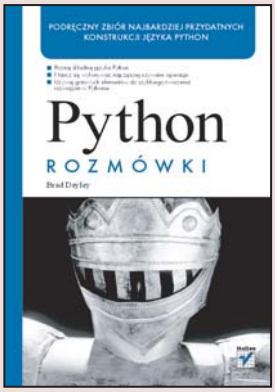
# Python. Rozmówki




Autor: Brad Dayley
Tłumaczenie: Anna Trojan
ISBN: 978-83-246-0950-5
Tytuł oryginału: Python Phrasebook
Format: B6, stron: 296

**Podręczny zbiór najbardziej przydatnych konstrukcji języka Python**

- Poznaj składnię języka Pyton
- Naucz się wykonywać najczęściej używane operacje
- Używaj gotowych elementów do szybkiego tworzenia rozwiązań w Pythonie

Prawdopodobnie słyszałeś już o zaletach Pythona. Jest przenośny i działa w niemal wszystkich systemach operacyjnych. Ma niezwykle czytelną i prostą składnię, a jego odmiany mogą współpracować z innymi językami programowania. Mówi się także, że pozwala skrócić czas pisania kodu kilka razy w porównaniu z językiem C++. To jeszcze nie wszystkie atuty Pythona, o czym wkrótce się przekonasz, pisząc swoje pierwsze programy w tym języku.

Dzięki książce „Python. Rozmówki" błyskawicznie poznasz najważniejsze zwroty i konstrukcje oraz podstawy składni tego języka programowania. Nauczysz się między innymi wykonywać w aplikacjach operacje na łańcuchach i pracować z typami danych, a także pisać programy wielowątkowe i sieciowe. Dowiesz się, jak zarządzać plikami i przetwarzać je oraz jak obsługiwać bazy danych.

- Składnia języka Python
- Przetwarzanie łańcuchów danych
- Korzystanie z typów danych
- Praca z plikami
- Tworzenie aplikacji wielowątkowych
- Komunikacja z bazami danych
- Obsługa komunikacji sieciowej
- Tworzenie usług sieciowych
- Przetwarzanie danych w formatach XML i HTML

**Wykorzystaj gotowe fragmenty kodu – zacznij pisać niezawodne programy w Pythonie**

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

# Spis treści

# 1

# Podstawy Pythona

Python jest zorientowanym obiektowo językiem programowania o wyjątkowych możliwościach i elastyczności. Widać w nim podobieństwa do języków skryptowych takich jak Perl, Scheme i TCL oraz do innych języków programowania, takich jak Java oraz C.

Niniejszy rozdział ma na celu ramowe przedstawienie tego języka programowania, co powinno pomóc w zrozumieniu kolejnych rozdziałów książki. Nie będzie on jednak wyczerpujący; powinien dać ogólny obraz języka oraz pomóc zrozumieć jego podstawy, tak by w celu uzyskania dokładniejszych informacji wystarczyło sięgnięcie do dokumentacji Pythona.

# Przyczyny popularności Pythona

Istnieje kilka przyczyn popularności Pythona. Jest on jednym z łatwiejszych języków, dzięki którym można rozpocząć swoją przygodę z programowaniem, a mimo to posiada ogromne możliwości, nadające się do zastosowania w większych aplikacjach. Poniższe punkty opisują tylko niektóre przydatne cechy Pythona:

- **Przenośność**: Python działa na prawie każdym systemie operacyjnym, w tym na Linuksie/Uniksie, Windows, Mac, OS/2 i innych.
- **Integracja**: Python może zostać zintegrowany z obiektami COM, .NET oraz CORBA. Istnieje implementacja Jython, która pozwala na używanie Pythona na dowolnej platformie Javy. IronPython jest implementacją, która umożliwia programistom Pythona dostęp do bibliotek .NET. Python może także zawierać opakowany kod w językach C czy C++.
- **Prostota**: Łatwo jest rozpocząć pisanie programów w Pythonie. Jasna, czytelna składnia sprawia, że tworzenie aplikacji i usuwanie z nich błędów jest naprawdę proste.
- **Możliwości**: Cały czas powstają rozszerzenia do Pythona, które służą funkcjom takim jak dostęp do bazy danych, edycja materiałów audio i video, GUI, tworzenie stron internetowych i tak dalej.

- **Elastyczność**: Python jest jednym z najbardziej elastycznych języków. Łatwo jest zacząć szybko tworzyć kod, który będzie rozwiązywał problemy związane z projektowaniem i programowaniem.
- **Open-source**: Python jest językiem o otwartym kodzie źródłowym, co oznacza, że można go swobodnie i za darmo używać i dystrybuować.

# Wywoływanie interpretera

Skrypty w Pythonie są wykonywane przez interpreter Pythona. W większości systemów operacyjnych można uruchomić interpreter Pythona, wykonując polecenie `python` w wierszu poleceń. Może to jednak wyglądać inaczej w zależności od systemu operacyjnego i środowiska, w którym się pracuje. Niniejszy podrozdział omawia standardowe metody wywoływania interpretera, by wykonywał on instrukcje Pythona oraz pliki ze skryptami.

Wywoływanie interpretera bez przekazania mu pliku ze skryptem w charakterze parametru powoduje wyświetlenie następujących wierszy:

```
bwd-linux:/book # python
Python 2.4.2 (#1, Apr  9 2006, 19:25:19)
[GCC 4.1.0 (SUSE Linux)] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

Wiersz poleceń Pythona wyświetla >>>. Jeśli wykona się polecenie, które wymaga więcej danych wejściowych, zostanie wyświetlony znak zachęty .... Z wiersza poleceń interpretera można wykonywać pojedyncze instrukcje Pythona, jak poniżej:

```
>>> print "Drukowanie łańcucha znaków"
Drukowanie łańcucha znaków
```

Wywołanie interpretera wraz ze skryptem jako parametrem, co zaprezentowano poniżej, rozpoczyna wykonywanie skryptu i kontynuuje aż do zakończenia skryptu. Kiedy skrypt zostaje zakończony, interpreter przestaje być aktywny.

```
bwd-linux:/book # python script.py
Wykonywanie skryptu
bwd-linux:/book #
```

Skrypty mogą także być uruchamiane z interpretera poprzez wykorzystanie wbudowanej w Pythona funkcji `execfile(`*`script`*`)`, gdzie *`script`* jest uruchamianym skryptem Pythona. Poniższy przykład prezentuje skrypt wykonywany za pomocą funkcji `execfile()`:

```
>>> execfile("script.py")
Wykonywanie skryptu
>>>
```

# Wbudowane typy

Najczęściej wykorzystywane typy Pythona, wbudowane w ten język, można pogrupować w kategorie wymienione w tabeli 1.1. Kolumna „Nazwa typu” pokazuje nazwę po-

wiązaną z każdym z wbudowanych typów i może być wykorzystywana w celu ustalenia, czy obiekt jest określonego typu, za pomocą funkcji `isinstance(object, typename)`, gdzie *`object`* to sprawdzany obiekt, natomiast *`typename`* jest nazwą typu, jak poniżej:

```
>>> s = "Prosty łańcuch znaków"
>>> print isinstance(s, basestring)
True
>>> print isinstance(s, dict)
False
>>>
```

**Tabela 1.1.** Popularne wbudowane typy Pythona

| Kategoria typu | Nazwa typu | Opis |
|---|---|---|
| None | `types.NoneType` | Obiekt `None` (obiekt null) |
| Liczby | `bool` | `True` lub `False` (Boolean) |
| | `int` | Liczba całkowita |
| | `long` | Długa liczba całkowita |
| | `float` | Liczba zmiennoprzecinkowa |
| | `complex` | Liczba zespolona |
| Zbiory | `set` | Zbiór zmienny |
| | `frozenset` | Zbiór niezmienny |
| Sekwencje | `str` | Łańcuch znaków |
| | `unicode` | Łańcuch znaków Unicode |
| | `basestring` | Typ bazowy dla wszystkich łańcuchów znaków |
| | `list` | Lista |
| | `tuple` | Krotka |
| | `xrange` | Sekwencja niezmienna |

**Tabela 1.1.** Popularne wbudowane typy Pythona — ciąg dalszy

| Kategoria typu | Nazwa typu | Opis |
|---|---|---|
| Odwzorowania | `dict` | Słownik |
| Pliki | `file` | Plik |
| Wywoływalne (Callable) | `type` | Typ dla wszystkich wbudowanych typów |
| | `object` | Rodzic wszystkich typów i klas |
| | `types.BuiltinFunctionType` | Wbudowana funkcja |
| | `types.BuiltinMethodType` | Wbudowana metoda |
| | `types.FunctionType` | Funkcja zdefiniowana przez użytkownika |
| | `types.InstanceType` | Obiekt klasy |
| | `types.MethodType` | Metoda związana |
| | `types.UnboundedMethodType` | Metoda niezwiązana |
| Moduły | `types.ModuleType` | Moduł |
| Klasy | `object` | Rodzic wszystkich klas |
| Typy | `type` | Typ dla wszystkich wbudowanych typów |

UWAGA

By możliwe było korzystanie z dowolnych obiektów typów takich jak `type` oraz `types.ModuleType`, moduł typów musi być zaimportowany.

## None

Typ `None` odpowiada obiektowi pustemu (ang. *null*), który nie posiada żadnej wartości. Typ `None` jest w Pythonie jedynym obiektem, który może być obiektem pustym. Składnia wykorzystywana w celu użycia tego typu w programach to po prostu `None`.

## Liczby

Typy liczbowe w Pythonie są bardzo proste. Typ `bool` posiada dwie możliwe wartości: `True` oraz `False`. Typ `int` przechowuje wewnętrznie 32-bitowe liczby całkowite. Typ `long` może przechowywać liczby ograniczone jedynie przez dostępną pamięć maszyny. Typ `float` wykorzystuje wbudowane liczby podwójnej precyzji do przechowywania 64-bitowych liczb zmiennoprzecinkowych. Typ `complex` przechowuje liczby jako pary liczb zmiennoprzecinkowych. Wartości te są dostępne za pomocą atrybutów `z.real` i `z.imag` obiektu `complex`.

## Zbiór

Typ zbioru reprezentuje nieuporządkowany zbiór unikalnych elementów. Istnieją dwa podstawowe typy zbiorów: zmienny (ang. *mutable*) `set` oraz niezmienny (ang. *immutable*) `frozenset`. Zbiory zmienne mogą być modyfikowane (można do nich dodawać i odejmować od nich elementy). Zbiory niezmienne nie mogą być modyfikowane po utworzeniu.

UWAGA

Wszystkie elementy umieszczone w zbiorze muszą być typu niezmiennego, dlatego też zbiory nie mogą zawierać elementów takich jak listy czy słowniki. Mogą jednak obejmować łańcuchy znaków oraz krotki.

## Sekwencje

W Pythonie istnieje kilka typów sekwencji. Sekwencje są uporządkowane i mogą być indeksowane przez liczby nieujemne. Sekwencje można łatwo przetwarzać i mogą one składać się z prawie każdego obiektu Pythona.

Dwa najczęściej dotychczas spotykane typy sekwencji to łańcuchy znaków oraz listy. W rozdziale 2., „Przetwarzanie łańcuchów znaków", omówione jest tworzenie i wykorzystywanie łańcuchów znaków. W rozdziale 3., „Zarządzanie typami danych", omówione zostały najpopularniejsze rodzaje sekwencji wraz z operacjami ich tworzenia i przetwarzania.

## Odwzorowania

Typ odwzorowań reprezentuje dwie grupy obiektów. Pierwszą z nich jest zbiór kluczy, które indeksują drugą grupę, zawierającą zbiór wartości. Każdy klucz indeksuje konkretną wartość z odpowiadającego mu zbioru. Klucz musi być typem niezmiennym. Wartością może być prawie każdy obiekt Pythona.

Słownik jest jedynym typem odwzorowania wbudowanym obecnie w Pythonie. W rozdziale 3. omówiono słowniki, ich tworzenie oraz przetwarzanie.

## Pliki

Typ pliku jest w Pythonie obiektem, który reprezentuje otwarty plik. Obiekty tego typu mogą być wykorzystywane do odczytywania danych z systemu plików i zapisywania ich do niego. W rozdziale 4., „Praca z plikami", omówiono obiekty plików, a także zaprezentowano niektóre z najczęściej wykorzystywanych sposobów związanych z używaniem plików w Pythonie.

## Wywoływalne

Obiekty typu wywoływalnego (ang. *callable*) obsługują operacje wywoływania funkcji w Pythonie, co oznacza, że mogą być wywoływane jako funkcje programu. Do kategorii tej zalicza się kilka typów wywoływalnych. Najczęściej spotykane są funkcje wbudowane w Pythona, funkcje zdefiniowane przez użytkownika, klasy oraz obiekty metod.

UWAGA

Klasy są uznawane za typy wywoływalne, ponieważ klasa wywoływana jest w celu utworzenia nowego obiektu tej klasy. Kiedy powoła się nowy obiekt klasy, obiekty metod tej klasy także stają się wywoływalne.

## Moduły

Typ modułu reprezentuje moduły Pythona, które zostały załadowane za pomocą instrukcji `import`. Instrukcja `import` tworzy obiekt typu modułu o tej samej nazwie co moduł Pythona. Następnie wszystkie obiekty wewnątrz modułu są dodawane do atrybutu `__dict__` nowoutworzonego obiektu typu modułu.

Dostęp do obiektów modułu można uzyskać bezpośrednio za pomocą składni z kropkami, ponieważ jest on tłumaczony na wyszukiwanie w słowniku. W ten sposób można użyć `module.object` zamiast dostępu do atrybutu poprzez `module.__dict__("object")` w celu uzyskania dostępu do obiektów modułu.

Przykładowo, moduł `math` zawiera obiekt liczbowy `pi`; poniższy kod ładuje moduł `math` i wywołuje obiekt `pi`:

```
>>> import math
>>> print math.pi
3.14159265359
```

# Podstawy składni Pythona

Język Python posiada wiele cech wspólnych z Perlem, językiem C oraz Javą. Istnieją jednak pomiędzy nimi także pewne zdecydowane różnice. Niniejszy podrozdział poświęcony jest krótkiemu opisowi składni, z którą można się spotkać w Pythonie.

## Używanie wcięć kodu

Jedną z pierwszych trudności, jakie napotykają programiści uczący się Pythona, jest brak nawiasów klamrowych do oznaczania bloków kodu dla definicji klas i funkcji czy do sterowania przepływem. Bloki kodu są oznaczane przez wcięcia wierszy, co jest surowo wymuszane.

Liczba spacji we wcięciu może być różna, jednak wszystkie instrukcje wewnątrz bloku muszą być wcięte o tyle samo. Oba bloki w poniższym przykładzie są poprawne:

```
if True:
    print "Prawda"
else:
  print "Fałsz"
```

Jednak drugi blok w kolejnym przykładzie wygeneruje błąd:

```
if True:
   print "Odpowiedź"
   print "Prawda"
else:
   print "Odpowiedź"
 print "Fałsz"
```

## Tworzenie wielowierszowych instrukcji

Instrukcje w Pythonie zazwyczaj kończą się znakiem końca wiersza. Python pozwala jednak na używanie znaku kontynuacji wiersza (\) do oznaczenia, że wiersz ma swój ciąg dalszy. Przykładowo:

```
total_sum = sum_item_one + \
            sum_item_two + \
            sum_item_three
```

Instrukcje zawarte wewnątrz nawiasów [], {} czy () nie wymagają użycia znaku kontynuacji wiersza. Przykładowo:

```
week_list = ['Poniedziałek', 'Wtorek', 'Środa',
             'Czwartek', 'Piątek']
```

## Cudzysłów

Python zezwala na używanie apostrofów (') i cudzysłowu (") bądź też trzech apostrofów ('''), lub trzech cudzysłowów (""") do oznaczenia literałów łańcuchów znaków, pod warunkiem, że ten sam typ znaku rozpoczyna i kończy łańcuch. Potrójne cudzysłowy mogą być wykorzystywane do rozciągnięcia łańcucha znaków na wiele wierszy. Przykładowo, wszystkie poniższe przykłady są poprawne:

```
word = 'słowo'
sentence = "To jest zdanie."
paragraph = """To jest akapit. Składa się
z kilku wierszy i zdań."""
```

## Formatowanie łańcuchów znaków

Python pozwala, by łańcuchy znaków były formatowane z użyciem predefiniowanego formatującego łańcucha znaków z listą zmiennych. Poniżej znajduje się przykład używania wielu formatujących łańcuchów znaków do wyświetlenia tych samych danych:

```
>>> list = ['Brad', 'Dayley', '"Python. Rozmówki"',
2007]

>>> letter = """
... Szanowny Panie %s,\n
... Dziękujemy za przesłanie nam Pańskiej książki, %s.
... Skontaktujemy się z Panem w %d roku."""

>>> display = """
... Tytuł: %s
... Autor: %s, %s
... Data: %d"""

>>> record = "%s|%s|%s|%08d"

>>> print letter % (list[1], list[2], list[3])
Szanowny Panie Dayley,
Dziękujemy za przesłanie nam Pańskiej książki,
"Python. Rozmówki".
Skontaktujemy się z Panem w 2007 roku.

>>> print display % (list[2], list[1], list[0],
list[3])
Tytuł: "Python. Rozmówki"
Autor: Dayley, Brad
Data: 2007

>>> print record % (list[0], list[1], list[2], list[3])
Brad|Dayley|"Python. Rozmówki"|00002007
```

## Wykorzystywanie instrukcji sterujących

Python obsługuje instrukcje `if`, `else` oraz `elif` dla warunkowego wykonywania kodu. Składnia jest następująca: `if` *`expression`*`:` *`block`*, gdzie *`expression`* to wyrażenie, a *`block`* — blok kodu, definiujący zachowanie warunkowe. Jeśli wyrażenie zwraca `True`, blok kodu jest wykonywany. Poniższy kod prezentuje przykład prostej serii bloków `if`:

```
if x == True:
   print "x jest prawdą"
elif y == True:
   print "y jest prawdą"
else:
   print "oba są fałszem"
```

Python obsługuje pętle warunkowe za pomocą instrukcji `while`. Składnia jest następująca: `while` *`expression`*`:` *`block`*, gdzie *`expression`* to wyrażenie, a *`block`* — blok kodu, definiujący zachowanie pętli. Dopóki wyrażenie zwraca `True`, blok jest wykonywany w pętli. Poniższy kod prezentuje przykład pętli warunkowej `while`:

```
x = 1
while x < 10:
   x += 1
```

Python obsługuje także instrukcję `for` dla pętli operujących na sekwencjach. Składnia jest następująca: `for` *`item`* `in` *`sequence`*`:` *`block`*, gdzie *`item`* to element sekwencji *`sequence`*, a *`block`* jest blokiem kodu definiującym zachowanie pętli. Po każdej pętli wybierany jest kolejny element z sekwencji i blok kodu jest wykonywany. Pętla `for` jest kontynuowana, dopóki w sekwencji nie skończą się elementy. Poniższe kody prezentują kilka różnych przykładów pętli sekwencyjnych `for`.

Pierwszy przykład wykorzystuje łańcuch znaków jako sekwencję, z której utworzy się listę kolejnych znaków z łańcucha:

```
>>> word = "Python"
>>> list = []
>>> for ch in word:
```

```
...     list.append(ch)
...
>>> print list
['P', 'y', 't', 'h', 'o', 'n']
```

Poniższy przykład wykorzystuje funkcję `range()` do utworzenia tymczasowej sekwencji liczb całkowitych rozmiaru listy, tak by elementy listy mogły być dodawane do łańcucha znaków w kolejności:

```
>>> string = ""
>>> for i in range(len(list)):
...     string += list[i]
...
>>> print string
Python
```

Kolejny przykład wykorzystuje funkcję `enumerate(string)` do utworzenia tymczasowej sekwencji. Funkcja `enumerate` zwraca wyliczenie w formie `(0, s[0]), (1, s[1])` i tak dalej, aż do końca sekwencji `string`, tak by pętla `for` mogła przypisać zarówno wartości `i`, jak i `ch` dla każdej iteracji w celu utworzenia słownika:

```
>>> dict = {}
>>> for i,ch in enumerate(string):
...     dict[i] = ch
...
>>> print dict
{0: 'P', 1: 'y', 2: 't', 3: 'h', 4: 'o', 5: 'n'}
```

Poniższy przykład wykorzystuje słownik jako sekwencję w celu wyświetlenia zawartości słownika:

```
>>> for key in dict:
...     print key, '=', dict[key]
...
```

```
0 = P
1 = y
2 = t
3 = h
4 = o
5 = n
```

Python udostępnia instrukcję `break` w celu przerwania wykonywania i wyjścia z bieżącej pętli. Zawiera także instrukcję `continue`, która przerywa wykonywanie bieżącej iteracji i rozpoczyna kolejną iterację bieżącej pętli. Poniższy przykład pokazuje użycie instrukcji `break` oraz `continue`:

```
>>> word = "Pithon. Rozmówki"
>>> string = ""
>>> for ch in word:
...     if ch == 'i':
...         string +='y'
...         continue
...     if ch == ' ':
...         break
...     string += ch
...
>>> print string
Python.
```

UWAGA

Po pętlach `for` i `while` można dodać instrukcję `else`, tak samo jak w przypadku instrukcji `if`. Instrukcja `else` jest wykonywana po tym, jak pętla z powodzeniem zakończy wszystkie iteracje. Jeśli napotykana jest instrukcja `break`, `else` nie jest wykonywane.

Obecnie nie ma w Pythonie instrukcji przełączającej (ang. *switch*). Najczęściej nie jest to problemem i można sobie poradzić dzięki serii instrukcji `if-elif-else`. Istnieje jednak wiele innych sposobów radzenia sobie z tym brakiem. Poniższy przykład pokazuje, jak można utworzyć prostą instrukcję przełączającą w Pythonie[1]:

```
>>> def a(s):
...     print s
...
>>> def switch(ch):
...     try:
...        {'1': lambda : a("jeden"),
...         '2': lambda : a("dwa"),
...         '3': lambda : a("trzy"),
...         'a': lambda : a("Litera a")
...        }[ch]()
...     except KeyError:
...        a("Nie znaleziono klucza")
...
>>> switch('1')
jeden
>>> switch('a')
Litera a
>>> switch('b')
Nie znaleziono klucza
```

[1] Warto zauważyć, że od dawna rozważano usunięcie notacji lambda z Pythona począwszy od przyszłej wersji 3.0; aktualnie nie zapowiada się, by miało się tak stać (PEP 3099) — *przyp. tłum.*

# Obiekty, moduły, klasy i funkcje Pythona

Niniejszy podrozdział poświęcony jest omówieniu podstawowych koncepcji związanych z obiektami, modułami, klasami oraz funkcjami języka Python. Podrozdział ten zakłada, że Czytelnik posiada podstawowe zrozumienie języków zorientowanych obiektowo, i ma dostarczyć wystarczającą ilość informacji, by możliwe było rozpoczęcie pracy z Pythonem oraz wykorzystywanie i tworzenie skomplikowanych modułów i klas.

## Wykorzystywanie obiektów

Python jest zbudowany wokół koncepcji „obiektu". Każdy fragment danych przechowywany i wykorzystywany przez Pythona jest obiektem. Listy, łańcuchy znaków, słowniki, liczby, klasy, pliki, moduły i funkcje — wszystkie one są obiektami.

Każdy obiekt w Pythonie posiada swoją tożsamość, typ oraz wartość. *Tożsamość* (ang. *identity*) wskazuje na lokalizację obiektu w pamięci. *Typ* (ang. *type*) opisuje reprezentację obiektu dla Pythona (więcej w tabeli 1.1). *Wartość* (ang. *value*) obiektu to po prostu dane w nim przechowywane.

Poniższy przykład pokazuje, w jaki sposób można uzyskać dostęp do tożsamości, typu i wartości obiektu w sposób programowy za pomocą odpowiednio: `id(object)`, `type(object)` oraz nazwy zmiennej:

```
>>> l = [1,2,3]
>>> print id(l)
9267480
>>> print type(l)
<type 'list'>
>>> print l
[1, 2, 3]
```

Po utworzeniu obiektu, tożsamość i typ nie mogą być zmienione. Jeśli wartość obiektu się zmienia, jest on uważany za obiekt zmienny (ang. *mutable*). Jeśli wartość obiektu nie może być modyfikowana, jest uznawany za obiekt niezmienny (ang. *immutable*).

Niektóre obiekty posiadają także atrybuty i metody. Atrybuty są wartościami powiązanymi z obiektem. Metody to wywoływalne funkcje, które wykonują operacje na obiekcie. Dostęp do atrybutów i metod obiektu uzyskuje się poprzez wykorzystanie składni z kropkami (.):

```
>>> class test(object):
...     def printNum(self):
...         print self.num
...
>>> t = test()
>>> t.num = 4
>>> t.printNum()
4
```

## Wykorzystywanie modułów

Cały Python zbudowany jest z modułów. Moduły są plikami Pythona, które pochodzą z podstawowych bibliotek dostarczanych wraz z tym językiem, a także modułów

tworzonych przez inne organizacje, tworzące rozszerzenia do Pythona oraz modułów napisanych samodzielnie przez programistę. Większe aplikacje lub biblioteki, które zawierają więcej modułów, zazwyczaj połączone są w pakiety. Pakiety pozwalają na połączenie kilku modułów pod jedną nazwą.

Moduły ładowane są do programu w Pythonie za pomocą instrukcji `import`. Po zaimportowaniu modułu tworzona jest dla niego i dla wszystkich zawartych w nim obiektów przestrzeń nazw. Następnie wykonywany jest kod z pliku źródłowego i tworzony jest obiekt modułu o takiej samej nazwie co plik źródłowy, dzięki czemu możliwy jest dostęp do przestrzeni nazw.

Istnieje kilka różnych sposobów importowania modułów. Poniższe przykłady obrazują niektóre z tych metod.

Moduły mogą być importowane bezpośrednio za pomocą nazwy pakietu bądź modułu. Dostęp do elementów z podmodułów musi odbywać się w sposób jawny, poprzez podanie pełnej nazwy pakietu:

```
>>> import os
>>> os.path.abspath(".")
'C:\\books\\python'
```

Moduły mogą być importowane bezpośrednio za pomocą nazwy modułu, jednak przestrzeń nazw może być nazwana inaczej. Dostęp do elementów z podmodułów musi odbywać się w sposób jawny, poprzez podanie pełnej nazwy pakietu:

```
>>> import os as computer
>>> computer.path.abspath(".")
'C:\\books\\python'
```

Moduły mogą być importowane za pomocą nazwy modułu wewnątrz pakietu. Dostęp do elementów z podmodułów musi odbywać się w sposób jawny, poprzez podanie pełnej nazwy pakietu:

```
>>> import os.path
>>> os.path.abspath(".")
'C:\\books\\python'
```

Moduły mogą być importowane poprzez wybranie konkretnych modułów z pakietu. Dostęp do elementów z podmodułów może odbywać się w sposób niejawny, bez podania pełnej nazwy pakietu:

```
>>> from os import path
>>> path.abspath(".")
'C:\\books\\python'
```

UWAGA

Python zawiera funkcję `reload(module)`, która przeładowuje moduł określony jako *`module`*. Jest ona wyjątkowo przydatna w czasie tworzenia programów, kiedy istnieje potrzeba uaktualnienia modułu i przeładowania go bez kończenia programu. Obiekty utworzone przed przeładowaniem modułu nie zostaną uaktualnione, dlatego należy bardzo uważać w przypadku wykonywania na nich jakichś operacji.

## Podstawy klas Pythona

Klasy Pythona są generalnie zbiorem atrybutów oraz metod. Klasy zazwyczaj służą jednemu z dwóch celów: tworzeniu nowego typu danych zdefiniowanego przez użytkownika lub rozszerzeniu możliwości istniejącego typu danych. Niniejszy podrozdział zakłada, że Czytelnik rozumie, czym są klasy, na bazie języka C, Javy czy innego języka zorientowanego obiektowo.

W Pythonie klasy są wyjątkowo łatwe do zdefiniowania; łatwo również utworzyć nowe obiekty klasy (ang. *instantiation*). Do zdefiniowania nowej klasy służy instrukcja `class` *`name`*`(`*`object`*`):`, gdzie *`name`* jest nazwą własnego typu obiektu, zdefiniowaną przez użytkownika, natomiast *`object`* określa obiekt Pythona, po którym się dziedziczy.

> UWAGA
>
> Dziedziczenie klas w Pythonie jest podobne do dziedziczenia z Javy, języka C i innych języków zorientowanych obiektowo. Metody i atrybuty klasy rodzica będą dostępne dla klasy potomnej, natomiast metody i atrybuty o tej samej nazwie w klasie potomnej nadpiszą te z klasy rodzica.

Cały kod zawarty w bloku następującym po instrukcji `class` wykonywany będzie za każdym razem, kiedy tworzony jest obiekt klasy. Próbka kodu o nazwie *testClass.py* ilustruje sposób tworzenia prostej klasy w Pythonie. Instrukcja `class` ustala nazwę typu klasy i dziedziczy po bazowej klasie `object`.

UWAGA

Instrukcja `class` definiuje jedynie typ obiektu klasy; nie tworzy samego obiektu klasy. Obiekt klasy nadal musi być utworzony poprzez bezpośrednie wywołanie klasy.

Funkcja `__init__` nadpisuje metodę odziedziczoną po klasie `object` i będzie wywoływana, kiedy tworzony będzie obiekt klasy. Obiekt klasy tworzony jest przez bezpośrednie wywołanie: `tc = testClass("Pięć")`. Po bezpośrednim wywołaniu klasy, zwracany jest obiekt tej klasy.

UWAGA

Możliwe jest określenie niezbędnych parametrów funkcji `__init__()` pod warunkiem, że parametry te dostarczy się w momencie wywoływania klasy w celu utworzenia obiektu klasy.

```
class testClass(object):
   print "Tworzenie nowej klasy\n====================="
   number=5
   def __init__(self, string):
      self.string = string
   def printClass(self):
      print "Liczba = %d"% self.number
      print "Łańcuch znaków = %s"% self.string

tc = testClass("Pięć")
tc.printClass()
tc.number = 10
tc.string = "Dziesięć"
tc.printClass()
```

*Plik testClass.py*

```
Tworzenie nowej klasy
=====================
Liczba = 5
Łańcuch znaków = Pięć
Liczba = 10
Łańcuch znaków = Dziesięć
```

*Dane wyjściowe dla kodu z pliku testClass.py*

UWAGA

Wewnątrz klasy konieczne jest stosowanie prefiksu `self.`, kiedy odnosi się do atrybutów i metod tej klasy. Choć `self` jest także wymieniane jako pierwszy argument we wszystkich metodach klasy, w rzeczywistości nie musi jednak być jawnie określane, kiedy wywołuje się metodę.

## Wykorzystywanie funkcji

Definiowanie i wywoływanie funkcji w Pythonie jest zazwyczaj stosunkowo łatwe, może jednak także stać się mocno zagmatwane. Najważniejsze, co należy zapamiętać, to że funkcje są obiektami języka Python i że przekazywane parametry są w rzeczywistości „stosowane" do obiektu funkcji.

By utworzyć funkcję, należy skorzystać z instrukcji `def` *`functionname`*`(`*`parameters`*`):`, gdzie *`functionname`* to nazwa funkcji, a *`parameters`* — jej parametry, a później zdefiniować funkcję w następującym po niej bloku kodu. Po zdefiniowaniu funkcji można ją wywoływać poprzez określenie nazwy funkcji oraz przekazanie jej odpowiednich parametrów.

Poniższe akapity pokazują niektóre z różnych sposobów wykonania pewnego zadania dla funkcji przedstawionej poniżej:

```
def fun(name, location, year=2006):
   print "%s/%s/%d" % (name, location, year)
```

- Pierwszy przykład prezentuje funkcję wywoływaną poprzez przekazanie wartości parametrów w kolejności. Warto zwrócić uwagę na fakt, iż parametr z rokiem posiada wartość domyślną, ustawioną w definicji funkcji, co oznacza, że ten parametr może zostać pominięty, a wtedy wykorzystana zostanie wartość domyślna.

  ```
  >>> fun("Robert", "Katowice")
  Robert/Katowice/2006
  ```

- Kolejny przykład pokazuje przekazywanie parametrów przez nazwę. Zaleta przekazywania parametrów przez nazwę polega na tym, że kolejność, w jakiej ukazują się one na liście parametrów, nie ma znaczenia.

  ```
  >>> fun(location="Berlin", year=2004,
  name="Aleksander" )
  Aleksander/Berlin/2004
  ```

- Poniższy przykład ilustruje możliwość łączenia różnych metod przekazywania parametrów. W przykładzie tym pierwszy parametr jest przekazany jako wartość, natomiast drugi i trzeci są przekazane przez nazwę.

  ```
  >>> fun("Amadeusz", year=2005, location="Wiedeń")
  Amadeusz/Wiedeń/2005
  ```

- Parametry mogą także być przekazane jako krotka (ang. *tuple*) za pomocą składni z *, jak pokazano na kolejnym przykładzie. Elementy krotki muszą odpowiadać parametrom, które są oczekiwane przez funkcję.

```
>>> tuple = ("Edward", "Kolonia", 2003)
>>> fun(*tuple)
Edward/Kolonia/2003
```

- Parametry mogą również być przekazywane jako słownik za pomocą składni z **, jak pokazano na poniższym przykładzie. Wpisy w słowniku muszą odpowiadać parametrom, które są oczekiwane przez funkcję.

```
>>> dictionary = {'name':'Franciszek',
'location':'Skierniewice', 'year':1999}
>>> fun(**dictionary)
Franciszek/Skierniewice/1999
```

- Wartości mogą być zwracane z funkcji za pomocą instrukcji `return`. Jeśli funkcja nie posiada instrukcji `return`, zwracany jest obiekt `None`. Poniższy przykład pokazuje prostą funkcję potęgowania, która przyjmuje liczbę i zwraca kwadrat tej liczby:

```
>>> def square(x):
...     return x*x
...
>>> print square(3)
9
```

UWAGA

Funkcje mogą być traktowane tak samo, jak wszystkie inne obiekty Pythona. Oprócz wywoływania, można je między innymi przypisywać jako wartość do listy bądź słownika, przekazywać jako argument, zwracać jako wartość i tak dalej.

- Operator `lambda` wbudowany w Pythona udostępnia metodę tworzenia funkcji anonimowych. Ułatwia to przekazanie prostych funkcji jako parametrów bądź przypisanie ich do nazw zmiennych. Operator `lambda` wykorzystuje następującą składnię w celu zdefiniowania funkcji:

  ```
  lambda <args> : <expression>
  ```

  Termin *`args`* odnosi się do listy argumentów, które są przekazywane do funkcji. Termin *`expression`* może być dowolnym poprawnym wyrażeniem Pythona. Poniższy kod prezentuje przykład zastosowania operatora `lambda` w celu przypisania anonimowej funkcji do zmiennej:

  ```
  >>> bigger = lambda a, b : a > b
  >>> print bigger(1,2)
  False
  >>> print bigger(2,1)
  True
  ```

## Przestrzenie nazw i zasięg

Zasięg w Pythonie powiązany jest z koncepcją przestrzeni nazw. *Przestrzenie nazw* (ang. *namespaces*) to generalnie słowniki, zawierające nazwy i wartości obiektów znajdujących się wewnątrz danego zakresu. Istnieją cztery główne typy przestrzeni nazw, z którymi można się spotkać: przestrzenie nazw globalne, lokalne, modułów oraz klas.

Globalne przestrzenie nazw są tworzone, kiedy program rozpoczyna wykonywanie. Globalna przestrzeń nazw początkowo zawiera wbudowane informacje o wykonywanym

module. Nowe obiekty są stopniowo dodawane do globalnej przestrzeni nazw w miarę definiowania ich w zakresie tej przestrzeni nazw. Globalna przestrzeń nazw jest dostępna ze wszystkich zakresów, jak pokazano na przykładzie, w którym globalna wartość `x` jest odczytywana za pomocą funkcji `globals()["x"]`.

UWAGA

Do globalnej przestrzeni nazw można zajrzeć poprzez wykorzystanie funkcji `globals()`, która zwraca obiekt słownika, zawierający wszystkie wpisy z tej przestrzeni nazw.

Lokalne przestrzenie nazw są tworzone, kiedy wywoływana jest funkcja. Lokalne przestrzenie nazw są zagnieżdżone wraz z funkcjami w miarę ich zagnieżdżania. Wyszukiwanie nazwy rozpoczyna się w najbardziej zagnieżdżonej przestrzeni nazw i postępuje stopniowo aż do globalnej przestrzeni nazw.

Instrukcja `global` zmusza nazwy do połączenia ich z globalną przestrzenią nazw zamiast z lokalną. W przykładowym kodzie wykorzystano instrukcję `global` do zmuszenia nazwy `x`, by wskazywała ona na globalną przestrzeń nazw. Kiedy `x` się zmienia, zmodyfikowany zostaje obiekt globalny.

UWAGA

Choć obiekty w zewnętrznie zagnieżdżonych przestrzeniach nazw mogą być widziane, modyfikowane mogą być jedynie najbardziej lokalne i globalne przestrzenie nazw. W przykładowym kodzie do zmiennej `b` z funkcji `fun` można odwoływać się z funkcji `sub`, jednak modyfikacja jej wartości w `sub` nie zmieni wartości w `fun`.

```
x = 1
def fun(a):
   b=3
   x=4
   def sub(c):
      d=b
      global x
      x = 7
      print ("Zagnieżdżona
funkcja\n====================")
      print locals()

   sub(5)
   print ("\nFunkcja\n====================")
   print locals()
   print locals()["x"]
   print globals()["x"]

print ("\nZmienne globalne\n====================")
print globals()

fun(2)
```

*Plik scope.py*

```
Zmienne globalne
====================
{'x': 1,
 '__file__':
'C:\\books\\python\\CH1\\code\\scope.py',
 'fun': <function fun at 0x008D757O>,
 't': <class '__main__.t'>,
 'time': <module 'time' (built-in)>,. . .}

Zagnieżdżona funkcja
====================
{'c': 5, 'b': 3, 'd': 3}

Funkcja
=================
{'a': 2, 'x': 4, 'b': 3, 'sub':
```

```
    <function sub at 0x008D75F0>}
4
7
```

*Dane wyjściowe dla kodu z pliku scope.py*

Przestrzeń nazw modułu jest tworzona, gdy moduł jest importowany, a obiekty wewnątrz modułu odczytywane. Dostęp do przestrzeni nazw modułu odbywa się za pomocą atrybutu `.__dict__` obiektu modułu. Dostęp do obiektów w przestrzeni nazw modułu można uzyskać bezpośrednio poprzez wykorzystanie nazwy modułu oraz składni z kropkami (`.`). Poniższy przykład pokazuje to na bazie wywołania funkcji `localtime()` modułu `time`:

```
>>> import time
>>> print time.__dict__
{'ctime': <built-in function ctime>,
 'clock': <built-in function clock>,
 ... 'localtime': <built-in function localtime>}
>>> print time.localtime()
(2006, 8, 10, 14, 32, 39, 3, 222, 1)
```

Przestrzeń nazw klasy jest podobna do przestrzeni nazw modułu, jest jednak tworzona w dwóch częściach. Pierwsza część jest tworzona, gdy klasa jest definiowana, natomiast druga część powstaje, kiedy tworzony jest obiekt klasy. Dostęp do przestrzeni nazw klasy może także odbywać się poprzez wykorzystanie atrybutu `.__dict__` obiektu klasy.

UWAGA

W poniższym kodzie warto zwrócić uwagę na fakt, iż `x` znajduje się w `t.__dict__`, natomiast `double` znajduje się w `tClass.__dict__`. Mimo to oba są dostępne za pomocą składni z kropkami obiektu klasy.

Dostęp do obiektów w przestrzeni nazw klasy może się odbywać w sposób bezpośredni za pomocą nazwy modułu oraz składni z kropkami. Na poniższym przykładzie widać to w przypadku instrukcji `print t.x` oraz `t.double()`:

```
>>> class tClass(object):
...     def __init__(self, x):
...         self.x = x
...     def double(self):
...         self.x += self.x
...
>>> t = tClass (5)
>>> print t.__dict__
{'x': 5}
>>> print tClass.__dict__
{'__module__': '__main__',
 'double': <function double at 0x008D7570>, . . . }
>>> print t.x
5
>>> t.double()
>>> print t.x
10
```

# Obsługa błędów

Obsługa błędów w Pythonie jest wykonywana poprzez wykorzystanie wyjątków, które są ujmowane w bloki `try` i obsługiwane w blokach `except`. Jeśli napotykany jest błąd, wykonanie kodu z bloku `try` jest zatrzymywane i przenoszone do bloku `except`, jak pokazano w poniższej składni:

```
try:
   f = open("test.txt")
except IOError:
   print "Nie można otworzyć pliku."
```

### Obsługa błędów

Wartość typu `exception` odnosi się albo do wyjątków wbudowanych w Pythona, albo do samodzielnie zdefiniowanego obiektu wyjątku. Wartość `error` jest zmienną, która przechwytuje dane zwracane przez wyjątek.

> **UWAGA**
>
> Blok `try` obsługuje także wykorzystywanie bloku `else` po ostatnim bloku `except`. Blok `else` jest wykonywany, jeśli blok `try` zakończy działanie bez otrzymania wyjątku.

Oprócz używania bloku `except` po bloku `try`, możliwe jest także wykorzystanie bloku `finally`. Kod z bloku `finally` będzie wykonany bez względu na to, czy wystąpi wyjątek. Jeśli wyjątek się nie pojawi, blok `finally` zostanie wykonany po bloku `try`. Jeśli wyjątek wystąpi, wykonanie jest natychmiast przenoszone do bloku `finally`, a następnie obsługa wyjątku jest kontynuowana, dopóki nie będzie on obsłużony[2]. Poniższy kod pokazuje przykład wykorzystania bloku `finally` w celu zmuszenia pliku do zamknięcia, nawet jeśli wystąpi wyjątek:

```
f = open("test.txt")
try:
   f.write(data)
   . . .
finally:
   f.close()
```

[2] Od wersji 2.5 Pythona możliwe jest już łączenie ze sobą dotychczas wzajemnie wykluczających się bloków `except` i `finally` w jeden blok `try-except-finally` — *przyp. tłum.*

Możliwe jest zgłoszenie wyjątku przez własny program poprzez użycie instrukcji `raise` *`exception`* `[,` *`value`*`]`. Wartość *`exception`* jest jednym z wbudowanych w Pythona wyjątków bądź też samodzielnie zdefiniowanym obiektem wyjątku. Wartością *`value`* jest obiekt Pythona, który tworzy się w celu podania szczegółów wyjątku. Zgłoszenie wyjątku przerywa bieżące wykonywanie kodu i zgłasza wyjątek. Poniższy przykład pokazuje, w jaki sposób można zgłosić ogólny wyjątek `RuntimeError` za pomocą prostej wartości komunikatu:

```
raise RuntimeError, "Błąd wykonania skryptu"
```

UWAGA

Jeśli wyjątek nie jest obsługiwany, program kończy się, a śledzenie wyjątku jest przesyłane do `sys.stderr`.

# Wykorzystywanie narzędzi systemowych

Jedną z najbardziej użytecznych cech Pythona jest zbiór modułów, które zapewniają dostęp do lokalnego systemu komputera. Moduły te umożliwiają dostęp do takich elementów jak system plików, system operacyjny oraz powłoka systemowa, a także do wielu funkcji systemowych.

Niniejszy podrozdział omawia wykorzystywanie modułów `os`, `sys`, `platform` oraz `time` w celu uzyskania dostępu do niektórych z najczęściej używanych informacji systemowych.

## Moduł os

Moduł `os` udostępnia przenośny, niezależny od platformy interfejs dla dostępu do popularnych usług operacyjnych, co pozwala na dodanie do programów obsługi na poziomie systemu operacyjnego. Poniższe przykłady ilustrują niektóre z najczęściej spotykanych zastosowań modułu `os`.

Funkcja `os.path.abspath(`*`path`*`)` modułu `os` zwraca bezwzględną ścieżkę (*`path`*) w formie łańcucha znaków. Ponieważ `abspath` bierze pod uwagę obecny bieżący katalog, elementy ścieżek takie jak `.` oraz `..` będą działały w następujący sposób:

```
>>> import os
>>> print os.path.abspath(".")
C:\books\python\ch1\
>>> print os.path.abspath("..")
C:\books\python\
```

Moduł `os.path` udostępnia funkcje `exists(`*`path`*`)`, `isdir (`*`path`*`)` oraz `isfile(`*`path`*`)` w celu sprawdzania istnienia plików oraz katalogów, jak zaprezentowano poniżej:

```
>>> print os.path.exists("/books/python/ch1")
True
>>> print os.path.isdir("/books/python/ch1")
True
>>> print
os.path.isfile("/books/python/ch1/ch1.doc")
True
```

Funkcja `os.chdir(`*`path`*`)` umożliwia prosty sposób zmiany bieżącego katalogu roboczego dla programu, na przykład:

```
>>> os.chdir("/books/python/ch1/code")
>>> print os.path.abspath(".")
C:\books\python\CH1\code
```

Atrybut `os.environ` zawiera słownik zmiennych środowiskowych. Można z tego słownika korzystać w pokazany poniżej sposób w celu uzyskania dostępu do zmiennych środowiskowych systemu:

```
>>> print os.environ['PATH']
C:\WINNT\system32;C:\WINNT;C:\Python24
```

Funkcja `os.system(`*`command`*`)` wykona funkcję systemową *`command`* tak, jakby znajdowała się ona w podpowłoce, jak pokazano w przypadku poniższego polecenia `dir`:

```
>>> os.system("dir")
Numer seryjny woluminu: 98F3-A875
Katalog: C:\books\python\ch1\code
2006-08-11  14:10      <DIR>          .
2006-08-11  14:10      <DIR>          ..
2006-08-10  16:00                 405 format.py
2006-08-10  10:27                 546 function.py
2006-08-10  15:07                 737 scope.py
2006-08-11  14:58                 791 sys_tools.py
               4 plik(ów)          3 717 bajtów
               2 katalog(ów)   7 880 230 400 bajtów
               ➥wolnych
```

Python udostępnia kilka funkcji typu `exec`, które służą do wykonywania aplikacji w rdzennym systemie. Poniższy przykład ilustruje wykorzystanie funkcji `os.execvp(`*`path`*`, `*`args`*`)` w celu wykonania aplikacji *update.exe* z parametrem wiersza poleceń `-verbose`:

```
>>> os.execvp("update.exe", ["-verbose"])
```

## Moduł sys

Moduł `sys` udostępnia interfejs dostępu do środowiska interpretera Pythona. Poniższe przykłady ilustrują niektóre z najczęstszych zastosowań modułu `sys`.

Atrybut `argv` modułu `sys` jest listą. Pierwszy element na liście `argv` jest ścieżką do modułu; reszta listy składa się z argumentów, które zostały przekazane do modułu na początku wykonania. Przykładowy kod pokazuje, w jaki sposób można wykorzystać listę `argv` dla dostępu do parametrów wiersza poleceń przekazanych do modułu Pythona:

```
>>> import sys
>>> print sys.argv
['C:\\books\\python\\CH1\\code\\print_it.py',
'text']
>>> print sys.argv[1]
text
```

Atrybut `stdin` modułu `sys` jest obiektem pliku, który zostaje utworzony na początku wykonywania kodu. W poniższym przykładowym kodzie `text` jest odczytywany z `stdin` (w tym przypadku z klawiatury, co jest wartością domyślną) za pomocą funkcji `readline()`:

```
>>> text = sys.stdin.readline()
Dane wejściowe
>>> print text
Dane wejściowe
```

Moduł `sys` posiada także atrybuty `stdout` oraz `stderr`, które wskazują na pliki używane jako standardowe wyjście oraz standardowe wyjście błędów. Pliki te domyślnie zwią-

zane są z ekranem. Poniższy fragment kodu pokazuje, w jaki sposób przekierować standardowe wyjście oraz standardowe komunikaty błędów do pliku w miejsce wyświetlania ich na ekranie:

```
>>> sOUT = sys.stdout
>>> sERR = sys.stderr
>>> sys.stdout = open("ouput.txt", "w")
>>> sys.stderr = sys.stdout
>>> sys.stdout = sOUT
>>> sys.stderr = sERR
```

## Moduł platform

Moduł `platform` udostępnia przenośny interfejs do informacji o platformie, na której uruchamiany jest program. Poniższe przykłady ilustrują niektóre z najczęstszych zastosowań modułu `platform`.

Funkcja `platform.architecture()` zwraca krotkę (*`bits`*, *`linkage`*), gdzie *`bits`* to liczba bitów wielkości słowa w systemie, natomiast *`linkage`* to powiązane informacje o pliku wykonywalnym Pythona:

```
>>> import platform
>>> print platform.architecture()
('32bit', '')
```

Funkcja `platform.python_version()` zwraca wersję pliku wykonywalnego Pythona dla celów zgodności:

```
>>> print platform.python_version()
2.4.2
```

Funkcja `platform.uname()` zwraca krotkę w formie (*`system, node, release, version, machine, processor`*). W krotce tej *`system`* odnosi się do aktualnie działającego systemu operacyjnego, *`node`* do nazwy hosta danej maszyny, *`release`* do głównej wersji systemu operacyjnego, *`version`* do informacji o wydaniu systemu operacyjnego w formie łańcucha znaków, natomiast *`machine`* oraz *`processor`* odnoszą się do informacji sprzętowych danej platformy.

```
>>> print platform.uname()
('Linux', 'bwd-linux', '2.6.16-20-smp',
 '#1 SMP Mon Apr 10 04:51:13 UTC 2006',
 'i686', 'i686')
```

## Moduł time

Moduł `time` udostępnia przenośny interfejs do funkcji związanych z czasem w systemie, na którym program jest wykonywany. Poniższe przykłady ilustrują niektóre z najczęstszych zastosowań modułu `time`.

Funkcja `time.time()` zwraca bieżący czas systemowy jako liczbę sekund, które upłynęły od 1 stycznia 1970 roku zgodnie z UTC (*Coordinated Universal Time*). Wartość ta jest zazwyczaj zbierana w kilku punktach programu i jest wykorzystywana w operacjach odejmowania w celu ustalenia czasu, który upłynął od jakiegoś zdarzenia.

```
>>> import time
>>> print time.time()
1155333864.11
```

Funkcja `time.localtime(secs)` zwraca czas, określany w sekundach, od 1 stycznia 1970 roku, w formie krotki (*year, month, day, hour, second, day of week, day of year, daylight savings*), zawierający rok, miesiąc, dzień, godzinę, sekundę, dzień tygodnia, dzień roku oraz przesunięcie związane z czasem letnim bądź zimowym. Jeśli czas nie jest podany, wykorzystywany jest czas bieżący, jak poniżej:

```
>>> print time.localtime()
(2006, 8, 11, 16, 4, 24, 4, 223, 1)
```

Funkcja `time.ctime(secs)` zwraca czas, określony w sekundach, od 1 stycznia 1970 w formie sformatowanego łańcucha znaków, nadającego się do wydrukowania. Jeśli czas nie został określony, wykorzystywany jest czas bieżący, jak poniżej:

```
>>> print time.ctime()
Fri Aug 11 16:04:24 2006
```

Funkcja `time.clock()` zwraca aktualny czas procesora w postaci liczby zmiennoprzecinkowej, która może być wykorzystywana do różnych funkcji mierzących czas:

```
>>>print time.clock()
5.02857206712e-006
```

Funkcja `time.sleep(secs)` zmusza bieżący proces do uśpienia na liczbę sekund określoną przez liczbę zmiennoprzecinkową *secs*:

```
>>>time.sleep(.5)
```