Politechnika Wrocławska

Programowanie Procduralne
Wykład #3(W#10) — Fizyka

Janusz Andrzejewski

Wstęp

- Parametry i argumenty funkcji:
  - a. pozycyjne i a. nazwane
  - p. domyślne
  - Rozpakowywanie a.
    - Pozycyjnych
    - Nazwanych
  - Dowolna ilość p.
    - Pozycyjnych
    - Nazwanych (słownikowych)
  - Wymuszanie argumentów nazwanych
- Funkcje jako typy pierwszoklasowe
  - Funkcje wewnętrzne i zmienne nielokalne

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.

Najprostsza funkcja

```
1  def najprostsza1():
2      pass
3
4  x=najprostsza1()
5  najprostsza1()
```

```
1  najprostsza1.__doc__
```

```
1  def najprostsza2():
2      """Nijaki komentarz"""
3
4  x=najprostsza2()
5  najprostsza2()
```

```
1  najprostsza2.__doc__
```

```
'Nijaki komentarz'
```

Instrukcja **pass**:

- Instrukcja która nic nie robi, instrukcja pusta
- Używamy jej gdy ze względów składniowych wymagana jest jakaś instrukcja

Dotychczasowe sposoby wywołania funkcji

```
1  def prosta1(x, y):
2      return x+y
3
4  x=prosta1(2, 3)
5  prosta1(2, 3)
6  y=prosta1(2, 3)-prosta1(3, 2)
```

```
1  def prosta2(x, y):
2      print("Suma:", x+y)
3
4  x=prosta2(2, 3)
5  prosta2(2, 4)
```

```
Suma: 5
Suma: 6
```

Wywołanie funkcji

- W instrukcji przypisania – podstawiając wartość funkcji do zmiennej
- W zwykłej instrukcji – pisząc tylko nazwę funkcji wraz z jej argumentami
- W wyrażeniach – ale już nie zawsze :)

```
1  y=prosta2(2, 3)-prosta2(4, 2)
```

```
Suma: 5
Suma: 6
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[28], line 1
----> 1 y=prosta2(2, 3)-prosta2(4, 2)

TypeError: unsupported operand type(s) for -: 'NoneType' and 'NoneType'
```

Argumenty pozycyjne i nazwane

```python
def ArazyBplusC(a, b, c):
    return a * b + c

#argumenty pozycyjne
x = ArazyBplusC(2, 3, 4)
print("Wart: ",x)
```

```
Wart:  10
```

```python
x1 = ArazyBplusC(2, 3, c=4)
x2 = ArazyBplusC(2, b=3, c=4)
x3 = ArazyBplusC(a=2, b=3, c=4)
x4 = ArazyBplusC(c=4, b=3, a=2)
print("Wart:", x1, x2, x3, x4)
```

```
Wart: 10 10 10 10
```

Argumenty nazwane:

- Argumenty do których odwołujemy się poprzez nazwy parametrów, operatora = oraz wartości która ma być przypisana.
- Stosujemy po to aby uniknąć pomyłek przy przekazywaniu
- Mogą występować w dowolnej kolejności
- Najpierw muszą być argumenty pozycyjne, potem mogą być argumenty nazwane

Parametry domyślne

- Parametry z ustaloną wartością
- Używane gdy odpowiadający jej argument nie ma podanej wartości

```
1  def ArazyBplusC(a, b=1, c=1):
2      #par b i c maja wartości
3      #domyślne, par a musi byc
4      #zawsze podany
5      return a * b + c
6
7  x = ArazyBplusC(1, 2, 3)
8  print("Wart:", x)
```

```
Wart: 5
```

```
1  x = ArazyBplusC(2)
2  #par. b i c - domyślne
3  print("Wart:", x)
```

```
Wart: 3
```

```
1  x = ArazyBplusC(2, 3)
2  #par. c -domysłne
3  print("Wart:", x)
```

```
Wart: 7
```

Argumenty nazwane i parametry domyślne

- Można je dowolnie, niezależnie od siebie używać

```
1  def ArazyBplusC(a, b=1, c=1):
2      #par b i c maja wartości
3      #domyślne, par a musi byc
4      #zawsze podany
5      return a * b + c
```

```
1  # można mieszać par domyślne
2  # z argumentami nazwanymi
3  x = ArazyBplusC(2, c=4)
4  #par b - domyslne
5  print("Wart:", x)
```

```
Wart: 6
```

```
1  x = ArazyBplusC(c=2, b=1, a=2)
2  print("Wart:", x)
```

```
Wart: 4
```

Uwaga na parametry domyślne(1/3):

```
1  #uwaga na par domyslne
2  def dodajEL(el, x=[]):
3      x.append(el)
4      print("wyn konc ",x)
5
6  #typowe wywoalanie
7  x = [1, 2]
8  dodajEL(3, x)
```

```
wyn konc  [1, 2, 3]
```

```
1  #pierwsze wyw. fun.
2  dodajEL('a')
```

```
wyn konc  ['a']
```

```
1  #drugie wyw. fun.
2  dodajEL('b')
```

```
wyn konc  ['a', 'b']
```

W przypadku gdy wartością parametru domyślnego jest obiekt mutowalny – wartość parametru domyślnego może się zmieniać ponieważ wartość ta jest ustalana tylko przy definiowaniu funkcji.

Uwaga na parametry domyślne(2/3):

```python
#uwaga na par domyslne
#Rozwiazanie problemu
def dodajEL(el, x=None):
    if x is None:
        x = []
    x.append(el)
    print("wyn konc ",x)
```

```python
#pierwsze wyw. fun.
dodajEL('a')
#drugie wyw. fun.
dodajEL('b')
```

```
wyn konc  ['a']
wyn konc  ['b']
```

```python
#uwaga na par domyslne
#Rozwiazanie problemu
#drugi sposób
def dodajEL(el, x=None):
    if not x:
        x = []
    x.append(el)
    print("wyn konc ",x)

#pierwsze wyw. fun.
dodajEL('a')
#drugie wyw. fun.
dodajEL('b')
```

```
wyn konc  ['a']
wyn konc  ['b']
```

Uwaga na parametry domyślne(3/3):

```
1  #uwaga na par domyslne
2  #Rozwiazanie problemu
3  #drugi sposób
4  def dodajEL(el, x=None):
5      if not x:
6          x = []
7      x.append(el)
8      print("wyn konc ",x)
```

```
1  dodajEL('a', 0)
```

```
wyn konc  ['a']
```

```
1  dodajEL('a', 1)
```

```
---------------------------------------------------------------
AttributeError                            Traceback (mo
Cell In[5], line 1
----> 1 dodajEL('a', 1)

Cell In[1], line 7, in dodajEL(el, x)
      5 if not x:
      6     x = []
----> 7 x.append(el)
      8 print("wyn konc ",x)

AttributeError: 'int' object has no attribute 'append'
```

```
1  #uwaga na par domyslne
2  #Rozwiazanie problemu
3  def dodajEL(el, x=None):
4      if x is None:
5          x = []
6      x.append(el)
7      print("wyn konc ",x)
8
9  dodajEL('a', 0)
```

```
---------------------------------------------------------------
AttributeError                            Traceback (mo
Cell In[8], line 9
      6     x.append(el)
      7     print("wyn konc ",x)
----> 9 dodajEL('a', 0)

Cell In[8], line 6, in dodajEL(el, x)
      4 if x is None:
      5     x = []
----> 6 x.append(el)
      7 print("wyn konc ",x)

AttributeError: 'int' object has no attribute 'append'
```

Rozpakowywanie argumentów

```
1  def ArazyBplusC(a, b, c):
2      return a * b + c
3
4  #typowe wywolanie
5  x = ArazyBplusC(1, 2, 3)
6  print("Wart", x)
```

```
Wart 5
```

```
1  kr = (1, 2, 3)
2  #argumenty pozycyjne
3  xkr = ArazyBplusC(*kr)
4  print("Wart", xkr)
```

```
Wart 5
```

```
1  li = [1, 2, 3]
2  #argumeny pozycyjne
3  xli = ArazyBplusC(*li)
4  print("Wart", xli)
```

```
Wart 5
```

```
1  sl = {"c":3, "a":1, "b":2}
2  #argumenty nazwane
3  xsl = ArazyBplusC(**sl)
4  print("Wart", x)
```

```
Wart 5
```

```
1  kr = (1, 2)
2  sl = {"c":3}
3  #argumenty mieszane
4  xks = ArazyBplusC(*kr, **sl)
5  print("Wart:", xks)
```

```
Wart: 5
```

Dowolna ilość parametrów pozycyjnych(1/5):

Parametr poprzedzony * (gwiazdką) użyty w definicji funkcji oznacza krotkę zawierającą nieokreśloną (dowolną) liczbę parametrów pozycyjnych. Argumenty przekazane do takiej funkcji są „pakowane” do krotki.

```
1  #dowolna(>0) ilość
2  #par. pozycyjnych
3  def suma(x, *s):
4      pom = x
5      for i in s:
6          pom += i
7      return pom
```

```
1  x = suma(1, 2)
2  print("Wart:",x)
```

```
Wart: 3
```

```
1  x = suma("a", "b", "c")
2  print("Wart:",x)
```

```
Wart: abc
```

```
1  x = suma([1], ["b"], ["c"])
2  print("Wart:",x)
```

```
Wart: [1, 'b', 'c']
```

```
1  x = suma(1)
2  print("Wart:",x)
```

```
Wart: 1
```

```
1  x = suma()
2  print("Wart:",x)
```

```
---------------------------------------------------------------------
TypeError                                 Traceback (most rece
Cell In[39], line 1
----> 1 x = suma()
      2 print("Wart:",x)

TypeError: suma() missing 1 required positional argument: 'x'
```

Dowolna ilość parametrów pozycyjnych(2/5):

```python
#dowolna(>0) ilość
#par. pozycyjnych
def suma(x, *s):
    pom = x
    for i in s:
        pom += i
    return pom
```

```python
kr=[1]
x=suma(kr, [1])
print("Wart:",x)
print("A co z kr?",kr)
```

```
Wart: [1, 1]
A co z kr? [1, 1]
```

```python
def suma(x, *s):
    #wersja uproszczona
    for i in s:
        x = x + i
    return x

x = suma(1, 2, 3, 4)
print("Wart:",x)
```

```
Wart: 10
```

```python
kr=[1]
x=suma(kr, [1])
print("Wart:",x)
print("A co z kr?",kr)
```

```
Wart: [1, 1]
A co z kr? [1]
```

Dowolna ilość parametrów pozycyjnych(3/5):

```
1  def suma(*s):
2      x = 0
3      for i in s:
4          x += i
5      return x
6
7  x=suma()
8  print("Wart1:",x)
9  x = suma(1, 2)
10 print("Wart2:",x)
```

```
Wart1: 0
Wart2: 3
```

```
1  x = suma("a", "b")
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recen
Cell In[60], line 1
----> 1 x = suma("a", "b")

Cell In[59], line 4, in suma(*s)
      2 x = 0
      3 for i in s:
----> 4     x += i
      5 return x

TypeError: unsupported operand type(s) for +=: 'int' and 'str'
```

Dowolna ilość parametrów pozycyjnych(4/5):

```
1  def suma(*s):
2      if s is None:
3          return None
4      else:
5          x = s[0]
6      for i in s[1:]:
7          x += i
8      return x
9
10 x = suma()
11 print("Wart:",x)
```

```
---------------------------------------
IndexError
Cell In[62], line 10
      7         x += i
      8     return x
---> 10 x = suma()
     11 print("Wart:",x)

Cell In[62], line 5, in suma(*s)
      3     return None
      4 else:
----> 5     x = s[0]
      6 for i in s[1:]:
      7     x += i

IndexError: tuple index out of range
```

Dowolna ilość parametrów pozycyjnych(5/5):

```python
def suma(*s):
    print("Czym jest s ?", s)
    print("Typ s:",type(s))

x = suma()
print("Wart:",x)
```

```
Czym jest s ? ()
Typ s: <class 'tuple'>
Wart: None
```

```python
def suma(*s):
    if not s:
        return None
    else:
        x = s[0]
    for i in s[1:]:
        x += i
    return x
```

```python
x = suma()
print("Wart:",x)
x = suma(1, 2)
print("Wart:",x)
x = suma('1', '2')
print("Wart:",x)
x = suma([1], [2], [3])
print("Wart:",x)
```

```
Wart: None
Wart: 3
Wart: 12
Wart: [1, 2, 3]
```

Dowolna ilość parametrów nazwanych(1/2):

Parametr poprzedzony ** (dwie gwiazdki) użyty w definicji funkcji oznacza słownik zawierający nieokreśloną (dowolną) liczbę parametrów nazwanych. Klucz w tym słowniku jest nazwą parametru a wartość słownika jest wartością parametru.

```
1  def slow(**sl):
2      print("Par. a = ", sl["a"])
3      print("Par. b = ", sl["b"])
```

```
1  slow(a=3, b="cos")
```

```
Par. a =  3
Par. b =  cos
```

```
1  slow(a=3, b="cos", c=3-2.0)
```

```
Par. a =  3
Par. b =  cos
```

Dowolna ilość parametrów nazwanych(2/2):

```
1  def iloczyn(x, **sl):
2      for key, value in sl.items():
3          x *= value
4      return x
```

```
1  x=iloczyn(2, i2=3, i5=5)
2  print("Wart:",x)
3  x=iloczyn(1.0, i2=3, i5=5)
4  print("Wart:",x)
```

```
Wart: 30
Wart: 15.0
```

```
1  x=iloczyn(2, 2)
2  print("Wart:",x)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent c
Cell In[8], line 1
----> 1 x=iloczyn(2, 2)
      2 print("Wart:",x)

TypeError: iloczyn() takes 1 positional argument but 2 were given
```

*args oraz **kwargs

- Argument *args „rozbija” krotkę args na argumenty pozycyjne oddzielone przecinkami
- Parametr *args zbiera argumenty pozycyjne w jedną **krotkę** o nazwie args.
- Argument **kwargs „rozbija” słownik kwargs na argumenty nazwane oddzielone przecinkami
- Parametr **kwargs zbiera argumenty nazwane w jeden słownik o nazwie kwargs.

UWAGA: Nazwy args oraz kwargs są tylko nazwami zwyczajowymi i są powszechnie stowowaną konwencją

Kolejność parametrów

1) Parametry pozycyjne
2) Parametry *args – opcjonalne parametry pozycyjne
3) Parametry **kwargs – opcjonalne parametry nazwane

```
1  # poprawna kolejność argumentów
2  def moja_funkcja(a, b, *args, **kwargs):
3      pass
```

```
1  #mozliwe sposoby wywołania
2  #funkcji moja_funkcja
3  moja_funkcja(1, 2)
4  moja_funkcja(1, 2, 3)
5  moja_funkcja(1, 2, 3, 4)
6  moja_funkcja(1, 2, 3, 4, x=3)
7  moja_funkcja(1, 2, 3, 4, x=3, yy="aa")
8  moja_funkcja(1, 2, x=3)
```

Wymuszenie argumentów wyłącznie nazwanych

Domyślne wartości parametrów funkcji które zapisane są po znaku * (gwiazdka) można zmienić jedynie poprzez argumenty nazwane (czyli przy wywołaniu funkcji)

```
1  def ParNazw(x, *, x0=0):
2      s=x0
3      for i in x:
4          s+=i
5      return s
```

```
1  li=[1, 2, 3]
2  s0=ParNazw(li)
3  print("s0:",s0)
4  s1=ParNazw(li, x0=3)
5  print("s1:",s1)
```

```
s0: 6
s1: 9
```

```
1  LI=["a", "b", "c"]
2  s2=ParNazw(LI, x0="")
3  print("s2:",s2)
```

```
s2: abc
```

Funkcje - typ pierwszoklasowy(1/5):

- Cechy typu pierwszoklasowego („first citizen”):
  - Można podstawić pod zmienną
  - Można umieszczać w argumentach funkcji
  - Można zwracać jako wartość funkcji
- W większości języków programowania (we wszystkich ?) typami pierwszoklasowymi są typy liczbowe
- W Pythonie do typu pierwszoklasowego należą ponadto funkcje oraz obiekty (czyli tak naprawdę wszystko – bo wszystko jest obiektem :))

Funkcje - typ pierwszoklasowy(2/5): funkcja jako zwykła zmienna

```
1  def dodaj(x, y):
2      return x + y
3
4  print("Typ funkcji:", type(dodaj))
```

```
Typ funkcji: <class 'function'>
```

```
1  fun=dodaj
2  f=fun(1, 2)
3  d=dodaj(1, 2)
4  print("Suma:", f, d)
```

```
Suma: 3 3
```

```
1  print("id fun-zmienna ",id(fun))
2  print("id org. funkcji",id(dodaj))
```

```
id fun-zmienna  140222578255296
id org. funkcji 140222578255296
```

Funkcje - typ pierwszoklasowy(3/5): funkcje jako argument funkcji

```python
def iloczyn(x):
    il=1
    for i in x:
        il *= i
    return il

def suma(x):
    su=0
    for i in x:
        su+= i
    return su

def glowna(x, fun):
    #jakis operacje na x
    x *= 2
    return fun(x)


x = (1, 2, 3)
s = glowna(x, suma)
i = glowna(x, iloczyn)
print("Suma:", s)
print("Iloczyn:",i)
```

```
Suma: 12
Iloczyn: 36
```

```python
def ilocz(x, y):
    return x * y

def suma(x, y):
    return x + y

def glowna(lista, fun,*, x0=0):
    pom=x0
    for i in lista:
        pom = fun(i, pom)
    return pom

lista = [1, 2, 3, 4]
sum = glowna(lista, suma)
ilo = glowna(lista, ilocz, x0=1)
print("Suma:", sum)
print("Ilocz:", ilo)
```

```
Suma: 10
Ilocz: 24
```

Funkcje - typ pierwszoklasowy(4/5): funkcje jako elementy struktur

```python
def iloczyn(x):
    il=1
    for i in x:
        il *= i
    return il

def suma(x):
    su=0
    for i in x:
        su+= i
    return su

def glowna1(x, LIfun):
    return LIfun[0](x), LIfun[1](x)
#Krotka o elemenatch które
#są funkcjami
li = (iloczyn, suma)
x = (1, 2, 3, 4)
y = glowna1(x, li)
print("Ilo:", y[0])
print("Sum:", y[1])

```

```
Ilo: 24
Sum: 10
```

```python
def glowna2(x, LIfun):
    li=[]
    for f in LIfun:
        li.append(f(x))
    return tuple(li)
```

```python
y = glowna2(x, li)
print("Wyn1:", y)
y = glowna2(x, li[1:])
print("Wyn2:", y)
y = glowna2(x, li[:-1])
print("Wyn2:", y)
```

```
Wyn1: (24, 10)
Wyn2: (10,)
Wyn2: (24,)
```

Funkcje - typ pierwszoklasowy(5/5): funkcje jako wartość funkcji

```
1  def ktoraFun(ilePar):
2      #funkcje wewnętrzne
3      def fun2p(x, y):
4          return x+y
5
6      def fun3p(x, y, z):
7          return x * y + z
8
9      if ilePar == 2:
10         return fun2p
11     elif ilePar == 3:
12         return fun3p
13     else:
14         print("Niewłasciwy
```

```
16 x=(1, 2, 3, 4)
17 ilePar=int(input("Podaj ile par ma fun:"))
18 fun=ktoraFun(ilePar)
19 print("Wyn:",fun(*x[:ilePar]))
20 ilePar=int(input("Podaj ile par ma fun:"))
21 fun=ktoraFun(ilePar)
22 print("Wyn:",fun(*x[:ilePar]))
```

```
Podaj ile par ma fun: 2
Wyn: 3
Podaj ile par ma fun: 3
Wyn: 5
```

Zmienne nielocalne(1/3):

```python
parJaki=3
def funZewn1(x):

    print("Zew1:parJaki:",parJaki)
    def funWew1(x):

        print("Wew1. parJaki:",parJaki)
        print("arg wew:",x)

    funWew1(x)

funZewn1(1)
```

```
Zew1:parJaki: 3
Wew1. parJaki: 3
arg wew: 1
```

```python
parJaki=3
def funZewn2(x):
    parJaki=5
    print("Zew2:parJaki:",parJaki)
    def funWew2(x):

        print("Wew2. parJaki:",parJaki)
        print("arg wew:",x)

    funWew2(x)

funZewn2(1)
```

```
Zew2:parJaki: 5
Wew2. parJaki: 5
arg wew: 1
```

Zmienne nielocalne(2/3):

```
1  parJaki=3
2  def funZewn3(x):
3      parJaki=5
4      print("Zew2:parJaki:",parJaki)
5      def funWew3(x):
6          nonlocal parJaki
7          parJaki +=2
8          print("Wew2. parJaki:",parJaki)
9          print("arg wew:",x)
10
11     funWew3(x)
12
13 funZewn3(1)
```

```
Zew2:parJaki: 5
Wew2. parJaki: 7
arg wew: 1
```

Zmienne nielocalne(3/3)

```
1  foo = 0 #1
2  def outer():
3      foo = 5 #2
4      def middle(
5          foo = 1
6          def inr
7              nor
8              #Kt
9              # #
10             foc
11             pri
12         inner()
13     middle()
14 outer()
```

```
1  foo = 0 #1 <- ✖ NIE TA
2  def outer():
3      foo = 5 #2 <- ✖ NIE TA
4      def middle():
5          foo = 10 #3 <- ○ TAK TA
6          def inner():
7              nonlocal foo # Tutaj
8              #Która zmienna sie zmieni?
9              # #3 sie zmieni
10             foo += 1
11             print(foo) # 11
12         inner()
13     middle()
14 outer()
```

Zmienne `global` a funkcje wewnętrzne

```
foo = 0 #1
def outer():
    foo = 5 #
    def midd
        foo =
        def
            g
            #
            #
            .
            l
        inne
    middle()
outer()
```

```
foo = 0 #1 <- ○ TAK TA
def outer():
    foo = 5 #2 <- ✖ NIE TA
    def middle():
        foo = 10 #2 <- ✖ NIE TA
        def inner():
            global foo # Tutaj
            #Która zmienna sie zmieni?
            # #1 ta sie zmieni
            foo += 1
            print(foo) # 1
        inner()
    middle()
outer()
```

```
1
```

Dziękuję za uwagę