

Zajęcia 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 ta 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 zmiennej jest tak jakby instancją obiektu).

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

Funkcja (tak jak wszystko) jest obiektem.

Zajęcia 2: Programowanie obiektowe

2024-12-16

Obiekt jest właściwie słownikiem, który trzyma `PyObject`y (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.

```
1 class Q:
2     def alfa(self, beta):
3         self.a = beta
4
5 Q.alfa # metoda jako element klasy
6 q = Q()
7 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 `__getattr__`, można ją nadpisać – wtedy szukanie pól będzie się odbywało po naszymu. Podobnie działa `__setattr__`.

```
1 class Q:
2     pass
3
4 q = Q()
5 q.__setattr__('e', 1)
6 q.__getattr__('e') # 1
```

```
1 class A:
2     def a(self):
3         self.a = 'a'
4
5 class B:
6     def a(self):
7         self.a = 'b'
8
9 class C(A,B): pass
10
11 c = C()
12 c.a()
13 c.a # 'a': wykona się pierwsze, tak się rozwiązuje diamond problem; do tego
      teraz metoda c.a została przesłonięta przez stringa
```

```

1 q = C()
2 q.__class__.__bases__ # (A,B), zdobywamy (bezpośrednie) klasy bazowe
3 A.__bases__ # (object,) - bazowa klasa
4 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.

```

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

```

Zajęcia 3: Meta programowanie

2025-01-13

```

1 @functools.cache # dekorator powodujący, że wywołania rekurencyjne też
    wywołują spamiętywanie
2 def fib(n):
3     if n==0 or n==1:
4         return 1
5     return fib(n-1)+fib(n-2)
6
7 fast_fib = functools.cache(fib) # zwraca funkcję ze spamiętywaniem

```

```

1 class A:
2     pass
3
4 type(A) # wypisuje 'type'
5 type(type) # też 'type'
6
7 B = type('B', (A,), {}) # tworzy klasę B dziedziczącą po A z pustym ciałem
8
9 C = type('C', (A,), {'__init__': lambda self: print(10)}) # ta klasa ma
    konstruktor

```

```

1 class A:
2     def __init_subclass__(cls):
3         print(f'Hello {cls}')
4
5 B = type('B', (A,), {}) # Hello <class '__main__.B'>

```

```

1 class A(B,C, metaclass=D): # teraz type(A) to 'D' -- ustalane w kolejności
    MRO, defaultowo metaklasa to type
2     @staticmethod # metoda jest statyczna, teraz f.__get__ nie podstawia self
        jako pierwszy argument
3     def f(arg):
4         pass
5

```

```
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
```