

Język Python

dr inż. Paweł Kędzierski

1 Zaczynamy

Interpreter Python uruchamia się komendą python:

```
% python
Python 1.6 (#1, Oct 13 2000, 01:16:18) [GCC 2.7.2.3] on linux2
Copyright (c) 1995-2000 CNRI.
All Rights Reserved.
Copyright (c) 1991-1995 Stichting Mathematisch Centrum, Amsterdam.
All Rights Reserved.
>>>
```

Gotowość do wprowadzenia (kolejnej) instrukcji sygnalizowana jest znakami „>>>”. Jeśli poprzednia instrukcja wymaga dokończenia, interpreter pokazuje „... ” (trzy kropki). W dalszych przykładach, w ten sam sposób będę odróżniał instrukcje od wyników ich działania.

Język Python został pomyślany tak, aby był prosty, zrozumiały dla czytającego i łatwy do nauczenia się, a jednocześnie elastyczny i zwięzły. Na przykład, zmiennych w ogóle nie trzeba deklarować — wystarczy nadać im wartość:

```
>>> x=15.0
>>> imie='Paweł'
```

Co więcej, w tej samej zmiennej można przechować daną dowolnego typu, więc nie będzie błędem napisanie po kolei:

```
>>> x=2
>>> x=1.1234
>>> x='Łańcuch znaków'
```

Nie znaczy to, że Python nie rozróżnia rodzaju danych — musi to robić chociażby dlatego, że ta sama operacja, np. dodawanie, ma różne znaczenie dla różnych typów: suma arytmetycznej dla liczb, łączenia dla napisów itp.

Raz utworzona zmienna istnieje tak długo, jak jest potrzebna — Python sam sprząta nieużywaną pamięć. Ale próba użycia zmiennej, której nie ma, spowoduje błąd (o nazwie `NameError`).

Jeżeli instrukcja, którą wydamy, da jakiś wynik (wartość), wartość ta zostanie wypisana*. Można więc używać Pythona jako kalkulatora:

```
>>> a=1
>>> a+5
6
>>> (a+5)/2.5
2.3999999999999999
```

Ale uwaga! Nie zawsze wynik może być zgodny z oczekiwaniem. Weźmy taki przykład:

```
>>> 1/5
0
```

Wbrew pozorom, jest to wynik prawidłowy[†]. Ponieważ Python rozpoznaje typ zmiennych po ich zapisie, w podanym wyrażeniu widzi on dwie liczby całkowite. Przy próbie dzielenia, stosuje do nich zatem operację dzielenia *całkowitego*, odrzucając resztę. Aby uzyskać dzielenie rzeczywiste, co najmniej jeden operand musi być liczbą rzeczywistą:

```
>>> 1/5.0
0.2
```

Niestety, problem taki pojawia się jeśli dane są podawane przez użytkownika programu, od którego nie można przecież wymagać stosowania się do takich reguł. Dlatego Python umożliwia zamianę typu na wymagany przez programistę. Poniższa tabelka zestawia odpowiednie funkcje:

Funkcja	Zamienia
<code>chr()</code>	wartość całkowitą na znak o odpowiednim kodzie ASCII
<code>complex()</code>	wartość całkowitą, rzeczywistą lub napisową na zespoloną
<code>ord()</code>	znak na jego kod ASCII
<code>float()</code>	wartość całkowitą lub napisową na rzeczywistą
<code>int()</code>	wartość rzeczywistą lub napisową na całkowitą
<code>str()</code>	jakąkolwiek wartość na napis
<code>eval()</code>	interpretuje wartość napisową tak, jak Python, i oblicza wynik

* Tylko podczas interaktywnej pracy z interpreterem. Programy w Pythonie, a także procedury z zewnętrznych modułów, wypiszą tylko to co jest objęte instrukcjami `print`.

† Z punktu widzenia konwencji tego języka, stosującego różnie działające operatory, w zależności od typu operandów.

A oto konkretny przykład: program prosi użytkownika o dane (funkcja `input()`), następnie zamienia typ na rzeczywisty i oblicza wynik dzielenia:

```
>>> a = input('Podaj a: ')
>>> b = input('Podaj b: ')
>>> print 'Wynik dzielenia: ', float(a)/b
```

1.1 Biblioteki podprogramów

Zestaw działań udostępnianych przez sam interpreter Pythona jest raczej skromny, nawet jeżeli chcielibyśmy używać go tylko jako kalkulatora. Natomiast bardzo bogaty zestaw różnych funkcji można znaleźć w bibliotekach podprogramów. Nawet standardowe funkcje matematyczne nie są „ładowane” bez potrzeby, aby nie zajmowały pamięci. Aby z nich skorzystać, trzeba wczytać odpowiednią bibliotekę (moduł) Pythona:

```
>>> import math
>>> math.sin(math.pi/2)
1.0
```

Instrukcja `import` służy właśnie do tego, aby załadować do pamięci interpretera żadaną bibliotekę podprogramów. Można to zrobić tak jak powyżej, ale jak widać, korzystanie z zawartości modułu nie jest wtedy zbyt wygodne, bo trzeba używać składni `moduł.funkcja()` lub `moduł.stała`. Dlatego często ładuje się z wybranego modułu albo wszystko hurtem, albo tylko wybrane elementy:

```
>>> from math import sin,pi # Wczytaj funkcję sin() i stałą pi
>>> sin(pi/2)
1.0
>>> from math import * # Ładuj wszystkie obiekty modułu math
>>> cos(pi)
-1.0
```

Wszystko od znaku `#` do końca linii jest traktowane jako komentarz (czyli po prostu ignorowane).

W rzeczywistości, można bardzo łatwo tworzyć własne moduły. Jeśli zapiszesz kod Pythona w pliku z rozszerzeniem `.py`, np. `funkcje.py`, to wydanie w interpreterze komendy

```
>>> import funkcje
```

wystarczy, aby móc z niego korzystać (jeszcze do tego wrócimy).

Standardowa instalacja Pythona zawiera dziesiątki różnych modułów, zawierające od podprogramów audiomedialnych do procedur obsługi plików zip. Przykład poniżej ładuje z modułu `string*` funkcję do konwersji wielkości liter i pokazuje jej zastosowanie:

* Modułu `string` użyłem dla prostoty przykładu. W rzeczywistości, jest on przestarzały, bo wszystkie funkcje operujące na napisach są dostępne bezpośrednio w samych napisach (patrz rozdział 2.1).

```
>>> from string import capwords
>>> capwords('JaKIś TAKI napis')
'Jaki\266 Taki Napis'
```

Przykład ten pokazuje coś jeszcze: wartości będące łańcuchami znaków wypisywane są w apostrofach i ze znakami nie-ASCII zamienionymi na kody numeryczne.

1.2 Formatowanie wyników

Na wypisywaniu ostatnio obliczonej wartości przez Python nie można polegać: z jednej strony, nie zawsze wygląd będzie nam odpowiadał, z drugiej zaś, wartości te nie są wypisywane, jeśli uruchamia się program zapisany w pliku (tzn. nie jest wpisywany interakcyjnie). Dlatego do wyprowadzania wyników należy używać wbudowanej instrukcji `print`:

```
>>> print "Cześć"
Cześć
>>> a,b,c = 1,2,3
>>> print "Wynik:", a+5
Wynik: 6
>>> print 'Współczynniki: a=%d, b=%d, c=%d' % (a,b,c)
Współczynniki: a=1, b=2, c=3
```

Stop. Ten przykład jest dość skondensowany i ilustruje kilka rzeczy na raz:

- Można przypisywać wartości kilku zmiennym jednocześnie (linia 3);
- Napisy można wpisywać w apostrofach lub cudzysłowach. Jest to bardzo wygodne:

```
>>> print 'Przykład "a"'
Przykład "a"
>>> print "Przykład 'b' "
Przykład 'b'
```

- Do „wstawiania” wartości w napis można użyć pól formatowanych `'%d'` oraz operatora `%` (patrz dwie ostatnie linie).

Wyjaśnienia wymaga to ostatnie. Operator `%` jest operatorem dwuargumentowym, którego lewym argumentem jest ciąg znaków, a prawym dane do sformatowania i wstawienia w ten ciąg znaków. Wynikiem jest także ciąg znaków, w którym pola oznaczone znakami `%` w lewym argumencie zostały zastąpione odpowiednio sformatowanymi wartościami danych z prawego argumentu. Jeśli danych jest więcej niż jedna, trzeba je wziąć w nawias (aby były potraktowane jako jeden argument).

Oznaczenia pól są podobne jak w języku C*. Po znaku % może wystąpić liczba całkowita określająca docelową szerokość pola — dodatnia oznacza dosunięcie wartości do prawej strony pola, ujemna — do lewej. Jeśli wartością do wstawienia jest liczba rzeczywista, można oprócz szerokości pola podać (po kropce) liczbę miejsc po przecinku. Wzorzec pola jest zakończony literą określającą typ wpisywanej wartości. Podane w tabeli 1 przykłady powinny wyjaśnić resztę.

Tablica 1: Formatowanie danych za pomocą operatora %. W przykładzie szóstym od dołu, podana szerokość pola (2) jest za mała, dlatego zostaje zignorowana. W wynikowych łańcuchach znaków zaznaczono spacje.

Kod typu wartości	Przykład	Wynik
c (znakowa)	'%4c' % 65	' 65'
	'%-4c' % 66	'66 '
d, i (całkowita dziesiętna)	'%4d' % 65	' 65'
	'%04d' % 65	'0065'
	'%-4i' % 65	'65 '
e, E (rzeczywista wykładnicza)	'%e' % 250	'2.500000e+02'
	'%.3e' % 250	'2.500e+02'
	'%10.1E' % 250	' 2.5E+02'
f (rzeczywista zmiennoprzecinkowa)	'%f' % 250	'250.000000'
	'%10.3f' % 250	' 250.000'
	'%.2f' % 250	'250.00'
	'%-10.3f' % 250	'250.000 '
	'%-5.0f' % 3.14159	'3.14159'
g, G (jak f lub e)	'%g' % 250	'250'
	'%g' % 250.01	'250.01'
	'%g' % 2500000	'2.5e+06'
o (całkowita ósemkowa)	'%2o' % 250	'372'
	'%05o' % 250	'00372'
s (ciąg znaków)	'%5s' % 'bis'	' bis'
	'%-5s' % 'bis'	'bis '
x, X (całkowita szesnastkowa)	'%x' % 250	'fa'
	'%04X' % 250	'00FA'

1.3 Zestaw *survival* poskramiacza węży

Żeby się umieć odnaleźć w gąszczu modułów, podprogramów i innych obiektów języka Python, niezwykle pomocne są dwie rzeczy: wbudowana funkcja `dir()` oraz łańcuchy dokumentacji obiektów `__doc__`. Wynikiem funkcji `dir()` jest lista nazw atrybutów podanego obiektu (w Pythonie prawie wszystko jest obiektem):

* Różnice są jednak istotne. Python dokona konwersji typu wyprowadzanej wartości na odpowiedni dla typu pola, natomiast C nie.

```
>>> import math
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil',
'cos', 'cosh', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp',
'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'sin',
'sinh', 'sqrt', 'tan', 'tanh']
```

W powyższym przykładzie, uzyskaliśmy listę nazw wszystkich obiektów z modułu `math`. Większość z nich to nazwy funkcji matematycznych dostarczanych przez ten moduł. Dwa specjalne atrybuty: `__doc__` i `__name__` występują w prawie każdym obiekcie (oprócz liczb). Pierwszy z nich to skrócona dokumentacja obiektu, drugi zaś to po prostu jego nazwa:

```
>>> print math.__doc__
This module is always available. It provides access to the
mathematical functions defined by the C standard.
>>> dir(math.acos)
['__doc__', '__name__', '__self__']
>>> print math.acos.__doc__
acos(x)
Return the arc cosine of x.
```

`dir()` i `__doc__` zastępują mapę i kompas. Niestety, chociaż Python jest w stanie zapewnić, że każdy obiekt będzie miał atrybut `__doc__`, to nie napisze dokumentacji za programistę. Dlatego częste są takie obiekty, których `__doc__` nie ma żadnej wartości.

1.4 Własne programy i moduły

Stworzenie osobnego programu w Pythonie jest bardzo proste. Wystarczy kolejne instrukcje, tak jak podaje się je interpreterowi, zapisać w zwykłym pliku tekstowym. Standardowym rozszerzeniem dla takich plików jest `.py` — pod Unixem nie trzeba tego przestrzegać, ale pod Windows właśnie po rozszerzeniu rozpoznawana jest zawartość pliku. Oczywiście, można programy pisać zarówno w `vi` jak i w Notatniku, jednak pod obydwo systemami polecam SciTE (który w wersji dla Linuxa potrzebuje bibliotek GTK+, ale są one standardowo dostępne we wszystkich nowszych dystrybucjach). Dla dobrze znających edytor `vi`, prawdopodobnie najlepszą alternatywą będzie `vim` z ustawionym podświetlaniem składni dla Pythona. Jest jeszcze i Emacs, który podobno potrafi wszystko :-).

OK, przejdźmy do rzeczy. Utało się, że pierwszy program robi tylko tyle:

```
print 'Cześć świecie'
```

Ważniejsze jest to, jak go uruchomić. Powiedzmy, że zapisaliśmy go w pliku `hello.py`, wtedy uruchomienie programu wyglądałoby tak*:

* Oczywiście, niezależnie od metody uruchomienia, Python musi być wcześniej zainstalowany w systemie.

```
python hello.py
```

Komendę tę należałoby wpisać w okienku DOS pod Windows lub z poziomu shella w Unixie. Istnieje jednak wygodniejszy sposób. Pod Windows, należy skojarzyć typ pliku `.py` z interpreterem Pythona — przy pierwszej próbie otwarcia pliku z takim rozszerzeniem, pojawi się okienko z pytaniem „Jakiego programu użyć do otwarcia tego pliku”. Wybór interpretera Pythona domyślnie spowoduje powstanie odpowiedniego skojarzenia, tak że następnym razem podwójne kliknięcie od razu uruchomi program.

Pod Uniksem, robi się to trochę inaczej*. Po pierwsze, trzeba nadać plikowi prawa wykonywania:

```
chmod +x hello.py
```

Oprócz tego, w pierwszej linii pliku powinna znaleźć się informacja, gdzie szukać interpretera dla reszty tekstu w pliku (pełna ścieżka dostępu po znakach `#!`), np. taka:

```
#!/usr/local/bin/python
```

Tutaj pojawia się jednak drobny problem — różne dystrybucje Linuxa (i inne Unixy) mają różne poglądy na to, gdzie instalować opcjonalne oprogramowanie. Żeby zapewnić lepszą przenośność między różnymi maszynami, lepiej jest podać pierwszą linijkę programu w formie:

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

Zapewni to znalezienie interpretera gdziekolwiek, o ile tylko jest w jednym z katalogów w ścieżce (tzn. zmiennej środowiska `$PATH`).

Jeśli zamierzamy utworzyć nie program, ale moduł Pythona (czyli zestaw podprogramów), nie są potrzebne żadne dodatkowe operacje. Komenda `import hello` potraktuje pierwszy znaleziony plik `hello.py` jako moduł Pythona, tj. załaduje go do pamięci interpretera i wykona zawarte w nim instrukcje. Wystarczy, że plik ten znajdzie się w którymś z katalogów wskazywanych przez zmienną środowiska `$PYTHONPATH` lub w katalogu bieżącym.

Jak sprawić, aby Twoje moduły i podprogramy miały wewnętrzną dokumentację (`__doc__`), opisuj w sekcji 5.2 na stronie 23.

2 Dane w Pythonie. Wprowadzanie i operowanie danymi.

Trzy różne typy danych były wprowadzone przy okazji dotychczasowych przykładów, mianowicie liczby całkowite, rzeczywiste oraz łańcuchy znaków. Wprowadzanie typów danych się nie deklaruje, ale interpreter Pythona musi je rozróżniać — inaczej przecież dodaje się liczby, a inaczej łańcuchy

* Graficzne systemy obsługi użytkownika dla Linuxa, takie jak GNOME i KDE, także umożliwiają stworzenie skojarzeń typu pliku z odpowiednim programem i ikoną.

znaków. Rozpoznawanie typu danej jest dość proste i zależy od tego, jak jest zapisana. Ciąg tylko cyfr będzie potraktowany jako liczba całkowita. Może ona być poprzedzona znakiem (+/-), zerem (co oznacza notację ósemkową) lub sekwencją 0x (zero iks) oznaczającą notację szesnastkową:

```
>>> +17
17
>>> 017
15
>>> -0x17
-23
```

Ciąg cyfr zawierający kropkę dziesiętną lub/i wykładnik (po literze e lub E) jest rozpoznawany jako liczba rzeczywista (zawsze w notacji dziesiętnej). Natomiast łańcuchy znaków muszą wystąpić w apostrofach lub cudzysłowach (i po tym właśnie są rozpoznawane):

```
>>> 10.0100
10.01
>>> 1e1
10.0
>>> -1e10
-10000000000.0
>>> +314.15E-02
3.1415
>>> "Dana napisowa"
'Dana napisowa'
```

Istnieje jeszcze jeden sposób podawania danych typu łańcuchowego, mianowicie między potrójnymi cudzysłowami. Umożliwia on w prosty i wygodny sposób podawanie wielu linii tekstu:

```
>>> opis="""To jest przykład
... tekstu, który rozciąga się
... na kilka linii."""
>>> print opis
To jest przykład
tekstu, który rozciąga się
na kilka linii.
```

Znaki „... ” wypisywane są przez interpreter Pythona i sygnalizują, że oczekuje on na kontynuację poprzedniej instrukcji.

2.1 Dane są obiektami!

Wynalezienie pojęcia obiektu zupełnie zmieniło sposób myślenia programistów. Na czym zatem polega jego użyteczność?

Pierwszą zaletą programowania obiektowego jest związanie danych z podprogramami, które je przetwarzają. Wydaje się to mało znaczące, ale niezwykle uprościło pracę, szczególnie nad dużymi programami. Otóż starszej generacji języki tzw. strukturalne, pozwalając tworzyć biblioteki podprogramów i własnych typów danych, zaniedbywały problem wygodnego z nich korzystania i modyfikacji. Po pierwsze, trudno zapamiętać jakie funkcje dostarcza duża biblioteka, do czego każda z nich służy i jak jej używać. Po drugie, jeśli chcemy zmodyfikować jakiś złożony typ danych, na którym operują podprogramy z tej biblioteki, wymaga to poprawienia praktycznie wszystkich tych podprogramów.

W podejściu obiektowym, wszystkie związane ze sobą dane oraz podprogramy do operacji na nich (zwane *metodami*) są zebrane w jeden obiekt. Python idzie tu jeszcze krok dalej: każdy obiekt zawiera także swoją dokumentację*. Już samo to ułatwia życie: jeśli np. zastanawiasz się, jakie masz narzędzia do operowania napisami, znajdziesz je w każdym obiekcie typu łańcuchowego, i każde z nich powinno być samo-udokumentowane:

```
>>> dir('')
['capitalize', 'center', 'count', 'endswith', 'expandtabs', 'find',
'index', 'isdigit', 'islower', 'isspace', 'istitle', 'isupper',
'join', 'ljust', 'lower', 'lstrip', 'replace', 'rfind', 'rindex',
'rjust', 'rstrip', 'split', 'splitlines', 'startswith', 'strip',
'swapcase', 'title', 'translate', 'upper']
```

```
>>> print ''.lower.__doc__
S.lower() -> string
Return a copy of the string S converted to lowercase.
```

2.2 Struktury danych, czyli dlaczego wygodnie jest programować w Pythonie

Kompilowane języki programowania z reguły wymagają jawnej deklaracji wszystkich typów danych i zmiennych, z których będą korzystać. Ma to nieocenione zalety z punktu widzenia szybkości i efektywności gotowego programu, który nie musi zawierać kodu odpowiedzialnego za sprawdzanie typu, formatu czy poprawności danych ani doboru odpowiednich operacji. Ponadto, niezbędne struktury danych mogą zostać stworzone dokładnie na miarę zadania.

Zalety te są jednak kosztowne z punktu widzenia programisty, bowiem dużą część jego czasu pochłania zaprojektowanie, zaprogramowanie i przetestowanie odpowiednich struktur danych. Jest to koszt w wielu przypadkach zupełnie niepotrzebny, jako że zwykle ważniejsza jest efektywność pracy człowieka, a nie programu.

* Obiektami są także podprogramy i biblioteki podprogramów (moduły).

I tu właśnie otwiera się pole do popisu dla gotowych, uniwersalnych struktur danych dostarczanych przez języki interpretowane. W Pythonie, dostępne są cztery rodzaje takich struktur:

- Łańcuchy (ang. string). Podawane w **cudzysłowach** lub **apostrofach**. Zwykle nazywam je napisami, bo przechowywanie napisów to ich najczęstsze zastosowanie. Jednak nie jest to żadnym ograniczeniem — łańcuch jest po prostu dowolnej długości ciągiem dowolnych bajtów.
- Zlepki (ang. tuple). Zlepek jest to kilka danych (dowolnego typu) wymienionych kolejno w nawiasach **okragłych**. Zlepki w Pythonie stosuje się zwykle tam, gdzie składniowo musi znaleźć się jedna dana, a chcemy mieć łącznie kilka — pierwszym przypadkiem, kiedy się z tym spotkaliśmy, był prawy operand operatora % do formatowania napisów.
- Listy (ang. list). Listy to także połączenia (sekwencje) danych, które mogą być tego samego lub różnych typów. Podawane są w nawiasach **kwadratowych**.
- Słowniki (ang. dictionary). Jest to najbardziej zaawansowana struktura danych. Jej zaletą jest to, że każda dana pamiętana w słowniku ma własny unikalny klucz, będący jej identyfikatorem. Wprawdzie elementy zlepku lub listy są ponumerowane, ale słownik ma tę zaletę, że klucz może być dowolnego typu. W rzeczywistości klucz także jest daną — powoduje to, że słowniki są bardziej uniwersalne niż np. rekordy w Pascalu. Na przykład, słownik można wykorzystać do łączenia danych w pary. Słowniki z kolei rozpoznawane są po nawiasach **klamrowych**.

Każdy z tych typów nadaje się do innych celów, i każdy ma inny zestaw metod — warto porównać wyniki komend `dir('')`, `dir(())`, `dir([])` i `dir({})`. Jednak między łańcuchami, zlepkami i listami istnieje znaczące podobieństwo, wszystkie trzy są bowiem tzw. *sekwencjami*.

2.3 Sekwencje, indeksy i klucze

Sekwencja to typ danej, której elementy mają określoną kolejność i określone numery porządkowe (indeksy). Numery te zaczynają się zawsze od zera. Dla każdej sekwencji, można odwołać się do jej dowolnego elementu podając jego indeks (niezależnie od rodzaju sekwencji, indeksy zawsze umieszcza się w nawiasach kwadratowych):

```
>>> z = ('a', 12, 0.001) # Zlepek trzech danych
>>> l1 = []             # Pusta lista
>>> l2 = ['ala', 'ola'] # Lista dwóch napisów
>>> print z[0], z[2], l2[1]
a 0.001 ola
>>> nap = 'JAKIŚ NAPIS' # Łańcuch też jest sekwencją
>>> nap[2]
'K'
```

O indeksach sekwencji najwygodniej jest myśleć w ten sposób, jak gdyby numerowały one *prze-
gródki* między elementami, natomiast dotyczyły elementów na prawo od danej przegródki. W ten
sposób łatwo jest sobie wyobrazić, że indeks [0] oznacza pierwszy element sekwencji (na prawo
od zerowej przegródki). Można także liczyć od końca (jeśli podany indeks jest ujemny) — indeks
[-1] dotyczy zatem elementu ostatniego (tego za przegródką pierwszą od końca).

Indeksowanie sekwencji pozwala jednak na więcej: można podawać zakresy indeksów (z dwu-
kropkiem):

```
>>> napis = 'Ala ma kota'  
>>> lista = ['a', 'b', 1, 2]  
>>> print napis[0:3], napis[-4:]  
Ala kota  
>>> lista[2:]  
[1, 2]  
>>> lista[:-2]  
['a', 'b']
```

Zakresy leżą jakby *między* podanymi przegródkami, zatem [1:-1] oznacza pod-sekwencję ele-
mentów od drugiego do przedostatniego. Opuszczenie w zakresie lewego indeksu oznacza zakres
od początku sekwencji, opuszczenie prawego — do końca. Powszechny zatem w Pythonie idiom
[:] oznacza zakres od początku do końca; jest więc najprostszym sposobem na uzyskanie kopii
oryginalnej sekwencji (zakres jest zawsze sekwencją tego samego typu).

Do zakresów można nawet użyć instrukcji przypisania. Przypisanie takie powoduje wymienienie
całego podzakresu na nowy, przy czym nie musi się zgadzać ani typ, ani nawet liczba elementów:

```
>>> lista = [1, 2, 3, 4]  
>>> lista[1:3]  
[2, 3]  
>>> lista[1:3] = ['napis']  
>>> lista  
[1, 'napis', 4]
```

Warto jeszcze zauważyć, że używanie zakresów jest często bezpieczniejsze niż pojedynczych in-
deksów, ponieważ odwołanie się do nieistniejącego elementu sekwencji jest w Pythonie błędem.
Weźmy taki przykład:

```
>>> napis='TEST'  
>>> napis[5] # Nie ma piątego elementu!  
Traceback (innermost last):  
  File "<stdin>", line 1, in ?  
IndexError: string index out of range  
>>> napis[5:6] # To samo znaczenie co przedtem  
,,
```

Najbardziej użyteczną z sekwencji jest lista. A to z tego prostego powodu, że tylko elementy listy można dowolnie zmieniać, dodawać, usuwać, przestawiać itd*. W żadnej innej sekwencji nie można nawet podmienić pojedynczego elementu:

```
>>> napis='napis próbny'
>>> napis[0] = 'N'    # Nie da się!
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> zlepek = (1, 2, 3)
>>> zlepek[1] = 5    # To samo...
Traceback (innermost last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
>>> lista = [1, 2, 3]
>>> lista[1] = 'a'   # A tak można!
>>> lista
[1, 'a', 3]
```

Mówi się że zlepki i łańcuchy są typami *niemutowalnymi*, a listy (a także słowniki) są *mutowalne* (ang. odpowiednio mutable i immutable). Ta podstawowa różnica ma konsekwencje np. w działaniu metod specyficznych dla różnych typów. Wynikiem działania metod obiektów łańcuchowych jest zawsze zmieniona *kopia*, działają one zatem jako funkcje (zwracają nową wartość):

```
>>> n='ala'
>>> n.upper()
'ALA'
>>> n
'ala'
```

Metody modyfikujące listy natomiast działają na oryginale, tj. na obiekcie dla którego zostają wywołane:

```
>>> l = [2, 3, 1]
>>> l.sort()
>>> l
[1, 2, 3]
```

Podstawowa cecha sekwencji — kolejność elementów — umożliwia wprawdzie takie sztuczki jak operowanie zakresami, ale w niektórych przypadkach okazuje się niewygodna. Nie można mieć w Pythonie sekwencji o indeksach nie będących liczbami całkowitymi, nie można też w sekwencji zapamiętać danych „wrywkowo”, np. tylko pod numerami 20 i 100:

* Tak samo można dowolnie zmieniać elementy słowników, jednak słownik nie jest sekwencją.

```
>>> dane = []
>>> dane[20] = 'Dana dwudziesta'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list assignment index out of range
```

Pomocne w takich kłopotach są właśnie słowniki. Użycie w powyższym przykładzie słownika wyglądałoby niemalże identycznie:

```
>>> dane = {}
>>> dane[20] = 'Dana dwudziesta' # Działa!
>>> dane[100] = 'Dana setna'
>>> print dane
{100: 'Dana setna', 20: 'Dana dwudziesta'}
```

ale reprezentacja jest nieco inna: każdy element słownika jest parą `klucz:wartość`, i nie obowiązuje żadna kolejność elementów. Klucze słowników są odpowiednikami indeksów sekwencji, ale mogą być dowolnego typu, nawet w tym samym słowniku.

3 Instrukcje sterujące

Zgodnie z filozofią języka, aby nie wprowadzać nadmiaru instrukcji, Python oferuje tylko trzy instrukcje sterujące: warunkową `if` oraz pętlę iteracyjną `for` i warunkową `while`. Rozwiązanie takie, choć często krytykowane przez programistów wychowanych na innych językach (a co z `repeat until` albo z `case ?`), ma istotne zalety — czas nauki jest krótszy, a tłumaczenie algorytmów na Python w dużym stopniu jednoznaczne. W połączeniu z innymi cechami składni języka, zwłaszcza sposobem definiowania struktury kodu (bloków instrukcji — o czym za chwilę), powoduje to, że programów w Pythonie praktycznie nie da się pisać niezrozumiale.

3.1 Zrób to pod warunkiem, że...

Zacznijmy od instrukcji warunkowej `if`. Jej składnia wygląda tak:

```
>>> if warunek: instrukcja
```

Chyba trudno o prostszą postać. Zwróć uwagę na dwukropek — jest on niezbędnym elementem składni wszystkich instrukcji sterujących (i nie tylko). Mówiąc ogólniej, dwukropek oznacza, że dalej następuje *blok* instrukcji (zatem może być więcej niż jedna).

Na razie, zajmijmy się jednak warunkiem. Dotychczas nie wspominałem, że Python posiada jakiś typ logiczny — z tego prostego powodu, że takiego nie ma. Co ciekawsze, warunek może być

zupełnie dowolnego typu! Mimo to rozróżnienie, czy warunek jest spełniony (prawdziwy) czy też nie, jest raczej intuicyjne:

- Po pierwsze, dostępne są operatory porównania: równe (==), różne (!=), większe (>), mniejsze (<), większe lub równe (>=) itd. oraz logiczne and, or i not. Wynik ich działania jest oczywiście zgodny z oczekiwaniem. W rzeczywistości, dają one wartość całkowitą 0 (nieprawda) lub 1 (prawda). Jest także specyficzny dla Pythona operator in: warunek „dana in sekwencja” jest prawdziwy, jeśli dana jest elementem sekwencji.
- Jeśli warunek jest dowolnego typu liczbowego, wartość zerowa oznacza że nie jest spełniony (nieprawdę). Wartość niezerowa jest równoznaczna ze spełnieniem warunku.
- Jeśli warunek jest innego typu, wartość pusta jest fałszywa, a dowolna inna — prawdziwa. W szczególności, warunkiem niespełnionym jest pusty napis '', zlepek (), pusta lista [] lub pusty słownik {}. Natomiast — uwaga — napis zawierający choćby tylko spacje jest warunkiem prawdziwym.
- Istnieje jeszcze specjalna wartość None, nie mająca typu i oznaczająca brak wartości. Jest ona zawsze warunkiem fałszywym.

Jeśli jednak cała lista powyższych możliwości jest zbyt skomplikowana przy pierwszym podejściu, zawsze można sobie poradzić korzystając tylko z operatorów wymienionych w punkcie pierwszym. Zwracam jednak uwagę na różnicę między operatorem przypisania = (który nadaje wartość), a operatorem porównania == (który służy do *sprawdzenia* wartości).

A oto i kilka przykładów*:

```
>>> x,y = 1,1
>>> if x==1 and y==1: print "Obie jedyнки"
Obie jedyнки
>>> if 'a' in 'Ala': print "Jest 'a'!"
Jest 'a'!
>>> if x: x+y
2
>>> if []: print l[0] # Tym razem nic
>>> l = ['a']
>>> if l: print l[0]
a
```

Instrukcja if ma klauzulę „w przeciwnym razie”, czyli else (po którym też niezbędny jest dwukropek):

* Z przykładów usunięto, dla oszczędności miejsca, znaki ... wypisywane przez interpreter Pythona w oczekiwaniu na kontynuację instrukcji po if. W odpowiedzi należy wprowadzić pustą linię (Enter) — objaśnienie znajduje się dalej w tekście.

```
>>> x = 0
>>> if x: print "X niezerowe!"
... else: print "Zero!"
Zero!
```

Zaraz po `else` może nastąpić kolejna instrukcja `if`. Ponieważ w Pythonie właśnie w ten sposób buduje się instrukcje wielokrotnego wyboru (odpowiednik `case` w Pascalu), zdefiniowane jest dodatkowe słowo kluczowe `elif`, będące skrótem od `else if`:

```
>>> if x==0: print "Zero!"
... elif x==1: print "Jeden!"
... elif x==2: print "Dwa!"
... elif x==3: print "Trzy!"
... else: print "Poddaję się!"
```

Wreszcie, pozostało wyjaśnić jak pod kontrolą instrukcji `if` (lub klauzul `elif` lub `else`) umieścić więcej niż jedną instrukcję, czyli *blok*. Python umożliwia to w sposób bardzo prosty, jednocześnie wymuszając na programiście czytelność zapisu. Mianowicie, za blok instrukcji uważane są wszystkie linie o takim samym wcięciu (ang. `indent`), czyli odstępem od początku linii. Zatem w poniższym przykładzie:

```
>>> if x:
...     print 'X jest niezerowe'
...     print 'ale to nie szkodzi'
... 
```

obie instrukcje `print` zostaną wykonane tylko wtedy, gdy `x` będzie warunkiem spełnionym.

W przypadku pracy interakcyjnej, interpreter Pythona zawsze oczekuje na kolejną linię bloku wypisując wielokropki. W tej sytuacji, wpisanie linii o innym wcięciu niż reszta bloku jest traktowane jako błąd. Żeby kontynuować program (bez wcięcia), należy najpierw zakończyć blok przez wpisanie pustej linii (Enter). W przypadku pisania programu w pliku tekstowym, te dodatkowe puste linie nie są potrzebne. Zawsze jednak wcięcia w kodzie w języku Python są znaczące, zatem niepotrzebne spacje na początku linii (lub w wydawałoby się pustych liniach odstępów) są traktowane jako błędy.

3.2 W koło Macieju, czyli pętle

Po przećwiczeniu, przy okazji instrukcji `if`, meandrów bloków instrukcji i warunków logicznych Pythona, instrukcje pętli zostały na deser. Zaczniemy od `while`, od razu na konkretnym przykładzie:

```
>>> while lista:
...     print lista.pop()
... 
```

Analogicznie do składni `if`, tutaj także mamy dwukropek i po nim blok instrukcji z wcięciem. Instrukcje w bloku pętli `while` są wykonywane tak długo, jak długo warunek po słowie `while` jest spełniony. Tutaj warunkiem jest `lista` (nazwa słusznie sugeruje, że chodzi mi o zmienną typu `lista`), jest on zatem prawdziwy wtedy, gdy `lista` zawiera przynajmniej jeden element. Instrukcja `print lista.pop()` powoduje zdjęcie (usunięcie) ostatniego elementu z listy oraz jego wydruk. Całość zatem spowoduje skasowanie wszystkich elementów listy, z wypisaniem ich na ekranie (od ostatniego). Pętla zakończy się wtedy, gdy `lista` będzie pusta.

Podany przykład reprezentuje przypadek, gdy *najpierw* należy sprawdzić prawdziwość warunku, a dopiero *później* (ewentualnie) wykonać odpowiednie instrukcje (`lista` może być bowiem pusta wcześniej, więc nie byłoby co usuwać ani wypisywać). Jest to naturalne zastosowanie pętli `while` także w innych językach programowania. Jednak częstym przypadkiem jest konieczność *najpierw* wykonania instrukcji w pętli, a *potem* sprawdzenia warunku.

Weźmy taką sytuację. Użytkownik programu ma wprowadzić zestaw danych (np. wyniki pomiarów). Liczba danych nie jest wcześniej znana — to użytkownik daje sygnał, że zakończył ich podawanie (np. wpisując zero lub pustą linię). Zatem kod programu wczytujący dane musi być wykonywany w pętli warunkowej, ale warunek zakończenia też jest wczytywany w tej samej pętli, nie może być zatem sprawdzony przed jej wykonaniem*. W Pythonie, odpowiedni kod wyglądałby tak:

```
>>> pomiary = []
>>> while 1:          # Warunek zawsze prawdziwy
...     dana = raw_input('Podaj daną:')
...     if not dana: break # Przerwanie pętli
...     pomiary.append(float(dana))
... 
```

Jak widać, w takich razach korzysta się z konstrukcji `while 1:` i przerywa pętlę instrukcją `break`. Rozwiązanie takie jest bardzo elastyczne, umożliwia bowiem przerwanie pętli w dowolnym miejscu (tutaj: przed zapamiętaniem zbędnej danej oznaczającej koniec wprowadzania) lub pod różnymi warunkami. Jest jeszcze instrukcja `continue`, powodująca przejście do następnego obrotu pętli (z opuszczeniem ewentualnych instrukcji poniżej).

Pozostała do omówienia pętla iteracyjna `for` działa nieco inaczej, niż uczą przyzwyczajenia wyniesione z innych języków. Jest także istotna różnica między znaczeniem operatora `in` w kontekście pętli `for` i w kontekście warunków logicznych w `if` i `while`. Mianowicie, `in` jest integralnym elementem składni pętli iteracyjnej, i oznacza jej wykonanie *dla każdego elementu podanej sekwencji*:

* w Pascalu, służy do tego pętla `repeat ... until`.

```
>>> for i in [1, 'a', 'nic']:
...     print i
...
1
a
nic
```

Jak widać, zmienna sterująca pętli `for` (tutaj: `i`) przyjmuje wartości kolejnych elementów podanej sekwencji. `for` wymaga sekwencji; można jej podać napis (wówczas wartościami zmiennej sterującej będą kolejne znaki), zlepek lub listę. Podanej listy nie można jednak modyfikować w pętli, prowadzi to do nieokreślonego zachowania. Proszę dla sprawdzenia wypróbować taką instrukcję:

```
>>> for i in lista: lista.remove(i) # Kłopoty
```

Rozwiązaniem jest w takich razach uruchomienie pętli na *kopii* listy, którą łatwo uzyskać używając notacji zakresów:

```
>>> for i in lista[:]: lista.remove(i) # OK
```

Aby umożliwić wykonywanie pętli iteracyjnej dla kolejnych liczb, jak w większości innych języków, Python dostarcza funkcję standardową `range()`. Służy ona do wygenerowania sekwencji liczb całkowitych. Można ją użyć na kilka sposobów:

```
>>> range(5)          # Pięć kolejnych liczb
[0, 1, 2, 3, 4]
>>> range(5,10)      # Liczby od 5 do 9
[5, 6, 7, 8, 9]
>>> range(5,10,2)    # Liczby od 5 do 9 co 2
[5, 7, 9]
```

`range(n)` liczy od 0 do $n-1$, i jest to dokładnie zakres indeksów sekwencji o n elementach. W ogólności, znaczenie parametrów funkcji `range()` jest podobne jak dla indeksów sekwencji — dlatego zakres od 5 do 10 oznacza sekwencję liczb 5–9 (przywołując przykład „przegródek”, są to liczby między piątą a dziesiątą „przegródką” na liście kolejnych liczb całkowitych od zera wzwyż).

Sprowadzając znów rzecz do konkretnego przykładu, poniżej ilustruję użycie pętli `for` do wypisania elementów pewnej listy z kolejnymi numerami (standardowa funkcja `len()` oblicza długość, tj. liczbę elementów dowolnej sekwencji):

```
>>> for nr in range(len(lista)):
...     print '%3d.' % (nr+1), lista[nr]
... 
```

I jeszcze jeden przykład, tym razem użycia pętli `for` do wypisania elementów słownika:

```
>>> for klucz in slownik.keys()
...     print klucz, ':', slownik[klucz]
... 
```

4 Wprowadzanie i wyprowadzanie danych. Praca z plikami.

4.1 Pytania do użytkownika. Funkcja `eval()`.

Jeśli chodzi o instrukcje służące do zapytania o dane użytkownika, zetknęliśmy się już z nimi we wcześniejszych przykładach. Są to mianowicie funkcje `input()` i `raw_input()`. W rzeczywistości, podstawową jest ta druga. Obie przyjmują opcjonalny parametr łańcuchowy, który ma służyć do poinformowania użytkownika, o jaką daną prosimy:

```
>>> nazw = raw_input('Podaj nazwisko: ')
Podaj nazwisko: _
```

W powyższym przykładzie, kreseczka w drugiej linii ma symbolizować kursor, bowiem w tym momencie program zatrzymuje się, czekając na wprowadzenie odpowiedzi przez człowieka (pod Windows, wyskoczyłoby okienko dialogowe z zapytaniem).

Wartością funkcji `raw_input()` jest zawsze dana typu łańcuchowego (napis). Jeśli pytamy o dane do obliczeń, należałoby zatem w programie dokonać samodzielnie konwersji do odpowiedniego typu, np. funkcją `float()`. Dla wygody, dostępna jest „inteligentniejsza” funkcja `input()`, równoznaczna z konstrukcją `eval(raw_input())`.

Rzeczona „inteligencja” siedzi właśnie w funkcji `eval()`. Jej działanie jest prawie dokładnie takie, jak interpretera języka Python. Jeśli podamy jej napis będący prawidłowym wyrażeniem w języku Python, to zostanie ono zinterpretowane, obliczone i jego wynik będzie wartością funkcji. Ma to swoje zalety i wady. Do zalet zalicza się możliwości podawania wartości od razu całych struktur danych, korzystania ze zmiennych zdefiniowanych w programie, ba nawet z funkcji:

```
>>> def suma(a,b):
...     return a+b
...
>>> lista = ['aaa', 'xxx']
>>> dana = input('Test: ')
Test: { lista[1]: suma(3,3)*3.14 } # To wpisał użytkownik
>>> print dana
{'xxx': 18.84}
```

Wadą jest, że użytkownik musi stosować się do reguł składni Pythona. Przykładowo, napisy trzeba podawać w apostrofach żeby zaznaczyć, że chodzi o typ napisowy. Nawet tak trywialna odpowiedź użytkownika, jak pusty łańcuch (tylko klepnięcie Enter), spowodowałaby spektakularną śmierć programu, z mało zrozumiałym dla laika komunikatem o błędzie składniowym:

```
>>> x = input('Podaj daną: ')
Podaj daną:
Traceback (innermost last):
  File "<stdin>", line 1, in ?
  File "<string>", line 0

^
SyntaxError: unexpected EOF while parsing
```

4.2 Jak korzystać z plików?

Operacje na plikach są w ogólnym zarysie podobne, niezależne od języka programowania, ponieważ są one wszystkim programom udostępniane w ten sam sposób przez system operacyjny. Plik (zwykle) jest w programie reprezentowany przez jakąś zmienną, którą należy *skojarzyć* z konkretnym plikiem (przez podanie jego nazwy i ewentualnie ścieżki dostępu). Aby korzystać z danych w pliku, lub aby do niego coś zapisać, należy najpierw plik *otworzyć*, a po użyciu należy go *zamknąć*. Zamknięcie pliku często jest opuszczane, ponieważ i tak robi to w końcu system operacyjny. Nie powinno się to jednak stać nawykiem, ponieważ może być przyczyną kłopotów*.

Python do tego schematu dodaje swoje trzy grosze, w znacznym stopniu upraszczając powyższą teorię. Odczyt pliku w Pythonie może bowiem wyglądać tak:

```
>>> dane = open('dane.dat').readlines()
```

Mimo że jest to tylko jedna linijka kodu, zawierają się w niej wszystkie elementarne operacje plikowe. Funkcja `open()` wykonuje dwie z nich, mianowicie skojarzenie z plikiem o nazwie `dane.dat` w bieżącym katalogu oraz otwarcie (standardowo do odczytu). Wartością tej funkcji jest *instancja*[†] obiektu plikowego — byłaby ona wartością zmiennej reprezentującej plik, gdybyśmy taką zmienną tutaj utworzyli. Jednak zamiast tworzyć zmienną plikową, od razu wywołujemy dla naszego obiektu metodę `readlines()` i to jej wynik zapamiętujemy w zmiennej `dane()` — będzie to lista wszystkich linii z pliku. Ponieważ sam obiekt plikowy nie zostaje zapamiętany, natychmiast po wykonaniu powyższej instrukcji Python go niszczy, przedtem wykonując operację zamknięcia.

* 1. System operacyjny nakłada ograniczenie na liczbę jednocześnie otwartych plików;
2. Nie można być pewnym, że wszystkie dane zostały zapisane, dopóki plik nie zostanie zamknięty, lub dopóki nie wykona się instrukcji `flush()`.

† Czyli konkretna wartość obiektu.

Jak widać, dość dużo dzieje się „za kurtyną”. Nie zawsze jednak wystarczają rozwiązania najprostsze; choćby dlatego że nie wszystkie pliki składają się z linii (tekstu).

Podsumowując podany przykład:

- Dla ułatwienia (programiście) życia, skojarzenie z plikiem oraz jego otwarcie wykonuje się jedną instrukcją `open()`.
- Zmienne plikowe są obiektami. W rzeczywistości, nie trzeba w ogóle tworzyć zmiennej do obsługi pliku — wystarczy *instancja* obiektu plikowego.
- Ponieważ Python z natury sprząta nieużytki, zamknięcie pliku oraz usunięcie niepotrzebnej instancji obiektu plikowego dokonywane jest automatycznie — w momencie, gdy przestanie być używana (np. nie jest już pamiętana w żadnej zmiennej).

Funkcja `open()` standardowo otwiera podany plik tylko do odczytu. Jeśli zamierzamy do pliku zapisywać, należy jako drugi parametr podać tryb otwarcia: `'w'` oznacza otwarcie do zapisu (powoduje to skasowanie poprzednio zapisanych danych!), natomiast `'a'` jest oznaczeniem trybu dopisywania (ang. *append*). Wartość `'r'` (jak *read*) jest wartością domyślną drugiego parametru funkcji `open()` (jeśli nie zostanie podany).

Do zapisu danych do pliku dostępne są dwie metody: `write()`, przyjmująca parametr typu łańcuchowego, oraz `writelines()`, wymagająca listy napisów. Żadna z nich nie dodaje automatycznie znaków końca linii, dlatego zadbać o to musi programista (znak końca linii podaje się jako `'\n'`):

```
>>> plik = open('wyniki.txt', 'a')           # Otwarcie do dopisywania
>>> plik.write('%10.5f%10.5f\n' % (x,y))    # Zapis dwóch liczb float
>>> linie = ['Linia 1\n', 'Linia 2\n']
>>> plik.writelines(linie)                 # Zapis kilku linii
>>> plik.close()                           # Zamknięcie pliku
```

4.3 Odczyt lub zapis pliku w pętli

W większości przypadków, gdy będziemy operować na niedużych plikach tekstowych, wystarczą nam metody `readlines()` oraz `writelines()`, zaprezentowane powyżej. Tutaj jednak chciałbym pokazać jak sobie radzić, gdy zapis lub odczyt wykonujemy porcjami, np. w pętli.

Przypuśćmy, że mamy za zadanie wypisać z pliku `obliczenia.txt` linie zawierające napis „WYNIK:”. Najprościej, można poradzić sobie tak:

```
>>> for linia in open('obliczenia.txt').readlines():
...     if linia.find('WYNIK:') >= 0: print linia
```

i wszystko będzie dobrze... przynajmniej do momentu w którym okaże się, że dwugigabajtowy plik `obliczenia.txt` nijak nie chce się zmieścić w 64 MB RAM. Znacznie lepiej byłoby czytywać po jednej linii:

```
>>> plik = open('obliczenia.txt')
>>> while 1:
...     linia = plik.readline()
...     if linia == '': break
...     if linia.find('WYNIK:') >= 0: print linia
>>> plik.close()
```

Korzystam tutaj z faktu, że po odczytaniu ostatniej linii, funkcja `readline()` zwróci pusty łańcuch. Tym razem, obiekt plikowy nie zostanie automatycznie zamknięty, dopóki jest wartością zmiennej `plik`. W dobrym stylu jest zatem jawne wywołanie metody `close()`.

Potrzeba wykonywania w pętli operacji zapisu do pliku często jest narzucona przez funkcję programu, wykonującego np. cykliczne obliczenia lub pomiary. Korzystając z `writelines()`, trzeba wyniki gromadzić na liście, a dopiero później zapisywać. Pomijając niepotrzebne marnotrawstwo pamięci, rozwiązanie takie ma tę oczywistą wadę, że niemożliwa jest kontrola wyników pośrednich w pliku (przez inne programy, człowieka itp.) — nic nie zostanie zapisane, aż do zakończenia roboczej pętli. Lepsze rozwiązanie problemu mogłoby wyglądać tak:

```
>>> plik = open('wyniki.txt', 'w')
>>> while not koniec_pomiarow():
...     plik.write( wynik_pomiaru() )
>>> plik.close()
```

Jest to najbardziej wydajny sposób, jednak np. nie gwarantuje *natychmiastowej* dostępności zapisanych danych na dysku — jak wyjaśniałem w opisie do metody `close()`, nic nie zostaje tak naprawdę zapisane, aż nie uzbiera się większa porcja (blok o rozmiarze zwykle 512 lub 1024 bajtów). Jeśli zależy nam na natychmiastowym zapisie, sprawę ułatwia dodanie w pętli dodatkowej instrukcji `plik.flush()`. Można jednak skorzystać z tymczasowej instancji obiektu plikowego otwieranego do dopisywania, a nasz przykład da się uprościć:

```
>>> while not koniec_pomiarow():
...     open('wyniki.txt', 'a').write(wynik_pomiaru())
```

Tutaj znów kupujemy wygodę kosztem efektywności: narzutem jest otwieranie i (automatyczne) zamykanie pliku w każdym obrocie pętli, ale o ileż mniej pisania, i okazji do zrobienia błędu.

5 Własne procedury i funkcje

Do definiowania podprogramów służy w Pythonie słowo kluczowe `def`, po którym musi wystąpić nagłówek podprogramu i dwukropek, a w kolejnych liniach blok instrukcji (linie o tym samym wcięciu) składających się na podprogram. W użyciu jest to znacznie prostsze niż tłumaczenie:

```
>>> def suma(a,b):
...     return a+b
...
>>> print suma(1,2)
3
```

Oto i mamy funkcję, która oblicza i zwraca sumę swoich parametrów. Instrukcja `return` powoduje zakończenie podprogramu i powoduje, że podana wartość będzie wartością tego podprogramu. Nie oznacza to wcale, że obecność `return` jest w podprogramie obowiązkowa:

```
>>> def hello(): # Nawiasy konieczne nawet bez parametrów
...     print 'Cześć, świecie!'
...
>>> hello()      # W wywołaniu również
Cześć, świecie!
```

W tym przykładzie nie ma `return`, zatem podprogram nie zwraca żadnej wartości. Można powiedzieć, że tym razem jest to procedura; jednak Python tak naprawdę nie rozróżnia między procedurami i funkcjami. Można np. napisać taką hybrydę:

```
>>> def dziwo(x):
...     if x>0 : return x
...     else: print 'Cześć'
```

która będzie się zachowywać jak funkcja dla dodatnich wartości `x`, a jak procedura dla ujemnych. Dlatego w tekście tym używam nazw funkcja, procedura i podprogram zamiennie; najczęściej jednak będę pisał „funkcja”, bo tak jest najkrócej*.

Instrukcji `return` można także użyć bez podania wartości — efektem jest wtedy natychmiastowe zakończenie podprogramu w miejscu jej użycia.

5.1 Funkcja jako wartość

Warto może zaznaczyć, że nazwa funkcji w Pythonie zachowuje się jak zwykła zmienna; tak też zostanie potraktowana, jeśli opuścimy nawiasy — zamiast uruchomić, Python poda jej wartość:

```
>>> suma
<function suma at 80d5380>
>>> suma, 1, 2
(<function suma at 80d5380>, 1, 2)
```

* Także w wielu innych językach, np. C, C++, podprogramy nazywa się funkcjami.

Analogia ta jest wcale nieprzypadkowa: identyfikator podprogramu *jest* zmienną i można go właśnie tak traktować. M.in. jej wartość, czyli samą funkcję, można przypisać innej zmiennej:

```
>>> suma = hello
>>> print suma
<function hello at 80d5408>
>>> suma()
Cześć świecie!
```

Cecha ta daje Pythonowi dużą elastyczność: np. nic nie stoi na przeszkodzie, żeby podać funkcję jako parametr innej funkcji:

```
>>> def uruchom(a):
...     a()
...
>>> uruchom(hello)
Cześć świecie!
```

5.2 Wewnętrzna dokumentacja — atrybut `__doc__`

Dobłą praktyką jest opisywanie własnych programów — przynajmniej, jeśli zamierza się ich użyć więcej niż raz. Oczywiście, robi się to przez komentowanie kodu, ale naprawdę warto skorzystać z mechanizmów języka, i zapewnić przyzwoity opis tworzonych podprogramów, obiektów, modułów w ich atrybutach `__doc__`. Zwłaszcza, że robi się to bardzo łatwo — jeśli pierwszym elementem bloku kodu obiektu jest łańcuch znaków, stanie się on wartością jego atrybutu `__doc__`:

```
>>> def p():
...     'Procedura p() wypisuje "hello"'
...     print 'hello'
...
>>> print p.__doc__
Procedura p() wypisuje "hello"
```

Oczywiście, dokumentacja może być dłuższa niż jedna linia — korzysta się wtedy z potrójnych cudzysłówów:

```
>>> def fun(x):
...     """Funkcja fun(liczba) -> liczba
...     Oblicza kwadrat podanej liczby.
...     """
...     return x*x
```

```
...
>>> print fun.__doc__
Funkcja fun(liczba) -> liczba
    Oblicza kwadrat podanej liczby.
```

W ten sam sposób dokumentuje się dowolne obiekty. Przykładowo, jeśli w pliku z kodem w Pythonie umieścisz na początku opis w postaci łańcucha znaków, po zaimportowaniu takiego pliku jako modułu, opis ten znajdzie się w jego atrybucie `__doc__`.

6 Moduł Numeric: obliczenia na tablicach liczb

7 Moduł Gnuplot: wykresy

Moduł Gnuplot służy do komunikacji z zewnętrznym programem o tej samej nazwie, umożliwiając rysowanie jedno- i dwuwymiarowych wykresów z poziomu programu w Pythonie. Oczywiście, oprócz modułu Gnuplot trzeba mieć również zainstalowany sam program gnuplot. Dobrą nowiną jest, że obydwie są dostępne dla systemów Unix oraz Windows, a nawet dla Maków. Wprawdzie wersja Windows ma pewne ograniczenia (podstawowe jest takie, że nie można jednocześnie pracować na dwóch oddzielnych wykresach), ale nie są one zbyt uciążliwe.

7.1 Najprostsze użycie i sterowanie programem gnuplot

Podstawowym elementem modułu Gnuplot jest klasa Gnuplot reprezentująca połączenie z programem odpowiedzialnym za rysowanie wykresu. Z punktu widzenia funkcjonalności, możemy myśleć o każdym obiekcie tej klasy jako o osobnym wykresie:

```
>>> import Gnuplot
>>> wyk = Gnuplot.Gnuplot()
```

Zwracam uwagę na nawiasy: powodują one *utworzenie* obiektu klasy Gnuplot.Gnuplot, a więc uruchomienie zewnętrznego programu i nawiązanie z nim połączenia. Instrukcja bez nawiasów, tj. `wykr=Gnuplot.Gnuplot` też jest bowiem składniowo poprawna, ale oznacza że zmienna `wykr` ma reprezentować *klasę* Gnuplot.Gnuplot, a nie obiekt. Próba skorzystania z takiej zmiennej skończyłaby się masą dziwnych błędów.

Obiektowi Gnuplot można podawać wszystkie komendy, jakie zrozumie program gnuplot, ponieważ będą one po prostu „przesłane” do programu:

```
>>> wyk('set data style linespoints')
>>> wyk('plot sin(x) title "Sinus"')
>>> wyk('replot "wyniki.dat" using 2:4 title "Wyniki"')
```

Jest to możliwość wystarczająca, jeśli ktoś zna program gnuplot bardzo dobrze, i funkcjonalność zapewniana przez moduł Gnuplot mu nie wystarcza (lub/i nie chce się go uczyć :-). Tylko taki sposób byłby jednak często niezbyt wygodny, bo tabelkę wyników do wykresu trzeba by samemu zapisać do pliku (ale niektóre opcje da się ustawić tylko tak).

Korzystanie z zestawu metod dostępnych dla obiektu Gnuplot pozwala większość zadań zrealizować prościej i szybciej, np.:

```
>>> wykrm.title('Tytuł wykresu')
>>> wykrm.xlabel('Oś X') # Podpis osi X
>>> wykrm.ylabel('Oś Y') # -- " -- Y
>>> wykrm.plot('sin(x)', [(x1,y1), (x2,y2), (x3,y3), ...],
...           wyniki) # Uniwersalna procedura do wykresów
>>> wykrm.replot() # Przerysowuje wykres (może dodawać nowe dane)
>>> wykrm.hardcopy() # Wysyła wersję PostScript wykresu na drukarkę
>>> wykrm.hardcopy('wydruk.ps') # Zapis wykresu do pliku
>>> wykrm.clear() # Kasuje listę linii na wykresie (następna
# komenda zadziała na czystym ukł. współrz.)
>>> wykrm.reset() # Przywraca standardowe opcje + wykrm.clear()
>>> wykrm.splot('sin(x*y)') # Procedura do wykresów dwuwymiarowych
```

Po szczegóły oraz opis reszty metod odsyłam do `dir(Gnuplot.Gnuplot)` i wewnętrznej dokumentacji (`__doc__`).

Z podanych przykładów, najważniejsze są metody `plot()` oraz `splot()`. Obie przyjmują dowolną liczbę parametrów, z których każdy definiuje jakiś wykres. Funkcję można podać w postaci napisowej; tabelkę w postaci sekwencji (zlepku, listy) par (x_i, y_i) lub trójek (x_i, y_i, z_i) , a nawet w postaci dwuwymiarowej tablicy (array) Numerical Pythona*. Dzięki tej dużej dozie „wbudowanej inteligencji”, zrobienie nawet dość złożonego wykresu może sprowadzić się do jednej krótkiej instrukcji. Jako ćwiczenie, proponuję wypróbować poniższy przykład:

```
>>> from Numeric import *
>>> import Gnuplot
>>> wykrm=Gnuplot.Gnuplot()
# Obliczenie przykładowej tabelki danych:
>>> dane=zeros((2,50),Float)
>>> dane[0] = arange(0,8*pi, (8./50.)*pi)
>>> dane[1] = sin(dane[0]) * dane[0]
>>> dane=transpose(dane)
# Wykres (a właściwie trzy)
>>> wykrm.plot('x', '-x', dane)
```

* W rzeczywistości, dane tablicowe zapisywane są do plików tymczasowych i w tej postaci dostarczane programowi gnuplot.

Warto także przestudiować programy `demo.py` i `test.py` z pakietu Gnuplot. Komentarze w `demo.py` są całkiem bogate, i chyba jaśniejsze niż dostępna z pakietem dokumentacja.

7.2 Więcej o operowaniu wykresami

Możliwość uzyskania wykresu jedną prostą instrukcją jest wygodna przy testowaniu programu lub wykonywaniu podręcznych obliczeń. Jeśli zależy nam na jakości i przejrzystości prezentowanych wyników, konieczna jest możliwość przyzwoitego opisu każdej krzywej, doboru rodzaju linii, punktów, kolorów lub stylu wykresu (np. histogramu). Osiąga się to przez stworzenie osobnych obiektów reprezentujących każdą serię danych, i przez nadanie tym obiektom odpowiednich atrybutów. Wspólną klasą takich obiektów, także definiowaną przez moduł Gnuplot, jest typ

7.3 No dobrze, ale jak przenieść wykres do dokumentu albo wydrukować?

W systemie Unix, najbardziej rozpowszechnionym (i wspieranym) formatem rysunków jest PostScript. Nieprzypadkowo, najwyższą jakość i możliwości opisu wykresów z gnuplota osiąga się właśnie w tym formacie, i dlatego metoda `hardcopy()` pakietu Gnuplot też go używa:

```
>>> wykrcopy('wyniki.ps')
```

Uzyskany plik można albo wydrukować (najczęściej przez GhostScript), albo wkleić w dokument LaTeX-a (np. używając LyX-a). Problemem są niestety polskie litery, jako że gnuplot nie wspiera znaków narodowych. Przychodzą mi do głowy tylko trzy rozwiązania:

- Nie używać polskich liter (najprostszy),
- Używać LaTeX-a i terminala `latex`; usunąć z wynikowego pliku komendy zmieniające font na `cm` (Computer Modern).
- Poprawić program gnuplot (najlepszy :-).

Metoda `hardcopy()` wywołana bez parametru wysyła wydruk, także w formacie PostScript, na standardową drukarkę. Pod Unixem, wymaga to prawidłowo skonfigurowanej komendy `lpr`. Nie wiem co się dzieje pod Windows, ale prawdopodobnie można wysłać wydruk bezpośrednio na drukarkę sekwencją komend (można zdefiniować własną procedurę `print()`):

```
>>> wykrcopy('set term pcl5')
>>> wykrcopy('set output "PRN"')
>>> wykrcopy('replot')
>>> wykrcopy('set term windows')
>>> wykrcopy('set output')
```

Tutaj używam terminala `pcl5`, który powinien działać na większości drukarek HP i podobnych. Ostatnie dwie instrukcje przywracają rysowanie w okienku Windows.

Pracując pod Windows, też warto zainstalować GhostScript. Jednak w tym systemie niewielu używa LaTeX-a (choć takowe wersje istnieją, zarówno darmowe, jak i za grube pieniądze), dlatego warto wiedzieć jak wykres z gnuplota przenieść — najlepiej w postaci wektorowej — do innego programu, np. Worda. W windowsowej wersji gnuplota działa wprawdzie kopiowanie wykresu przez Schowek, ale pozostawia (moim zdaniem) sporo do życzenia. Dlatego proponuję poeksperymentować samodzielnie z różnymi terminalami. Do wektorowych należą `cgm` (Computer Graphics Metafile), `corel` (dla CorelDraw), `dxg` (format AutoCAD-a), `hpgl` i `pcl5` (dla ploterów i drukarek Hewlett-Packard). Formaty `cgm` i `hpgl` powinny być czytane przez MS Worda, nie wiem jak pozostałe. Odpowiednie komendy gnuplota brzmią (przykładowo):

```
gnuplot> set terminal cgm
gnuplot> set output 'wyniki.cgm'
gnuplot> replot
```

Oczywiście, można to samo zrobić z poziomu Pythona. Więcej o terminalach i ich możliwościach można dowiedzieć się używając wbudowanego systemu pomocy programu Gnuplot:

```
gnuplot> help set term
```