**Function definitions**

https://docs.python.org/3/reference/compound\_stmts.html#function-definitions

https://www.python-course.eu/python3\_functions.php

**INTRODUCTION**

Why are we using functions?   
(a) To provide the structure for your code.   
(b) To reuse code (repeating code should be avoided; shorter code).   
(c) To make a safe namespace [scopes].   
(d) It is easier to remove bugs.

It lowers the cost for development and maintenance of the software.

Robić funkcję na całym kontenerze bo po kolei jest długo

+-------------+----------------------------+

| Statements | Examples |

+-------------+----------------------------+

| Calls | my\_func(12, word="alpha") | nazwa I argumenty

| def, return | def func(x, y=1): |return – funkcja zwraca wartosc

| | return x + y |

| global | def changer(): |

| | global x; x = "beta" |

| nonlocal | def changer(): # Python 3 | PEP 3104

| | nonlocal x; x = "beta" |

| yield | generators |

| lambda | anonymous functions |funkcje bez nazwy

+-------------+----------------------------+

**CREATING FUNCTIONS**

A function in Python is defined by a 'def' statement (a compound statement). The argument list may be empty. If a 'return' line is not present then 'return None' is assumed. Function names should be in a joined\_lower style.

The execution of a 'def' statement binds the function name in the current local namespace to a function object. The function body is executed when the function is called.

A function definition may be wrapped by one or more decorator expressions. Decorator expressions are evaluated when the function is defined, in the scope that contains the function definition.

# Syntax.

@decorator # optional, dodaje warstwę logiki oprócz ciałą funkji (np.

#zapis do pliku, extra komunikat na ekranie

def function\_name(argument\_list): #może być bez argumentów

docstring # optional, łańcuch dokumentacyjny

statements

return result # optional, może być w kilku miejscach

def do\_nothing(): # a placeholder, it can be changed in the future

pass # return None

# Recommended docstrings.

def first\_function(argument1, argument2):

"""Return the sum of arguments.""" # one line, to jest docstring

statements

def second\_function(argument):

"""First line.

(empty line) #docstring wyjdzie w help!

More lines with detailes.

"""

statements

def func(): ... # definition (statement)

func() # call (expression)

other\_name = func # assignment

other\_name() # call

func.atrr = value # defining an attribute for func

dir(func) # the list of attributes, see dir(len),lista atrybutów

func.\_\_name\_\_ # string, przechowuje nazwe funkcji jako string

func.\_\_doc\_\_ # docstring, przechowuje docstring

func(x, y) # two arguments

func((x, y)) # one argument (tuple)

**RETURNING MULTIPLE VALUES**

def find\_minmax(x, y):

"""Finding min and max simultaneously."""

return (x, y) if x < y else (y, x) # make a tuple with values

a, b = find\_minmax(5, 3) # (a, b) = (3, 5)

**EXERCISES**

# Create a function print\_in\_squares(word).

If word = "Python", them the result is

+---+---+---+---+---+---+

| P | y | t | h | o | n |

+---+---+---+---+---+---+

Try to build the whole string 'result' and print it at the end.

# Create a function print\_reversed\_pyramid(height).

>>> print\_reversed\_piramid(4)

\*\*\*\*\*\*\*

\*\*\*\*\*

\*\*\*

\*

**Polimorphism**

**INTRODUCTION**

Polimorphism means that the same function name can be used for different types. This makes programming more intuitive and easier.

def times(x, y):

"""Return the product of arguments."""

# The arguments should support the \* operator (the \_\_mul\_\_ method).

return x \* y

times(2, 3) # 6, int \* int

times(2, 3.14) # 6.28, int \* float

times("Bum!", 3) # "Bum!Bum!Bum!", str \* int

# Polymorphic len() function.

# Based on the \_\_len\_\_ method.

len("Python") # 6, the length of a string

len(["Python", "C++", "Java"]) # 3, the number of items

len({"Name": "Adam", "Age": 30}) # 2, the number of keys

len(set([1, 3, 5, 7])) # 4, the number of elements

Polymorphism is a very important concept in Object-Oriented Programming. Polymorphism in class methods: different (unrelated) classes can have a method with the same name.

**Builtin functions**

https://docs.python.org/3/library/functions.html

**INTRODUCTION**

Builtin functions are always available.

# Builtin objects in Python 2.

import \_\_builtin\_\_ # Py2

dir(\_\_builtin\_\_) # list of objects

# Builtin objects in Python 3.

import builtins # Py3

dir(builtins) # list of objects

zip is builtins.zip # True

zip = my\_better\_zip

Type names: basestring(), bool(), bytearray(), bytes(), complex(), dict(), float(), frozenset(), int(), list(), long() [Py2], set(), str(), tuple(), unicode() [Py2].

Mathematical functions and tools: abs(), all(), any(), ascii() [Py3], bin() , chr(), divmod(), enumerate(), eval(), exec(), filter() format(), hex(), iter(), len(), map(), max(), min(), next() [Py2.6+], oct(), open(), ord(), pow(), range(), repr(), reversed(), round(), sorted(), sum(), unichr() [Py2], xrange() [Py2], zip().

OOP: classmethod(), delattr(), dir(), getattr(), hasattr(), isinstance(), issubclass(), object(), property(), setattr(), staticmethod(), super(), type().

**FUNCTIONS FOR ITERABLES**

# zip(\*iterables) return a zip object #przyjmuje sekwencje jako argumenty, zwraca krotki, coś jak Transpose ale pracuje poki nie braknie par

# zip(\*iterables, strict=False) in Py3.10

# The strict=True is used to test if the lengths of iterables are identical.

# Iterate over several iterables in parallel, producing tuples with an item from each one.

# By default, zip() stops when the shortest iterable is exhausted.

list(zip([1, 2, 3], 'abc')) # [(1, 'a'), (2, 'b'), (3, 'c')]

list(zip([1, 2], 'abcd')) # [(1, 'a'), (2, 'b')], different lengths

# map(function, iterable, ...) return a map object (iterator) in Py3

def cube(x):

return x \* x \* x

list(map(cube, range(10))) # map object

list(cube(x) for x in range(10)) # generator object

# [0, 1, 8, 27, 64, 125, 216, 343, 512, 729]

L = [(1, 2), (3, 4, 5)]

list(map(sum, L)) # [3, 12] map object

list(sum(t) for t in L) # [3, 12] generator object

L = [1, 2, 3]

M = [4, 5, 6]

list(map(sum, zip(L, M))) # [5, 7, 9]

list(sum(t) for t in zip(L, M))

def calc(x, y): # two arguments and two iterables in map()

return 2 \* x + y

R = range(10)

list(map(calc, R, R))

list(calc(x, y) for (x, y) in zip(R, R))

# [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]

# filter(function or None, iterable) return a filter object (iterator) in Py3

def check(x):

return x % 2 != 0 and x % 3 != 0

list(filter(check, range(30))) # filter object

list(x for x in range(30) if check(x)) # generator object, equivalent

# [1, 5, 7, 11, 13, 17, 19, 23, 25, 29]

list(filter(None, [-2,-1,0,1,2])) # [-2, -1, 1, 2] filter object

list(x for x in [-2,-1,0,1,2] if x) # generator object, equivalent

**Scopes**

PEP 3104 - Access to Names in Outer Scopes. The specification for the nonlocal statement.

https://realpython.com/python-namespaces-scope/   
Namespaces and Scope in Python

**INTRODUCTION**

A 'scope' defines the visibility of a name/variable within a block. #zakres

A 'namespace' is a collection of currently defined symbolic names along with information about the object that each name references. You can think of a namespace as a dictionary in which the keys are the object names and the values are the objects themselves.

There are four types of namespaces in a Python program (LEGB rule):   
(1) Local   
(2) Enclosing – jak definijemy funkcje wewnątrz funkcji to jest to ta outer warstwa  
(3) Global   
(4) Builtin

def my\_len(sequence): # sequence is local

"""Return the length of a sequence."""

length = 0 # length is local

for item in sequence: # item is local

length += 1

return length

X = 99 # global name

def func(): # local namespace

#print(X) # UnboundLocalError, X will be local

X = 88 # local name

print(X) # 88, reference to the local X

func() # 88

print(X) # 99, X is global here

X = 99 # global name

def func():

global Z # global statement (declaration)

print(X) # reading global name

Y = 88 # local name

Z = 77 # Z is created

#print(Z) # NameError, Z is not defined

X = 55 # global X changed

func() # 55, here the function body is executed

#print(Y) # NameError, Y is not defined

print(Z) # 77, Z is global

X = 99 # global name

def func():

global X # global statement

print(X) # OK

X = 88 # global X changed

func() # 99

print(X) # 88

# PEP 227 - Statically Nested Scopes

#enclosing

x = 1

def outer():

#from math import \* # SyntaxError: import \* is not allowed ...

import math # possible

x = 2

def inner(): # nested scope

print(x)

inner()

outer() # print 2 from nested scope, not 1 from global scope

#print(math.sin(1)) # NameError: name 'math' is not defined

def make\_score(score=0):

for i in [1, 2, 3, 4]:

def increment(step=i):

#global score

#NameError: name 'score' is not defined

nonlocal score # if not, there is an exception (Py3)

#UnboundLocalError: local variable 'score' referenced before assignment

# 'score' is not local and it is not global

score = score + step

print(score)

increment()

make\_score() # 1 3 6 10

#nonlocal – gdy definiujemy zakres zmiennej, która jest w enclosing

**Passing arguments**

**DEFAULT ARGUMENTS**

Default parameter values are evaluated from left to right when the function definition is executed. This means that the expression is evaluated once, when the function is defined, and that the same “pre-computed” value is used for each call.

def print\_word(word, n=1):

for i in range(n):

print("{} {}".format(i, word))

print\_word("spam") # default n=1

print\_word("spam", 3) # n=3

print\_word(n=4, word="spam") # using keywords 'n' and 'word', wywołanie funkcji z użyciem słów kluczowych (nazw argumentów)

# Problems with mutable default arguments.

def tester1(x, L=[]): # L is created once

L.append(x)

print("tester1 {}".format(L))

def tester2(x, L=None):

if L is None: # explicit test for None

L = []

L.append(x)

print("tester2 {}".format(L))

tester1(1) # [1]

tester1(2) # [1, 2]

tester1(3) # [1, 2, 3]

tester2(1) # [1]

tester2(2) # [2]

tester2(3) # [3]

# Using an external list.

M = []

tester1(5, M) # M == [5]

tester2(6, M) # M == [5, 6]

**EXCESS POSITIONAL ARGUMENTS**

# '\*identifier' is initialized to a tuple with excess positional arguments

def unique\_letters(\*arguments): #dowolna liczba arg, krotka

letters = set()

for word in arguments: # 'arguments' is a tuple

letters.update(word)

return letters

unique\_letters("abc", "cde", "efg") # set(['a', 'e', 'f', 'b', 'c', 'g', 'd'])

**KEYWORD ARGUMENTS**

# '\*\*identifier' is initialized to a new ordered mapping receiving

# any excess keyword arguments

def university(\*\*keywords):

for key in sorted(keywords): # keywords is a dict

print("{} : {}".format(key, keywords[key]))

university(rektor="Jacek Popiel", kanclerz="Monika Harpula")

# kanclerz : Monika Harpula

# rektor : Jacek Popiel

**EXAMPLE**

def f(a1, a2=v2, \*a3, \*\*a4): pass

# a function call f(x1, x2, x3, x4, b1=x5, b2=x6)

# a1=x1, positional argument

# a2=x2, positional argument (default value will not be used)

# a3=(x3, x4), excess positional arguments

# a4={"b1":x5, "b2":x6}, excess keyword arguments

# The most general form is

def g(\*arguments, \*\*keywords): pass

**UNPACKING ARGUMENTS**

args = (1, 5)

range(\*args) # range(1, 5)

def describe(name, age=0, position=None):

print("{} age {} position {}".format(name, age, position))

person = {"name": "Bogdan", "age": 30, "position": "teacher"}

describe("Adam", 20, "student")

# Adam age 20 position student

describe(\*\*person)

#describe(name="Bogdan", age=30, position="teacher") # the same

# Bogdan age 30 position teacher

**FORCING ARGUMENTS**

def func(pos1, pos2, /, pos\_or\_kwd, \*, kwd1, kwd2): pass

def func\_pos\_only(arg, /): pass # func\_pos\_only(25)

def func\_kwd\_only(\*, arg): pass # func\_kwd\_only(arg=34)

# Exercise: help(sorted)

sorted(iterable, /, \*, key=None, reverse=False)

# Exercise: help(abs)

abs(x, /)

**Lambdas**

https://docs.python.org/3/reference/expressions.html#lambda

**INTRODUCTION**

Lambda expressions (sometimes called lambda forms) are used to create anonymous functions. Functions created with lambda expressions cannot contain statements or annotations.

# Syntax.

lambda arguments: expression # yields a function object

# The unnamed object behaves like a function object defined with:

def <lambda>(arguments):

return expression

sum\_items = lambda x, y, z: x + y + z

print ( sum\_items(2, 3, 4) ) # 9

def sum\_items(x, y, z): # the same result

return x + y + z

# Default arguments.

sum\_items = lambda x=0, y=0, z=0: x + y + z

print ( sum\_items() ) # 0

print ( sum\_items(6, 7) ) # 13

**CUSTOM SORTING**

number\_list.sort(reverse=True)

number\_list.sort(key=lambda x: -x)

L = [(2, "gamma"), (-7, "alpha"), (4, "beta"), (4, "delta")] # a list of records

L.sort(key=lambda t: (-t[0], len(t[1])))

# t[0] decreasing, for the same t[0] use the length of t[1]

# [(4, 'beta'), (4, 'delta'), (2, 'gamma'), (-7, 'alpha')]

L.sort(key=lambda t: (t[1], abs(t[0])))

# t[1] increasing (lexicographic order), for the same t[1] use abs(t[0])

# [(-7, 'alpha'), (4, 'beta'), (4, 'delta'), (2, 'gamma')]

**ACTIONS**

def show\_sequence(sequence, action=None):

for item in sequence:

print ( item )

if action:

action(item)

show\_sequence([1, 2, 3, 4], action=lambda x: print(x\*x)) # Python 3

M = []

show\_sequence([1, 2, 3, 4], action=lambda x: M.append(x\*x))

# Example: BFS or DFS in graphs.

**EXERCISES**

Define the following functions using lambda: abs(x), min(x, y), max(x, y).

**Generators**

https://docs.python.org/3/reference/expressions.html#yieldexpr

https://rmariano.eu/posts/exploring-generators-and-coroutines/   
Exploring Generators and Coroutines   
PEP-255 (Simple Generators) [yield is a statement; lazy evaluation]   
PEP-342 (Coroutines via Enhanced Generators) [yield is an expression; send(), throw(), close()]   
PEP-380 (Syntax for delegating to a Sub-Generator) [return in generators; yield from]   
PEP-525 (Asynchronous Generators)

https://snarky.ca/how-the-heck-does-async-await-work-in-python-3-5/   
How the heck does async/await work in Python 3.5? [long essey]

**INTRODUCTION**

Use cases of generators:   
(a) Working with data streams or large files, like CSV files.   
(b) Generating an infinite sequence.

**GENERATOR EXPRESSIONS**

This is an expression that returns an iterator.

# sum of squares 0, 1, 4, ... 81

alist = [i\*i for i in range(10)] # list comprehension

gen = (i\*i for i in range(10)) # generator expression

result = sum(i\*i for i in range(10))

**GENERATOR FUNCTIONS**

This is a function which returns a generator iterator. It looks like a normal function except that it contains yield expressions (instead of return) for producing a series of values.

def iter\_squares(n):

for i in range(n):

yield i\*i

result = sum(iter\_squares(10))

def my\_range(stop):

"""My version of range(stop)."""

value = 0

while value < stop:

yield value # yield instead of return

value = value + 1

# Example 1 - using for loop (hidden iteration protocole).

for i in my\_range(10):

print(i)

# Example 2 - explicit iteration protocole.

gen = my\_range(3)

print(next(gen)) # 0

print(next(gen)) # 1

print(next(gen)) # 2

print(next(gen)) # StopIteration, gen is exhausted

print(next(gen)) # StopIteration

**INFINITE ITERATORS**

def iter\_squares2(): # an infinite iterator

i = 0

while True:

yield i\*i

i += 1

def fibonacci(): # an infinite iterator

"""The Fibonacci sequence."""

a, b = 0, 1

yield a

yield b

while True:

a, b = b, a + b

yield b

# Example 1 - using for loop (hidden iteration protocole).

for i in fibonacci():

print(i)

if i > 100: # we have to break the loop manually

break

# 0 1 1 2 3 5 8 13 21 34 55 89 144

# Example 2 - the generator is alive!

fib = fibonacci() # fib is a generator object

for i in fib:

print(i)

if i > 100:

break

print(next(fib)) # 233

print(next(fib)) # 377

**SPEED VS MEMORY**

List comprehensions can be faster to evaluate than the equivalent generator expression.

import sys

import timeit

sqr\_list = [i\*i for i in range(1000000)]

print(sys.getsizeof(sqr\_list)) # 8697464 bytes

sqr\_gen = (i\*i for i in range(1000000))

print(sys.getsizeof(sqr\_gen)) # 120 bytes

t1 = timeit.Timer(lambda: sum(sqr\_list))

print(t1.timeit(1)) # 0.00961924200237263

t2 = timeit.Timer(lambda: sum(sqr\_gen))

print(t2.timeit(1)) # 0.05263053500675596 (5 times slower)

**EXAMPLES**

https://realpython.com/introduction-to-python-generators/

# Problem: couting rows in a text file.

txt\_gen = txt\_reader("some\_file.txt")

line\_count = 0

for line in txt\_gen:

line\_count += 1

def txt\_reader(file\_name): # possible MemoryError

file = open(file\_name, "r")

result = file.read() # loads everything into memory at once

result = result.rstrip() # remove last "\n" if present

result = result.split("\n") # divide to lines

return result # list of strings without "\n"

def txt\_reader(file\_name): # a generator function

for line in open(file\_name, "r"):

yield line # lines with "\n" (the last line may be an exception)

\

**Recursion**

https://en.wikipedia.org/wiki/Recursion

**INTRODUCTION**

A class of objects or methods exhibits recursive behavior when it can be defined by two properties:   
(1) a simple base case (or cases) - a terminating scenario that does not use recursion to produce an answer,   
(2) a recursive step - a set of rules that reduces all other cases toward the base case.

def print\_stars(n):

if n > 0: # pass for the base case n=0

print("\*")

print\_stars(n-1)

print\_stars(5)

# 0! = 1, 1! = 1, n! = n\*(n-1)!

def factorial(n):

if n == 0 or n == 1: # base cases

return 1

else:

return n \* factorial(n-1)

import math

print(math.factorial(10)) # Python 2.6+

print(factorial(10))

# The Fibonacci sequence

# f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2)

def fibonacci(n):

if n == 0 or n == 1: # base cases

return n

else:

return fibonacci(n-1) + fibonacci(n-2)

# binomial(n, k) = factorial(n) / (factorial(k) \* factorial(n-k))

# binomial(n, 0) = binomial(n, n) = 1

def binomial(n, k):

if k == 0 or k == n:

return 1

else:

return binomial(n-1, k-1) + binomial(n-1, k)

\

**Annotations**

PEP 3107 - Function Annotations

PEP 526 - Syntax for Variable Annotations

https://docs.python.org/3/howto/annotations.html   
Annotations Best Practices

**FUNCTION ANNOTATION**

Function arguments may have an annotation of the form ': expression' following the parameter name. Any parameter may have an annotation, even those of the form \*identifier or \*\*identifier. Functions may have 'return' annotation of the form '-> expression' after the parameter list.

The annotation values are available as values of a dictionary keyed by the parameters’ names in the \_\_annotations\_\_ attribute of the function object.

# Syntax.

def function\_name(arg1: expression, arg2: expression = value) -> expression:

statements

def function\_name(\*arguments: expression, \*\*keywords: expression)-> expression:

statements

# Function annotations are usually used for type hints.

def sum\_two\_numbers(a: int, b: int) -> int:

return a + b

sum\_two\_numbers.\_\_annotations\_\_

# {'a': <class 'int'>, 'b': <class 'int'>, 'return': <class 'int'>}

**VARIABLE ANNOTATION**

# When annotating a variable or a class attribute, assignment is optional:

class C:

field: 'annotation' # a class attribute annotation

# Variable annotations are usually used for type hints.

count: int = 0 # a variable annotation

some\_list: List[int] = [1, 3] # from 'typing' module

some\_list: list[int] = [1, 3] # Py3.9+

some\_dict: Dict[str, float] = {"a": 1.2} # from 'typing' module

some\_dict: dict[str, float] = {"a": 1.2} # Py3.9+

coords: Tuple[float, float] = (1.2, 3.4) # from 'typing' module

coords: tuple[float, float] = (1.2, 3.4) # Py3.9+

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.