Akademia Pythona

IV Funkcje

KN Pythona - Kurs Pythona

KN Pythona wita na kursie Pythona.

Plan:

- Zaawansowane zagadnienia dotyczące funkcji
- Iteracje i składanie list - część 2.

Funkcje

Koncepcje projektowania funkcji:

- Sprzęganie: używanie argumentów jako danych wejściowych oraz instrukcji return do zwracania wyników.
- Sprzęganie: używanie zmiennych globalnych wyłącznie tam, gdzie to konieczne.
- Sprzęganie: nie należy modyfikować argumentów mutowalnych, o ile kod wywołujący tego nie oczekuje.
- Spójność: każda funkcja powinna mieć jeden zunifikowanyc cel.
- Rozmiar: każda funkcja powinna być relatywnie mała.
- Sprzęganie: należy unikać bezpośredniego modyfikowania zmiennych zdefiniowanych w innym pliku modułu.

Funkcje rekurencyjne

Sumowanie z użyciem rekurencji

```
def mysum(L):
    if not L:
        return 0
    else:
        return L[0] + mysum(L[1:])

mysum([1, 2, 3, 4, 5]) # 15
```

Funkcje rekurencyjne

Sumowanie z użyciem rekurencji

```
def mysum(L):
    first, *rest = L
    return first if not L else first + mysum(rest)

mysum([1, 2, 3, 4, 5]) # 15
```

Funkcje rekurencyjne

Obsługa dowolnych struktur

```
def sum_tree(L):
    tot = 0
    for x in L:
        if not isinstance(x, list):
            tot += x
        else:
            tot += sum_tree(x)
    return tot

L = [1, [2, [3, 4], 5], 6, [7, 8]]
print(sum_tree(L)) # 36
```

Funkcje rekurencyjne

```
import sys
sys.setrecursionlimit(10**10)
```

Obiekty funkcji

Pośrednie wywołania funkcji

```
def echo(message):
    print(message)

x = echo
x('abc') # abc


def indirect(func, arg):
    func(arg)

indirect(echo, 'abc') # abc
```

Harmonogram

```
schedule = [(echo, 'abc'), (echo, 'xyz')]
for func, arg in schedule:
    func(arg) # abc, xyz
```

Introspekcja funkcji

```
def func(a):
    b = 'abc'
    return b * a

func(3) # 'abcabcabc'
```

Introspekcja funkcji

```
func.__name__ # 'func'
dir(func)
func.__code__
dir(func.__code__)
func.__code__.to_varnames # ('a', 'b')
func.__code__.to_argcount # 1
```

Atrybuty funkcji

```
func.count = 0
func.count += 3
func.count # 3
```

Adnotacje funkcji

```
def func(a: 'expl', b: (1, 10) = 4, c: float) -> int:
    return a + b + c

func.__annotations__
```

Funkcje anonimowe

Wyrażenia **lambda** zostały zaczerpnięte z języka LISP, który z kolei zaczerpnął je z rachunku lambda, formy logiki symbolicznej.

Funkcje anonimowe

```
lambda argument1, ...: wyrażenie wykorzystujące argumenty
```

Funkcje anonimowe

Funkcje anonimowe:

- **lambda** jest wyrażeniem, a nie instrukcją.
- Ciałem **lambda** jest pojedyńcze wyrażenie, a nie blok instrukcji.

Funkcje anonimowe

```
def func(x, y, z): return x + y + z

func = lambda x, y, z: x + y + z

x = lambda a='raz', b='dwa', c='trzy': a + b + c
x('las') # lasdwatrzy
```

Funkcje anonimowe

```
def title(a):
    pref = a
    return lambda x: pref + " " + x

a_func = title('abc')
a_func('def') # abcdef
```

Funkcje anonimowe

```
lower = lambda x, y: x if x < y else y

showall = lambda x: map(sys.stdout, x)

showtable = lambda x: [sys.stdout.write(line) for line in x
```

Funkcje anonimowe

```
action = (lambda x: (lambda y: x + y))

act = action(2)
act(2) # 4 (2 + 2 = 4)
```

Odwzorowywanie funkcji na sekwencje - map

```
counters = [1, 2, 3, 4]
updated = []
for x in counters:
    updated.append(x + 10)

updated # 11, 12, 13, 14
```

Odwzorowywanie funkcji na sekwencje - map

```
def inc(x): return x + 10

map(inc, counters)

map((lambda x: x + 10), counters)
```

Narzędzia programowania funkcyjnego - filter i reduce

```
filter((lambda x: x > 0), range(-100, 100)) # 1..99

from functools import reduce

reduce((lambda x, y: x + y), [1, 2, 3, 4]) # 10
reduce((lambda x, y: x * y), [1, 2, 3, 4]) # 24

import operator
reduce(operator.add, [1, 2, 3, 4]) # 10
```

Listy składane kontra map

```
list(map(ord, 'abcdef'))
```

```
[ord(x) for x in 'abcdef']
```

Warunki

```
list(map((lambda x: x ** 2), filter((lambda x: x % 2 == 0)
```

```
[x ** 2 for x in range(5) if not x % 2]
```

Generatory

Funkcje generatorów, które tworzy się tak samo jak zwykłe funkcje, ale zamiast instrukcji **return** do zwracania wyników cząstkowych wykorzystuje się instrukcję **yield**, która zawiesza wykonanie funkcji, zachowując jej stan, co pozwala wznowić ją w przypadku, gdy odbiorca znów poprosi o dane.

Generatory

Wyrażenia generatorów mają składnię zbliżoną do list składanych, ale zwracają obiekt generujący wyniki cząstkowe na żądanie, zamiast tworzyć naraz całą listę wyników.

Generatory

```
def gensquares(N):
    for i in range(N):
        yield i ** 2

list(gensquares(100))
```

Generatory a protokół iteracyjny

Pętle for i inne konteksty iteracyjne wymagają obiektów implementujących interfejs iteratorów. Instrukcje **yield** kompilowane są jako generatory, które obsługują interfejs iteracji i metodę **next** wznawiającą wykonywanie operacji.

Wyrażenia generatorów

```
def times_four(S):
    for c in S:
        yield c * 4

(c * 4 for c in S) # <generator object at 0x...>
```