazwami atrybutów są nazwy, które istniały w przestrzeni nazw klasy w czasie tworzenia jej obiektu. Tak więc, jeśli definicja klasy wygląda następująco:

[3] czyli funkcji zdefiniowanych dla obiektów klas (*przyp. tłum.*).

```
class MojaKlasa:
    "Prosta, przykładowa klasa"
    i = 12345
    def f(x):
        return 'witaj świecie'
```

to `MojaKlasa.i` i `MojaKlasa.f` są prawidłowymi odniesieniami do jej atrybutów, których wartością jest odpowiednio liczba całkowita i obiekt metody. Atrybutom klasy można przypisywać wartości, w ten sposób można zmienić wartość `MojaKlasa.i` poprzez przypisanie. `__doc__` jest także prawidłową nazwą atrybutu klasy, którego wartością jest napis dokumentacyjny należący do klasy: `"Prosta, przykładowa klasa"`.

Konkretyzację klasy przeprowadza się używając notacji wywołania funkcji. Należy tylko udać, że obiekt klasy jest bezparametrową funkcją, która zwraca instancję (konkret) klasy. Oto przykład (używa definicji klasy z poprzedniego ćwiczenia):

```
x = MojaKlasa()
```

w którym tworzy się nowy *konkret* klasy i wiąże się ten obiekt z nazwą zmiennej lokalnej `x` poprzez przypisanie do niej.

Operacja konkretyzacji („wywołanie" obiektu klasy) tworzy pusty obiekt. Dla wielu klas występuje konieczność stworzenia swojego konkretu w pewnym znanym, początkowym stanie. Dlatego też, można zdefiniować dla klas specjalną metodę o nazwie `__init__()`, tak jak poniżej:

```
    def __init__(self):
        self.dane = []
```

W momencie konkretyzacji klasy, automatycznie wywołana zostanie metoda `__init__()` dla nowopowstałego konkretu klasy.

Tak więc, w tym przykładzie, nowy, zainicjalizowany konkret klasy można uzyskać poprzez:

```
x = MojaKlasa()
```

Oczywiście, dla zwiększenia wygody w użyciu, metoda `__init__()` może posiadać argumenty. W tym przypadku, argumenty dane na wejściu operatora konkretyzacji klasy, przekazywane są do `__init__()`. Na przykład:

```
>>> class Zespolona:
...     def __init__(self, rzeczywista, urojona):
...         self.r = rzeczywista
...         self.i = urojona
...
>>> x = Zespolona(3.0,-4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Obiekty konkretu klasy

No dobrze, więc co można zrobić z obiektem konkretu klasy? Jedyna operacją zrozumiałą dla obiektu konkretu klasy jest odniesienie do jego atrybutów. Istnieją dwa rodzaje prawidłowych nazw atrybutów takiego obiektu.

Pierwszy z nich nazywany jest przeze mnie *atrybutami danych*. Są to odpowiedniki «zmiennych konkretu klasy» w Smalltalku i «danymi składowymi» w C++. Atrybuty danych nie muszą być deklarowane. Podobnie jak zmienne lokalne, są one tworzone w momencie pierwszego przypisania wartości do nich. Jeżeli `x` jest na przykład konkretem klasy `MojaKlasa`, to poniższy fragment kodu spowoduje wydrukowanie wartości `16` bez powstania komunikatu o błędzie:

```
x.licznik = 1
while x.licznik < 10:
    x.licznik = x.licznik * 2
print x.licznik
del x.licznik
```

Drugim rodzajem odniesienia do atrybutu są *metody*. Metoda jest funkcją, która „należy” do obiektu konkretu klasy. (W Pythonie, określenie «metoda» nie przynależy tylko i wyłącznie do konkretów klasy: inne typy obiektowe też mogą mieć metody, np. listy mają metody o nazwie `append`, `insert`, `remove`, `sort`, itd. Jednakże, określenie «metoda» użyte poniżej, odnosi się do funkcji należącej do instancji klasy, chyba, że zaznaczono inaczej.)

Poprawna nazwa metody obiektu konkretu zależy od jego klasy. Z definicji, wszystkie atrybuty klasy, które są funkcjami (zdefiniowanymi przez użytkownika), są poprawnymi nazwami metod konkretu tej klasy. Tak więc, w naszym przykładzie `x.f` jest funkcją, ale `x.i` już nie, ponieważ `MojaKlasa.i` nie jest funkcją. `x.f` nie jest tym samym co `MyClass.f` — jest to *obiekt metody*, a nie obiekt funkcji.

9.3.4 Obiekty metod

Metodę wywołuje się w następujący sposób:

```
x.f()
```

w naszym przykładzie zwróci ona napis `"witaj świecie"`. Nie jest jednakże konieczne wywoływać metodę w ten bezpośredni sposób: `x.f` jest obiektem i można go zachować i wywołać później, np.:

```
xf = x.f
while 1:
    print xf()
```

będzie po wsze czasy drukować „`witaj świecie`”.

Co właściwie dzieje się, gdy wywoływana jest metoda? Można było zauważyć, że `x.f()` została wywołana bez argumentu, pomimo że definicja funkcji `f` zawiera jeden argument formalny `self`. Co się z nim stało? Pewne jest, że Python zgłosi wyjątek, gdy funkcja, która wymaga podania argumentów, wywoływana jest bez nich — nawet, jeżeli żaden z nich nie jest wewnątrz jej używany...

Odpowiedź jest dość oczywista: jedną ze szczególnych cech metody jest to, że obiekt, na rzecz którego jest wywoływana, przekazywany jest jako pierwszy argument funkcji. W naszym przykładzie, wywołanie `x.f()` odpowiada dokładnie wywołaniu `MyClass.f(x)`. Ogólnie rzecz biorąc, wywołanie metody z listą *n* argumentów równoznaczne jest wywołaniem odpowiedniej funkcji klasy z lista argumentów stworzoną poprzez włożenie obiektu konkretu klasy przed pierwszy element (argument wywołania) listy argumentów.

Jeśli w dalszym ciągu nie jest zrozumiałe jak działają metody, spróbujmy spojrzeć na ich implementację, co powinno rozjaśnić nieco to zagadnienie. Gdy używa się odniesienia do atrybutu konkretu klasy, który nie jest atrybutem danych, następuje przeszukanie klasy konkretu. Jeżeli znaleziono nazwę w klasie, która odnosi się do funkcji w niej zdefiniowanej, tworzony jest obiekt metody jako paczka zawierająca odniesienia do obiektu konkretu klasy i obiektu funkcji klasy. Kiedy obiekt metody jest wywoływany z listą argumentów, następuje jego rozpakowanie, tworzona jest nowa lista argumentów z obiektu konkretu klasy i oryginalnej listy argumentów, a następnie obiekt funkcji wywoływany jest z nowo utworzoną listą.

9.4 Luźne uwagi

[Ten podrozdział być może powinien być umieszczony gdzieś indziej...]

Nazwy atrybuty danych przesłaniają te same nazwy metod. Przyjęło się używać pewnej konwencji w nazewnictwie, aby uniknąć przypadkowego konfliktu nazw, co może powodować trudne do znalezienia błędy w dużych programach. Na przykład, rozpoczynanie nazw metod od dużej litery, dodawanie przedrostka do nazwy atrybutu danych (zazwyczaj jest to znak podkreślenia, „_”) lub używanie czasowników dla metod i rzeczowników dla danych.

Atrybuty danych mogą być użyte w metodach w ten sam sposób, w jaki używane są przez zwykłych użytkowników („klientów”) obiektu. Innymi słowy, klasy w Pythonie nie są użyteczne jako środek do implementacji „czystych” abstrakcyjnych typów danych. W rzeczywistości, nic w Pythonie nie umożliwia ukrywania danych — wszystko to oparte jest na konwencji. (Z drugiej strony, implementacja Pythona, która jest całkowicie napisana w C, może całkowicie ukryć szczegóły implementacyjne i kontrolować zakres dostępu do obiektu, jeśli jest to konieczne. Jest do pewna droga postępowania dla wszelkich rozszerzeń Pythona pisanych w C).

Klienci powinni używać atrybutów danych z pewną dozą ostrożności — mogą narobić dużo bałaganu w niezmiennikach metod poprzez naruszanie ich atrybutów danych. Proszę zauważyć, że można dodawać nowe atrybuty danych do obiektu konkretu bez naruszania poprawności działania metod pod warunkiem, że uniknięto konfliktu nazw — trzeba to znów zaznaczyć: trzymanie się konwencji w nazewnictwie może zaoszczędzić bólu głowy w tym względzie.

Nie istnieje skrócony sposób dostępu z wnętrza metody do atrybutów danych (lub innych metod)! Moim zdaniem, zwiększa to czytelność ich kodu: nie ma szansy pomieszania nazw zmiennych lokalnych i zmiennych konkretu podczas przeglądania programu źródłowego.

Przyjęta ogólnie konwencją jest nazywanie pierwszego argumentu metody `self`.[4] To nic więcej niż konwencja: nazwa `self` nie ma zupełnie żadnego specjalnego znaczenia w Pythonie (proszę jednak zauważyć, że nie przestrzeganie tej konwencji może spowodować powstanie mniej czytelnego kodu oraz to, że niektóre narzędzia służące do *przeglądania definicji klas* mogą polegać właśnie na tej konwencji)[5]

Każdy obiekt funkcji, który jest atrybutem klasy definiuje metodę dla konkretu tej klasy. Nie jest konieczne aby definicja takiej funkcji była tekstowo umieszczona w definicji jej klasy: przypisanie obiektu funkcji do lokalnej zmiennej w klasie powoduje to samo jakby była ona tam umieszczona. Na przykład:

```
# Funkcja zdefiniowana poza obrębem klasy
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1
    def g(self):
        return 'witaj świecie'
    h = g
```

W tym wypadku `f`, `g` i `h` są wszystkie atrybutami klasy `C`, których wartością są obiekty funkcji i w konsekwencji stają się metodami konkretów klasy `C` — `h` jest dokładnie odpowiednikiem `g`. Proszę zauważyć, że w praktyce taka kombinacja służy jedynie do wprowadzenia czytelnika takiego programu w stan głębokiego zmieszania i frustracji.

Metody mogą wywoływać inne metody poprzez użycie atrybutów metod obiektu `self`, np.:

```
class Worek:
    def __init__(self):
        self.dane = []
    def dodaj(self, x):
        self.dane.append(x)
    def dodaj_dwa_razy(self, x):
        self.dodaj(x)
        self.dodaj(x)
```

[4] `self` można przetłumaczyć w tym kontekście jako «ja sam» lub «moja,mój». Dwa ostatnie określenia być może są bardziej adekwatne jeżeli weźmiemy pod uwagę, że zaraz pod `self` następuje odniesienie do atrybutu konkretu, np. „`self.atrybut`” (*przyp. tłum.*).

[5] Dlatego też zrezygnowałem z tłumaczenia tej nazw w przykładach na słowo takie jak «ja», albo «się» (*przyp. tłum.*).

Metody mogą używac odniesień do nazw globalnych w ten sam sposób jak zwykłe funkcje. Zasięg globalny związany z metodą jest po prostu tym modułem, który zawiera definicje klasy (sam klasa nigdy nie jest używana jako zasięg globalny). Zazwyczaj rzadko można spotkać odniesienia do zmiennych globalnych w metodach. Istnieje parę dobrych powodów do używania zmiennych globalnych: funkcje i moduły importowane do przestrzeni nazw globalnych mogą zostać użyte przez metody, tak samo jak funkcje i klasy należace do tej samej przestrzeni. Zwykle klasa zawierająca tę metodę sama jest zdefiniowana w globalnym zasięgu. W następnym podrozdziale będzie się można dowiedzieć, dlaczego w metodzie trzeba czasami użyć odniesienia do jej własnej klasy!

9.5 Dziedziczenie

Bez istnienia mechanizmu dziedziczenia, ta cecha języka, która określana jest mianem „klasy" nie byłaby jego warta. Poniżej podano składnię definicji klasy dziedziczącej:

```
class NazwaKlasyPotomnej(NazwaKlasyBazowej):
    <instrukcja-1>
    .
    .
    .
    <instrukcja-N>
```

Nazwa `NazwaKlasyBazowej` musi być zdefiniowana w zasięgu zawierającym definicję klasy pochodnej. Zamiast nazwy klasy bazowej dopuszcza się również wyrażenie. Jest to szczególnie przydatne, jeśli nazwa klasy bazowej zdefiniowana jest w innym module, np.:

```
class NazwaKlasyPochodnej(modul.NazwaKlasyBazowej):
```

Wykonanie definicji klasy pochodnej następuje w taki sam sposób jak dla klasy bazowej. Klasa jest zapamiętywana w momencie stworzenia obiektu klasy. Ten mechanizm używany jest w procesie rozstrzygania odniesień do atrybutów konkretu takiej klasy: jeśli poszukiwany atrybut nie jest znajdowany w klasie, poszukiwany jest w klasie bazowej. Ta zasada stosowana jest rekurencyjnie w przypadku, gdy klasa bazowa jest pochodną innej.

W konkretyzacji takiej klasy nie ma nic szczególnego: `NazwaKlasyPochodnej()` tworzy nowy konkret klasy. Odniesienia do metod rozstrzygane są w następujący sposób: poszukuje się odpowiedniego atrybutu klasy, jeśli jest to konieczne, schodzi się z poszukiwaniem w głąb drzewa dziedziczenia. Gdy odniesienie wskazuje na obiekt funkcji, metoda uznawana jest za poprawną.

Klasy pochodne mogą przesłaniać metody ich klas bazowych. Metody nie są obsługiwane w żaden uprzywilejowany sposób, gdy wywołują inne metody tego samego konkretu klasy, a więc metoda klasy bazowej, która wywołuje metodę zdefiniowaną w tej samej klasie może uruchomić metodę swojej klasy pochodnej, która tamtą przesłania (uwaga dla programistów C++: wszystkie metody w Pythonie zachowują sie jak *wirtualne*).

Przesłonięcie metody w klasie pochodnej jest raczej rozszerzeniem zbioru obsługiwanych funkcji niż zastąpieniem elementu noszącego taką samą nazwę jak ta metoda. Istnieje prosty sposób wywołania metody klasy bazowej: po prostu „`NazwaKlasyPodstawowej.nazwa_metody(self, argumenty)`". Ten mechanizm można stosować, jeżeli klasa podstawowa zdefiniowana jest w globalnym zasięgu nazw.

9.5.1 Dziedziczenie wielorakie

W Pythonie występuje ograniczona forma dziedziczenia wielorakiego. Poniżej podano przykład definicji klasy dziedziczącej z wielu klas podstawowych:

```
class NazwaKlasyPochodnej(Bazowa1, Bazowa2, Bazowa3):
    <instrukcja-1>
    .
    .
    .
    <instrukcja-N>
```

Aby wyjaśnić semantykę dziedziczenia wielorakiego w Pythonie, konieczne jest poznanie zasady znajdowania odniesień atrybutów klasy. Jest to zasada „najpierw w głąb, potem na prawo”. Jeśli atrybut nie zostanie znaleziony w klasie `NazwaKlasyPochodnej`, zostanie poszukany w `Bazowa1`, a potem (rekurencyjnie) w klasach bazowych klasy `Bazowa1` i jeśli tam nie zostanie znaleziony, poszukiwanie zostanie przeniesione do klasy `Bazowa2`.

(Niektórym bardziej naturalne wydaje się być poszukiwanie nazw „w szerz” — przeszukiwanie `Bazowa2` i `Bazowa3` przed przeszukaniem klas bazowych klasy `Bazowa1`. Wymaga to jednak znajomości miejsca zdefiniowania konkretnego atrybutu (czy jest on zdefiniowany w `Bazowa1` lub w jednej z jej klas bazowych), zanim można by wywnioskować konsekwencje konfilktu nazw atrybutu z klasy `Bazowa2`. Poszukiwanie „najpierw w głąb” nie rozróżnia atrybutów zdefiniowanych bezpośrednio od dziedziczonych).

Nieograniczone użycie dziedziczenia wielorakiego może w oczywisty sposób stać się koszmarem w procesie pielęgnacji oprogramowania, zwłaszcza że w Pythonie unikanie konfliktów nazw opiera się na umowie. Jednym ze bardziej znanych przykładów problemu z dziedziczeniem wielorakim jest sytuacja gdy dwie klasy bazowe dziedziczą z tej samej klasy-przodka.[6]. Nie wiadomo jak bardzo taka semantyka jest użyteczna w połączeniu z faktem, że mamy pojedynczą kopię „zmiennych konkretu” lub atrybutów danych wspólnej klasy bazowej.

9.6 Zmienne prywatne

Python posiada ograniczony mechanizm implementacji zmiennych prywatnych klasy. Każdy identyfikator o nazwie `__pomyje` (przynajmniej dwa poprzedzające znaki podkreślenia i co najwyżej jeden znak podkreślenia zakończający nazwę) jest zastępowany przez `_nazwaklasy__pomyje`, gdzie `nazwaklasy` jest nazwą bieżącej klasy z usuniętymi znakami podkreślenia[7]. Kodowanie to występuje zawsze, bez względu na pozycję składniową identyfikatora. Może to zostać użyte do zdefiniowania prywatnych konkretów klasy i zmiennych klasowych, metod, jak również obiektów globalnych, a nawet do przechowywania zmiennych konkretów tej klasy w konkretach *innych* klas. Ucięcie nazwy może wystąpić po przekroczeniu przez identyfikator długości 255 znaków. Poza granicami klasy, lub gdy nazwa klasy składa się tylko ze znaków podkreślenia, proces takiego kodowania nigdy nie zachodzi.

Kodowanie nazw ma na celu uzyskanie łatwego sposobu definiowania „prywatnych” zmiennych konkretu oraz metod bez martwienia się o zmienne konkretu zdefiniowane przez inne klasy pochodne, lub mucking with instance variables by code outside the class. Trzeba zauważyć, że zasady kodowania nazw przeznaczone są głównie w celu uniknięcia nieprzyjemnych wypadków — dla zdeterminowanego programisty wciąż możliwe jest uzyskanie bezpośredniego dostępu do do zmiennej uważaną za prywatną. Jest to nawet szczególnie użyteczne np. w przypadku debuggerów. Jest to właściwie jedyny powód, dla którego ta dziura w bezpieczeństwie nie została załatana (Pluskwa: dziedziczenie klasy o nazwie takiej samej jak jedna z klas bazowych powoduje bezpośredni dostęp do zmiennych prywatnych klasy bazowej).

Kod przekazany instrukcji `exec` i funkcji `eval()` lub `evalfile()` nie zawiera nazwy bieżącej klasy; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. To samo ograniczenie dotyczy `getattr()`, `setattr()` i `delattr()` oraz odwołań do kluczy słownika `__dict__`.

Poniżej przedstawiono przykład klasy, która implementuje swoje wersje metod `__getattr__` i `__setattr__`. Za ich pomocą wszystkie atrybuty konkretu przechowywane są w zmiennej prywatnej (przypomina to sposób działania Pythona 1.4 i poprzednich wersji):

[6] Zwłaszcza w C++... (*przyp. tłum.*)
[7] Jeśli takie występują (*przyp. tłum.*)

```
class VirtualAttributes:
    __vdict = None
    __vdict_name = locals().keys()[0]

    def __init__(self):
        self.__dict__[self.__vdict_name] = {}

    def __getattr__(self, name):
        return self.__vdict[name]

    def __setattr__(self, name, value):
        self.__vdict[name] = value
```

9.7 Sztuczki i chwyty

Czasami potrzebe jest użycie typu danych spotykanego w Pascalu pod nazwąi rekordu lub struktury w C. Trzeba po prostu zgrupować pewną liczbę nazwanych elemtów w jednym „zasobniku”. W prosty sposób da to się wyrazić za pomocą pustej definicji klasy, tzn.:

```
class Pracownik:
    pass

janek = Pracownik() # Tworzy pusty rekord pracownika

# Wypełnienie pól w rekordzie
janek.nazwa = 'Janek Kos'
janek.miejsce = 'laboratorium komputerowe'
janek.pensja = 1000
```

We fragmencie pythonowatego kodu, który wymaga użycia szczególnego abstrakcyjnego typu danych, można skorzystać z jakiejś klasy, która emuluje metody wymaganego typu. Na przykład, jeśli istnieje funkcja, która formatuje dane na wyjściu jakiegoś pliku, to można zdefiniować klasę z metodami `read()` i `readline)`, które pobierają dane z łańcucha znaków zamiast z pliku i przekazać ją do danej funkcji.

Metody konkretu klasy również posiadają swoje atrybuty: `m.im_self` jest obiektem (wskazanie na), dla którego dana metoda ma zostać wywołana, a `m.im_func` jest obiektem funkcji, która implementuje daną metodę.

9.7.1 Wyjątki mogą być klasami

Wyjątki definiowane przez użytkownika nie są już ograniczone tylko do obiektów napisów — można również użyć klas. Poprzez użycie tego mechanizmu można stworzyć rozszerzalna hierarchię wyjątków.

Istnieją dwie poprawne formy instrukcji zgłoszenia wyjątku:

```
raise Klasa, konkret

raise konkret
```

W pierwszej formie, `konkret` musi być konkretem klasy `Klasa` lub klasy pochodnej od niej. Druga forma jest skróconym zapisem dla:

```
raise konkret.__class__, konkret
```

Klauzula `except` instrukcji `try` może zawierać również listę klas. Klasa w tej klauzuli odpowiada zgłoszonemu

wyjątkowi, jeśli jest tej samej klasy co wyjątek lub jego klasą bazową (i nie inaczej — lista klas pochodnych w klauzuli `except` nie odpowiada wyjątkom, które są ich klasami bazowymi). Na przykład, wynikiem działania poniższego kodu będzie wyświetlenie B,C,D w takim właśnie porządku:

```
class B:
    pass
class C(B):
    pass
class D(C):
    pass

for c in [B, C, D]:
    try:
        raise c()
    except D:
        print "D"
    except C:
        print "C"
    except B:
        print "B"
```

Trzeba zauważyć, że jeżeli klauzula `except` byłaby odwrócona (tzn. „`except B`” na pierwszym miejscu), program wyświetliłby B, B, B — uruchamiany jest kod pierwzej pasującej klauzuli.

Gdy wyświetlany jest komunikat o błędzie w przypadku niewyłapanego wyjątku klasowego, wyświetlana jest nazwa klasy, potem dwukropek i spacja, a następnie konkret klasy wyjątku przekształcony do napisu za pomocą funkcji wbudowanej `str()`.

ROZDZIAŁ

DZIESIĄTY

Co teraz?

Mam nadzieję, że lektura tego przewodnika wzbudziła zainteresowanie użytkowaniem Pythona. Co teraz powinno się zrobić?

Należy przeczytać lub przynajmniej przekartkować, «Opis biblioteki». Stanowi on kompletny (aczkolwiek dość suchy) opis typów, funkcji i modułów, co może zaoszczędzić mnóstwo czasu podczas pisania programów w Pythonie. Standardowa dystrybucja Pythona zawiera *mnóstwo* kodu napisanego zarówno w C jak i w samym Pythonie. Można spotkać moduły do czytania skrzynek pocztowych na UNIKSIE, odbierania treści dokumentów poprzez HTTP, generowania liczb losowych, przetwarzania opcji podanych w linii poleceń, pisania skryptów CGI, kompresji danych i dużo, dużo więcej. Pobieżne przeglądnięcie «Opisu biblioteki» na pewno da wyobrażenie na temat tego, co jest dostępne.

Główną witryną Pythona jest http://www.python.org. Zawiera ona kod, dokumentację i odnośniki do innych stron na całym świecie dotyczących Pythona. Witryna ta jest mirrorowana w wielu miejscach świata: Europie, Japonii i Australii. Mirror może być szybszy niż witryna główna, co zależy głównie od lokalizacji Czytelnika. Bardziej nieformalną witryną jest http://starship.python.net, która zawiera wiele stron domowych ludzi zajmujących się Pythonem oraz stworzone przez nich oprogramowanie, które można stamtąd załadować (ang. download).

W celu zgłoszenia błędów implementacji Pythona oraz zadawania pytań na jego temat, można zapisać się na listę dyskusyjną comp.lang.python lub wysłać listy do python-list@cwi.nl. Do listy dyskusyjnej i skrzynki pocztowej prowadzi bramka, tak więc wiadomości tam wysyłane są automatycznie przesyłane dalej do innych. Obecnie przez bramkę przechodzi około 35–45 wiadomości dziennie zawierających pytania (i odpowiedzi), sugestie na temat nowych cech języka i zgłoszenia nowych modułów. Przed wysłaniem wiadomości należy sprawdzić Listę Często Zadawanych Pytań (FAQ) pod adresem http://www.python.org/doc/FAQ.html lub spojrzeć na zawartość katalogu „Misc” znajdującego się w standardowej dystrybucji źródłowej Pythona. Lista FAQ zawiera wiele odpowiedzi na wciąż napływające pytania i być może zawiera już rozwiązanie danego problemu.

Społeczność pythonowską można wspomóc poprzez przyjączenie się do organizacji Python Software Activity (PSA), która obsługuje witrynę python.org, ftp i serwery pocztowe oraz organizuje warsztaty Pythona. O tym jak się dołączyć do PSA, można dowiedzieć się pod http://www.python.org/psa/.

DODATEK

A

Interaktywna edycja i operacje na historii poleceń

Niektóre wersje interpretera zawierają mechanizmy edycji aktualnej linii poleceń i operacji na historii poleceń, podobne do mechanizmów zawartych w powłoce Korna i GNU Bash. Jest on zaimplementowany za pomocą biblioteki *GNU Readline*. która umożliwia edycję w stylu Emacsa i edytora vi. Biblioteka ta posiada własną dokumentację, której nie będę tutaj duplikował, jakkolwiek wyjaśnię pewne podstawy jej używania. Interaktywna edycja i i historia poleceń, które są tutaj opisane, dostępne są jako opcja na platformach UNIX i CygWin.

Rozdział ten *nie* opisuje własności edycyjnych pakietu PythonWin Marak Hammonda oraz opartego na bibliotece Tk środowiska IDLE, który jest dydtrybuowany wraz z Pythonem. I zupełnie inną parą kaloszy są własności linii poleceń emulatora DOS-owego na NT i innych okien DOS-owych w Windows.

A.1 Edycja linii poleceń

Edycja linii poleceń jest aktywna, jeśli jest zawarta w dystrybucji, w każdym momencie wystąpienia pierwszego lub drugiego znaku zachęty. Bieżący wiersz może być edytowany za pomocą zwykłych znaków sterujących Emacsa. Najważniejszymi z nich są: C-A (Control-A), które przesuwa kursor na początek linii, C-E na koniec, C-B jedną pozycję w lewo, C-F na prawo. Znaki backspace usuwają znaki na lewo od kursora, C-D na prawo. C-K usuwa resztę linii na prawo od kursora, C-Y wstawia ostatnio usunięty napis. C-podkreślenie powraca do stanu przed ostatnią zmianą—może to być powtarzane, aby wrócić do wcześniejszych stanów edycji.

A.2 Zastępowanie poleceń historycznych

Działa to jak następuje: wszystkie wprowadzone, niepuste wiersze wejściowe zachowywane są w buforze historii. W chwili, gdy wprowadzony jest nowy wiersz, znajdujemy się w nowym wierszu, na końcu bufora historii. C-P przesuwa nas jeden wiersz wstecz w buforze, C-N jeden wiersz naprzód. Każdy wiersz w buforze może być przedmiotem edycji. W chwili modyfikacji wiersza na jego przodzie pojawia się znak ’asteriks’. Naciśnięcie klawisza Return (Enter) powoduje wprowadzenie bieżącego wiersza do interpretacji. C-R rozpoczyna proces inkrementacyjnego poszukiwania wstecz a C-S w przód.

A.3 Funkcje klawiszowe

Funkcje przypisywane kombinacjom klawiszy wraz z innymi parametrami mogą być dostosowane do potrzeb użytkownika, poprzez wstawienie odpowiednich poleceń do pliku „$HOME/.inputrc”. Funkcje klawiszowe posiadają następującą formę:

```
nazwa-kombinacji-klawiszowej: nazwa-funkcji
```

lub

```
"napis": nazwa-funkcji
```

a opcje mogą zostać ustawione za pomocą

```
set nazwa-opcji wartość
```

Na przykład:

```
# Wolę styl vi:
set editing-mode vi
# Edycja za pomocą jednego wiersza:
set horizontal-scroll-mode On
# zmiana niektórych funkcji klawiszowych:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Trzeba zauważyć, że funkcją przypisaną do klawisza TAB w Pythonie jest funkcja wstawiająca znak tabulacji zamiast funkcji dopełniania nazw plików, która w bibliotece Readline jest domyślną. Jeżeli to jest naprawdę konieczne, mozna to pythonowe przypisanie zmienić poprzez wstawienie

```
TAB: complete
```

do pliku „$HOME/.inputrc”. Oczywiście spowoduje to utrudnione wprowadzanie wierszy kontynuacji, które mają byc wcięte…

Automatyczne dopełnianie nazw zmiennych i modułów jest dostępne. Aby to uzyskać w trybie interaktywnym interpretera, należy dodać następujące polecenie w pliku „$HOME/.pythonrc.py”:

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

Klawiszowi TAB zostanie przypisana funkcja dopełniania nazw, tak więc naciśnięcie dwa razy tego klawisza spowoduje wyświetlenie propozycji nazw, które pasują do wprowadzonego już napisu: instrukcji, bieżących zmiennych lokalnych i dostępnych w tym kontekście nazw modułów. Dla wyrażeń kropkowanych, jak `string.a`, wyrażenie aż do kropki zostanie określone, a następnie pojawią się propozycje atrybutów wynikowego obiektu. Proszę zauważyć, że może to spowodować wykonanie kodu aplikacji, jeśli obiekt z metodą `__getattr__()` jest częścią wyrażenia.

A.4 Komentarz

Wyżej wspomniana właściwość jest ogromnym krokiem naprzód w porównaniu z poprzednimi wersjami interpretera, aczkolwiek pozostały pewne niespełnione życzenia. Byłoby miło, aby automatycznie sugerowano poprawną indentację w wierszach kontynuacji (parser wie czy następny symbol ma być wcięty). Mechanizm uzupełniania nazw mógłby korzystać z tablicy symboli interpretera. Polecenie sprawdzenia (a nawet proponowania) dopasowania nawiasów, cudzysłowiów itd. również byłoby bardzo pomocne.

INDEKS

Symbole

C

D

F

I

M

N

O

P

R

S

U

Z