

PODSTAWY PROGRAMOWANIA W PYTHON

Dzień 13



AGENDA

DAY 13

- type hinty
- dostarczanie argumentów do programu z konsoli
- lambda - funkcje bez nazwy
- generatory, komenda **yield**
- dekoratory

Type hints

Python “sugeruje” typy

Type hints, czyli jak wspomóc IDE

```
def funkcja_a(arg1: str, arg2: dict, arg3: float = 4.0) -> list:  
    arg1.format()  
    arg2.items()  
    arg3.is_integer()  
    return []
```

Argumenty programu

Jak przekazać coś podczas wywołania

argumenty programów

```
python plik.py  
python plik.py opcja1 opcja2 opcja3
```

```
import sys
```

sys.argv – lista argumentów przekazanych do Python

```
['plik.py', 'opcja1', 'opcja2', 'opcja3']
```

sys.argv[0] – ścieżka uruchomionego pliku

sys.argv[1:] – lista parametrów programu (jako stringi)

Lambda

anonimowa funkcja

Lambda definicja

```
x = lambda a: a**2
```

wartość zwracana

```
kwadrat = x(4)
```

argumenty

```
policz = lambda zdanie, znak: zdanie.count(znak)
```

```
print(policz("ała ma kota", 'a'))
```


Lambda

przykład użycia

```
pracownicy = [{"imie": "Waldemar", "wiek": 34, "zarobki": 4000.00},  
              {"imie": "Joanna", "wiek": 23, "zarobki": 3000.00},  
              {"imie": "Adam", "wiek": 50, "zarobki": 6000.00}]
```

```
posortowani = sorted(pracownicy, key=lambda p: p["wiek"])
```

Generator

funkcja imitująca iterable

Generatory

yield

```
def kwadraty_lista(lista_a):  
    temp = []  
    for x in lista_a:  
        temp.append(x ** x)  
  
    return temp
```

```
def kwadraty_generator(lista_a):  
    for x in lista_a:  
        yield x ** 2
```

Zwraca wartość zachowując stan wykonania funkcji, wracając do niego przy następnym wywołaniu.

Generator comprehension

```
kwadraty_lista = [x ** 2 for x in [1, 2, 3, 4, 5]]  
kwadraty_generator = (x ** 2 for x in [1, 2, 3, 4, 5])
```

Context manager

with – zautomatyzuj wejście i wyjście

context managers

przykład – bez menedżera

```
import openpyxl

plik = openpyxl.load_workbook("example.xlsx")

# coś robimy
#...

#
plik.save("example-copy.xlsx")
plik.close()
```

context managers

przykład – definicja menadżera

```
from contextlib import contextmanager
```

```
@contextmanager
```

```
def otworz_excel(plik):
```

```
    xlsx = openpyxl.load_workbook(plik)
```

```
    yield xlsx
```

```
    nowa_nazwa = "example-copy.xlsx"  
    xlsx.save(nowa_nazwa)
```

przygotowanie – wejście, kod wykonany na początku bloku with

yield – zwrócenie obiektu, kod powyżej tego wyrażenia jest wejściowym, poniżej wyjściowym

czyszczenie, zamykanie – kod wykonywany po wyjściu z bloku with

context managers

użycie

```
from contextlib import contextmanager  
import openpyxl
```

```
@contextmanager  
def otworz_excel(plik):  
    xlsx = openpyxl.load_workbook(plik)
```

```
    yield xlsx
```

```
    nowa_nazwa = "example-copy.xlsx"  
    xlsx.save(nowa_nazwa)
```

```
with otworz_excel("example.xlsx") as plik_excel:  
    # coś robimy  
    print(plik_excel.sheetnames)
```

przygotowanie – wejście, kod wykonany na początku bloku with

yield – zwrócenie obiektu, kod powyżej tego wyrażenia jest wejściowym, poniżej wyjściowym

czyszczenie, zamykanie – kod wykonywany po wyjściu z bloku with

Dekorator

dodatkowa funkcjonalność do funkcji

dekoratory

nagłówek dekoratora, jako parametr przyjmuje funkcję

funkcja “udekorowana”,
to ją faktycznie będziemy
wywoływać

```
def logger(func):  
    def inner(*args, **kwargs):  
        print(f"Arguments were: {args}, {kwargs}")  
        return func(*args, **kwargs)  
    return inner
```

dekorator zwraca obiekt
definicji “udekorowanej”
funkcji

wywołanie funkcji
“dekorowanej”, ważne jest,
aby przekazać parametry

- <http://simeonfranklin.com/blog/2012/jul/1/python-decorators-in-12-steps/>



Thanks!!