Wstęp do programowania
semestr zimowy 2024/2025

Dr Anna Muranova
UWM w Olsztynie

Wykład 3

3 typy instrukcji

Istnieje **tylko** trzy typa instrukcje:

- Instrukcja prosta
- Instrukcja warunkowa
- Pętla

Pętla

Pętla – element języka programowania, który pozwala na wykonanie instrukcji dopóki nie zostanie spełniony warunek kończący pętlę.

- Pętli `while` używa się jeżeli nie potrzebujesz iteratora (nie wiesz ile razy dokładnie będzie wykonywać się kod).
- Pętli `for` używa się kiedy potrzebujesz iteratora. (Jakaś wartość zmiennie się tak samo w każdej pętli i od tej wartości zależy wykonywanie pętli)

Pętli w Python: while

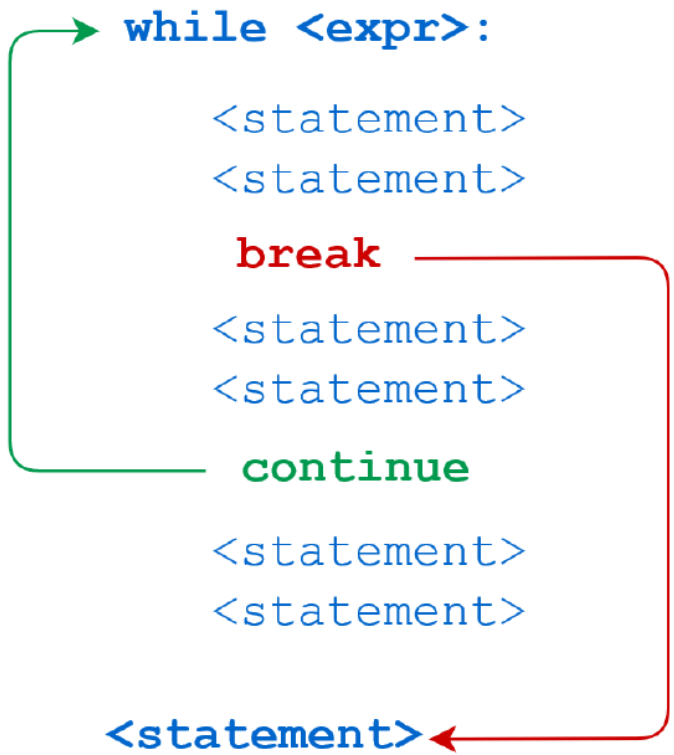
Pętlę `while` w praktyce stosuje się zawsze tam, gdzie mamy potrzebę przetwarzania nieokreślonej z góry liczby danych, np: należy zakończyć pętlę gdy tylko zostanie wciśnięty określony klawisz klawiatury.

```
while <expr>:
    <statement(s)>
```

```
i = 0
while i < 5:
    print(i)
    i = i + 2
```

```
0
2
4
```

break/continue



. . . break . . .

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
1
2
3
```

. . . . continue . . .

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
1
2
4
5
6
```

. . . while . . . else

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

```
1
2
3
4
5
i is no longer less than 6
```

. . . while . . . else

Uwaga! `else` nie wykonuje się, jeżeli pętla jest zakończona z powodu `break`.
Porownaj:

```
i = 1
while i < 6:
    print(i , end=' ')
    i += 1
    if i==3:
        break
else:
  print("i is no longer less than 6")
```

1 2

```
i = 1
while i < 6:
    print(i, end=' ')
    i += 1
    if i==3:
        break
print("i is no longer less than 6")#błąd logiczny!!!
```

1 2 i is no longer less than 6

. . . while . . . else

Porównaj jeszcze:

```
i = 1
while i < 6:
    print(i, end=' ')
    i += 1
    if i==3:
        break
else:
  print("i is no longer less than 6")
```

1 2

```
i = 1
while i < 6:
    print(i, end = ' ')
    i += 1
    if i==3:
        break
    else:
        print("i is no longer less than 6")#błąd logiczny!!!
```

1 i is no longer less than 6
2

Zastosowanie pętli `while` w praktyce

Przykład pętli o nieznanej z góry liczbie iteracji.

Wyznaczyć taką najmniejszą liczbę naturalną, że suma liczb od niej mniejszych i podzielnych przez 7 jest większa od założonej wartości.

```
print("Podaj liczbe ")
varMin = int(input())
suma = 0
i = 1
while (suma < varMin):
        if (i % 7 == 0):
            suma += i
        i+=1
print("Wyznaczona liczba = ",i)
print("Suma = " ,suma)
```

Podaj liczbe 23
Wyznaczona liczba = 22
Suma = 42

Inny sposób dla poprzedniego przykładu

Przykład pętli o nieznanej z góry liczbie iteracji.

Wyznaczyć taką najmniejszą liczbę naturalną, że suma liczb od niej mniejszych i podzielnych przez 7 jest większa od założonej wartości.

```
print("Podaj liczbe ")
varMin = int(input())
suma = 0
i = 0
while (suma < varMin):
        suma += i
        i+=7
print("Wyznaczona liczba = ",i - 6)
print("Suma = " ,suma)
```

Podaj liczbe 23
Wyznaczona liczba = 22
Suma = 42

Pętli w Python: for

Pętlę `for` stosuje się zawsze tam, gdy znamy liczbę iteracji, np: znamy liczbę danych, jaką należy wczytać, wypisać lub zmienić, chcemy policzyć średnią z określonej liczby danych, mamy za zadanie wczytać znaną liczbę danych z pliku lub wypisać określoną liczbę danych na ekran, itd.

```
for i in <collection>:
    <loop body>
```

```
for x in range(6):
    print(x)
```

range

Generuje nam ciąg liczb (dedykowany typ **range**). Trzeba zamienić na listę "by podejrzeć".
Uwaga : wszystkie argumenty muszą być w typie całkowitym. Jeden argument – to "koniec" – ciąg tworzą liczby naturalne od 0 do $n-1$. Krok domyślny to 1.

Dwa argumenty - to "początek" i "koniec". Krok domyślny to 1. Wtedy wyświetlone są liczby całkowite z przedziału lewostronnie domkniętego .

Trzy argumenty – to "początek", "koniec" oraz krok.

```
print(list(range(5)))#[0, 1, 2, 3, 4]
print(list(range(1, 11)))#[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(list(range(0, 30, 5)))#[0, 5, 10, 15, 20, 25]
print(list(range(0, 10, 3)))#[0, 3, 6, 9]
print(list(range(0, -10, -1)))#[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
print(list(range(0)))#[]
print(list(range(1, 0)))#[]
```

Pętli w Python: for

Liczby parzyste od 0 do n:

```
n = 24
for x in range(0, n+1, 2):
  print(x)
```

lub

```
n = 24
for x in range(0, n+1):
    if n % 2 == 0:
        print(x)
```

Co robi? Która będzie ostatnia liczba?

```
for x in range(2, 30, 3):
  print(x, end=' ')
```

break w for

```
for x in range(6):
  if x==3: break
  print(x)
```

```
0
1
2
```

. . . for . . . else

```
for x in range(6):
  print(x, end=' ')
else:
  print("Finally finished!")
```

0 1 2 3 4 5 Finally finished!
lub

```
for x in range(6):
  if x == 3: break
  print(x, end=' ')
else:
  print("Finally finished!")
```

0 1 2

. . . for . . . else

Porównaj:

```
for x in range(6):
  if x == 3: break
  print(x, end=' ')
else:
  print("Finally finished!")
```

0 1 2

```
for x in range(6):
  if x == 3: break
  print(x, end=' ')
print("Finally finished!")
```

0 1 2 Finally finished!

. . . for . . . else

Porównaj:

```
for x in range(6):
  if x == 3: break
  print(x, end=' ')
else:
  print("Finally finished!")
```

0 1 2

```
for x in range(6):
  if x == 3: break
  print(x)
  else:
      print("Finally finished!")
```

. . . for . . . else

```
for x in range(6):
  if x == 3: break
  else:
      print("Finally finished!")
  print(x)
```

Finally finished! 0 Finally finished! 1 Finally finished! 2

Przykład działania instrukcji `continue` w `for`

```
for i in range(0,10):
    if i == 4:
        continue
    print(i, end =' ')
```

```
0 1 2 3 5 6 7 8 9
```

Przykład działania instrukcji continue w for

```
print("Podaj zakres liczb ")
print("od: ")
zakrMin = int(input())
print("od: ")
zakrMax = int(input())
suma = 0
for i in range(zakrMin,zakrMax+1):
if i % 2 : continue
suma += i
print("Suma liczb parzystych = ",suma)
```

od 5 do 19
Suma liczb parzystych = 84
Zastosowanie instrukcji continue oraz dzielenia i % 2 , jako warunku w pętli if spowodowało sumowanie jedynie liczb parzystych.

Jak to można zrobić inaczej?

Zagnieżdżone pętle

```
for i in <collection>:
    <loop body>
    for i in <collection>:
        <loop body>

while <expr>:
    <statement(s)>
     while <expr>:
        <statement(s)>
```

Zagnieżdżone pętle

```
for i in range(3):
    for j in range(3):
        print(i, "*", j, "=", i * j)
```

```
0 * 0 = 0 0 * 1 = 0 0 * 2 = 0 1 * 0 = 0 1 * 1 = 1 1 * 2 = 2 2 * 0 = 0 2 * 1
= 2 2 * 2 = 4
```

. . . pass . . .

Pętla for nie może być pusta, używaj pass.

```
for x in [0, 1, 2]:
  pass
```

Uwaga: najczęściej pass jest używany, jeżeli kod będzie dodany później.

Zagnieżdżone pętle: ciekawostki

```
for i in range(3):
    print(i, end=' ')
    for i in range(3):
        pass
```

0 1 2

```
for i in range(3):
    for i in range(3):
        pass
    print(i, end = ' ')
```

2 2 2

Zagnieżdżone pętle: ciekawostki

```
a = 0
b = 0
for i in range(3):
    for i in range(3):
        a += 1
    b += 1
print('a=',a)
print('b=',b)
```

a= 9 b= 3

Przykład `while`

W 2005 roku posadziłam choinka mierzącą 1,20 m. Przyrasta o 30 cm rocznie. Postanowiłam wyciąć gdy przekroczy 7 m. W którym roku będę ścinała drzewo?

```
year = 2005;
h = 1.2;


while h < 7:
        year += 1;
        h += 0.3;

print(year)
```

2025

Przykład `for`

Oblicz 10tą liczbą Fibonacciego ($F_0 = 0, F_1 = 1$).

```
f0 = 0
f1 = 1

for i in range(2,11):
    f = f1
    f1 = f1 + f0
    f0 = f

print(f1)
```

lub

```
f0, f1 = 0, 1

for i in range(2,11):
    f0, f1 = f1, f1 + f0

print(f1)
```

55

Debugowanie

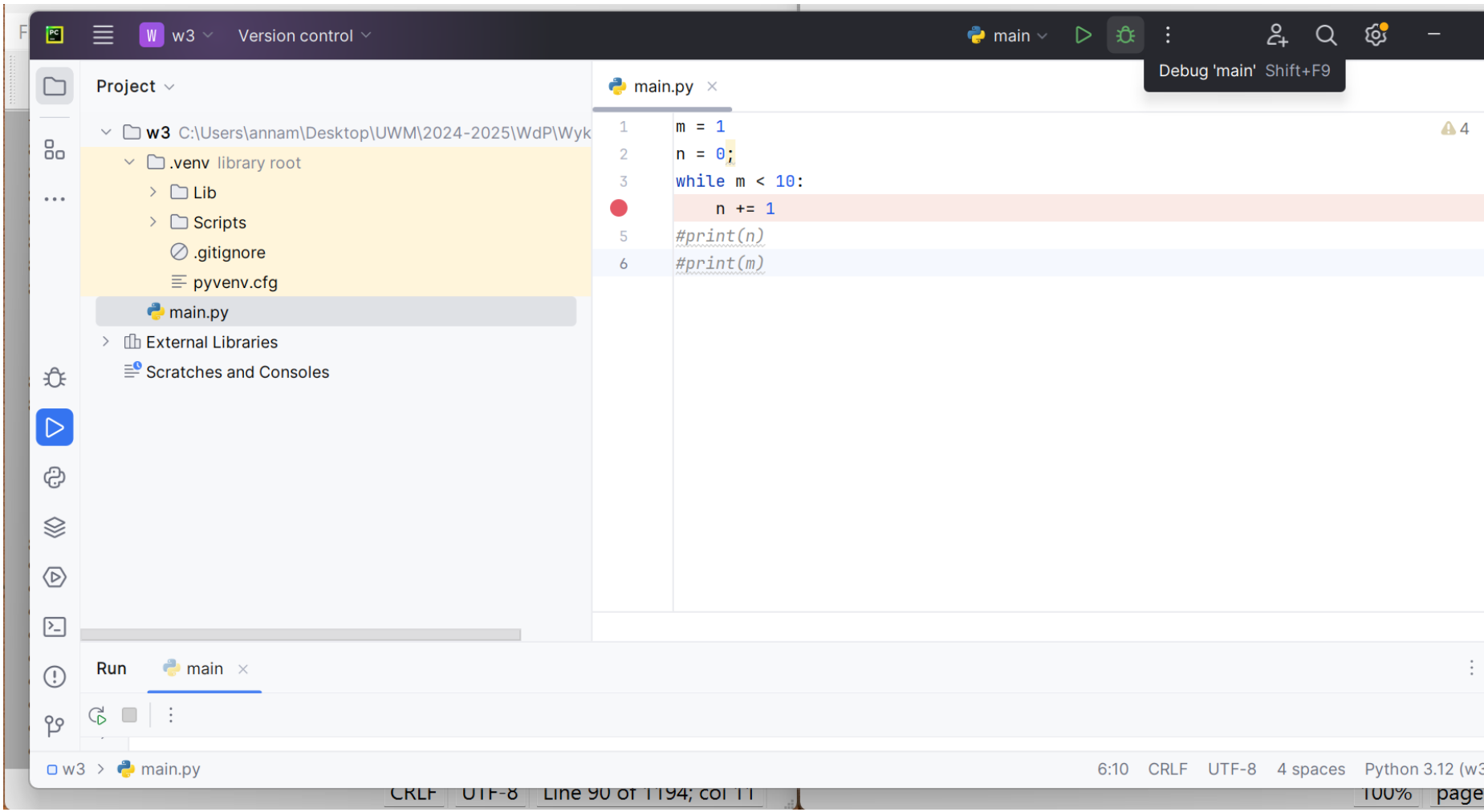
Debugger albo debuger – program komputerowy służący do dynamicznej analizy innych programów, w celu odnalezienia i identyfikacji zawartych w nich błędów, zwanych z angielskiego bugami (robakami). Proces nadzorowania wykonania programu za pomocą debuggera określa się mianem *debugowania*. Podstawowym zadaniem debuggera jest sprawowanie kontroli nad wykonaniem kodu, co umożliwia zlokalizowanie instrukcji odpowiedzialnych za wadliwe działanie programu. Współczesne debuggery pozwalają na efektywne śledzenie wartości poszczególnych zmiennych, wykonywanie instrukcji krok po kroku czy wstrzymywanie działania programu w określonych miejscach. Debugger jest standardowym wyposażeniem większości współczesnych środowisk programistycznych.

Debugowanie

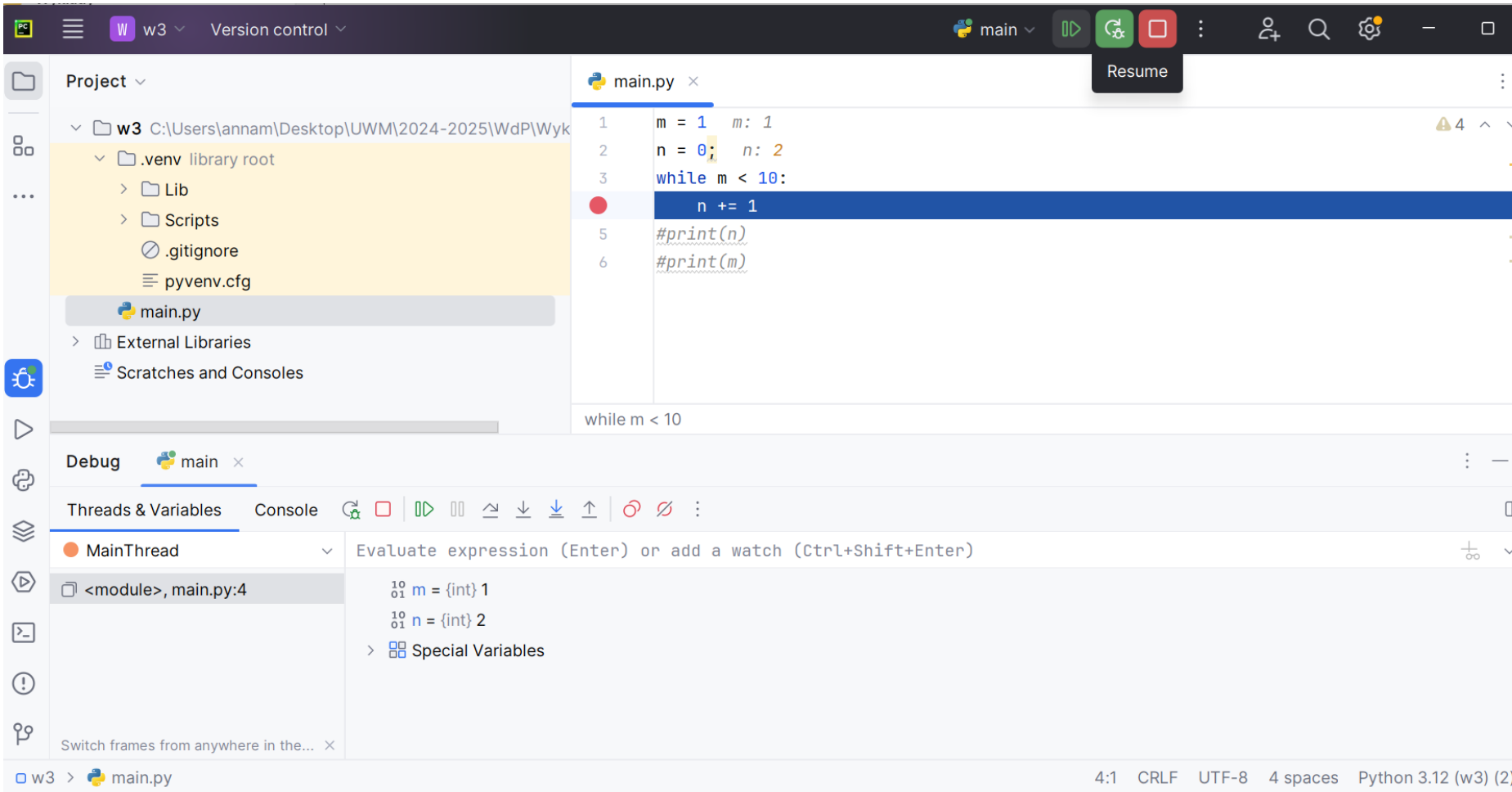
Niepoprawny program

```
m = 1
n = 0;
while m < 10:
    n += 1
#print(n)
#print(m)
```

Debugowanie



Debugowanie



Funkcje w Python

Funkcja - podprogram zwracający wynik na podstawie przekazanych argumentów.
Funkcje definiuje się używając słowa `def`. Po nim następuje nazwa identyfikująca funkcji, następnie para nawiasów, które mogą zawierać kilka nazw zmiennych (parametrów, argumentów), a na końcu dwukropek. Poniżej zaczyna się blok wyrażeń, które są częścią tej funkcji.
Ogólna składnia funkcji:

```
def nazwa_funkcji(parametr_1, parametr_2, ):
    polecenie_1
    polecenie_2
    ... i td.

    return wartosc
```

Instrukcja return powoduje zakończenie wykonywania funkcji i zwrócenie wartości.
W języku Python nazwy funkcji powinny być zgodne z konwencją `snake_case`, czyli wszystkie słowa z małych liter, oddzielone znakiem podkreślenia `_`. Ta konwencja określona jest przez twórców tego języka i większość programistów się jej trzyma. Nie jest nam ona obca, bo używamy jej również przy nazywaniu zmiennych.

Najprostsze funkcje

```
def przywitanie():
    # Blok należący do funkcji.
    print ("Pozdrowienia z mojej funckji!")
# Koniec funkcji.

przywitanie()   # Wywołanie funkcji.
przywitanie()   # Ponowne wywołanie funkcji.
```

```
przywitanie()   # Wywołanie funkcji.

def przywitanie():
    print ("Pozdrowienia z mojej funckji!")
```

Czyste funkcje

Funkcja czysta – funkcja, która nie ma żadnych skutków ubocznych, tzn. nie modyfikuje ani przekazanych argumentów, ani globalnego stanu programu.

```
a = 1

def foo():
    a = 2
    print (a)#2

foo() # Wywołanie funkcji.
print(a)#1
```

```
a = 1

def foo():
    print(a)#1

foo() # Wywołanie funkcji.
print(a)#1
```

Czyste funkcje i zmienne globalne 1

Funkcja, która nie jest czysta, jest związana ze zmiennymi globalnymi.

```
a = 1

def foo():
    a +=1
    print (a)

foo() # Wywołanie funkcji.
print(a)
```

```
a = 1

def foo():
    global a
    a += 1
    print (a)#2

foo() # Wywołanie funkcji.
print(a)#2
```

Czyste funkcje i zmienne globalne 2

Funkcja, która nie jest czysta, jest związana ze zmiennymi globalnymi.

```
def foo():
    a = 1
    print (a)

foo() # Wywołanie funkcji.
print(a)
```

```
def foo():
    #global a = 1 #nie dziala
    global a
    a = 1
    print (a)#1

foo() # Wywołanie funkcji.
print(a)#1
```

Funkcji z argumentami

Funkcję można wywołać z dowolnej liczby miejsc w programie. Wartości przekazywane do funkcji to argumenty.

Nie ma praktycznego ograniczenia długości funkcji, ale dobry projekt ma na celu funkcje, które wykonują jedno dobrze zdefiniowane zadanie. Złożone algorytmy należy podzielić na prostsze funkcje, gdy jest to możliwe.

Funkcja ma rozdzielaną przecinkami listę parametrów zero, z których każda ma nazwę, do której można uzyskać dostęp wewnątrz treści funkcji.

Domyślnie argumenty są przekazywane do funkcji według wartości, co oznacza, że funkcja odbiera kopię przekazanego obiektu.

Funkcja z parametrami

```
def przywitanie_imienne(imie, zyczenia):
    print ("Witaj" + imie + ". Zycze Tobie " + zyczenia)

przywitanie_imienne("Jacek", "zdrowia")#Wywołanie funkcji.
```

Witaj Jacek. Zycze Tobie zdrowia

```
def my_sum(a, b):
    print (a + b)

my_sum(3, 5)#Wywołanie funkcji.
```

8

Argumenty vs. Parametry

- Parametr to zmienna zdefiniowana w nawiasach podczas definiowania funkcji. Po prostu są one zapisywane, gdy deklarujemy funkcję.
- Argument to wartość przekazywana do funkcji podczas jej wywołania. Może to być zmienna, wartość lub obiekt przekazywany do funkcji lub metody jako dane wejściowe. Są one zapisywane podczas wywoływania funkcji.

```
def my_sum(a, b): #a,b, parametry
    print (a + b)

my_sum(3, 5)#Wywołanie funkcji, 3,5 argumenty
```

```
8
```

Funkcja z parametrami: wywołanie

```
def my_dif(a, b):
    print (a - b)

my_dif(3, 5)#Wywołanie funkcji. -2
my_dif(a = 3, b = 5)#2
my_dif(b = 3, a = 5)#2
#błędy:
#my_dif(a = 3, b)
#my_dif(b = 3, 5)
```

Argumenty 1

Wewnątrz funkcje są kopie argumentów. Nie zmieniać się wartości zewnątrz:

- ```
  def my_sum(a, b):
      a += b
      print (a)

  a = 2
  my_sum(a,2)
  print(a)
  4
  2
  ```
- ```
  def my_sum(a, b):
      a += b
      print(a)

  x = 2
  my_sum(x,2)
  print(x)
  4
  2
  ```

Argumenty 2

Funkcje dziala na wszystkich typach, dla ktorych dzialaja jej operacje:

```
def my_sum(a, b):
    a += b
    print (a)

my_sum('Hello', 'World')
my_sum(2.0, 3.5)
```

HelloWorld
5.5

```
def my_dif(a, b):
    print (a - b)

my_dif('Hello', 'World')
```

Funkcja, która zwraca wartość

- ```
  def przywitanie():
      return "Witaj"

  przywitanie() # Wywołanie funkcji.
  print(przywitanie() )
  a = przywitanie()
  print(a)
  ```
  Witaj
  Witaj
- ```
  def my_sum(a, b):
      a = + b
      return a

  x = 2
  y = 3
  print(my_sum(x,y))
  z = my_sum(x,y)
  print(z)
  ```
  3
  3

Funkcja zwraca None

W Python domyślnie funkcja zwraca **None**

- ```
  def my_sum(a, b):
      a = + b

  print(my_sum(3,5))
  ```
  None
- ```
  def przywitanie():
      print ("Witaj")

  przywitanie() # Wywołanie funkcji.
  print(przywitanie() )
  a = przywitanie()
  print(a)
  ```
  Witaj
  Witaj
  None
  Witaj
  None

Nie dziala po pierwszym return!

- ```
  def func():
      return 5
      print('Hello')
      return 3

  x = func()
  print(x)#5
  ```
- ```
  def my_abs(x):
      if x < 0:
          return -x
  #    else:
  #        return x
      return x


  print(my_abs(-10))#10
  print(my_abs(10))#10
  ```

Funkcja z argumentami domyślanymi

Ostatni parametr lub parametry w podpisie funkcji może być przypisany argument domyślny, co oznacza, że obiekt wywołujący może pominąć argument podczas wywoływania funkcji, chyba że chcą określić inną wartość.
Domyślny argument:

```
def przywitanie_imienne(imie, zyczenia = "zdrowia" ):
    print ("Witaj, " + imie + ". Zycze Tobie " + zyczenia)

przywitanie_imienne("Jacek") # Wywołanie funkcji z domyslnym argumentem
przywitanie_imienne("Jacek", "szczęścia") # Wywołanie funkcji.
przywitanie_imienne(imie ="Jacek")
```

Witaj, Jacek. Zycze Tobie zdrowia
Witaj, Jacek. Zycze Tobie szczęścia
Witaj, Jacek. Zycze Tobie szczęścia

```
def przywitanie_imienne(zyczenia = "zdrowia", imie ):
    print ("Witaj, " + imie + ". Zycze Tobie " + zyczenia)

przywitanie_imienne("Jacek") # Wywołanie funkcji z domyslnym argumen
przywitanie_imienne("Jacek", "szczęścia")
```

Funkcje: domyślny argument

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

  print(suma(12, 4))#16
  print(suma(3, 4, 5, 7))#19
  ```

- ```
  def prod(a, b=1, c=1, d=1):
      return a * b * c * d

  print(prod(12, 4))#48
  print(prod(3, 4, 5, 7))#420
  ```

Funkcje: dwie funkcje z rożną ilością argumentów nie są możliwe

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

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


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

Funkcja: dla różnych typów argumentów

```
def aFunction(a):
    if type(a) == str:
         print('you gave string as argument')
    if type(a) == int:
         print('you gave a int as argument')

aFunction('test')#you gave string as argument
aFunction(2.0)
aFunction(5)#you gave a int as argument
```

Przykłady funkcje

- Suma:

```
def suma(n):
    s = 0
    for i in range(1,n):
        s+=i
    return s

print(suma(6))#15
```

- Silnia:

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

print(silnia(6))#120
```

Przykłady funkcje: zwracanie dwóch wartości

```
def sumaprod(n):
    s = 0
    p = 1
    for i in range(1,n):
        s+=i
        p*=i
    return s, p

print(sumaprod(6))
a , b = sumaprod(6)
print(a)
print(b)
a = sumaprod(6)[0]
b = sumaprod(6)[1]
print(a,' ',b)
```

```
(15, 120)
15
120
15 120
```