Faculty of Geodesy
and Cartography
WARSAW UNIVERSITY OF TECHNOLOGY

Informatyka Geodezyjna – wykłady/ćwiczenia, rok akad. 2021/2022

Wyk. 6: Python – funkcje w Pythonie

Kinga Węzka
kinga.wezka@pw.edu.pl
Katedra Geodezji i Astronomii Geodezyjnej

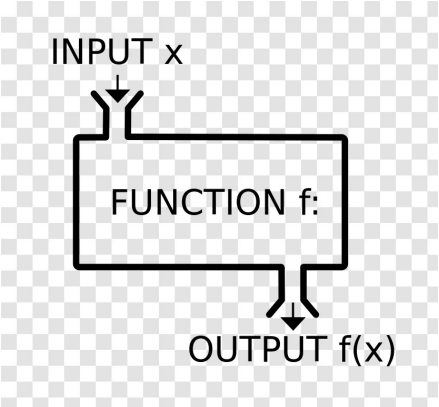
Warsaw University
of Technology

Plan

1. Funkcje w Pythonie – deklaracja i wywołanie
2. Reguły dotyczące zakresów: Zmienne globalne, lokalne, zasięg (przysłanianie)
   - Rozwiązywanie konfliktów w zakresie nazw w Pythonie
   - Reguły dotyczące zakresów
3. Przekazywanie argumentów i wywołania funkcji
   - Kolejność argumentów w funkcji
   - Środowisko wykonawcze funkcji – podsumowanie
4. Dokumentacja funkcji (docstring)
5. Adnotacje w funkcjach
6. Funkcje, moduły - importowanie – `if __name__ == '__main__'`
7. *Materiał dodatkowy
   - Generator `yield`
   - Funkcja anonimowa: `lambda`
   - Funkcje rekurencyjne
8. Podsumowanie

Funkcje

- Funkcje są podstawowym sposobem unikania powtarzalności kodu w Pythonie. Parametryzacja kodu w funkcji oznacza, że mamy tylko jedną kopię kodu operacji do uaktualnienia w przyszłości.
- Dobrze zdefiniowane funkcje mogą być wielokrotnie wywoływane w wielu programach.
- Ciało funkcji (kod zagnieżdżony wewnątrz instrukcji definicji funkcji) wykonywane jest, kiedy funkcja jest później wywoływana za pomocą wyrażenia wywołania. Ciało funkcji z każdym wywołaniem funkcji wykonywane jest na nowo.
- W pythonie niektóre funkcje są nazywane metodami (metody to funkcje w klasach).



- Deklaracja funkcji: `def ... return`:

```
def my_function(par1, par2):
    wynik = par1 + par2
    return(wynik)
```

- Wywołanie funkcji:

```
wynik1 = my_function(arg1, arg2)
```

- Zmiana nazwy funkcji:

```
# przypisanie obiektu
inna_nazwa = my_function
# wywołanie my_function
wynik2 = inna_nazwa(arg1, arg2)
```

- Słowo klucz (ang. *keywords*): `def` rozpoczyna instrukcje deklaracji funkcji. Instrukcja `def` tworzy obiekt i przypisuje go do nazwy: `my_function`.
- Słowo klucz: `return` zwraca wynik funkcji. Instrukcja ta przesyła wynikowy obiekt z powrotem do wywołującego.
- Wyrażenia w nawiasach są nazywane **parametrami** (ang. *formal parameter*) – `(par1, par2)` – są one definicją zmiennej w deklaracji funkcji:
- Podczas wywołania funkcji w miejsce parametrów podajemy **argumenty**, (ang. *actual parameter*) – `(arg1, arg2)` – są to wartości przekazane do parametru funkcji.

In[1]:

```
1  def dodawanie(par1, par2):
2      wynik = par1 + par2
3      return(wynik)
4
5  arg1, arg2 = 4, 3
6  # wywołanie 1
7  wynik1 = dodawanie(arg1, arg2)
8  print('wynik1 = ', wynik1)
9
10 # wywołanie 2
11 wynik2 = dodawanie(2, 8)
12 print('wynik2 = ', wynik2)
```

Out[1]:

```
1  wynik1 = 7
2  wynik2 = 10
```

- Podczas wywołania funkcji w miejsce parametrów podajemy **argumenty**, czyli wartości dla których chcemy znać wynik funkcji.
- Funkcja domyślnie zwraca obiekt `None`, kiedy sterowanie wyjdzie poza jej ciało bez trafienia `return`.

In[2]:

```
1  def dzialania(par1, par2):
2      dodaj  = par1 + par2
3      odejmij = par1 - par2
4      return(dodaj, odejmij)
5
6  arg1, arg2 = 4, 3
7  # wywołanie 1
8  wynik1 = my_function(arg1, arg2)
9  print('wynik1 = ', wynik1)
10 # wywołanie 2
11 d, o = my_function(2, 8)
12 print('wynik2 = ', d, o)
```

Out[2]:

```
1  wynik1 = (7, 1)
2  wynik2 = 10, -6
```

- Podczas wywołania funkcji w miejsce parametrów podajemy **argumenty**, czyli wartości dla których chcemy znać wynik funkcji.
- Funkcja domyślnie zwraca obiekt `None`, kiedy sterowanie wyjdzie poza jej ciało bez trafienia `return`.
- Funkcja może zwracać kilka wartości, są one zdefiniowane za pomocą **krotki**, więc wynik funkcji należy przypisać do krotki:

- Przypisanie zmiennej w środku funkcji tworzy **zmienną lokalną**. Nazywamy ją lokalną, ponieważ istnieje tylko w bloku kodu funkcji i nie można używać jej poza nią.
- Zmienne x, z są globalne, ponieważ są przypisane poza funkcją (poziom modułu),

In[3]:

```
1 z = 8          # Zmienna globalna
2 def funkcja(start):
3     a = 4      # Zmienna lokalna
4     wynik = start + a
5     return(wynik)
6
7 funkcja(z) # wywołanie funkcji
8 print(a)   # brak dostępu do zmiennej a na poziomie globalnym
```

Out[3]:

```
1 NameError: name 'a' is not defined
```

- Zmienna a jest **globalna**, ponieważ została wymieniona w instrukcji `global`. Bez instrukcji `global` zmienna a byłaby lokalna ze względu na miejsce przypisania (w funkcji).
- **Zmienna a będzie zmienną globalną tylko wtedy gdy wywołamy funkcję**

In[4]:

```
x = 99          # Zmienna z zakresu globalnego -nieużywana
z = 88          # Zmienna z zakresu globalnego -używana
def funkcja1(start):
    global a    # global: zmienia a z lokalnej na globalną
    a = 4
    wynik = a + start
    return(a, z, wynik)
print(funkcja1(z)) # wywołanie funkcji
print(a)           # dostęp do zmiennej a na poziomie globalnym
```

Out[4]:

```
(4, 88, 92)
4
```

In[5]:

```
def tester(start):
    state = start # każde wywołanie otrzymuje nową zmienną state
    def nested(label):
        nonlocal state # Pamięta state z zakresu funkcji
        print(label, state)
        state += 1 # Można zmienić, jeśli nonlocal
    return(nested)

funkcja = tester(98)
funkcja('zielone')
```

Out[5]:

```
zielone 98
```

In[6]:

```
def tester(start):
    state = start # każde wywołanie otrzymuje nową zmienną state
    def nested(label):
        #nonlocal state # Pamięta state z zakresu funkcji
        print(label, state)
        state += 1 # tylko jeśli nonlocal
    return(nested)

funkcja = tester(98)
funkcja('zielone')
```

Out[6]:

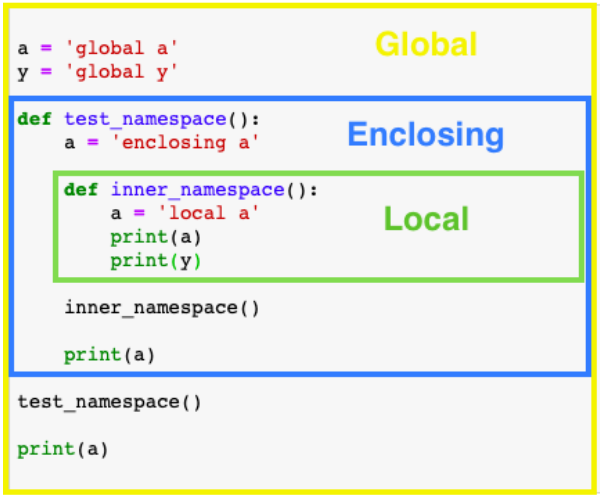
```
UnboundLocalError: local variable 'state' referenced before
    assignment
```

W Pytonie wyróżniamy dwie deklaracje przestrzeni nazw:

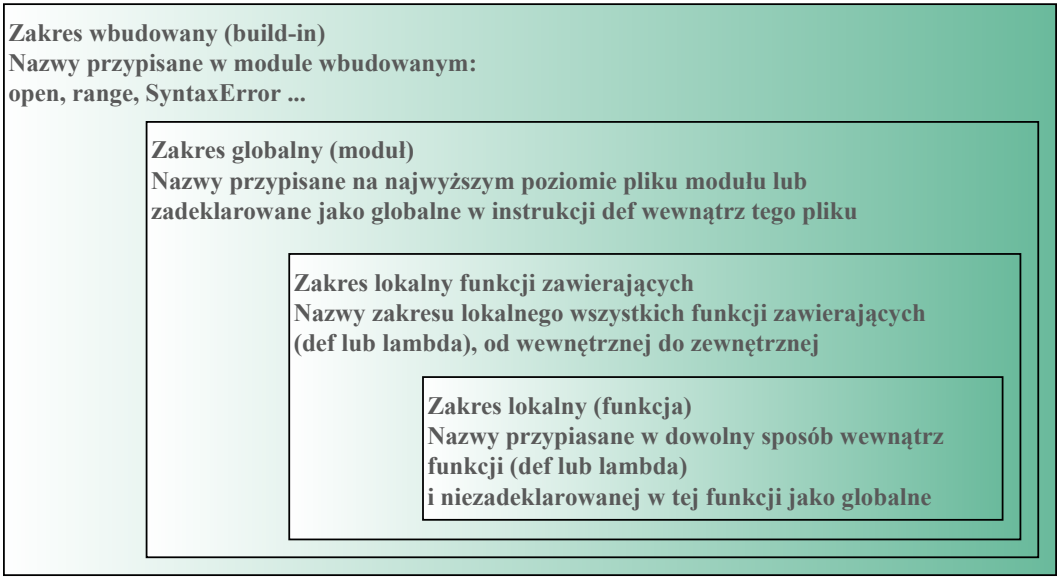
- `global wyrazenie` – instrukcja `global` - jeśli użyje się jej wewnątrz klasy lub instrukcji definicji funkcji, to powoduje ona, że wszystkie wystąpienia ciągu `nazwa` w wybranym kontekście będą traktowane jak referencje do zmiennej globalnej (poziomu modułu) zmiennej `nazwa` — niezależnie od tego, czy do zmiennej nazwa zostanie przypisana wartość i czy zmienna nazwa jest już zdefiniowana.
- `nonlocal wyrazenie` – instrukcja `nonlocal` - jeśli użyje się jej wewnątrz zagnieżdżonej funkcji, to wszystkie wystąpienia ciągu `nazwa` w wybranym kontekście będą traktowane jak referencje do zmiennej lokalnej o tej nazwie w bieżącym zasięgu funkcji okalającej - niezależnie od tego, czy zmiennej nazwa została przypisana wartość, czy nie. **Dzięki tej instrukcji zagnieżdżone instrukcje def mają dostęp do odczytu i zapisu zmiennych znajdujących się w funkcjach je zawierających.** `nonlocal` działa prawie jak `global`, jednak ma zastosowanie do zmiennych w zakresie lokalnym zawierającej instrukcji `def`, a nie do zmiennych z modułu zawierającego.

Rozwiązywanie konfliktów w zakresie nazw w Pythonie czasami nazywane jest regułą **LEGB** — od angielskich nazw kolejnych zakresów (Lutz, 2011, Rozdział 17. Zakresy):

- Kiedy wewnątrz funkcji użyjemy nazwy bez kwalifikatora, Python przeszuka cztery zakresy -— lokalny (**L, od ang. *local***), następnie zakres lokalny instrukcji `def` lub wyrażeń lambda zawierających daną funkcję (**E, od ang. *enclosing***), później globalny (**G, od ang. *global***), a na końcu wbudowany (**B, od ang. *built-in***), zatrzymując się w pierwszym miejscu, w którym nazwa ta zostanie odnaleziona. Jeśli nazwa nie zostanie znaleziona, Python zgłasza błąd. Nazwy muszą być przypisane przed pierwszym użyciem.
- Kiedy przypisujemy zmienną wewnątrz funkcji (zamiast odnieść się do niej w wyrażeniu), Python zawsze tworzy lub modyfikuje tę zmienną w zakresie lokalnym, o ile nie została ona w tej funkcji zadeklarowana jako globalna lub nielokalna.
- Kiedy przypisujemy zmienną poza jakąkolwiek funkcją (na przykład na najwyższym poziomie pliku modułu czy w sesji interaktywnej), zakres lokalny jest tym samym co zakres globalny — przestrzenią nazw modułu.




```
local a
global y
enclosing a
global a
```


Zakres wbudowany (build-in)
Nazwy przypisane w module wbudowanym:
open, range, SyntaxError ...
Zakres globalny (moduł)
Nazwy przypisane na najwyższym poziomie pliku modułu lub
zadeklarowane jako globalne w instrukcji def wewnątrz tego pliku
Zakres lokalny funkcji zawierających
Nazwy zakresu lokalnego wszystkich funkcji zawierających
(def lub lambda), od wewnętrznej do zewnętrznej
Zakres lokalny (funkcja)
Nazwy przypiasane w dowolny sposób wewnątrz
funkcji (def lub lambda)
i niezadeklarowanej w tej funkcji jako globalne

Reguły dotyczace zakresów - na podstawie (Lutz, 2011, Rozdział 17. Zakresy)

- **Moduł zawierający funkcję jest zakresem globalnym**. Każdy **moduł** (plik zawierający funkcje) jest zakresem globalnym, czyli przestrzenią nazw, w której znajdują się zmienne tworzone (przypisane) na najwyższym poziomie pliku modułu. Zmienne globalne stają się atrybutami obiektu modułu dla świata zewnętrznego, jednak wewnątrz samego modułu mogą być używane jako proste zmienne.
- **Zakres globalny rozciąga się jedynie na jeden plik**. Zmienne przypisane na najwyższym poziomie pliku są globalne jedynie w odniesieniu do kodu tego jednego pliku.
- **Każde wywołanie funkcji tworzy nowy zakres lokalny**. Za każdym razem, gdy wywołujemy funkcję, tworzymy nowy zakres lokalny – to znaczy przestrzeń nazw, w której znajdują się zmienne utworzone w tej funkcji. Każdą instrukcję `def` (i każde wyrażenie `lambda`) można sobie wyobrazić jako tworzącą nowy zakres lokalny. Ponieważ Python pozwala na wywoływanie funkcji przez nie same w celu wykonania pętli (**rekurencja**), zakres lokalny tak naprawdę odpowiada wywołaniu funkcji. Krótko mówiąc, każde wywołanie tworzy nową lokalną przestrzeń nazw.

- **Przypisane nazwy są lokalne, o ile nie zostaną zadeklarowane jako globalne lub nielokalne**. Domyślnie wszystkie nazwy przypisane wewnątrz definicji funkcji umieszczane są w zakresie lokalnym (przestrzeni nazw powiązanej z wywołaniem funkcji). Jeśli potrzebujemy przypisać zmienną istniejącą na najwyższym poziomie modułu zawierającego funkcję, możemy to zrobić, deklarując to w instrukcji `global` wewnątrz funkcji. Jeśli potrzebujemy przypisać zmienną istniejącą w instrukcji `def` zawierającej inną funkcję, od Pythona 3.0 możemy to zrobić, deklarując to w instrukcji `nonlocal`.
- **Wszystkie pozostałe nazwy są lokalne dla zakresu zawierającego, globalne lub wbudowane**. Nazwy, które nie zostały przypisane wewnątrz definicji funkcji, są lokalne dla zawierającego je zakresu (w przypadku instrukcji `def` zawierającej tę funkcję), globalne (należące do przestrzeni nazw modułu) lub wbudowane (znajdujące się w udostępnianym przez Pythona module builtins).

Podstawy przekazywania argumentów (Lutz, 2011, p.465), (Cena et al., 2016) Można definiować funkcje ze zmienną liczbą argumentów. Argumenty do funkcji, można przekazywać w następujący sposób:

- argument **pozycyjny** (ang.*positional*);
- argument **domyślny** z wartościami domyślnymi (ang.*default, keyword*) ;
- argument **spakowany** (***args**) – przydaje się gdy nie wiemy, ile argumentów przyjmie funkcja. (***args**) grupuje (w postaci listy) wszystkie argumenty, które należy później rozpakować;
- argument **spakowany** (****kwargs**) – (ang. *keword argument* służy do zbiorczej reprezentacji wszystkich niewymienionych jawnie na liście parametrów argumentów postaci `nazwa=wartość`. Argumenty tego typu są dostępne w ciele funkcji w postaci słownika.

In[8]:

```
def my_function(par1, par2 = 3):
    return(par1 + par2)

x = 6
y = 10
wynik1 = my_function(7)
wynik2 = my_function(1, 5)
wynik3 = my_function(1, par2 = y)
wynik4 = my_function(1, par2 = 10)
```

Out[8]:

```
wynik1 = 10
wynik2 = 6
wynik3 = 11
wynik4 = 11
```

- Pojedyńczy symbol *arg rozpakuje sekwencje/kolekcje do zwykłych argumentów:

In[9]:

```
1 def funk(a, b):
2     return a + b
3
4 values = [1, 2]
5 s = funk(*values)
6 s = funk(1, 2)
```

Out[9]:

```
1 s1 = 3
```

In[10]:

```
1  def funk(*val):
2      w = []; m = []
3      for x in val:
4          w.append(x + x)
5          m.append(x * x)
6      return(w, m)
7
8  values = [1, 2]
9  s1 = funk(*values)
10 s2 = funk(*[1, 2, 3])
```

Out[10]:

```
1 s1 = ([2, 4], [1, 4])
2 s2 = ([2, 4, 6], [1, 4, 9])
```

- Podwójny symbol **kwarg rozpakuje sekwencje – słownik do zwykłych argumentów:

In[11]:

```
def funk(**val):
    w = []; m = []
    for x in val.values():
        w.append(x+x)
        m.append(x*x)
    return(w, m)
values = { 'a': 1, 'b': 2 }
s1 = funk(**values)
s2 = funk(**{ 'a': 1, 'b': 2, 'c': 3 })
s3 = funk(a = 4, b = 3, c = 6, d = 6)
```

Out[11]:

```
s1 = ([2, 4], [1, 4])
s2 = ([2, 4, 6], [1, 4, 9])
s3 = ([8, 6, 12, 12], [16, 9, 36, 36])
```

- Argumenty spakowane można stosować wspólnie:, n:

In[12]:

```
def funk(*val, **kval):
    pow_val = [i**2 for i in val] # elem. listy do kwadratu
    sum_pow_val = sum(pow_val) # suma wszystkich elementów
    w = []
    for v in kval.values():
        w.append(sum_pow_val + v)
    return(pow_val, sum_pow_val, w)
values1 = [1, 2]
values2 = { 'c': 10, 'd': 15 }
s = funk(*values1, **values2)
w1, w2, w3 = s[0], s[1], s[2]
s1, s2, s3 = funk(*values1, **values2)
```

Out[12]:

```
s : ([1, 4], 5, [15, 20])
w1, w2, w3 : [1, 4] 5 [15, 20]
```

In[13]:

```
def dodaj1(*args):
    return sum(args)
d1 = dodaj(1, 2, 3)
d2 = dodaj(*[1,2,3,4,5])
def dodaj2(**kwargs):
    return kwargs
d3 = dodaj2(test='a', b=1)
d4 = dodaj2(test='a', b=1, a=3)
d5 = dodaj2(test='a', b=1, a=3, li=[1,2,3])
```

Out[13]:

```
d1 = 6
d1 = 6
d3 = {'test': 'a', 'b': 1}
d4 = {'test': 'a', 'b': 1, 'a': 3}
d5 = {'test': 'a', 'b': 1, 'a': 3, 'li': [1, 2, 3]}
```

Kolejność argumentów w funkcji !!!

W przekazywaniu różnych rodzajów argumentów do funkcji Pythona ważne jest zachowanie kolejności, wygląda ono następująco:

In[14]:

```
def my_function(a, b = 1, *args, **kwargs):
    print(a)
    print(b)
    print(args)
    print(kwargs)
    return('jakieś wyniki w postaci krotki')
```

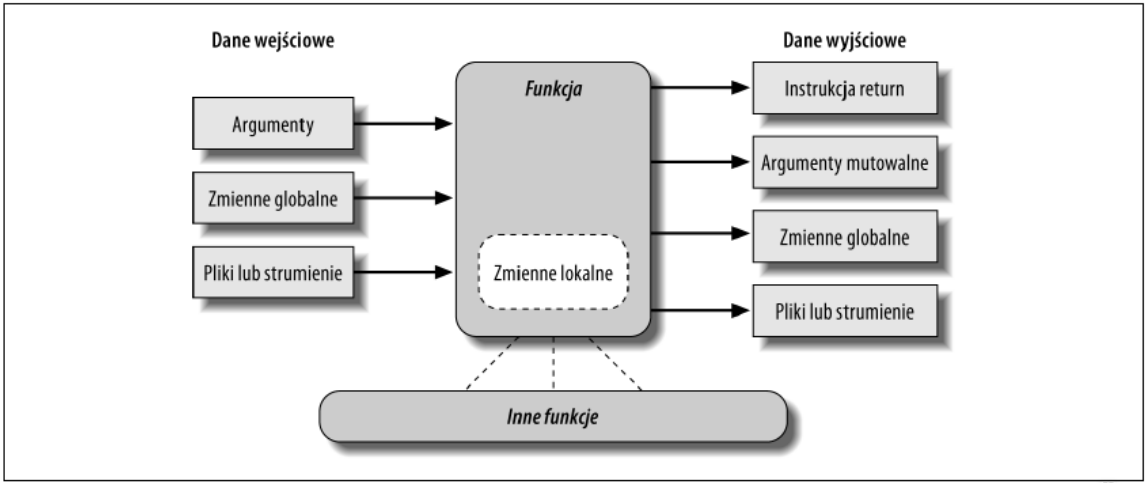


Figure: Środowisko wykonawcze funkcji (Lutz, 2011, p.493)

PEP 257 – Docstring Conventions https://www.python.org/dev/peps/pep-0257/

- **Docstring** to literał łańcuchowy występujący jako pierwsza instrukcja w definicji modułu, funkcji, klasy lub metody. Taki ciąg dokumentów staje się specjalnym atrybutem `__doc__` danego obiektu.
- Wszystkie moduły, funkcje i klasy powinny mieć dokumentacje.
- Metody publiczne (w tym konstruktor `__init__` - programowanie obiektowe) powinny również mieć dokumentacje. Pakiet może być udokumentowany w module docstring pliku `__init__.py` w katalogu pakietu.
- Literały łańcuchowe (docstring) występujące w innym miejscu aniżeli po instrukcjach rozpoczynających funkcje klasy, metody mogą również działać jako dokumentacja. Nie są one rozpoznawane przez kompilator kodu bajtowego Pythona i nie są dostępne jako atrybuty obiektów wykonawczych (tj. Nieprzypisane do `__doc__`).

PEP 257 – Docstring Conventions https://www.python.org/dev/peps/pep-0257/

In[15]:

```
def my_funk(a, b):
    '''
    Funkcja wykonująca dodawanie dwóch argumentów
    INPUT:
        a [float]- liczba zmiennoprzecinkowa [bez jednostek]
        b [float]- liczba zmiennoprzecinkowa [bez jednostek]
    OUTPUT:
        suma [float] - wynik dodawania [bez jednostek]
    '''
    suma = a + b
    return(suma)
my_funk.__doc__  # dostęp do dokumentacji funkcji
help(my_funk)    # dostęp do dokumentacji funkcji
```

Adnotacje funkcji - składnia dodawania dowolnych opisów metadanych funkcje Pythona. PEP 3107 – Function Annotations https://www.python.org/dev/peps/pep-3107/:

- Adnotacje funkcji (jej parametrów i zwracanych wartości) są całkowicie opcjonalne.
- Adnotacje funkcji są jedynie sposobem na powiązanie dowolnych wyrażeń Pythona z różnymi częściami funkcji w czasie kompilacji.
- Sam w sobie Python nie przypisuje żadnego szczególnego znaczenia ani znaczenia adnotacjom. Pozostawiony sam sobie, Python po prostu udostępnia te wyrażenia, jak opisano w Uzyskiwanie dostępu do adnotacji funkcji poniżej.

In[16]:

```
1 def f(x) -> 'funckcja f(x) zwraca x':
2     return x
3 print(f.__annotations__)
```

```
{'return': 'funckcja f(x) zwraca x'}
```

- Adnotacje są słownikami:

In[17]:

```
def kinetic_energy(m:'in KG', v:'in M/S')->'Joules':
    return 1/2*m*v**2
print(kinetic_energy.__annotations__)
```

```
{'return': 'Joules', 'v': 'in M/S', 'm': 'in KG'}
```

In[18]:

```
def kinetic_energy(m:'in KG', v:'in M/S')->'Joules':
    return 1/2*m*v**2

ke = kinetic_energy(20,3000)
jednostka = kinetic_energy.__annotations__['return']
print( ke, jednostka)
```

```
90000000.0 Joules
```

- Adnotacje są słownikami:

In[19]:

```
def foo(a: 'szerokosc', b: 'długosc', c: 'pole') -> 'pole':
    return a + b

print(foo.__annotations__)
print(foo.__annotations__['return'])
print(foo.__annotations__['c'])
print(foo.__annotations__['a'])
```

```
{'a': 'szerokosc', 'b': 'długosc', 'c': 'pole', 'return': '
    pole'}
pole
pole
szerokosc
```

Moduł to zbiór powiązanych klas i funkcji (np. plik z kodem zawierającym funkcje itp.)

Moduł – plik o nazwie **my_methods.py** zawiera funkcje:

In[20]:

```
def dodaj(a,b):
    return a + b

def odejmij(a,b):
    return a + b

def podziel(a,b):
    return a / b
```

Funkcje z pliku **my_methods.py** można importować i wykorzystywać w innych programach.

- 1 sposób importu

In[21]:

```
from my_methods import *

dodaj(a,b)
```

- 2 sposób importu

In[22]:

```
import my_methods
my_methods.dodaj(a,b)
my_methods.odejmij(a,b)
```

Plik o nazwie **my_methods.py**:

In[23]:

```
1  def dodaj(a,b):
2      return a + b
3  def odejmij(a,b):
4      return a + b
5
6  print('plik1: ', __name__)
7  if __name__ == '__main__':
8      # True jeśli wyk. ten plik
9      print(dodaj(2,3)
10     print('ukryte dla innych')
```

Out[23]:

```
1  plik1: __main__
2  5
3  ukryte dla innych
```

Plik **my_algorytm.py** importujący funkcje.

In[24]:

```
1  import my_methods as mm
2
3  print(mm.dodaj(4,4))
4  print(mm.odejmij(4,2))
5
6  print('plik2: ', __name__)
```

Out[24]:

```
1  plik1: my_method
2  8
3  2
4  plik2: my_algorytm
```

- Tworzenie funkcji: `def ... yield`, - funkcja generator

  `yield`: przerywa wykonania funkcji, zapisuje jej stan oraz zwraca wartości

In[25]:

```
def my_function(arg1, arg2):
    for x in [arg1, arg2]:
    yield(x*x)

my_generator = my_function(2, 5)        # wywołanie funkcji
print('wynik 1: ', my_generator)
for i in my_generator:                  # wywołanie generatora
    print('wynik 2:', i)
```

Out[25]:

```
wynik 1: <generator object my_function at 0x7f2b801d00c0>
wynik 2: 4
wynik 2: 25
```

- Instrukcja `yield` jest używana wyłącznie przy definiowaniu funkcji generującej i występuje wówczas w jej treści. Użycie wewnątrz definicji funkcji instrukcji yield jest warunkiem wystarczającym, aby stała się ona funkcją generującą, zamiast zwykłej funkcji.
- Wywołanie funkcji generującej zwraca tak zwany iterator generujący, lub po prostu **generator**. Wykonywanie treści funkcji generującej następuje każdorazowo po wywołaniu metody `next()` generatora, aż do czasu, gdy funkcja wygeneruje wyjątek.
- Przy wykonywaniu instrukcji `yield` następuje zamrożenie stanu generatora, a wartość wyrażenia podanego po słowie `yield` jest przekazywana do punktu wywołania metody `next()`. "Zamrożenie" oznacza w tym przypadku zapamiętanie całego stanu lokalnego, wraz z istniejącymi lokalnymi dowiązaniami nazw, wskaźnikiem instrukcji oraz wewnętrznym stosem wartościowania: zachowywana jest wystarczająca ilość informacji, aby przy kolejnym wywołaniu metody `next()` wykonywanie funkcji mogło być kontynuowane tak, jakby instrukcja `yield` była zwykłym wywołaniem funkcji.
- Instrukcji `yield` nie wolno używać w klauzuli `try` (`try ... finally`). Ograniczenie wynika z tego, że nie ma gwarancji przyszłego wznowienia generatora, więc nie ma też gwarancji, że byłby kiedykolwiek wykonany zestaw klauzuli `finally`.

In[26]:

```
def createGenerator(mylist):
    for i in mylist:
    return i
mygenerator = createGenerator(range(2)) # wywołanie funkcji
```

Out[26]:

```
return 0
```

In[27]:

```
def createGenerator(mylist):
    for i in mylist:
        yield i #przerywa wykonanie, zapisuje stan i zwraca wart.
mygenerator = createGenerator(range(2)) # wywołanie funkcji
for i in mygenerator:
    print('yield', i)
```

Out[27]:

```
yield: <generator object createGenerator at 0x7f2b8062e2a0>
yield 0
yield 1
```

- Tworzenie funkcji: `lambda`, **Składnia** wygląda następująco:

```
lambda arguments : expression
```

In[28]:

```
my_function = lambda arg1, arg2: a + b
wynik = my_function(5,4)
```

Co jest równoważne zapisowi :

In[29]:

```
def my_function(arg1, arg2):
    wynik = arg1 + arg2
    return(wynik)

wynik = my_function(5, 5)     # wywołanie funkcji
print(wynik)
```

Funkcje rekurencyjne Funkcje rekurencyjne to **funkcje, które wywołują same siebie** bezpośrednio lub za pośrednictwem innych funkcji. Istotne jest zapewnienie, że rekurencja będzie przerwana i nie wystąpi pętla nieskończona.

- to że funkcja wywołuje (i zwraca) samą siebie oznacza, że żeby otrzymać wynik, znów musi się wykonać. Czyli funkcja nie zwraca gotowego produktu, tylko przepis jak ten produkt otrzymać.
- rekurencja ma tę zaletę, że taki kod jest krótki i czytelny dla programisty w porównaniu z iteracją. Wadą jest to, że pochłania więcej zasobów niż iteracja.

In[30]:

```
def countdown(number):
    """
    odliczanie wartości w dół (do zera)
    """
    if number == 0:           # przypadek podstawowy - aby
        zatrzymać rekurencję
        print('complited')
    else:
        print('number', number) # przypadek rekurencyjny
        countdown(number -1)  # wywołanie rekurencyjne funkcji
print('countdown', countdown(8))
```

In[31]:

```
def accumulated_sum(number):
    """
    sumowanie wszystkich liczb przed podaną
    np sum_number(3) =0+1+2+3 =6
    """
    if number == 0:    # przypadek podstawowy - aby zatrzymać
        rekurencję
        return 0
    else:              # przypadek rekurencyjny
        return number + accumulated_sum(number -1)

print(accumulated_sum(4))  #10
```

- Zbieżności - różnica między bieżącą wartością z a jej poprzednią wartością jest mniejsza niż 0,25

In[32]:

```
def convergence(z , n): # funkcja przyjmująca 2 argumenty
    fz = z ** 4 - 1    # funkcja f(z)
    dfz = 4 * z ** 3   # pierwsza ochodna f(z)
    while n > 0:       # pętla do momentu kiedy n = 0
        z = z - fz / dfz # Newton-Raphson formula
        print('iteracje:', fz / dfz)
        if abs(fz / dfz)<0.25: # warunek zbieżności
            break
        return convergence(z, n-1) # po każdej iteracji

print('wynik:', convergence(4, 10))
```

Funkcje w Pythonie są wywoływane w wyrażeniach, przekazuje się do nich wartości i zwraca się z nich wyniki.

Najważniejsze koncepcje funkcji w Pythonie (Lutz, 2011, p.427):

- instrukcja `def` tworzy obiekt i przypisuje go do nazwy;
- instrukcja `return` przesyła wynikowy obiekt z powrotem do wywołującego;
- instrukcja `yield` (generator) odsyła wynikowy obiekt z powrotem do wywołującego, jednak zapamiętuje, gdzie zakończyła działanie
- **zmienne lokalne** to wszystkie zmienne wewnątrz funkcji (`def`);
- instrukcja `global` deklaruje zmienne, które mają być przypisane, na poziomie modułu.
- instrukcja `nonlocal` deklaruje zmienne z funkcji zawierającej, które mają być przypisane.
- **Argumenty przekazywane są przez przypisanie** (referencję obiektu). Argumenty, zwracane wartości i zmienne nie są deklarowane.
- wyrażenie `lambda` tworzy obiekt i zwraca go jako wynik (można użyć do tworzenia funkcji)

A. Cena, M. Gągolewski, and M. Bartoszuk. *Przetwarzanie i analiza danych w języku Python*. PWN, 2016. URL http://eczyt.bg.pw.edu.pl/han/ibuk/https/libra.ibuk.pl/book/000168290.

M. Lutz. *Python. Wprowadzenie*. Helion, 2011.

Dziękuje za uwagę

Kinga Węzka kinga.wezka@pw.edu.pl