zaj1-blok2

October 22, 2015

1 Wprowadzenie do Pythona

(dla naukowców)

2 Funkcje

Do tworzenia funkcji służy słowo kluczowe `def`

```
In [3]: def pole_trojkata(a, h): # Argumenty funkcji
            """
            Pole trójkąta o podstawie a i wysokości h
            """
            # To jest normalna stała znakowa.
            # Pierwsza stała znakowa w funkcji, klasie czy module nazwana
            # jest docstringiem i jest dostępna podczas wykonania programu
            return 0.5 * a * h


In [4]: pole_trojkata(10, 10)

Out[4]: 50.0

In [5]: pole_trojkata.__doc__.strip()

Out[5]: 'Pole trójkąta o podstawie a i wysokości h'
```

3 Obsługa styuacji wyjątkowych

Do obsługi sytuacji wyjątkowych służą słowa: `try`, `catch` `raise`.

```
In [7]: import math
        def rownanie_kwadratowe(a, b, c):
            d = b**2 - 4*a*c
            if d < 0:
                raise ValueError("Równanie nie ma rozwiązań rzeczywistych")
            if d == 0:
                return (-b/(2*a), None)
            delta = math.sqrt(d)
            return (-b+delta)/(2*a), (-b-delta)/(2*a)

In [8]: rownanie_kwadratowe(1, 0, -1)

Out[8]: (1.0, -1.0)
```

```
In [9]: rownanie_kwadratowe(1, 2, 1)

Out[9]: (-1.0, None)

In [10]: rownanie_kwadratowe(1, 0, 1)


        ---------------------------------------------------------------------------

        ValueError                                Traceback (most recent call last)

        <ipython-input-10-45325fa8843e> in <module>()
    ----> 1 rownanie_kwadratowe(1, 0, 1)


        <ipython-input-7-4e49bf685f6b> in rownanie_kwadratowe(a, b, c)
          3     d = b**2 - 4*a*c
          4     if d < 0:
    ----> 5         raise ValueError("Równanie nie ma rozwiązań rzeczywistych")
          6     if d == 0:
          7         return (-b/(2*a), None)


        ValueError: Równanie nie ma rozwiązań rzeczywistych
```

3.1 Obsługa sytuacji wyjątkowych

```
In [11]: print("Podaj oddzielone przecinakmi współczynniki równania")
         a, b, c = map(int, input().split(','')) # Ten kawałek może być enigmatyczny!
         try:
             print("Wynik równania {}".format(rownanie_kwadratowe(a, b, c)))
         except ValueError:
             print("Równanie nie ma wyników")

Podaj oddzielone przecinakmi współczynniki równania
1, 2, 3
Równanie nie ma wyników
```

polecenie `pass` nie robi nic

```
In [11]: # Pełna składnia insrukcji:
         try:
             pass # Tutaj spodziewamy się wyjątku
         except (IndexError, KeyError) as e:
             pass # Klauzula Catch może złapać wiele typów wyjątków.
             #Wyjątek jest widoczny jako zmienna e
         except ValueError as e:
             pass
         else:
             pass # Wykona się jeśli blok try nie rzuci wyjątku
                  # (może samo rzucić wyjątkiem, który nie będzie złapany)
         finally:
             pass # Wykona się zawsze
```

4 Klasy

```
In [13]: class Foo(object): # Klasa o nazwie Foo dziedziczy po typie object

             def __init__(self, foo): # Konsruktor
                 self.foo = foo # Utworzenie artybutu

             def print_foo(self):
                 print(foo)
```

Zmienna `self` jest odpowiednikiem `this` z Javy/C++, różnica jest taka że znajduje się na liście argumentów, oraz musi być zawsze używana jawnie

```
In [14]: inst = Foo('foo')
         inst.foo
```

```
Out[14]: 'foo'
```

```
In [15]: print(inst)
```

```
<__main__.Foo object at 0x7fea087cd908>
```

Atrybut do obiektu można przypiąć również poza metodą `__init__`:

```
In [16]: inst.bar = 'bar'
         inst.bar
```

```
Out[16]: 'bar'
```

4.1 Dziedziczenie

Python (oczywiście!) wspiera dziedziczenie:

- Wspiera nawet wielodziedziczenie
- Klasa dziedzicząca może nadpisywać metody i atrybuty.

```
In [17]: class Foo(object):

             def foo(self):
                 print("foo")

             def bar(self):
                 print("foo")

         foo = Foo()
         foo.foo()
         foo.bar()
```

```
foo
foo
```

```
In [17]: class Baz(Foo): # Dziedziczy

             def bar(self):
                 print ("bar")

         baz = Baz()
         baz.foo()
         baz.bar()
```

```
foo
bar
```

4.2 Wywoływanie metod z nadklasy

```
In [18]: class BazFoo(Foo): # Dziedziczy

             def bar(self):
                 super().bar() # Super wybiera metodę z nadklasy
                 Foo.foo(self) # Tak też można!
                 print ("bar")

In [21]: bazfoo = BazFoo()
         bazfoo.bar()

foo
foo
bar
```

Super jest bardzo przydatne przy wielodziedziczenu, ale tutaj raczej odsyłałam do lektury: http://www.artima.com/weblogs/viewpost.jsp?thread=236275

4.3 Prywatność

W Pythonie nie ma zmiennych prywatnych. Domyślnie wszystko jest dostępne.

Ma to sporo wad, ale też dużo zalet. Główną jest to że zawsze można naprawić cudzy kod (bez ingerencji w kod źródłowy) po prostu nadpisując źle działające jego części.

Zgodnie z konwencją zmienne zaczynające się od podkreślenia są prywatnawe, są dostępne jak każde inne, ale użytkownik kodu powinien przeczytać dokumentację zanim zacznie ich używać. Ta zmienna: `_foo` jest prywatnawa.

Zmienne z dwoma podkreśleniami na początku są chronione za pomocą name mangling, na przykład takie: `__foo`.

Zmienne ktore zaczynają i kończą się dwoma podkreśleniami są to zmienne zarezerwowane przez twórców Pythona, na przykład `__init__`, `__del__`.

4.4 Prywatność

```
In [23]: class Foo():

             def __init__(self):
                 self.normal = 1
                 self._private = 2
                 self.__mangled = 3

In [27]: foo = Foo()
         foo.__dict__ #__dict__ przechowuje słowinik zawierający atrybuty klasy

Out[27]: {'_private': 2, '_Foo__mangled': 3, 'normal': 1}
```

4.5 Duck typing

If it walks like a duck, if it quacks like a duck: it is a duck!

W Pythonie dość rzadko sprawdza się typy różnych obiektów. Zakłada się raczej że jeśli coś wspiera operacje dostępne dla kaczki to jest kaczką.

```
In [35]: class FooImpostor(object):
             def foo(self):
                 print ('impostor')
```

```
        def expect_foo(foo):
            foo.foo()

        expect_foo(Foo())
        expect_foo(FooImpostor())
```

```
foo
impostor
```

5 Wszystko jest obiektem

Obiekt to coś co:

- Ma/może mieć atrybuty
- Można to przypisać to zmiennej
- Można zserializować na dysk

5.1 Funkcja jest obiektem

```
In [36]: def foo():
             print("foo")
         foo()
```

```
foo
```

```
In [37]: foo.bar = "10"
         print(foo.bar)
```

```
10
```

```
In [38]: bar = foo
         bar()
```

```
foo
```

5.2 Przykład wykorzystania przekazania funkcji jako obiektu

Posortujemy listę pythona w kolejności malejącej. Do sortowania służy wbudowana funkcja `sorted`, przyjmuje ona (między innymi) funkcję generującą klucz sortujący obiekty będą posortowane (rosnąco) względem wartości klucza.

```
In [39]: def key(x):
             return -x
```

```
In [40]: l = list(range(10))
```

```
In [41]: l
```

```
Out[41]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
In [42]: sorted(l, key=key)
```

```
Out[42]: [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

5.3 Problem z "funkcja jest obiektem"

Nie da się przeładowywać funkcji — w danej przestrzeni nazw może istnieć tylko jedna funkcja o danej nazwie

5.4 Typ jest obiektem

```
In [43]: class Foo(object):
             pass

In [44]: Bar = Foo

In [45]: Bar()

Out[45]: <__main__.Foo at 0x7f8ae809a748>

In [46]: class Bar(object): pass
         class Baz(object): pass

         types = [Foo, Bar, Baz]
```

5.5 Domknięcia (*z ang.* closure)

Domknięcie to: obiekt wiążący funkcję oraz wolne zmienne znajdujące się w środowisku jej definicji.
Przykład:

```
In [47]: x = 10
         def foo(y):
             return x*y
         foo(2)

Out[47]: 20
```

- Funkcja foo przyjmuje jeden argument `y`
- Druga (`x`) zmienna pochodzi z **domknięcia**.

5.6 Modyfikacja zmiennej z domknięcia:

- Generalnie ułatwianie modyfikcji globalnego kontekstu jest złym pomysłem.
- By powiedzieć: "Drogi Interpreterze wiem co robię pozwól mi zmodyfikowac globalną zmienną y" można użyć słowa `global` które oznacza zmienną jako istniejącą globalnie.

```
In [48]: x = 10
         def foo():
             global x
             x = 20
         foo()
         x

Out[48]: 20
```

6 Keyword arguments

Brak możliwości przeładowania funkcji powoduje że funkcie w Pythonie przyjmują dużo argumentów

```
In [49]: def minimize(fun, x0, args=(), method=None, jac=None, hess=None,
                      hessp=None, bounds=None, constraints=(), tol=None,
                      callback=None, options=None):
             pass
```

Wydaje się żę jest do dość niewygodne, ale da się z tym żyć.
Weźmy prostszy przykład:

```
In [50]: def foo(foo, bar=None, baz=None):
             print(foo, bar, baz)

In [51]: foo('foo')
         foo('foo', 'bar')

foo None None
foo bar None
```

7 Keyword Arguments

```
In [52]: foo('foo', baz='foobaz')

foo None foobaz
```

Taka składnia oznacza:

- Pierwszy argument ma wartość foo
- Argument o nazwie `baz` przyjmuje wartość `foobaz`
- Pozostałe argumenty przyjmują wartość domyślną

```
In [53]: foo(baz='foobaz', foo='foo') # Kolejność keyword argumentów nie ma znaczenia

foo None foobaz

In [54]: foo(baz='foobaz', 'foo') # Ale najpierw należy podać wszystkie "zwykłe argumenty"


          File "<ipython-input-54-c119fcd00c21>", line 1
        foo(baz='foobaz', 'foo') # Ale najpierw należy podać wszystkie "zwykłe argumenty"
                          ^
    SyntaxError: non-keyword arg after keyword arg


In [ ]:
```