**MODULE 2**

1. The print() function is a **built-in** function. It prints/outputs a specified message to the screen/consol window.

2. Built-in functions, contrary to user-defined functions, are always available and don't have to be imported. Python 3.7.1 comes with 69 built-in functions. You can find their full list provided in alphabetical order in the [Python Standard Library](https://docs.python.org/3/library/functions.html).

3. To call a function (**function invocation**), you need to use the function name followed by parentheses. You can pass arguments into a function by placing them inside the parentheses. You must separate arguments with a comma, e.g., print("Hello,", "world!"). An "empty" print() function outputs an empty line to the screen.

4. Python strings are delimited with **quotes**, e.g., "I am a string", or 'I am a string, too'.

5. Computer programs are collections of **instructions**. An instruction is a command to perform a specific task when executed, e.g., to print a certain message to the screen.

6. In Python strings the **backslash** (\) is a special character which announces that the next character has a different meaning, e.g., \n (the **newline character**) starts a new output line.

7. **Positional arguments** are the ones whose meaning is dictated by their position, e.g., the second argument is outputted after the first, the third is outputted after the second, etc.

8. **Keyword arguments** are the ones whose meaning is not dictated by their location, but by a special word (keyword) used to identify them.

9. The end and sep parameters can be used for formatting the output of the print() function. The sep parameter specifies the separator between the outputted arguments (e.g., print("H", "E", "L", "L", "O", sep="-"), whereas the end parameter specifies what to print at the end of the print statement.

Look at the table below:

|  |  |  |
| --- | --- | --- |
| **Priority** | **Operator** |  |
| 1 | +, - | unary |
| 2 | \*\* |  |
| 3 | \*, /, % |  |
| 4 | +, - | binary |

Note: we've enumerated the operators in order **from the highest (1) to the lowest (4) priorities**.

**Key takeaways**

1. An **expression** is a combination of values (or variables, operators, calls to functions - you will learn about them soon) which evaluates to a value, e.g., 1 + 2.

2. **Operators** are special symbols or keywords which are able to operate on the values and perform (mathematical) operations, e.g., the \* operator multiplies two values: x \* y.

3. Arithmetic operators in Python: + (addition), - (subtraction), \* (multiplication), / (classic division - returns a float if one of the values is of float type), % (modulus - divides left operand by right operand and returns the remainder of the operation, e.g., 5 % 2 = 1), \*\* (exponentiation - left operand raised to the power of right operand, e.g., 2 \*\* 3 = 2 \* 2 \* 2 = 8), // (floor/integer division - returns a number resulting from division, but rounded down to the nearest whole number, e.g., 3 // 2.0 = 1.0)

4. A **unary** operator is an operator with only one operand, e.g., -1, or +3.

5. A **binary** operator is an operator with two operands, e.g., 4 + 5, or 12 % 5.

6. Some operators act before others - **the hierarchy of priorities**:

* unary + and - have the highest priority
* then: \*\*, then: \*, /, and %, and then the lowest priority: binary + and -.

7. Subexpressions in **parentheses** are always calculated first, e.g., 15 - 1 \* (5 \* (1 + 2)) = 0.

8. The **exponentiation** operator uses **right-sided binding**, e.g., 2 \*\* 2 \*\* 3 = 256.

# Keywords

Take a look at the list of words that play a very special role in every Python program.

['False', 'None', 'True', 'and', 'as', 'assert', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

They are called **keywords** or (more precisely) **reserved keywords**. They are reserved because **you mustn't use them as names**: neither for your variables, nor functions, nor any other named entities you want to create.

The meaning of the reserved word is **predefined**, and mustn't be changed in any way.

# Key takeaways

1. The print() function **sends data to the console**, while the input() function **gets data from the console**.

2. The input() function comes with an optional parameter: **the prompt string**. It allows you to write a message before the user input, e.g.:

name = input("Enter your name: ") print("Hello, " + name + ". Nice to meet you!")

3. When the input() function is called, the program's flow is stopped, the prompt symbol keeps blinking (it prompts the user to take action when the console is switched to input mode) until the user has entered an input and/or pressed the *Enter* key.

NOTE

You can test the functionality of the input() function in its full scope locally on your machine. For resource optimization reasons, we have limited the maximum program execution time in Edube to a few seconds. Go to Sandbox, copy-paste the above snippet, run the program, and do nothing - just wait a few seconds to see what happens. Your program should be stopped automatically after a short moment. Now open IDLE, and run the same program there - can you see the difference?

Tip: the above-mentioned feature of the input() function can be used to prompt the user to end a program. Look at the code below:

name = input("Enter your name: ") print("Hello, " + name + ". Nice to meet you!") print("\nPress Enter to end the program.") input() print("THE END.")

3. The result of the input() function is a string. You can add strings to each other using the concatenation (+) operator. Check out this code:

num1 = input("Enter the first number: ") # Enter 12 num2 = input("Enter the second number: ") # Enter 21 print(num1 + num2) # the program returns 1221

4. You can also multiply (\* - replication) strings, e.g.:

myInput = ("Enter something: ") # Example input: hello print(myInput \* 3) # Expected output: hellohellohello

**MODULE 3**

Now we need to update our **priority table**, and put all the new operators into it. It now looks as follows:

|  |  |  |
| --- | --- | --- |
| **Priority** | **Operator** |  |
| 1 | +, - | unary |
| 2 | \*\* |  |
| 3 | \*, /, //, % |  |
| 4 | +, - | binary |
| 5 | <, <=, >, >= |  |
| 6 | ==, != |  |

These two instructions are:

* break - exits the loop immediately, and unconditionally ends the loop's operation; the program begins to execute the nearest instruction after the loop's body;
* continue - behaves as if the program has suddenly reached the end of the body; the next turn is started and the condition expression is tested immediately.

**Key takeaways**

1. There are two types of loops in Python: while and for:

* the while loop executes a statement or a set of statements as long as a specified boolean condition is true, e.g.:

# Example 1 while True: print("Stuck in an infinite loop.") # Example 2 counter = 5 while counter > 2: print(counter) counter -= 1

* the for loop executes a set of statements many times; it's used to iterate over a sequence (e.g., a list, a dictionary, a tuple, or a set - you will learn about them soon) or other objects that are iterable (e.g., strings). You can use the for loop to iterate over a sequence of numbers using the built-in range function. Look at the examples below:

# Example 1 word = "Python" for letter in word: print(letter, end="\*") # Example 2 for i in range(1, 10): if i % 2 == 0: print(i)

2. You can use the break and continue statements to change the flow of a loop:

* You use break to exit a loop, e.g.:

text = "OpenEDG Python Institute" for letter in text: if letter == "P": break print(letter, end="")

* You use continue to skip the current iteration, and continue with the next iteration, e.g.:

text = "pyxpyxpyx" for letter in text: if letter == "x": continue print(letter, end="")

3. The while and for loops can also have an else clause in Python. The else clause executes after the loop finishes its execution as long as it has not been terminated by break, e.g.:

n = 0 while n != 3: print(n) n += 1 else: print(n, "else") print() for i in range(0, 3): print(i) else: print(i, "else")

4. The range() function generates a sequence of numbers. It accepts integers and returns range objects. The syntax of range() looks as follows: range(start, stop, step), where:

* start is an optional parameter specifying the starting number of the sequence (0 by default)
* stop is an optional parameter specifying the end of the sequence generated (it is not included),
* and step is an optional parameter specifying the difference between the numbers in the sequence (1 by default.)

Example code:

for i in range(3): print(i, end=" ") # outputs: 0 1 2 for i in range(6, 1, -2): print(i, end=" ") # outputs: 6, 4, 2

Here are all of them:

* & (ampersand) - bitwise conjunction;
* | (bar) - bitwise disjunction;
* ~ (tilde) - bitwise negation;
* ^ (caret) - bitwise exclusive or (xor).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bitwise operations (&, |, and ^)** | | | | |
| Arg A | Arg B | Arg B & Arg B | Arg A | Arg B | Arg A ^ Arg B |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |

|  |  |
| --- | --- |
| **Bitwise operations (~)** | |
| Arg | ~Arg |
| 0 | 1 |
| 1 | 0 |

the arguments of these operators **must be integers**; we must not use floats here.

And here is the **updated priority table**, containing all the operators introduced so far:

|  |  |  |
| --- | --- | --- |
| **Priority** | **Operator** |  |
| 1 | ~, +, - | unary |
| 2 | \*\* |  |
| 3 | \*, /, //, % |  |
| 4 | +, - | binary |
| 5 | <<, >> |  |
| 6 | <, <=, >, >= |  |
| 7 | ==, != |  |
| 8 | & |  |
| 9 | | |  |
| 10 | =, +=, -=, \*=, /=, %=, &=, ^=, |=, >>=, <<= |  |

**Key takeaways**

1. Python supports the following logical operators:

* and → if both operands are true, the condition is true, e.g., (True and True) is True,
* or → if any of the operands are true, the condition is true, e.g., (True or False) is True,
* not → returns false if the result is true, and returns true if the result is false, e.g., not True is False.

2. You can use bitwise operators to manipulate single bits of data. The following sample data:

* x = 15, which is 0000 1111 in binary,
* y = 16, which is 0001 0000 in binary.

will be used to illustrate the meaning of bitwise operators in Python. Analyze the examples below:

* & does a *bitwise and*, e.g., x & y = 0, which is 0000 0000 in binary,
* | does a *bitwise or*, e.g., x | y = 31, which is 0001 1111 in binary,
* ˜ does a *bitwise not*, e.g., ˜ x = 240, which is 1111 0000 in binary,
* ^ does a *bitwise xor*, e.g., x ^ y = 31, which is 0001 1111 in binary,
* >> does a *bitwise right shift*, e.g., y >> 1 = 8, which is 0000 1000 in binary,
* << does a *bitwise left shift*, e.g., y << 3 = , which is 1000 0000 in binary,

# Key takeaways

1. The **list is a type of data** in Python used to **store multiple objects**. It is an **ordered and mutable collection** of comma-separated items between square brackets, e.g.:

myList = [1, None, True, "I am a string", 256, 0]

2. Lists can be **indexed and updated**, e.g.:

myList = [1, None, True, 'I am a string', 256, 0] print(myList[3]) # outputs: I am a string print(myList[-1]) # outputs: 0 myList[1] = '?' print(myList) # outputs: [1, '?', True, 'I am a string', 256, 0] myList.insert(0, "first") myList.append("last") print(myList) # outputs: ['first', 1, '?', True, 'I am a string', 256, 0, 'last']

3. Lists can be **nested**, e.g.: myList = [1, 'a', ["list", 64, [0, 1], False]].

You will learn more about nesting in module 3.1.7 - for the time being, we just want you to be aware that something like this is possible, too.

4. List elements and lists can be **deleted**, e.g.:

myList = [1, 2, 3, 4] del myList[2] print(myList) # outputs: [1, 2, 4] del myList # deletes the whole list

Again, you will learn more about this in module 3.1.6 - don't worry. For the time being just try to experiment with the above code and check how changing it affects the output.

5. Lists can be **iterated** through using the for loop, e.g.:

myList = ["white", "purple", "blue", "yellow", "green"] for color in myList: print(color)

6. The len() function may be used to **check the list's length**, e.g.:

myList = ["white", "purple", "blue", "yellow", "green"] print(len(myList)) # outputs 5 del myList[2] print(len(myList)) # outputs 4

7. A typical **function** invocation looks as follows: result = function(arg), while a typical **method** invocation looks like this:result = data.method(arg).

# Key takeaways

1. You can use the sort() method to sort elements of a list, e.g.:

lst = [5, 3, 1, 2, 4] print(lst) lst.sort() print(lst) # outputs: [1, 2, 3, 4, 5]

2. There is also a list method called reverse(), which you can use to reverse the list, e.g.:

lst = [5, 3, 1, 2, 4] print(lst) lst.reverse() print(lst) # outputs: [4, 2, 1, 3, 5]

# Key takeaways

1. If you have a list l1, then the following assignment: l2 = l1 does not make a copy of the l1 list, but makes the variables l1 and l2 **point to one and the same list in memory**. For example:

vehiclesOne = ['car', 'bicycle', 'motor'] print(vehiclesOne) # outputs: ['car', 'bicycle', 'motor'] vehiclesTwo = vehiclesOne del vehiclesOne[0] # deletes 'car' print(vehiclesTwo) # outputs: ['bicycle', 'motor']

2. If you want to copy a list or part of the list, you can do it by performing **slicing**:

colors = ['red', 'green', 'orange'] copyWholeColors = colors[:] # copy the whole list copyPartColors = colors[0:2] # copy part of the list

3. You can use **negative indices** to perform slices, too. For example:

sampleList = ["A", "B", "C", "D", "E"] newList = sampleList[2:-1] print(newList) # outputs: ['C', 'D']

4. The start and end parameters are **optional** when performing a slice: list[start:end], e.g.:

myList = [1, 2, 3, 4, 5] sliceOne = myList[2: ] sliceTwo = myList[ :2] sliceThree = myList[-2: ] print(sliceOne) # outputs: [3, 4, 5] print(sliceTwo) # outputs: [1, 2] print(sliceThree) # outputs: [4, 5]

5. You can **delete slices** using the del instruction:

myList = [1, 2, 3, 4, 5] del myList[0:2] print(myList) # outputs: [3, 4, 5] del myList[:] print(myList) # deletes the list content, outputs: []

6. You can test if some items **exist in a list or not** using the keywords in and not in, e.g.:

myList = ["A", "B", 1, 2] print("A" in myList) # outputs: True print("C" not in myList) # outputs: True print(2 not in myList) # outputs: False

# Key takeaways

1. **List comprehension** allows you to create new lists from existing ones in a concise and elegant way. The syntax of a list comprehension looks as follows:

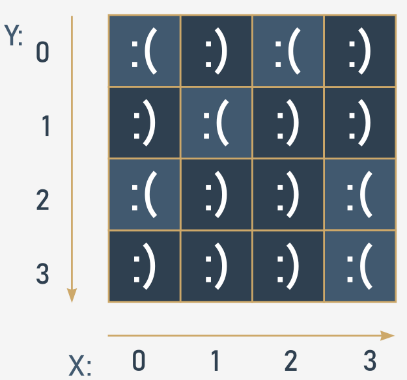
[expression for element in list if conditional]  
which is actually an equivalent of the following code:

for element in list: if conditional: expression

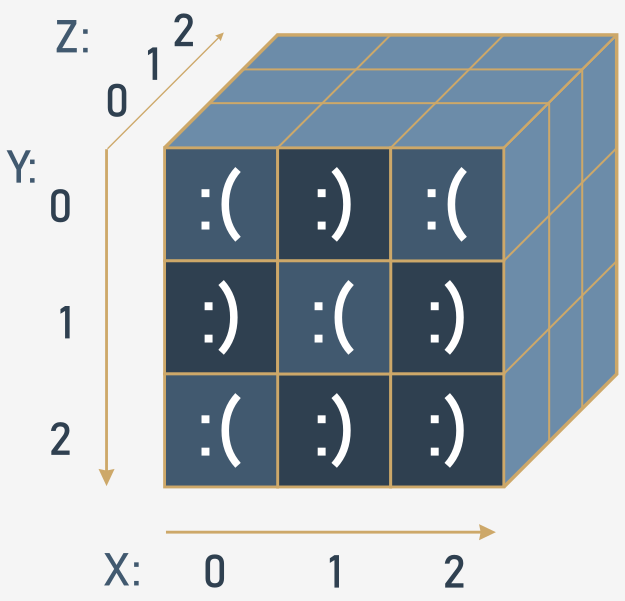
Here's an example of a list comprehension - the code creates a five-element list filled with with the first five natural numbers raised to the power of 3:

cubed = [num \*\* 3 for num in range(5)] print(cubed) # outputs: [0, 1, 8, 27, 64]

2. You can use **nested lists** in Python to create **matrices** (i.e., two-dimensional lists). For example:

  
  
# A four-column/four-row table - a two dimensional array (4x4) table = [[":(", ":)", ":(", ":)"], [":)", ":(", ":)", ":)"], [":(", ":)", ":)", ":("], [":)", ":)", ":)", ":("]] print(table) print(table[0][0]) # outputs: ':(' print(table[0][3]) # outputs: ':)'

3. You can nest as many lists in lists as you want, and therefore create n-dimensional lists, e.g., three-, four- or even sixty-four-dimensional arrays. For example:

  
  
# Cube - a three-dimensional array (3x3x3) cube = [[[':(', 'x', 'x'], [':)', 'x', 'x'], [':(', 'x', 'x']], [[':)', 'x', 'x'], [':(', 'x', 'x'], [':)', 'x', 'x']], [[':(', 'x', 'x'], [':)', 'x', 'x'], [':)', 'x', 'x']]] print(cube) print(cube[0][0][0]) # outputs: ':(' print(cube[2][2][0]) # outputs: ':)'

# Key takeaways

1. A **function** is a block of code that performs a specific task when the function is called (invoked). You can use functions to make your code reusable, better organized, and more readable. Functions can have parameters and return values.

2. There are at least four basic types of functions in Python:

* **built-in functions** which are an integral part of Python (such as the print() function). You can see a complete list of Python built-in functions at <https://docs.python.org/3/library/functions.html>.
* the ones that come from **pre-installed modules** (you'll learn about them in *Module 5* of this course)
* **user-defined functions** which are written by users for users - you can write your own functions and use them freely in your code,
* the lambda functions (you'll learn about them in *Module 6* of this course.)

3. You can define your own function using the def keyword and the following syntax:

def yourFunction(optional parameters): # the body of the function

You can define a function which doesn't take any arguments, e.g.:

def message(): # defining a function print("Hello") # body of the function message() # calling the function

You can define a function which takes arguments, too, just like the one-parameter function below:

def hello(name): # defining a function print("Hello,", name) # body of the function name = input("Enter your name: ") hello(name) # calling the function

**Key takeaways**

1. You can pass information to functions by using parameters. Your functions can have as many parameters as you need.

An example of a one-parameter function:

def hi(name): print("Hi,", name) hi("Greg")

An example of a two-parameter function:

def hiAll(name1, name2): print("Hi,", name2) print("Hi,", name1) hiAll("Sebastian", "Konrad")

An example of a three-parameter function:

def address(street, city, postalCode): print("Your address is:", street, "St.,", city, postalCode) s = input("Street: ") pC = input("Postal Code: ") c = input("City: ") address(s, c, pC)

2. You can pass arguments to a function using the following techniques:

* **positional argument passing** in which the order of arguments passed matters (Ex. 1),
* **keyword (named) argument passing** in which the order of arguments passed doesn't matter (Ex. 2),
* a mix of positional and keyword argument passing (Ex. 3).

Ex. 1 def subtra(a, b): print(a - b) subtra(5, 2) # outputs: 3 subtra(2, 5) # outputs: -3 Ex. 2 def subtra(a, b): print(a - b) subtra(a=5, b=2) # outputs: 3 subtra(b=2, a=5) # outputs: 3 Ex. 3 def subtra(a, b): print(a - b) subtra(5, b=2) # outputs: 3 subtra(5, 2) # outputs: 3

It's important to remember that **positional arguments mustn't follow keyword arguments**. That's why if you try to run the following snippet:

def subtra(a, b): print(a - b) subtra(5, b=2) # outputs: 3 subtra(a=5, 2) # Syntax Error

# Key takeaways

1. You can use the return keyword to tell a function to return some value. The return statement exits the function, e.g.:

def multiply(a, b): return a \* b print(multiply(3, 4)) # outputs: 12 def multiply(a, b): return print(multiply(3, 4)) # outputs: None

2. The result of a function can be easily assigned to a variable, e.g.:

def wishes(): return "Happy Birthday!" w = wishes() print(w) # outputs: Happy Birthday!

Look at the difference in output in the following two examples:

# Example 1 def wishes(): print("My Wishes") return "Happy Birthday" wishes() # outputs: My Wishes # Example 2 def wishes(): print("My Wishes") return "Happy Birthday" print(wishes()) # outputs: My Wishes # Happy Birthday

3. You can use a list as a function's argument, e.g.:

def hiEverybody(myList): for name in myList: print("Hi,", name) hiEverybody(["Adam", "John", "Lucy"])

4. A list can be a function result, too, e.g.:

def createList(n): myList = [] for i in range(n): myList.append(i) return myList print(createList(5))

# Key takeaways

1. A variable that exists outside a function has a scope inside the function body (Example 1) unless the function defines a variable of the same name (Example 2, and Example 3), e.g.:

Example 1:

var = 2 def multByVar(x): return x \* var print(multByVar(7)) # outputs: 14

Example 2:

def mult(x): var = 5 return x \* var print(mult(7)) # outputs: 35

Example 3:

def multip(x): var = 7 return x \* var var = 3 print(multip(7)) # outputs: 49

2. A variable that exists inside a function has a scope inside the function body (Example 4), e.g.:

Example 4:

def adding(x): var = 7 return x + var print(adding(4)) # outputs: 11 print(var) # NameError

3. You can use the global keyword followed by a variable name to make the variable's scope global, e.g.:

var = 2

print(var) # outputs: 2

def retVar():

global var

var = 5

return var

print(retVar()) # outputs: 5

print(var) # outputs: 5

# Key takeaways

1. A function can call other functions or even itself. When a function calls itself, this situation is known as **recursion**, and the function which calls itself and contains a specified termination condition (i.e., the base case - a condition which doesn't tell the function to make any further calls to that function) is called a **recursive** function.

2. You can use recursive functions in Python to write **clean, elegant code, and divide it into smaller, organized chunks**. On the other hand, you need to be very careful as it might be **easy to make a mistake and create a function which never terminates**. You also need to remember that **recursive calls consume a lot of memory**, and therefore may sometimes be inefficient.

When using recursion, you need to take all its advantages and disadvantages into consideration.

The factorial function is a classic example of how the concept of recursion can be put in practice:

# Recursive implementation of the factorial function def factorial(n): if n == 1: # the base case (termination condition) return 1 else: return n \* factorial(n - 1) print(factorial(4)) # 4 \* 3 \* 2 \* 1 = 24

# Sequence types and mutability

Before we start talking about **tuples** and **dictionaries**, we have to introduce two important concepts: **sequence types** and **mutability**.

A **sequence type is a type of data in Python which is able to store more than one value (or less than one, as a sequence may be empty), and these values can be sequentially (hence the name) browsed**, element by element.

As the for loop is a tool especially designed to iterate through sequences, we can express the definition as: **a sequence is data which can be scanned by the**for**loop**.

You've encountered one Python sequence so far - the list. The list is a classic example of a Python sequence, although there are some other sequences worth mentioning, and we're going to present them to you now.

The second notion - **mutability** - is a property of any of Python's data that describes its readiness to be freely changed during program execution. There are two kinds of Python data: **mutable** and **immutable**.

**Mutable data can be freely updated at any time** - we call such an operation in situ.

*In situ* is a Latin phrase that translates as literally *in position*. For example, the following instruction modifies the data in situ:

list.append(1)

**Immutable data cannot be modified in this way**.

Imagine that a list can only be assigned and read over. You would be able neither to append an element to it, nor remove any element from it. This means that appending an element to the end of the list would require the recreation of the list from scratch.

You would have to build a completely new list, consisting of the all elements of the already existing list, plus the new element.

The data type we want to tell you about now is a **tuple**. **A tuple is an immutable sequence type**. It can behave like a list, but it mustn't be modified in situ.

# Key takeaways: tuples

1. **Tuples** are ordered and unchangeable (immutable) collections of data. They can be thought of as immutable lists. They are written in round brackets:

myTuple = (1, 2, True, "a string", (3, 4), [5, 6], None) print(myTuple) myList = [1, 2, True, "a string", (3, 4), [5, 6], None] print(myList)

Each tuple element may be of a different type (i.e., integers, strings, booleans, etc.). What is more, tuples can contain other tuples or lists (and the other way round).

2. You can create an empty tuple like this:

emptyTuple = () print(type(emptyTuple)) # outputs: <class 'tuple'>

3. A one-element tuple may be created as follows:

oneElemTup1 = ("one", ) # brackets and a comma oneElemTup2 = "one", # no brackets, just a comma

If you remove the comma, you will tell Python to create a variable, not a tuple:

myTup1 = 1, print(type(myTup1)) # outputs: <class 'tuple'> myTup2 = 1 print(type(myTup2)) # outputs: <class 'int'>

4. You can access tuple elements by indexing them:

myTuple = (1, 2.0, "string", [3, 4], (5, ), True) print(myTuple[3]) # outputs: [3, 4]

5. Tuples are immutable, which means you cannot change their elements (you cannot append tuples, or modify, or remove tuple elements). The following snippet will cause an exception:

myTuple = (1, 2.0, "string", [3, 4], (5, ), True) myTuple[2] = "guitar" # a TypeError exception will be raised

However, you can delete a tuple as a whole:

myTuple = 1, 2, 3,

del myTuple

print(myTuple)

# NameError: name 'myTuple' is not defined

# Key takeaways: dictionaries

1. Dictionaries are unordered**\***, changeable (mutable), and indexed collections of data. (**\***In Python 3.6x dictionaries have become ordered by default.

Each dictionary is a set of *key : value* pairs. You can create it by using the following syntax:

myDictionary = { key1 : value1, key2 : value2, key3 : value3, }

2. If you want to access a dictionary item, you can do so by making a reference to its key inside a pair of square brackets (ex. 1) or by using the get() method (ex. 2):

polEngDict = { "kwiat" : "flower", "woda" : "water", "gleba" : "soil" } item1 = polEngDict["gleba"] # ex. 1 print(item1) # outputs: soil item2 = polEngDict.get("woda") print(item2) # outputs: water

3. If you want to change the value associated with a specific key, you can do so by referring to the item's key name in the following way:

polEngDict = { "zamek" : "castle", "woda" : "water", "gleba" : "soil" } polEngDict["zamek"] = "lock" item = polEngDict["zamek"] # outputs: lock

4. To add or remove a key (and the associated value), use the following syntax:

myPhonebook = {} # an empty dictionary myPhonebook["Adam"] = 3456783958 # create/add a key-value pair print(myPhonebook) # outputs: {'Adam': 3456783958} del myPhonebook["Adam"] print(myPhonebook) # outputs: {}

You can also insert an item to a dictionary by using the update() method, and remove the last element by using the popitem() method, e.g.:

polEngDict = {"kwiat" : "flower"} polEngDict = update("gleba" : "soil") print(polEngDict) # outputs: {'kwiat' : 'flower', 'gleba' : 'soil'} polEngDict.popitem() print(polEngDict) # outputs: {'kwiat' : 'flower'}

5. You can use the for loop to loop through a dictionary, e.g.:

polEngDict = { "zamek" : "castle", "woda" : "water", "gleba" : "soil" } for item in polEngDict: print(item) # outputs: zamek # woda # gleba

6. If you want to loop through a dictionary's keys and values, you can use the items() method, e.g.:

polEngDict = { "zamek" : "castle", "woda" : "water", "gleba" : "soil" } for key, value in polEngDict.items(): print("Pol/Eng ->", key, ":", value)

7. To check if a given key exists in a dictionary, you can use the in keyword:

polEngDict = { "zamek" : "castle", "woda" : "water", "gleba" : "soil" } if "zamek" in polEngDict: print("Yes") else: print("No")

8. You can use the del keyword to remove a specific item, or delete a dictionary. To remove all the dictionary's items, you need to use the clear() method:

polEngDict = { "zamek" : "castle", "woda" : "water", "gleba" : "soil" } print(len(polEngDict)) # outputs: 3 del polEngDict["zamek"] # remove an item print(len(polEngDict)) # outputs: 2 polEngDict.clear() # removes all the items print(len(polEngDict)) # outputs: 0 del polEngDict # removes the dictionary

9. To copy a dictionary, use the copy() method:

polEngDict = { "zamek" : "castle", "woda" : "water", "gleba" : "soil" } copyDict = polEngDict.copy()

:#b -> converts to binary

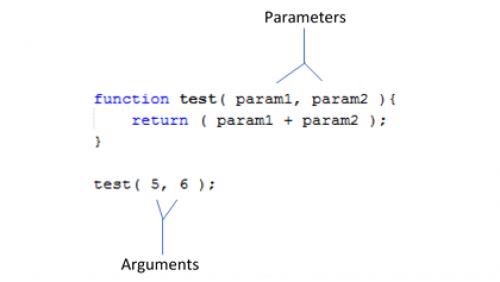
:#o -> converts to octal

:#x -> converts to hexadecimal

:#0 -> converts to decimal as above example

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

<https://getkt.com/blog/types-of-function-arguments-in-python/>



**min**("aAbByYzZ") - minimum element of the sequence passed as an argument

**max**("aAbByYzZ") - maximum element of the sequence passed as an argument

"aAbByYzZaA".**index**("b") - searches the sequence from the beginning, in order to find the first element of the value specified in its argument; its absence will cause a ValueError exception

**list**("abcabc") - takes its argument (a string) and creates a new list containing all the string's characters, one per list element.

"abcabc**".count**("b") - counts all occurrences of the element inside the sequence

'aBcD'.**capitalize(**) - if the first character inside the string is a letter (note: the first character is

element with an index equal to 0, not just the first visible character), it will

be converted to upper- case;all remaining letters from the string will be converted to lower-case.

**center**()- adding some spaces before and after the string

print('[' + 'gamma'.center(20, '\*') + ']')

"epsilon".**endswith**("on") - checks if the given string ends with the specified argument and returns True or False

print(“etata”.**find**('eta')), print('kappa'.**find**('a', 2)), print('kappa'.**find**('a', 1, 4))

**isalnum()** checks if the string contains only digits or alphabetical characters (letters), and returns True or False 'lambda30'.isalnum()

**isalpha()** - letters only

**isdigit()** - digits only

**islower()** - lower-case letters only.

**isspace()** - identifies whitespaces only

**isupper()** - upper-case letters only.

**join()** - ",".join(["omicron", "pi", "rho"])

**lower()** method makes a copy of a source string, replaces all upper-case letters with their lower-case counterparts

**lstrip()** method returns a newly created string formed from the original one by removing all leading whitespaces

The one-parameter lstrip() method does the same as its parameterless version, but removes all characters enlisted in its argument (a string), not just whitespaces:

print("www.cisco.com".lstrip("w."))

**replace()** method returns a copy of the original string in which all occurrences of the first argument have been replaced by the second argument

print("This is it!".replace("is", "are", 1))

print("This is it!".replace("is", "are", 2))

**rfind()** do nearly the same things as their counterparts (the ones devoid of the r prefix), but start their searches from the end of the string, not the beginning (hence the prefix r, for right).

**lstrips**, like rstrip() but affect the opposite side of the string.

**split()** - it splits the string and builds a list of all detected substrings.

**startswith()** method is a mirror reflection of endswith() - it checks if a given string starts with the specified substring

**trip()** - makes a new string lacking all the leading and trailing whitespaces

**swapcase()** method makes a new string by swapping the case of all letters within the source string

**title()** - changes every word's first letter to upper-case, turning all other ones to lower-case

**upper()** - method makes a copy of the source string, replaces all lower-case letters with their upper-case counterparts

**sorted()** - returns a new list

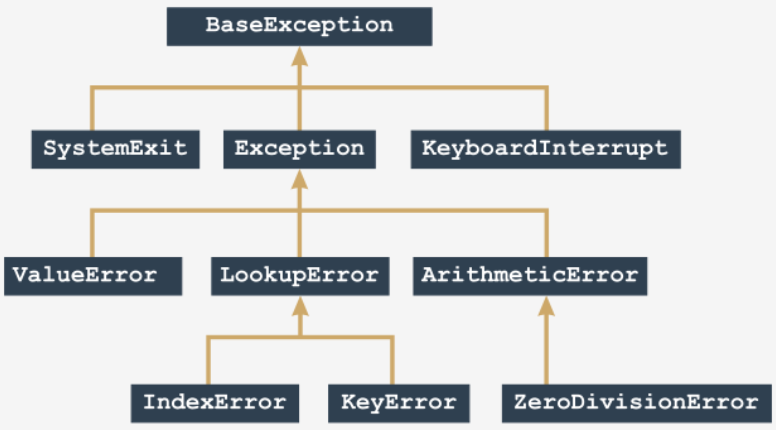
firstGreek = ['omega', 'alpha', 'pi', 'gamma']

firstGreek2 = sorted(firstGreek)

**sort()** - no new list is created :

secondGreek = ['omega', 'alpha', 'pi', 'gamma']

secondGreek.sort()



PART 2

MODULE 6

\_\_init\_\_

\_\_dict\_\_

hasattr - function which is able to safely check if any object/class contains a specified property

\_\_name\_\_ property contains the name of the class, is absent from the object - it exists only inside classes

\_\_module\_\_ is a string, too - it stores the name of the module which contains the definition of the class

\_\_bases\_\_ is a tuple. The tuple contains classes (not class names) which are direct superclasses for the class.

setattr()

getattr()

isinstance()

\_\_str\_\_()

issubclass(ClassOne, ClassTwo)

The function returns True if ClassOne is a subclass of ClassTwo, and False otherwise.

isinstance(objectName, ClassName) - functions returns True if the object is an instance of the class, or False otherwise

**DECORATORS:**

def make\_pretty(func):

def inner():

print("I got decorated")

func()

return inner

**@make\_pretty**

def ordinary():

print("I am ordinary")

# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
def smart\_devide(f):  
 def decor(x,y):  
 print(f"I'll be devide {x} / {y}")  
 if y == 0:  
 print("It's impossible")  
 return  
 return f(x,y)  
 return decor  
  
@smart\_devide  
def divide(a,b):  
 return a/b  
  
# \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
  
def smart\_devide\_args(f):  
 def decor(\*args):  
 x = args[0]  
 y = args[1]  
 z = args[2]  
 print(f"I'll be devide {x} / {y} plus {z}")  
 if y == 0:  
 print("It's impossible")  
 return  
 return f(\*args)  
 return decor  
  
  
@smart\_devide\_args  
def divide\_args(\*args):  
 a = args[0]  
 b = args[1]  
 c = args[2]  
 return a/b+c

**ITERATORS:**

li = [1,4,6,9,10]  
iterat = iter(li)  
print(next(iterat),next(iterat))  
print(iterat.\_\_next\_\_(),iterat.\_\_next\_\_(),iterat.\_\_next\_\_())  
  
print(int())  
  
class my\_iter():  
  
 def \_\_init\_\_(self,table):  
 self.table = table  
 self.lenght = len(table)  
  
 def \_\_iter\_\_(self):  
 self.i = 0  
 return self  
  
 def \_\_next\_\_(self):  
 if self.i < self.lenght:  
 self.i += 1  
 return self.i\*\*2  
 else:  
 raise StopIteration  
  
f\_iter = my\_iter(li)  
aaa = iter(f\_iter)  
for z in li:  
 print(next(aaa))  
  
# print(aaa.\_\_next\_\_())  
# print(next(aaa),next(aaa),next(aaa),next(aaa),next(aaa))

**GENERATORS:**

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

def gen2x\_5(max = 0):  
 i = 0  
 while i < max:  
 yield 2\*i+5  
 i +=1  
  
for x in gen2x\_5(25):  
 print(x)

* 1. Tupla / list / dictionaries – sortowanie jak wygląda

Tuples are immutable

Dictionary: unordered collection of data values

* 1. Przekazywanie argumentów do funkcji – przez \*args pozycyjne (lista) lub przez \*kwargs (słownik)
  2. Typowanie dynamiczne vs ograniczenie typu do wartości :

<https://realpython.com/python-type-checking/>

<https://geek.justjoin.it/jak-okielznac-typy-w-pythonie-czyli-python-3-i-type-annotation/>

* 1. Polymorfizm: <https://www.edureka.co/blog/polymorphism-in-python/>
  2. Class inheritance – limitations of number superclasses
  3. Słowo kluczowe self
  4. Generators
  5. Garbage collector: <https://stackify.com/python-garbage-collection/>
  6. Python interpretowany język vs języki compilowane
  7. Wielowątkowość – dostęp do zasobów Global Interpreter Log: <https://realpython.com/python-gil/>
  8. Decorators - build decorators:

<https://www.datacamp.com/community/tutorials/decorators-python>

* 1. Slice on lists
  2. Range vs xrange: range returns list but xrage returns generator, deprecated in Python 3 : <https://www.geeksforgeeks.org/range-vs-xrange-python/>
  3. Monkey patching:

<https://medium.com/@chipiga86/python-monkey-patching-like-a-boss-87d7ddb8098e>

<https://riptutorial.com/python/example/9909/monkey-patching>

* 1. Lambda function, map, filter: <https://www.programiz.com/python-programming/anonymous-function>
  2. Anagram solutions

Shallow copy vs deep\_copy: <https://realpython.com/copying-python-objects/>

python -c “import platform; print(platform.python\_implementation())”