Wstęp do programowania
semestr zimowy 2024/2025

Dr Anna Muranova
UWM w Olsztynie

Wykład 4

Funkcja `main()`

W wielu językach programowania (jak Java czy C++), nie jest możliwe, aby wyrażenia podobne do tych na końcu programu widniały ot tak, samodzielnie. Wymagane jest by były one częścią specjalnej funkcji o nazwie `main()` (funkcja główna).
Pomimo tego, że nie jest to wymagane w języku Python, nie będzie to zły pomysł, jeżeli włączymy je w logiczną strukturę programu.

```
x = 5
y = 3
print(x+y)#8
```

```
def main():
    x = 5
    y = 3
    print(x + y)

main()
```

```
8
```

Część zaawansowana

Zanim interpreter języka Python uruchomi Twój program, definiuje on kilka specjalnych zmiennych. Jedną z nich jest zmienna `__name__`, do której automatycznie przypisywana jest wartość "`__main__`"w momencie, gdy dany kod uruchamiany jest niezależnie, jako samodzielny program. Z drugiej strony, gdy dany kod importowany jest z poziomu innego programu, wartość zmiennej `__name__` ustawiana jest jako nazwa tego właśnie modułu. Oznacz to, że wiemy, czy moduł uruchamiany jest jako niezależny program czy może używany jest przez inny program. Bazując na tej wiedzy, możemy zdecydować, czy uruchamiać wybraną część kodu.
Załóżmy przykładowo, że napisaliśmy szereg funkcji, wykonujących proste obliczenia matematyczne. Wywołania tych funkcji możemy zamieścić w funkcji `main`. Jest jednak bardziej prawdopodobne, że funkcje te będą zaimportowane przez inny program, do zupełnie innych celów. W tym przypadku, nie będziemy raczej chcieli wywoływać funkcji `main`.

Część zaawansowana

```
def main():
    x = 5
    y = 3
    print(x + y)

if __name__ == "__main__":
    main()
```

```
8
```

W tym kodzie znajdziemy wyrażenie `if`, aby sprawdzić wartość zmiennej `__name__`. Jeżeli będzie to `__main__`, wtedy wywołana będzie funkcja main. W innym przypadku możemy założyć, że powyższy program został zaimportowany z poziomu innego programu, nie będziemy zatem chcieli wywołać funkcji `main`, gdyż ów nowy program będzie używać powyższych funkcji wedle potrzeb. Możliwość warunkowego uruchamiania funkcji `main` może się okazać bardzo potrzebna, gdy piszemy program, który potencjalnie będzie używany przez innych. Umożliwia nam też dodanie tej funkcjonalności, której użytkownik kodu nie będzie potrzebował. Najczęściej będą to procedury testowe sprawdzające poprawne funkcjonowanie funkcji.

Funkcje: nie znana ilość argumentów

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

print(suma(12, 4)) #16
```

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

def suma(a, b, c):
    return a + b + c

print(suma(12, 4))
print(suma(3, 4, 5))
```

Funkcje: *args

- ```
  def suma(a, b=0, c=0, d=0):
      return a + b + c + d

  print(suma(12, 4))
  print(suma(3, 4, 5, 7))
  ```
- ```
  def suma(*values):
      s = 0
      for i in values:
          s+=i
      return s

  print(suma(12, 4))# 16
  print(suma(3, 4, 5, 7)) #29
  ```

Funkcje wbudowane sum() i len()

- ```
  def suma(*values):
      #print(values)
      return sum(values)

  print(suma(12, 4))
  print(suma(3, 4, 5, 7))
  #print(sum(3, 4, 5, 7))błąd
  print(sum((3,4,5,7)))#omówimy później
  ```
- ```
  def srednia(*values):
      return (sum(values)) / (len(values))

  print(srednia(2, 3, 4, 6))
  print(srednia(45))
  ```

Funkcja długości, napisana samodzielne

- ```
  def dlug(*values):
      i = 0
      for x in values:
      i += 1
      return i

  print(dlug(12, 4)) #2
  print(dlug(3, 4, 5, 7)) #4
  ```

- ```
  def dlug(*values):
      i = 0
      for _ in values:
      i += 1
      return i

  print(dlug(12, 4)) #2
  print(dlug(3, 4, 5, 7)) #4
  ```

Argumenty oprocz *args:

- ```
  def func(potega, *values):
      s = 0
      for i in values:
          s+=i**potega
      return s


  print(func(2, 3, 4, 6)) #61
  print(func(3, 4)) #64
  print(func(3)) #0
  ```
- ```
  def func(*values, potega):
      s = 0
      for i in values:
          s+=i**potega
      return s
  #print(func(2,3, 5))blad
  #print(func(potega = 5, 2,3))blad
  print(func(2,3,potega = 5)) #275
  ```

Funkcje: dwie funkcje z rożną ilością argumentów nie są możliwe, ale!

```
def area(l, b):
  return l * b

def area(r):
  return 3.14 * r ** 2


area(3, 4)
area(7)
```

```
def area(*values):
  if len(values)==2:
      return values[0]*values[1]
  if len(values)==1:
      return 3.14 * values[0] ** 2

print(area(3, 4)) #12
print(area(7)) #153.86
```

Funkcje: dwie funkcje z rożną ilością argumentów nie są możliwe, ale jeszcze raz

```
def area(l= None, b= None, r = None):
    if r == None:
        return l*b
    else:
        return 3.14 * r ** 2


print(area(l=3, b=4))#12
print(area(r=7))#153.86
```

Funkcje rekurencyjne

Funkcje rekurencyjne są to funkcje, których charakterystyką jest to, iż przed zakończeniem bieżącego wywołania funkcji następuje kolejne wywołanie tej funkcji (czyli w ciele danej funkcji następuje wywołanie tejże funkcji).

Ich zawsze można i lepiej(!) zastąpić pętlą.

Silnia

```
def silnia(n):
    if n:
        return silnia(n-1)*n
    else:
        return 1

print(silnia(5)) #120
print(silnia(0)) #1
```

```
def silnia(n):
    s = 1
    for i in range(1,n+1):
        s*=i
    return s

print(silnia(5)) #120
print(silnia(0)) #1
```

Potęgowanie

```
def pot(n, p):
    if p ==0:
        return 1
    else:
        return n*pot(n,p-1)

print(pot(5,2))#25
print(pot(8,3))#512
```

```
def pot(n, p):
    prod = 1
    for i in range (1,p+1):
        prod *= n
    return prod

print(pot(5,2))#25
print(pot(8,3))#512
```

Potęgowanie binarne

```
def pot(n, p):
    if not p:
        return 1
    if p%2 == 0:
        temp = pot(n,p//2)
        return temp*temp
    else:
        temp = pot(n,p//2)
        return temp*temp*n

print(pot(2,10))#1024
```

Potęgowanie binarne: ilość operacje

```
global count
count = 0
def pot(n, p):
    global count
    if not p:
        return 1
    if p%2 == 0:
        temp = pot(n,p//2)
        count += 1
        return temp*temp
    else:
        temp = pot(n,p//2)
        count += 2
        return temp*temp*n

print(pot(2,10))
print(count)#6

count = 0
print(pot(2,100))
print(count)#10
```

Potęgowanie zwykle: ilość operacje

```
count = 0
def pot(n, p):
    global count
    if p ==0:
        return 1
    else:
        count += 1
        return n*pot(n,p-1)

print(pot(2,10))
print(count) #10

count = 0
print(pot(2,100))
print(count) #100
```

Potęgowanie binarne

Uwaga! Potęgowanie binarne można zrobić nie rekurencyjne, ale dlatego będziemy potrzebowali listy. Dlatego zrobimy to później.

Ciąg Fibonacciego

```
def Fib(n):
    if n < 2:
        return n
    else:
        return Fib(n-1) + Fib(n-2)

print(Fib(12)) #144
```

```
def Fib1(n):
    if (n == 0):
        return 0
    i0 = 0
    i1 = 1
    for j in range(1,n):
        i0, i1 = i1, i0+i1
    return i1

print(Fib1(12)) #144
```

Dlaczego rekurencja w ciągu Fibonacciego jest złym rozwiązaniem

```
count = 0
def Fib(n):
    global count
    count += 1
    if n < 2:
        return n
    else:
        return Fib(n-1) + Fib(n-2)

print(Fib(12))
print(count)#465

#print(Fib(30))
#print(count) #2692537
```

Dlaczego rekurencja w ciągu Fibonacciego jest złym rozwiązaniem

```
count = 0
def Fib1(n):
    global count
    if (n == 0):
        return 0
    i0 = 0
    i1 = 1
    for j in range(1,n):
        i0, i1 = i1, i0+i1
        count += 1
    return i1

print(Fib1(12))
print(count)#11

print(Fib1(30))
print(count) #29
```

Algorytm Euklidesa

Algorytm Euklidesa —- algorytm wyznaczania największego wspólnego dzielnika dwóch liczb. Został opisany przez greckiego matematyka, Euklidesa w jego dziele „Elementy", w księgach siódmej oraz dziesiątej.
Największym wspólnym dzielnikiem dwóch liczb jest największa z liczb, która dzieli obie te liczby bez reszty
Dla liczb całkowitychk, m oraz n załóżmy, że k jest wspólnym czynnikiem liczb a oraz b. Niech

$$a = bq + r.$$

Wtedy k jest wspólnym czynnikiem liczb b oraz r. I odwrotnie. Tzn,

$$NWD(a, b) = NWD(q, r)$$

Dzialamy, dopoki $r \neq 0$.

Algorytm Euklidesa

```
def NWD(a, b):
    if b:
        return NWD(b,a%b)
    return a

print(NWD(12,18)) #6
```

Na ćwiczeniach: napisać wersja nierekurencyjna

Funkcje wbudowane

Built-in Functions

A	E	L	R
abs()	enumerate()	len()	range()
aiter()	eval()	list()	repr()
all()	exec()	locals()	reversed()
anext()			round()
any()	F	M	
ascii()	filter()	map()	S
	float()	max()	set()
B	format()	memoryview()	setattr()
bin()	frozenset()	min()	slice()
bool()			sorted()
breakpoint()	G	N	staticmethod()
bytearray()	getattr()	next()	str()
bytes()	globals()		sum()
		O	super()
C	H	object()	
callable()	hasattr()	oct()	T
chr()	hash()	open()	tuple()
classmethod()	help()	ord()	type()
compile()	hex()		
complex()		P	V
	I	pow()	vars()
D	id()	print()	
delattr()	input()	property()	Z
dict()	int()		zip()
dir()	isinstance()		
divmod()	issubclass()		_
	iter()		__import__()

Link do dokumentacji
https://docs.python.org/3/library/functions.html

Funkcje wbudowane

```
print (abs(-10))#10

def abs(x):
    return 5

print (abs(-10))#5
```

- `abs()` – wartosc bezwzgledna
- `min()` – minimum
- `max()` – maksymum
- `pow()` – potega

Slowo kluczowe pass

Instrukcja pass służy jako symbol zastępczy dla przyszłego kodu, zapobiegając błędom z pustych bloków kodu.
Jest zwykle używana, gdy kod jest zaplanowany, ale jeszcze nie został napisany.

```
def future_function():
    pass

# this will execute without any action or error
future_function()
print(future_function())#None
```

Slowo kluczowe assert

`assert` – pozwala użyć asercji, aby móc skontrolować działanie programu. Jeśli warunek asercji nie zostanie spełniony kompilator zgłosi błąd AssertionError.

```
x = "hello"

#if condition returns True, then nothing happens:
assert x == "hello"

#if condition returns False, AssertionError is raised:
assert x == "goodbye"
```

Możemy użyć `assert`, żeby sprawdzić działanie funkcje:

```
assert abs(-10) == 10
assert abs(10) == 10
assert abs(0) == 0
```

Slowo kluczowe assert

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

def main():
    assert suma(3,5)==8
    assert suma(-2,1)==-1
    assert suma('a', 'b') == 'ab'
    #assert suma(0.2,0.3) == 0.5
    #assert suma (249.99, 22.41) == 272.40
    #print(suma (249.99, 22.41))

if __name__ == "__main__":
    main()
```

Funkcje: import math

Link do dokumentacji https://docs.python.org/3/library/math.html

- ```
  import math

  a=0
  b=math.sin(2*math.pi)
  print(b)
  print(math.isclose(a,b, rel_tol=1e-09, abs_tol=1e-09))
  ```
- ```
  import math

  print(math.e)
  print(math.exp(1))
  print(math.ceil(math.e))
  print(math.floor(math.e))
  print(math.factorial(10))
  ```

Funkcje: import math

Link do dokumentacji https://docs.python.org/3/library/math.html

- ```
  import math

  print(math.gcd())
  print(math.gcd(20,16))
  print(math.gcd(20,15,35,25))
  ```
- ```
  import math

  #print(math.prod())
  #print(math.prod(0,2))
  print(math.prod((0,2)))
  print(math.prod((12,3,21)))
  print(math.prod((12,3,21), start=2))
  ```

Funkcje: potęga

```
import math

print(math.pow(2,100))#1.2676506002282294e+30
print(pow(2,100))#1267650600228229401496703205376
print(2**100)#1267650600228229401496703205376

#print(math.pow(-2,0.5))
print(pow((-2),0.5))#(8.659560562354934e-17+1.4142135623730951j)
print((-2)**0.5)#(8.659560562354934e-17+1.4142135623730951j)

print(math.exp(1e-5) - 1)  # gives result accurate to 11 places
print(math.expm1(1e-5) )
```

Oprócz tego `math.exp(x)` dokładniej niż `math.e ** x` oraz `pow(math.e, x)`.

Funkcje: reszta

```
import math

print(math.remainder(8,3))#-1.0
print(math.fmod(8,3))#2.0
print(8%3)#2

print(math.remainder(8.5,3))#-0.5
print(math.fmod(8.5,3))#2.5
print(8.5%3)#2.5

print(math.remainder(-1e-100,1e100))#-1e-100
print(math.fmod(-1e-100,1e100))#-1e-100
print(-1e-100 % 1e100)#1e+100
```

docstrings

```
print(len.__doc__)#Return the number of items in a container.
print(print.__doc__)
```

Docstringi języka Python to literały ciągów znaków, które pojawiają się zaraz po definicji funkcji, metody, klasy lub modułu.

```
def square(n):
    '''Take a number n and return the square of n.'''
    return n**2

print(square.__doc__)
```

Mozna tez uzywac potrojnych cudzoslowow:

```
def square(n):
    """Take a number n and return the square of n."""
    return n**2

print(square.__doc__)
```

docstrings vs komentarze

W Pythonie używamy symbolu hash #, aby napisać komentarz jednowierszowy. Ale jeśli nie przypiszemy ciągów znakow (napisow) do żadnej zmiennej, działają one tez jak komentarze. Na przykład,

```
'napis'
"napis"
print(2+3)
```

Docstringi języka Python to literały ciągów znaków, które pojawiają się zaraz po definicji funkcji, metody, klasy lub modułu.

```
"I am a single-line comment"

'''
I am a
multi-line comment!
'''

print("Hello World")
```

Kiedykolwiek napisy są obecne zaraz po definicji funkcji, modułu, klasy lub metody, są one skojarzone z obiektem jako ich atrybut__doc__. Możemy później użyć tego atrybutu, aby pobrać ten docstring.

Standardowe konwencje pisania docstringów

Jednowierszowe:

- Mimo że są jednowierszowe, nadal używamy potrójnych cudzysłowów wokół tych docstringów, ponieważ można je później łatwo rozszerzyć.
- Cudzysłowy zamykające znajdują się w tym samym wierszu co cudzysłowy otwierające.
- Nie ma pustego wiersza ani przed, ani po docstringu.
- Nie powinny być opisowe, raczej muszą być zgodne ze strukturą „Zrób to, zwróć tamto" kończącą się kropką.

```
def multiplier(a, b):
    """Take two numbers and return their product."""
    return a*b
```

Wielowierszowe cidocstringi składają się z wiersza podsumowującego, tak jak jednowierszowe ciągi dokumentacyjne, po którym następuje pusty wiersz, a następnie bardziej szczegółowy opis.
Dokument PEP 257 zawiera standardowe konwencje pisania wielowierszowych docstringow dla różnych obiektów.

Docstringi wielowierszowe dla funkcje

- Docstring dla funkcji powinien podsumowywać jej zachowanie i dokumentować jej argumenty i wartości zwracane.
- Powinien również zawierać listę wszystkich wyjątków, które mogą zostać zgłoszone, oraz inne opcjonalne argumenty.

```
def add_binary(a, b):
    '''
    Return the sum of two decimal numbers in binary digits.

            Parameters:
                    a (int): A decimal integer
                    b (int): Another decimal integer

            Returns:
                    binary_sum (str): Binary string of the sum of a and
    '''
    binary_sum = bin(a+b)
    return binary_sum

print(add_binary.__doc__)
print(add_binary(3, 5))
```

Błędy i wyjątki

Występują (przynajmniej) dwa charakterystyczne typy błędów:

- błędy składni (syntax errors)

```
while True print('Witaj świecie')
#brakuje :
```

- wyjątki (exceptions)

```
print(4/0)
#dzielenie przez 0
print( '2' + 2)
#str + int
print(a)
#a nie zdefiniowane
```

Błędy składni

```
while True print('Witaj świecie')
#brakuje :
```

Parser (interfejs do Pythona) powtarza błędną linię i wyświetla małe „strzałki" wskazujące token w linii, w której wykryto błąd. Błąd może być spowodowany brakiem tokenu przed wskazywanym tokenem. W przykładzie błąd jest wykryty na funkcji `print()`, ponieważ brakuje przed nią dwukropka (':'). Nazwa pliku i numer linii są drukowane, abyś wiedział(a), gdzie szukać, w przypadku, gdy dane wejściowe pochodzą ze skryptu.

Wyjątki 1

Nawet jeśli instrukcja lub wyrażenie jest poprawne składniowo, może ona wywołać błąd podczas próby jej wykonania. Błędy zauważone podczas wykonania programu są nazywane wyjątkami (exceptions) i nie zawsze są niedopuszczalne: już niedługo nauczysz w jaki sposób je obsługiwać. Większość wyjątków nie jest jednak obsługiwana przez program przez co wyświetlane są informacje.

```
print(4/0)
#dzielenie przez 0
print( '2' + 2)
#str + int
print(a)
#a nie zdefiniowane
```

Ostatni wiersz komunikatu o błędzie wskazuje, co się stało. Wyjątki występują w różnych typach, a typ jest drukowany jako część komunikatu.

W przykładach: `ZeroDivisionError`, `NameError` i `TypeError`

Wyjątki 2

Ciąg wydrukowany jako typ wyjątku jest nazwą wbudowanego wyjątku, który wystąpił. Jest to prawdą dla wszystkich wbudowanych wyjątków, ale nie musi być prawdą dla wyjątków zdefiniowanych przez użytkownika (choć jest to przydatna konwencja). Standardowe nazwy wyjątków są wbudowanymi identyfikatorami (nie zarezerwowanymi słowami kluczowymi).

Pozostała część linii dostarcza szczegółów na temat typu wyjątku oraz informacji, co go spowodowało.

Wbudowane wyjątki:
`https://docs.python.org/pl/3/library/exceptions.html#bltin-exceptions`

Obsługa wyjątków: try except

Możliwe jest pisanie programów, które obsługują wybrane wyjątki.

```
x = 'napis'
try:
  print(x + 3)
except:
  print("An exception occurred")
```

Obsługa wyjątków: try, kilka except

```
#x = 'napis'
try:
  print(x + 3)
except TypeError:
  print("TypeError")
except:
  print("Unknown Error has occured")
```

Obsługa wyjątków: try, nie pasuje except

Porównaj:

```
x = 'napis'
try:
  print(x + 3)
except NameError:
  print("NameError")
```

z

```
x = 'napis'
try:
  print(x + 3)
except TypeError:
  print("TypeError")
```

oraz z

```
x = 'napis'
try:
  print(x + 3)
except:
  print("Error")
```

Try

Instrukcja try działa następująco.

- W pierwszej kolejności wykonywane są instrukcje pod klauzulą try - pomiędzy słowami kluczowymi try i except.
- Jeżeli nie wystąpi żaden wyjątek, klauzula except jest pomijana i zostaje zakończone wykonywanie instrukcji try.
- Jeśli wyjątek wystąpi podczas wykonywania klauzuli try, reszta klauzuli jest pomijana. Następnie, jeśli jego typ pasuje do wyjątku nazwanego po słowie kluczowym except, wykonywana jest klauzula except, a następnie wykonanie jest kontynuowane po bloku try/except.
- Jeśli wystąpi wyjątek, który nie pasuje do wyjątku nazwanego w klauzuli except, jest on przekazywany do zewnętrznych instrukcji try; jeśli nie zostanie znaleziona obsługa, jest to nieobsłużony wyjątek i wykonanie zatrzymuje się z komunikatem błędu.

Jeszcze przykład

Poniższy przykład, prosi użytkownika o wprowadzenie danych, dopóki nie zostanie wprowadzona poprawna liczba całkowita.

```
while True:
    try:
        x = int(input("Proszę podaj liczbę: "))
        break
    except ValueError:
        print("Ups! To nie była poprawna liczba. Spróbuj ponownie...")
```

Bardziej rozbudowany sposób

- Blok try pozwala przetestować fragment kodu pod kątem błędów.
- Blok except pozwala obsłużyć błąd.
- Blok else pozwala wykonać kod, gdy nie wystąpi żaden błąd.
- Blok finally pozwala wykonać kod, niezależnie od wyniku bloków try i except.

Przykład

```
# Python code to illustrate
# working of try()
def divide(x, y):
    try:
        # Floor Division : Gives only Fractional
        # Part as Answer
        result = x // y
    except ZeroDivisionError:
        print("Sorry ! You are dividing by zero ")
    else:
        print("Yeah ! Your answer is :", result)
    finally:
        # this block is always executed
        # regardless of exception generation.
        print('This is always executed')

# Look at parameters and note the working of Program
divide(3, 2)
divide(3, 0)
```

Dlaczego potrzebny finally?

Porównaj:

```
def divide(x, y):
    try:
        result = x // y
    except ZeroDivisionError:
        return None
    else:
        return result
    finally:
        print("This is always executed")

divide(3, 2)#This is always executed
divide(3, 0)#This is always executed
```

Dlaczego potrzebny finally?

oraz

```
def divide(x, y):
    try:
        result = x // y
    except ZeroDivisionError:
        return None
    else:
        return result
    print("This is always executed")

divide(3, 2)#nic sie nie wyswietli
divide(3, 0)#nic sie nie wyswietli
```

Głównym celem bloku finally jest zapewnienie, że określony kod czyszczący zostanie wykonany bez względu na to, co się stanie w bloku try. Nawet jeśli nie wystąpi żaden wyjątek do obsłużenia, blok finally i tak wykona się po zakończeniu bloku try.