Język Programowania Python

Notatki

„Zaczynając ten kurs nie wiedziałem jaki jest jego cel i teraz też nie wiem."

Maciej Mikołajczak
Kraków, 2024/2025

Spis treści

1. Podstawy

2024-12-09

Każda zmienna w Pythonie jest typu PyObject, przez to nie ma typów, ale implementacja jest bardziej złożona – żeby dodać dwie zmienne trzeba wewnątrz dodawania ifować, czy da się dodać te konkretne typy.

Wykorzystanie zmiennej globalnej w funkcji działa tak jak się spodziewamy. Pisanie do niej stworzy nową zmienną lokalną, nie zajrzy do zmiennej globalnej.

Instrukcja `dir` pozwala na zobaczenie wszystkich składowych danej zmiennej (każda zmienna jest tak jakby instancją obiektu).

Możemy podglądnąć działanie Pythona za pomocą `dis` (Python disassembler).

Funkcja (tak jak wszystko) jest obiektem.

2. Programowanie obiektowe

2024-12-16

Obiekt jest właściwie słownikiem, który trzyma PyObjecty (tak jak zwykły słownik). Obiekt ma pole `__dict__`, które trzyma ten słownik.

Metody mają pierwszy parametr, który jest wskaźnikiem na instancję klasy. Zwykle nazywamy go `self`, ale można dowolnie. Metody należą do klasy (są w jej słowniku), można na niej wywoływać metody, podając odpowiednią instancję obiektu.

```
class Q:
    def alfa(self, beta):
        self.a = beta

Q.alfa # metoda jako element klasy
q = Q()
Q.alfa(q,1) # to samo co q.alfa(1)
```

Coś takiego jest dość wolne, dlatego typy wbudowane trzymane są inaczej.

Typy wbudowane mają metody typu `__mul__` i `__pow__`, które odpowiadają za operatory arytmetyczne `*` i `**`. Zaimplementowanie takich funkcji spowoduje, że będziemy mogli używać tych ładnych składni na naszych obiektach. Podobnie działają rzutowania.

Szukając atrybutu Python wywołuje metodę `__getattribute__`, można ją nadpisać – wtedy szukanie pól będzie się odbywało po naszemu. Podobnie działa `__setattr__`.

```
class Q:
    pass

q = Q()
q.__setattr__('e', 1)
q.__getattribute__('e') # 1
```

```
class A:
    def a(self):
        self.a = 'a'

class B:
    def a(self):
        self.a = 'b'

class C(A,B): pass

c = C()
c.a()
c.a # 'a': wykona się pierwsze, tak się rozwiązuje diamond problem; do tego
    teraz metoda c.a została przesłonięta przez stringa
```

```
q = C()
q.__class.__bases__ # (A,B), zdobywamy (bezpośrednie) klasy bazowe
A.__bases__ # (object, ) - bazowa klasa
q.__class__.__mro__ # posortowane klasy bazowe (wszystkie, nie tylko bezpośrednie), w kolejności szukania metod
```

MRO wywołuje się na klasach bazowych, dostaje listę list MRO klas bazowych i swoją listę dziedziczenia. Wyznacza kolejność: wyciąga pierwszą w kolejności klasę, która występuje jako pierwsza (lub nie występuje) w każdej liście. Następnie usuwa tę klasę i kontynuuje. W ten sposób klasa późniejsza w MRO podklasy jest późniejsza w MRO klasy.

```
match: # podobne do switcha, ale potrafi matchować typy
    case 5:
        print(5) # prosty switch
    case ['up'] | ['go', 'up']:
        pass # wartość jest jedną z tych list
    case ['pick', obj, 'up'] | ['pick', 'up', obj]:
        pass # wartość jest jedną z tych list, gdzie obj jest dowolną zmienną dowolnego typu
    case {'text': str(message), 'c': str(color)}:
        pass # wartość ewaluuje się do słownika o zadanych polach, które są stringami
```

3. Meta programowanie

2025-01-13

```
@functools.cache # dekorator powodujący, że wywołania rekurencyjne też wywołują spamiętywanie
def fib(n):
    if n==0 or n==1:
        return 1
    return fib(n-1)+fib(n-2)

fast_fib = functools.cache(fib) # zwraca funkcję ze spamiętywaniem
```

```
class A:
    pass

type(A) # wypisuje 'type'
type(type) # też 'type'

B = type('B', (A,), {}) # tworzy klasę B dziedziczącą po A z pustym ciałem

C = type('C', (A,) {'__init__': lambda self: print(10)}) # ta klasa ma konstruktor
```

```
class A:
    def __init_subclass__(cls):
        print(f'Hello {cls}')

B = type('B', (A,), {}) # Hello <class '__main__.B'>
```

```
class A(B,C, metaclass=D): # teraz type(A) to 'D' -- ustalane w kolejności MRO, defaultowo metaklasa to type
    @staticmethod # metoda jest statyczna, teraz f.__get__ nie podstawia self jako pierwszy argument
    def f(arg):
        pass

```

```
6       @classmethod # nie wywołuje się na instancji, tylko na klasie, więc
            zamiast self jest type(self)
7       def g(cls, arg):
8           pass
9
10      @property # przy a.val wywołuje a.val(), wywołanie a.val = 10 powoduje błąd
11      def val(self):
12          return 10*self._val
13
14      @val.setter # teraz mamy setter, wywołanie a.val = 10 ma sens
15      def val(self, v): # nazwa tej funkcji może być dowolna
16          self._val = v
```