Politechnika Wrocławska

Programowanie Procduralne
Wykład #4(W#11) – Fizyka

HR EXCELLENCE IN RESEARCH

Janusz Andrzejewski

Wstęp

- Funkcje lambda
- Dopełnienia funkcji
- Funkcje wyższego rzędu
- Dekoratory
  - Proste
  - Zapamiętujące stan
  - Z parametrem
  - Zagnieżdżone

Plan czyli czego się będziemy uczyli?

- W#1 – Podsumowanie
- – Typy mutowalne oraz niemutowalne a przekazywanie argumentów do funkcji.
- – Zasięg zmiennych — zmienne lokalne, nielokalne oraz globalne.
- – Argumenty funkcji: pozycyjne, nazwane, domyślne, rozpakowywanie, słownikowe
- – Funkcje: „first citizen object”.
- – Funkcje: funkcje wyższego rzędu.
- – Funkcje anonimowe.
- – Dopełnienia funkcji.
- – Zastosowania funkcji.
- Laboratoria: od teorii do praktyki czyli rozwiązywanie problemów związanych z omówionymi zagadnieniami na wykładzie.

Funkcje lambda

```
1  def funIdn(x):
2      return x
3
4  y=funIdn({'a':2, 1:2})
5  print("y= ",y)
```

```
y=  {'a': 2, 1: 2}
```

```
1  flIdn=lambda x: x
2  yL=flIdn([1, 2, 3])
3  print("yL=",yL)
```

```
yL= [1, 2, 3]
```

Funkcje lambda:

- Mają postać pojedynczego wyrażenia
- Budowa:
  - Słowo kluczowe lambda
  - Argumenty funkcji oddzielone przecinkami, nie ma nawiasów !!
  - Po dwukropku – definicja funkcji w postaci **JEDNEGO** wyrażenia – nie ma instrukcji return, wartością funkcji jest wartość wyrażenia
  - **Nie może być żadnych instrukcji**

Synonimy funkcji lambda:

1. Funkcje lambda
2. Funkcje anonimowe
3. Wyrażenia lambda
4. Forma lambda (Lambda form)
5. Abstrakcja lambda (Lambda abstractions)
6. Stałe funkcyjne (Function literals)

Zastosowania:

- Jako szybka forma definiowania funkcji
- W tzw. formie *inline*

```
(lambda x: x*3)(2)
```

```
6
```

```
(lambda x, y: x*y)(2,3)
```

```
6
```

```
fn=lambda : print("Cos tam 6")
fn()
```

```
Cos tam 6
```

```
temp={"FnaK": lambda st_f: 273.15 + (st_f - 32) * 5 / 9,
       "CnaK": lambda st_c: 273.15 + st_c}
yf0 = temp["FnaK"](0)
yc0 = temp["CnaK"](0)
print("O F to ",yf0," w K")
print("O C to ",yc0," w K")
```

```
O F to  255.3722222222222  w K
O C to  273.15  w K
```

Uwagi:

- Nie mogą zawierać instrukcji

```
1 def najprostsza():
2     pass
3
4 najprostsza()
```

```
1 najpr = lambda : pass
```

```
  Cell In[57], line 1
    najpr = lambda : pass
                     ^
SyntaxError: invalid syntax
```

```
1 def najprostsza():
2     """Najproszta fun"""
3
4 najprostsza()
```

```
1 najpr = lambda : """Najproszta fun"""
2 print(najpr())
```

```
Najproszta fun
```

```
1 print("Notka:", najpr.__doc__)
```

```
Notka: None
```

Funkcje lambda:

- Można stosować wszystkie sposoby przekazywania argumentów/parametrów takie jak do „zwykłej” funkcji

```
1 (lambda a, b, c: a + b + c)(1, 2, 3)
```

6

```
1 (lambda a, b, c=3: a + b + c)(1, 2)
```

6

```
1 (lambda a, b, c=3: a + b + c)(1, b=2)
```

6

```
1 (lambda *args: sum(args))(1,2,3)
```

6

```
1 (lambda **kwargs: sum(kwargs.values()))(one=1, two=2, three=3)
```

6

```
1 (lambda a, *, b=0, c=0: a + b + c )(1, b=2, c=3)
```

6

Funkcje lambda — podsumowanie

- Za
  - Zwięzła forma
  - Możliwość programowania funkcyjnego
- Przeciw:
  - Problemy z czytelnością
  - Narzucanie funkcjonalnego sposobu myślenia
  - Ciężka składnia ze słowem kluczowym lambda

Domknięcie (Closure) (1/2)

- Domknięcie – jest to funkcja gdzie wszystkie wartości zmiennych z wyjątkiem tych przekazanych przez parametry są wzięte z funkcji otaczającej.

```
1  def fun_otaczająca(x):
2      y = 4
3      def fun_wewn(z):
4          print(f"x = {x}, y = {y}, z = {z}")
5          return x + y + z
6      return fun_wewn
7
8
```

```
1  for i in range(3):
2      domkn = fun_otaczająca(i)
3      print(f"domkn({i+5}) = {domkn(i+5)}")
```

```
x = 0, y = 4, z = 5
domkn(5) = 9
x = 1, y = 4, z = 6
domkn(6) = 11
x = 2, y = 4, z = 7
domkn(7) = 13
```

Domknięcie (2/2)

```
1  from math import sin, pi
2  def fun_factory(x):
3      y = 2*x
4      def fun_wewn(z):
5          #w domknieciu można oczywiście
6          #używać zmiennych lokalnych
7          a=sin(x*z)
8          b=sin(y*z)
9          return a+b
10     return fun_wewn
11
12 domkn=fun_factory(pi/2)
13 print(f"domkn({1}) = {domkn(1)}")
```

```
domkn(1) = 1.0000000000000002
```

Domknięcie & lambda

```python
def fun_otaczająca1(x):
    y = 4
    def fun_wewn(z):
        return x + y + z
    return fun_wewn

def fun_otaczająca2(x):
    y = 4
    return lambda z: x+y+z
```

```python
for i in range(3):
    domkn1 = fun_otaczająca1(i)
    domkn2 = fun_otaczająca2(i)
    print(f"domkn1({i+5}) = {domkn1(i+5)}")
    print(f"domkn2({i+5}) = {domkn2(i+5)}")
```

```
domkn1(5) = 9
domkn2(5) = 9
domkn1(6) = 11
domkn2(6) = 11
domkn1(7) = 13
domkn2(7) = 13
```

Funkcje wyższego rzędu

- Funkcja wyższego rzędu (higher-order function) – w informatyce jest to funkcja, która robi przynajmniej jedną z dwu rzeczy:
  - Zwraca inna funkcję
  - Przyjmuje jako argument inne funkcje
- Przykłady:
  - Domknięcia
  - Pewne wbudowane funkcje w standardowej bibliotece pythona jak: **`sort()`**, **`sorted()`**, **`min()`**, and **`max()`** czy też **`map()`** oraz **`filter()`**
  - Dekoratory

Przykłady:

```
1  def dodaj(x, fun):
2      return fun(x)+x
3
4  y=dodaj(2, lambda x: 2*x)
5  print("y = ",y)
```

```
y =  6
```

```
1  def doTwice1(x, f):
2      return f(f(x))
3
4  def doTwiceMaker1(f):
5      return lambda x: f(f(x))
6
7  def doTwiceMaker2(f):
8      def twoF(x):
9          return f(f(x))
10     return twoF
11
12 kwadrat = lambda x: x*x
```

```
1  doTwice1(2, kwadrat)
```

```
16
```

```
1  y1 = doTwiceMaker1(kwadrat)
2  print(y1(2))
```

```
16
```

```
1  y2 = doTwiceMaker2(kwadrat)
2  print(y2(2))
```

```
16
```

Dekorator:

- Służy do modyfikowania działania funkcji bez modyfikowania jej kodu
- Jest funkcją, której argumentem i zwracanym wynikiem również są funkcje

Dekorator — wstęp

```
1  def dekoruj(funkcja):
2       print("W funkcji dekorującej, dekorując",
3             funkcja.__name__)
4       def funkcja_obudowująca(parametr):
5          print("Wykonuję", funkcja.__name__)
6          wartość=funkcja(parametr)
7          print("Wartość funkjci wynosi ",wartość)
8          return wartość
9       return funkcja_obudowująca
10
11 def moja_funkcja(parametr):
12       return parametr-3
13
14 moja_funkcja = dekoruj(moja_funkcja)
15 y=moja_funkcja(3)
16 print("y = ",y)
```

```
W funkcji dekorującej, dekorując moja_funkcja
Wykonuję moja_funkcja
Wartość funkjci wynosi  0
y =  0
```

Dekorator

```
1  def dekoruj1(funkcja):
2       print("W funkcji dekorującej, dekorując",
3             funkcja.__name__)
4       def funkcja_obudowująca(parametr):
5          print("Wykonuję", funkcja.__name__)
6          wartość=funkcja(parametr)
7          print("Wartość funkjci wynosi ",wartość)
8          return wartość
9       return funkcja_obudowująca
10
11 @dekoruj1
12 def moja_funkcja(parametr):
13      return parametr-3
14
15 print("Zaczynam !")
16 y=moja_funkcja(3)
17 print("y = ",y)
```

```
W funkcji dekorującej, dekorując moja_funkcja
Zaczynam !
Wykonuję moja_funkcja
Wartość funkjci wynosi  0
y =  0
```

Dekorator

- Dekorator składa się z dwóch części — zdefiniowania funkcji, która będzie obudowywać czy dekorować inną, a następnie użycia znaku @ i nazwy tej obudowującej funkcji w linii tuż nad definicją opakowywanej funkcji. Funkcja dekorująca powinna przyjmować w parametrze funkcję i zwracać funkcję
- Składnia przy pomocy @ nazywa się też składnią ciasteczka (pie syntax)

Dekorator — dowolna funkcja (1/2)

```python
def udokumentuj(func):
    def func_obudowuj(*args, **kwargs):
        print('Nazwa funkcji:', func.__name__)
        print('Argumenty pozycyjne:', args)
        print('Argumenty nazwane:', kwargs)
        result = func(*args, **kwargs)
        print('Wynik:', result)
        return result
    return func_obudowuj

@udokumentuj
def dodaj(x, y):
    return x+y

@udokumentuj
def pomnoz(x, y, z):
    return x*y*z

dodaj(2, 3)
pomnoz(1, 2, z=1)
```

```
Nazwa funkcji: dodaj
Argumenty pozycyjne: (2, 3)
Argumenty nazwane: {}
Wynik: 5
Nazwa funkcji: pomnoz
Argumenty pozycyjne: (1, 2)
Argumenty nazwane: {'z': 1}
Wynik: 2
```

Problem z „nazwami” funkcji:

```
1 print
```

```
<function print(*args, sep=' ', end='\n', file=None, flush=False)>
```

```
1 print.__name__
```

```
'print'
```

```
1 dodaj
```

```
<function __main__.udokumentuj.<locals>.func_obudowuj(*args, **kwargs)>
```

```
1 dodaj.__name__
```

```
'func_obudowuj'
```

Dekorator — dowolna funkcja (2/2)

```python
import functools

def udokumentuj(func):
    @functools.wraps(func)
    def func_obudowuj(*args, **kwargs):
        print('Nazwa funkcji:', func.__name__)
        print('Argumenty pozycyjne:', args)
        print('Argumenty nazwane:', kwargs)
        result = func(*args, **kwargs)
        print('Wynik:', result)
        return result
    return func_obudowuj

@udokumentuj
def dodaj(x, y):
    return x+y

dodaj(2, 3)
```

```
Nazwa funkcji: dodaj
Argumenty pozycyjne: (2, 3)
Argumenty nazwane: {}
Wynik: 5
5
```

```python
dodaj
```

```
<function __main__.dodaj(x, y)>
```

```python
dodaj.__name__
```

```
'dodaj'
```

dekorator **`@functools.wraps`** używa funkcji **`functools.update_wrapper()`** do aktualizacji specjalnych atrybutów takich jak **`__name__`** i **`__doc__`**, które są używane w introspekcji.

Funkcje – obiekty

```
1 def prosta(x, y):
2     return 2*x+y
3
4 print(dir(prosta))
```

```
['__annotations__', '__builtins__', '__call__', '__class__', '__closure__', '__code__', '__defaults__',
'__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattr
ibute__', '__getstate__', '__globals__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__kwdef
aults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce_
_', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
```

```
1 prosta.zm1=2 #dodaje atrybut zm1
2 print(dir(prosta))
```

```
['__annotations__', '__builtins__', '__call__', '__class__', '__closure__', '__code__', '__defaults__',
'__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__get__', '__getattr
ibute__', '__getstate__', '__globals__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__kwdef
aults__', '__le__', '__lt__', '__module__', '__name__', '__ne__', '__new__', '__qualname__', '__reduce_
_', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', 'zm1']
```

Dekorator — zachowywanie stanu

```python
import functools

def ileRazy(func):
    @functools.wraps(func)
    def func_obudowuj(*args, **kwargs):
        func.ileRazy+=1
        print("Funkcje ",func.__name__,
              "wywołujesz ",func.ileRazy," razy")
        result = func(*args, **kwargs)
        print('Wynik:', result)
        return result
    func.ileRazy=0
    return func_obudowuj

@ileRazy
def dodaj(x, y):
    return x+y

dodaj(1, 3)
dodaj(1, 4)
```

```
Funkcje  dodaj wywołujesz  1  razy
Wynik: 4
Funkcje  dodaj wywołujesz  2  razy
Wynik: 5

5
```

Dekorator – zachowywanie stanu

```python
import functools

def ileRazy(func):
    @functools.wraps(func)
    def func_obudowuj(*args, **kwargs):
        func.ileRazy+=1
        return func(*args, **kwargs)
    func.ileRazy=0
    return func_obudowuj

@ileRazy
def dodaj(x, y):
    return x+y

s=0
for i in range(10):
   s+=dodaj(i, i*i)
print("Suma = ", s)
```

```
Suma =  330
```

```
---------------------------------------------------------------
AttributeError                            Traceback (most rec
Cell In[129], line 20
     17    s+=dodaj(i, i*i)
     18 print("Suma = ", s)
     19 print("Do obliczenia sumy ",
---> 20       "użyłeś ", dodaj.ileRazy,
     21      "wywołań funkcji")

AttributeError: 'function' object has no attribute 'ileRazy'
```

```python
import functools

def ileRazy(func):
    @functools.wraps(func)
    def func_obudowuj(*args, **kwargs):
        func_obudowuj.ileRazy+=1
        return func(*args, **kwargs)
    func_obudowuj.ileRazy=0
    return func_obudowuj

@ileRazy
def dodaj(x, y):
    return x+y

s=0
for i in range(10):
   s+=dodaj(i, i*i)
print("Suma = ", s)
print("Do obliczenia sumy ",
      "użyłeś ", dodaj.ileRazy,
     "wywołań funkcji")
```

```
Suma =  330
Do obliczenia sumy  użyłeś  10 wywołań funmcji
```

Pomiar czasu wykonywania:

```python
import functools
import time

def czasomierz(func):
    @functools.wraps(func)
    def wrapper_timer(*args, **kwargs):
        start_time = time.perf_counter()    # 1
        value = func(*args, **kwargs)       # 2
        end_time = time.perf_counter()      # 3
        run_time = end_time - start_time    # 4
        print(f"Skończono {func.__name__} w",
              f"{run_time:.3g} sekund")
        return value
    return wrapper_timer

@czasomierz
def strata_czasu(num_times):
    for _ in range(num_times):
        sum([i**2 for i in range(10000)])
```

```
1 strata_czasu(1)
```

```
Skończono strata_czasu w 0.000564 sekund
```

```
1 strata_czasu(999)
```

```
Skończono strata_czasu w 0.471 sekund
```

```
1 strata_czasu(1)
```

```
Skończono strata_czasu w 0.000455 sekund
```

```
1 strata_czasu(999)
```

```
Skończono strata_czasu w 0.476 sekund
```

Spowalniacz

```python
import functools
import time

def spowalniacz(func):
    """Odczekaj 1 sekunde zanin wywołasz funkcje"""
    @functools.wraps(func)
    def wrapper_spowalniacz(*args, **kwargs):
        time.sleep(1)
        return func(*args, **kwargs)
    return wrapper_spowalniacz

@spowalniacz
def zmniejsz(x):
    return x-1

def odliczanie(odkad):
    while odkad > 0:
        print(odkad)
        odkad = zmniejsz(odkad)
    print("Start !")
```

```python
odliczanie(3)
```

```
3
2
1
Start !
```

Dekorator z parametrem (1/4)

```
1  def dwaRazy(func):
2      @functools.wraps(func)
3      def dekorator(*args, **kwargs):
4          func(*args, **kwargs)
5          return func(*args, **kwargs)
6      return dekorator
7
8  @dwaRazy
9  def pozdrowienia():
10     print("Wszystkiego najlepszego !!")
11
12 pozdrowienia()
```

```
Wszystkiego najlepszego !!
Wszystkiego najlepszego !!
```

Dekorator z parametrem (2a/4)

```python
def powtarzaj(ileRazy):
    def dekorator(func):
        @functools.wraps(func)
        def wrapper_powtarzaj(*args, **kwargs):
            for _ in range(ileRazy):
                value = func(*args, **kwargs)
            return value
        return wrapper_powtarzaj
    return dekorator

@powtarzaj(3)
def pozdrowienia():
    print("Wszystkiego najlepszego !!")

pozdrowienia()
```

```
Wszystkiego najlepszego !!
Wszystkiego najlepszego !!
Wszystkiego najlepszego !!
```

Dekorator z parametrem (2b/4)

```python
def powtarzaj(ileRazy):
    def dekorator(func):
        @functools.wraps(func)
        def wrapper_powtarzaj(*args, **kwargs):
            for _ in range(ileRazy):
                value = func(*args, **kwargs)
            return value
        return wrapper_powtarzaj
    return dekorator

"""
@powtarzaj(3)
def pozdrowienia():
    print("Wszystkiego najlepszego !!")
"""
def pozdrowienia():
    print("Wszystkiego najlepszego !!")

pozdrowienia = powtarzaj(3)(pozdrowienia)
pozdrowienia()
```

Dekorator z parametrem (2c/4)

```
1  def fun1(x):
2      def fun2(fun):
3          return fun(x)
4      return fun2
5
6  def nic(x):
7      return x+2
8
9  z=fun1(2)(nic)
10 print(z)
11
12 y=fun1(2)
13 z=y(nic)
14 print(z)
```

```
1  def funA(fun):
2      def funB(x):
3          return fun(x)
4      return funB
5
6  def nic(x):
7      return x+2
8
9  z=funA(nic)(2)
10 print(z)
11
12 y=funA(nic)
13 z=y(2)
14 print(z)
```

Dekorator z parametrem (3/4)

```
1  def powtarzaj(_func=None, *, ileRazy=2):
2      def dekorator(func):
3          @functools.wraps(func)
4          def wrapper_powtarzaj(*args, **kwargs):
5              for _ in range(ileRazy):
6                  value = func(*args, **kwargs)
7              return value
8          return wrapper_powtarzaj
9
10     if _func is None:
11         print("jeden") # dla testu
12         return dekorator
13     else:
14         print("dwa")   # dla testu
15         return dekorator(_func)
```

Dekorator z parametrem (4/4)

```
1 @powtarzaj(ileRazy=3)
2 def pozdrowienia(kto):
3     print("Wszystkiego najlepszego, ", kto," !!")
```

```
jeden
```

```
1 pozdrowienia("Aniu")
```

```
Wszystkiego najlepszego,  Aniu  !!
Wszystkiego najlepszego,  Aniu  !!
Wszystkiego najlepszego,  Aniu  !!
```

```
1 @powtarzaj
2 def hallo():
3     print("Hallo")
```

```
dwa
```

```
1 hallo()
```

```
Hallo
Hallo
```

Zagnieżdżone dekoratory (1/3)

```python
import functools

def udokumentuj(func):
    @functools.wraps(func)
    def func_obudowuj(*args, **kwargs):
        print('Nazwa funkcji:', func.__name__)
        print('Argumenty pozycyjne:', args)
        print('Argumenty nazwane:', kwargs)
        result = func(*args, **kwargs)
        print('Wynik:', result)
        return result
    return func_obudowuj


def dwaRazy(func):
    @functools.wraps(func)
    def dekorator(*args, **kwargs):
        func(*args, **kwargs)
        return func(*args, **kwargs)
    return dekorator
```

Zagnieżdżone dekoratory (2/3)

```
1  @udokumentuj
2  @dwaRazy
3  def dodaj(x, y):
4      return x+y
5
6
7  dodaj(2, 3)
```

```
Nazwa funkcji: dodaj
Argumenty pozycyjne: (2, 3)
Argumenty nazwane: {}
Wynik: 5
5
```

```
1  @dwaRazy
2  @udokumentuj
3  def dodaj1(x, y):
4      return x+y
5
6
7  dodaj1(2, 3)
```

```
Nazwa funkcji: dodaj1
Argumenty pozycyjne: (2, 3)
Argumenty nazwane: {}
Wynik: 5
Nazwa funkcji: dodaj1
Argumenty pozycyjne: (2, 3)
Argumenty nazwane: {}
Wynik: 5
5
```

Zagnieżdżone dekoratory (3/3)

```python
import functools

def parametrized(dec):
    def layer(*args, **kwargs):
        def repl(f):
            return dec(f, *args, **kwargs)
        return repl
    return layer

@parametrized
def dekoratorek(func, n, k):
    # 1 par musi byc funkcją, pozostale
    # moga byc dowolnymi parametrami
    @functools.wraps(func)
    def wraped(*args, **kwargs):
        return n * func(*args, **kwargs)+k
    return wraped
```

```python

@dekoratorek(2, 3)
def function(a):
    return 10 + a

print(function(3))    # 2*(10+3)+3=29

@dekoratorek(3, 2)
def function_again(a):
    return 10 + a

print(function_again(3))    #  3*(10+3)+2
```

```
29
41
```

Dziękuję za uwagę