Python fundamentals

Python is ...

Python is a general purpose programming language. It can be used virtually for any projects – make an application, create a website or a game, or do data analysis. Python is used a lot in an Artificial Intelligence area.

Python is a free portable high level language. First, Python is free to use by anyone. You don't need to pay for it. Second, Python can be used on any operating system. The same code [most probably] will work on any computer – Windows, Mac OSX, Linux. Third, "high level" means that Python is easy to understand by humans. So, it is easy to learn, easy to write programs. There is a downside – high level programming languages tend to have a bit lower performance. However, Python is still good enough for most tasks.

Python is an interpreted language. When you run a Python code, it is converted into a bytecode and executed by the Python virtual machine. Python code goes through *an interpreter*, which converts your code into a machine code. This is different to other programming languages, like C or C++, where you use a compiler to convert your code into a machine executable application.

Learning to write programs in Python requires a lot of practice. First of all, you need to practice in logical thinking, then practice in problem solving, and, finally, practice in coding. Coding is an easy part. However, before you code anything, you need to develop a good understanding of what you want to do.

Special note: object-oriented VS procedural programming. Python is a so-called object-oriented programming (OOP) language, however we are not going to touch this side of Python in our course. We will use procedural programming or top-to-bottom programming only.

Python syntax

Traditionally, the very first program people learn to code in any programming language is following:

```
#Display Hello World message to the screen
print("Hello World")
```

```
## Hello World
```

Here it is – your very first program. It takes an input "Hello World", executes a command `print()` and you get a result – text on the screen. You will do something similar with all your programs in this course and beyond.

Comments

Comments help you to make code more readable. You can provide any extra information that might be useful to better understand the code, provide name of the program, version of Python it requires, author's name, etc.

Comment is indicated by hash `#` and it spans for the entire line. If you need multiline comments, then you should use triple quotation mark, that is, single quotation marks repeated three times `'''`, in the beginning and at the end of comments.

```
# One-line comment

# This is a two-line comment, however,
# in fact there are just two one-line comments

'''
Real
multiline
comments

You can have a lot of text here.
'''
```

Variable names

Any piece of data you have in your program is called a **variable**. Each variable should have a unique name, so you can call for it to extract or change information it stores.

```
# we create a variable with name "x" and
# assign it a value "3"
x = 3

# call for that variable and print its value
print(x)
```

```
## 3
```

Python variables are automatically created when they are initialised. Variables are defined when they are assigned a value by using an assignment statement. When a variable represents a value stored in memory, we say that the variable *references* the value.

You have to create a variable before you can use it. This is a very common mistake students do in their code. If you get an error message: `NameError: name 'z' is not defined`, it means that the variable `z` was not created yet.

Special note: Variable Reference vs Copy. Whenever the variable is used, it takes the value by referring to it in memory, it does not take the value from a copy.

```
# Example
num1 = 3
num2 = num1
```

Variables `num1` and `num2` now refer to the same object, i.e. the value 3. In this case, Python uses referencing to memory. Understanding the references and copying is critical when working with large data sets, otherwise your program can produce unexpected results. You will see examples later.

Variable names can include letters, numbers and underscore. However, they should start from a letter. Small and large case letters are different.

```
# legitimate variable names
x = 1
x2 = 2
x_2 = 3

# case does matter
# below two variables are different objects
test = 3
Test = 5
```

All variable names should be meaningful. It is not compulsory for the program execution, but it makes a code easy to read and understand.

```
# meaningful names for variables
long_name_variable = 1
anotherLongName = 2    # camelCase style
```

A traditional convention regarding variable names advises to use a lower case for variables that you plan to change in the process, and an upper case for variables that you don't plan to change, that is, you assume them being constant.

```
# variable that can take different values
var = 1

# variable that stores an important value
# that you don't plan to change
CONSTANT = 3.14159
```

You can use almost any names for your variables except "protected" words.

```
# protected words in Python
import keyword
print(keyword.kwlist)
```

```
## ['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', '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']
```

All these words are special values or commands or operators or statements. You will learn them later. You cannot use them as a variable name – Python would not allow you that.

Another important consideration while choosing variable names is not to overlap names of Python functions.

```
# this is a Python function to print on screen
print("Hello World")

# you might try to create a variable with name "print"
print = 3

# above is not a mistake, however it is a serious inconvenience
# as you can not use function "print()" anymore
# you reassigned a value instead of a function to the name "print"

print("Hello World") # results in an error

# you need to delete the problem variable to get the function back
del print

print("Hello World") # now it works
```

There are some other rules regarding variable names in Python, e.g. name of the class should start with capital letter, use of single and double underscore in the beginning and at the end of the name. We skip that part as it is not relevant to data analysis and as we don't use OOP.

Writing a code

It is traditional in Python to start each new command on a new line. That insures the clarity of the code. It is possible to have multiple commands on the same line, but it goes against the general priciples of Python.

```
# good style
x = 1
y = 2
x + y
```

```
## 3
```

```
# poor style
x = 1; y = 2; x + y
```

```
## 3
```

Technically both examples are identical and the code works fine. However, the second code might be treated as a poor programming style, especially if there are long and complex commands.

Data types

There are five main data types: Number, String, List, Tuple, Dictionary. You will spend first three weeks of the course to learn and use these data types.

```
my_number = 1
my_string = "string"
my_list = [1,2,"three"]
my_tuple = (1,2,"three")
my_dictionary = {'a':1,'b':"test", 'c':my_list}

# for any of the above variables you can try function "type()"
type(my_string)
```

Data types – Numeric

The most important data type for us is numeric – different formats of numbers. Any type of data analysis is always a number-crunching exercise. *Side note:* even if we work with text data, e.g. text mining or sentiment analysis, we still work with numbers by converting words or letters in to numbers.

```
# int - Integer, whole number with no decimal point
a = 1
b = -100

# float - numbers with decimal point
c = 1.0
d = -99.99

# complex - Complex numbers
e = 3.14j

# bool - Boolean, reference  one of two values:  True or False
x = True
y = False

# try function type() for each variable above
type(x)
```

Different numbers are used for different purposes and required different amount of memory for storage and time for processing. Integers are quick and easy to process. Floats are not so much.

You can do a conversion between different data types. Sometimes conversion might not be possible, or it might result in a partial loss of the information.

```
# Integer to float - no problem at all
a = 1
float(a)
```

```
## 1.0
```

```
# Float to integer - we lost all information after decimal point
b = 3.14
int(b)
```

```
## 3
```

```
# Boolean to integer
c = True
int(c)
```

```
## 1
```

Python operators

There are many different type of operators in Python. You already know some of them from mathematics. Mathematical expressions are calculated in a similar way as on your calculator. (*Side note:* An expression is a combination of values, variables, and operators. Resulting value is typically assigned to a variable.) Below you can see the most popular operators on the examples of working with numerical variables. You will use these operators on a regular basis. We skip "less popular" operators. You can find the full list on the internet by googling "Python operators". There are huge number of resources, e.g. – https://www.w3schools.com/python/python_operators.asp or https://www.programiz.com/python-programming/operators

There are the same operators as

Arithmetic Operators

```
# Arithmetic Operators +, -, *, /, %, **, //

# Addition
1 + 2

# Subtraction
1 - 2

# Multiplication
2 * 3

# Division
6 / 4

# Modulus
6 % 4

# Floor Division
6 // 4

# Exponent
2 ** 10
```

You have seen the "main" assignment operator `=`. There are many others which expand assignment functionality, they do some operation and then assign the value. It is a matter of your personal preferences if you are going to use this "advanced" functionality.

```
# Below pairs of expressions are identical
x += 3
x = x + 3

y -= 3
y = y - 3

# ... and so on, check the link above
```

Comparison Operators

Next very important set of operators are comparison operators: equal and not-equal, greater and less. All comparison operators result in value True or False, which in turn could be treated as 1 and 0.

```
3 == 2      # equal
```

```
## False
```

```
3 != 2      # not equal
```

```
## True
```

```
3 > 2       # greater than
```

```
## True
```

```
3 < 2       # less than
```

```
## False
```

```
3 > 3       # greater than
```

```
## False
```

```
3 >= 3      # greater or equal
```

```
## True
```

Logical Operators

Logical operators work only with boolean values and result boolean values as well. In most cases, logical operators are used with comparison expressions as they result in boolean values. Logical operators are: and, or, not.

```
# "and" results in True only if all elements are True
True and True
```

```
## True
```

```
True and False
```

```
## False
```

```
True and True and True and False
```

```
## False
```

```
# "or" results in False only if all elements are False
True or False
```

```
## True
```

```
True or True
```

```
## True
```

```
False or False
```

```
## False
```

```
True or True or True or False
```

```
## True
```

```
# "not" just reverse the given boolean value
not True
```

```
## False
```

```
not False
```

```
## True
```

Membership Operators

Membership operators `in` and `not in` test if a value is a member of the group. The result of the expression with the membership operator is a boolean value – `True` or `False` – the same as for comparison operators.

```
x = [1, 2, 3]      # a group this three elements, you will learn such "groups" later
1 in x             # test if 1 is in the variable "x" - it is True
```

```
## True
```

```
4 in x             # test if 4 is in the variable "x" - it is False
```

```
## False
```

```
4 not in x         # test if 4 is not in the variable "x" - it is True
```

```
## True
```

Identity Operators

Identity operators `is` and `is not` check if two values are identical, that is, if two values are in the same part of the memory. They look similar to comparison operators equal and not equal, however they are not the same. Two variables that are equal might be not identical. The result of the expression with the identity operators is a boolean value – `True` or `False`.

```
x = 1
y = 1
x is y           # yes, they are identical, and they are equal if you compare them
```

```
## True
```

```
x = [1, 2]
y = [1, 2]
x is y           # no, they are not identical
```

```
## False
```

```
x == y           # however, they are equal
```

```
## True
```

```
x is not y       # this is true as they are not identical
```

```
## True
```

```
x = 99
type(x) is int   # test if type of "x" is integer
```

```
## True
```

Precedence and Associativity

Python follows the standard algebraic rules for the order of operation (operator precedence) – BODMAS: (1) Brackets; (2) Orders (exponentiation `**`); (3) Division (`/`, `//`), Multiplication (`*`), Remainder (`%`) – equal level, left to right; (4) Addition (`+`) and Subtraction (`-`) – equal level, left to right.

Higher precedence is performed first. Operators on the same precedence level are evaluated left to right (associativity).

```
# try to calculate "manually" and then check results in Python
5 * (2 + 2)                    # =
5 * 2 + 2                      # =
4 * 2 / 2 * 4                  # =
4 * 2 / (2 * 4)                # =
```

All comparison operators have lower precedence than arithmetic operators. All logical operators have lower precedence than comparison operators.

```
# try to calculate "manually" and then check results in Python
3 * 2 > 4                     # =
3 * (2 > 4)                   # =
1 and 3 * 2 > 4               # =
0 and 3 * 2 > 4               # =
0 or 3 * 2 > 4                # =
```

If you are in doubts about the order of execution in your formula – use round brackets to set correct precedence explicitly. Parenthesis are always go first.

Builtin Functions

Functions are programs that do something. These programs are already prepared and available for you to use. You can write your own functions.

In general, functions have the following format:

```
output = function_name(input_argument)
```

`input_argument` is an input, there might be more than one input value or there might be no inputs at all. However, there are always parentheses (round brackets) after the name of the function. If there are no input arguments, then parentheses will be empty. But parentheses should always be there.

`output` is a variable to store a result of running function. Some function might not produce any results but do some action, e.g. print on the screen or create a graph or delete a variable from memory – there will be no output value to store in the variable `output`.

There is Python Standard Library – a set of functions that are always available, and then there are huge number of external libraries that can be loaded to bring extra functionality in to Python.

Python Standard Library

Python has a set of functions that are always available. You have seen some of them already.

```
x = int(3.14)     # convert value to integer
print(x)          # print value referenced by the variable on the screen
```

```
## 3
```

```
abs(-99)          # get absolute value
```

```
## 99
```

```
pow(2,10)         # returns 2 in power 10
```

```
## 1024
```

```
2 ** 10           # the same result as above
```

```
## 1024
```

```
range(10)         # create a sequence of ten numbers from 0 to 9
```

```
## range(0, 10)
```

```
range(1,20,2)     # create a sequence from 1 to 20 with step 2
```

```
## range(1, 20, 2)
```

```
del x             # delete variable "x", del is not a function but a statement
                  # you saw it in the list of "protected" words
```

```
help(abs)         # get help on function abs()
help("del")       # get help on statement del
help()            # start interactive session with builtin help. Press "q" to quit.

name = input("Enter your name: ")  # interactive communication with the user
print(name)
```

Here you can find a full list of functions from the standard library – https://docs.python.org/3/library/functions.html

Statement `import`

Besides the standard library with its functions there are more than 10,000 other libraries (or modules, or packages) available for Python. Some of these modules might have just one or two functions. Other modules might have thousands functions and many new data types and new data structures. Typically, these modules have some focus, e.g., module for data analysis, for 3D games, for artificial intelligence.

To load any module into a running session of Python you need to use statement `import`. Typically, you do it on the top of your code.

```
# import one package at a time
import math
import random

# import several packages simultaneously
import math, random
```

Modules `math` and `random` should be available on your computer. If you use Anaconda distributive, then you already have all modules required for this course. If you got different distributive or just want to install something, then you can use commands `pip` (https://packaging.python.org/en/latest/tutorials/installing-packages/) or `conda` (https://docs.conda.io/projects/conda/en/latest/user-guide/concepts/installing-with-conda.html)

After loading the module, you get access to extra functions and data types. To use these functions, you write the name of the module, then dot and then the name of the function.

```
# Some functions from "math"
math.sqrt(9)       # square root
```

```
## 3.0
```

```
math.floor(3.14)  # floor function
```

```
## 3
```

```
math.pi   # constant - value of pi, this is not a function, so no parenthesis
```

```
## 3.141592653589793
```

```
math.sin(math.pi*2)   # sine function for value in radians
```

```
## -2.4492935982947064e-16
```

```
dir(math)  # list of all functions in the module "math"
```

```
## ['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
     'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb', 'copysign', 'cos', 'cosh', 'degrees
    ', 'dist', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fmod', '
    frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan',
     'isqrt', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi
    ', 'pow', 'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', '
    trunc']
```

```
# Some functions from "random"
random.randint(1,10)  # random integer between 1 and 10
```

```
## 1
```

```
random.random()       # random float number between 0 and 1
```

```
## 0.6780565279304465
```

```
dir(random)  # list of all functions in the module "math"
```

```
## ['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom',
    'TWOPI', '_Sequence', '_Set', '__all__', '__builtins__', '__cached__', '__doc__', '
    __file__', '__loader__', '__name__', '__package__', '__spec__', '_accumulate', '_acos',
    '_bisect', '_ceil', '_cos', '_e', '_exp', '_inst', '_log', '_os', '_pi', '_random', '
    _repeat', '_sha512', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn', '
    betavariate', 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss', 'getrandbits
    ', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randint', 'random',
    'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform', '
    vonmisesvariate', 'weibullvariate']
```

When you use function `dir()` to look inside the module (or any other object), ignore all names with one or two underscore in front of them. These functions are not for us as Python users, we don't use them. Remaining *objects* are usable.

Not all of them are functions – some of them are constants, some of them might be a collection of other objects – like a module inside a module. Typically, it happens with very large and complex packages. For example:

```
import numpy              # import a package - you will learn/use it in week 4
dir(numpy)                # check content of the package, it is a big package
dir(numpy.random)         # check content of one "object" inside that package
numpy.random.sample()     # use a function from that "sub-package" inside a package
                          # to get a random float number between 0 and 1
```

Please note a dot-notation and hierarchical structure of the command: package name, then dot, sub-package name, dot again, function name with round brackets at the end.

You will learn more statements as we go through the course. For example, control flow operators in the next section are statements too. At the same time, you might never use some statements. So, there is no need to memorise all statement. If you need, you can always find the full list of simple statements here – https://docs.python.org/3/reference/simple_stmts.html; and compound statements here – https://docs.python.org/3/reference/compound_stmts.html.

Control Flow Operators

There are two main types of control operators in Python: selection and repetition. Control flow operators are identical in all programming languages. There might be different syntax but the meaning is always the same. Selection is used to make decisions. We can change the flow of the program – If something we do one thing, otherwise, we do another thing. Repetition is used to repeat the same operation multiple times – five, ten, thousand, mullion times, as many as you want.

Selection operators

There are three types of selection operators: if, if-else, and if-elif-else.

if

"If" selection has the following structure:

```
if expression:
  <do something>
```

The *expression* should be a boolean value – `True` or `False`. If the value is `True` then the code `<do something>` will be executed. If the value is `False` then the program will "jump over" the code `<do something>` and it will not be executed. It is very common to use a comparison operator in the *expression* to get a boolean value.

```
age = 21
if age >= 18:               # comparison resulting in a boolean value
  print("You can vote!")    # code to be executed if comparison results in value "True"
```

```
## You can vote!
```

Important note: indentation is extremely important. Only code with indentation is considered to be inside a selection structure block, so it will be or will be not executed depending on the result of a comparison. Also, you must use consistent indentation as the start and end of the blocks are indicated by indentation. Your indentation might be one tabulation or four spaces ("traditional" approach in Python), or it can be just one or two spaces – however, indentation should be consistent through your entire code. Python uses indentation to signify its block structure. The block inside a selection operator can have as many lines as you need as far as all of them have the same indentation.

```
import random
x = random.gauss(0,1)  # create random normal value

if x > 0:              # comparison resulting in a boolean value
  x = x + 1            # three lines of calculations to execute
  y = x / 2            # all lines have the same indentation
  z = x + y
print(z)               # end of the block as there is no indentation
                       # this line will be executed regardless the result of comparison
                       # and you might get an error if "z" was not defined beforehand
```

Exercise

Create a code for a simple game: generate a random number between 1 and 10, then ask the user to guess that number. Display on the screen: (1) the user's guess; (2) true random number; if user guessed correctly, then (3) congratulations!

if-else

Operator "if" is one-sided – the block of code will be executed if the *expression* is `True` and ignored if the *expression* is `False`. Operator "if-else" allows us to execute one set of commands if the *expression* is `True` and a different set if the *expression* is `False`. The syntax is following:

```
if expression:              # boolean value
  <do something>            # code to execute if expression is "True"
else:                       # no indentation, so previous block is over
  <do something different> # block of code to execute if expression is "False"
```

Important note: pay attention to indentations and to colons at the end of lines with "if" and "else".

```
mark = 55                   # try to put a different value
if mark >= 50:              # comparison resulting in a boolean value
  print("You passed!")     # code to be executed if comparison above results in value "True"
else:
  print("You failed!")     # an alternative code if comparison above results in value "False"
```

```
## You passed!
```

The *expression* in the selection structure should result in a single boolean value. *Side note:* later you will see how to get multiple boolean values in one comparison, but we cannot use them with the selection operators.

The *expression* in the selection structure can be very large and complex, it can involve multiple comparisons combined by logical operators to result in a single boolean value – `True` or `False`.

```
temperature = 25            # try to put a different value
if temperature > 10 and temperature < 35:       # two comparisons with "and" operator
  print("Normal temperature")
else:
  print("Extreme temperature")
```

```
raining = True; highUV = True   # try to put different values
if raining or highUV:           # two boolean values with "or" operator
  print("Take an umbrella!")
```

Side note: Technically, selection operators will work with other values than boolean. It goes against Python principles of clarity, explicit definitions, and so on. So, please don't use in general. However, it will work and sometimes it can be used. See an example below:

```
x = 1           # any number is treated as True, only 0 (zero) means False
# x = "test"    # any none-empty string is True, only x = "" means False
if x:
  print("That was True")
else:
  print("That was False")
```

Exercise

Improve the code for the above simple game. Initial steps are the same: generate a random number between 1 and 10, then ask the user to guess that number. Display on the screen: (1) the user's guess; (2) true random number; (3) now it is non-optional – congratulations message if user guessed correctly; or commiserations message otherwise.

if-elif-else

If you need a lot of selections based on many comparisons, you can have a nested if-else structure. Beware of indentations!

```
mark = 55                  # try to put a different value
if mark >= 85:
  print("HD")
else:
  if mark >= 75:
    print("D")
  else:
    if mark >= 65:
      print("C")
    else:
      if mark >= 50:
        print("P")
      else:
        print("F")
```

Even after multiple selections only one result will be printed. Try to check that manually following this nested selection structure.

It is perfectly fine to use nested structures, however, it might be difficult to control correct indentations everywhere, and the code might be difficult to read. So, there is "if-elif-else" structure which does the same job as a nested structure above.

```
mark = 55                  # try to put a different value
if mark >= 85:
  print("HD")
elif mark >= 75:
  print("D")
elif mark >= 65:
  print("C")
elif mark >= 50:
  print("P")
else:
  print("F")
```

Two lines from the original code `else: ... if ...:` are combined in one statement, and there is no need to multiple indentations.

Exercise

Improve the code for the above simple game one more time. Initial steps are the same: generate a random number between 1 and 10, then ask the user to guess that number. Display on the screen: (1) the user's guess; (2) true random number; (3) it is non-optional and there are three options: * congratulations message if user guessed correctly; * "Too low!" if the user's guess is below the true number; * "Too high!" if the user's guess is above the true number;

Ternary Expressions for selection operators

There is one more way to use selection operator "if-else" by combining both blocks of `<do something>` in one line. The syntax is following:

```
value = true_code if expression else false_code
```

In this command, *value* is a variable to store result; *expression* is a comparison resulting in a boolean value; *true_code* is a code to run if *expression* is True; and *false_code* is a code to run if *expression* is False. Below is a couple examples.

```
mark = 55     # try to put a different value
grade = "Pass" if mark >= 50 else "Fail"
print(grade)


mark = 55     # try to put a different value
print("Pass") if mark >= 50 else print("Fail")
```

The code to execute in case of True or False conditions of the last example already has print command, so there is no need to store result in any variable *value* as above. The same as with normal "if-else" selection, only one block of code will be evaluated. Obviously, that you cannot have multiple lines of code in these two block.

Use ternary expressions wisely. If the code to execute becomes large, the compactness and readability of your code will suffer. It is perfectly fine if you would not use ternary expressions at all – it depends on your personal programming style.

Repetition operators

Repetitions or loops are used when you need to repeat a set of instructions multiple times.Thee are two types of loops: while-loop and for-loop.

while-loop

While-loop is a good choice when you don't know, in advance, how many times the set of instructions should be repeated.

```
while expression:            # boolean value
  <do something while true>  # code to execute if expression is "True"
else:
  <do something else>
```

As long as *expression* is `True`, the block of commands inside `while` will be executed. When the *expression* becomes `False`, the loop terminates and *else* clause is executed. Part *else* is optional and in most cases it is not used for while-loop.

```
import random
a = 0                  # initial value for "a" as we plan to use it on the next line
while a < 10:          # loop while "a" is less than 10
  print(a)             # print current value of "a"
  a = a + random.random()  # increase "a" by a positive random number
else:                  # when loop terminated do something different
  print("done")        # print finishing sentence
```

Try to run above code multiple times. Each time you will get different outputs and there will be different number of iterations – different number of loops. We are confident that eventually value of `a` will become greater than 10 and while-loop will stop. However, we don't know how many iterations it would require as increments of `a` are random.

Similar to selection structures, indentation in repetition structures is extremely important. Python uses indentation to signify a block of code to execute inside the loop.

Important note: You need to be confident that while-loop will stop eventually. That is, the comparison expression will become `False` on some stage. Otherwise, Python goes in to infinite loop that never stops. Don't try to run the code below but understand what the problem is. While-loop can be dangerous.

```
import random
a = 0                  # initial value for "a" as we plan to use it on the next line
while a >= 0:          # loop while "a" equals zero or more
  print(a)             # print current value of "a"
  a = a + random.random()  # increase "a" by a positive random number
```

for-loop

For-loop is less dangerous and way more popular option than while-loop. For-loop runs a predefined number of iterations and then stops. It has the following form:

```
for counter in sequence:
  <do something>          # code to execute during each iteration
else:
  <do something else>     # code to execute after iteration stops
```

Clause *else* is optional and it is executed only once after finishing all iterations. The *sequence* is an iterable object, it is a collection of some elements. For-loop runs once for each element of the *sequence*. The value of that element corresponding to each loop is called a *counter* (or *iterating variable*). You can use the value of *counter* inside the loop or you can ignore it.

```
x = 0
sequence = range(4)       # make a sequence of four numbers: 0, 1, 2, 3
for i in sequence:        # run four iterations - one for each element of "sequence"
  print(i)                # print current value of counter "i"
  x = x + i               # do summation of counter values
else:
  print("==Total Sum==") # print final statement after finishing all loops
  print(x)                # print total
```

```
## 0
## 1
## 2
## 3
## ==Total Sum==
## 6
```

Later you will learn more different types of iterable objects that can be used inside for-loops. All of them have one common property – they have a length, that is, a count of how many elements are inside that object. As a result, they can be used in a for-loop for the same number of loops as the number of elements.

```
len(range(4))          # length for the sequence of 4 numbers
```

```
## 4
```

```
fruits = ['banana', 'apple', 'mango']    # this is a list
len(fruits)            # length of the list
```

```
## 3
```

```
for i in fruits:       # for-loop for each element of the list
  print(i)
```

```
## banana
## apple
## mango
```

```
for i in range(len(fruits)):   # for-loop for a sequence of indexes of list elements
  print(i)
  print(fruits[i])
```

```
## 0
## banana
## 1
## apple
## 2
## mango
```

Altering Control Flow

There are three statements that allows to alter control flow: `break`, `continue` and `pass`. For example, we might be not interested to run all pre-defined iterations in the for-loop but want to terminate the loop. These statements helps us to do it.

break

Statement `break` does precisely that – it breaks the loop. For example, we use for-loop to guess a secret number:

```
import random
secret = random.randint(1,10)    # get a random integer number between 1 and 10

for i in range(1,100):           # we don't know how large is "secret" number
  print( "we try number", i)     # we try all numbers in a sequence 1 to 99
  if i == secret:                # check if "i" is the same as "secret"
    break                        # no need to continue as we found the right number
```

```
## we try number 1
## we try number 2
```

```
print("solution is", i)
```

```
## solution is 2
```

Statement `break` can be used to get out from while-loop

```
x = 0                # initial value, we will use "x" later, so we need to define it
while True:          # this is a bad idea as expression "True" would never become False
    x += 1
    print(x)
    if x >= 5:       # if value of "x" becomes 5
        break        # statement "break" is activated and break while-loop
    print("===")
```

```
## 1
## ===
## 2
## ===
## 3
## ===
## 4
## ===
## 5
```

Statement `break` will stop on only "its own" loop. If there is a loop inside a loop, then you should double check to what loop does `break` belong. See an example below – there are two while-loops and two statements `break`

```
x = 0
while True:
    while True:
        x += 1
        print(x)
        if x >= 10:
            break
        print("===")
    if x > 20:
        break
```

pass

Statement `pass` is used for development and it should not be included in the final version of your code. Let's assume that you write a code (below) and you don't have a clear idea (yet!) of what to put in True-block of the selection structure. You cannot leave an empty space – Python would not allow that, there should be something. Statement `pass` creates "something" to fill the space which does not do anything at all. `pass` is a placeholder. You will replace it later by a meaningful code.

```
for x in range(10):
    if (x == 3 or x == 4):      # you have not decided yet what to do if True
        pass
    else:                       # but you have a good idea of what to do if False
        print(x)
```

You can test your code and be sure if it works correctly for the False-condition. You will take care about the True-condition later.

continue

Statement `continue` is sort-of opposite to `break`, it forces the loop to proceed to the next iteration. Statement `continue` allows to skip all remaining code in the current iteration and goes directly to the next iteration.

```
for i in range(10):     # run for-loop for a sequence of numbers from 0 to 9
    if i % 2 == 0:      # check if counter is an even number
        continue        # skip the remaining part of the block if comparison is True
    print(i)            # print the value of the counter
```

```
## 1
## 3
## 5
## 7
## 9
```

Above code will get to the command `print()` only for odd numbers that create False in the selection structure.

Exercise

Final improvement for the guessing game. First step is the same: generate a random number between 1 and 10. Then ask the user to guess that number and display on the screen: * congratulations message if the user guessed correctly; * "Too low!" if the user's guess is below the true number; * "Too high!" if the user's guess is above the true number;

If user guessed incorrectly, ask the user to guess the number again – give them a chance to win. Then again and again. Also, count the number of unsuccessful attempts and show that number on the screen at the end of the game.

There might be different conditions to end the game: (1) user guessed correct number; or (2) user reached the limit of attempts (if you included that limit)

Fun fact: there is a winning strategy for the above guessing game. If you follow that strategy, then you can guess correctly the number between 1 and 1000(!!!) in 10 attempts or less.

Data types – String

String is a text data. Variable `x` below is a string.

```
x = "Hello World!"
type(x)
```

```
## <class 'str'>
```

String assignment

String is any piece of information enclosed in quotation marks. Here are some examples:

```
a = 'single quotation'
b = "double quotation"
c = '''triple quotation'''
```

It does not matter what quotation marks to use but you should be consistent. If you open double-quotation mark then you should close double-quotation mark as well.

If you want to include a quotation mark as a part of the string, you should enclose your entire string in a different kind of quotation marks

```
a = "It's a wonderful world!"   # single quotation as a part of the string
b = ''' Tim says: "Yes, you can do this - there is no couldn't shouldn't wouldn't." '''
print(b)                        # single and double quotations as a part of the string
```

```
##  Tim says: "Yes, you can do this - there is no couldn't shouldn't wouldn't."
```

String indexing and slicing

Strings are iterable objects as every string is a collection of individual characters, which can be accessed by **indexing** and **slicing**. That is, we can use an index to get a particular character. Indexes in Python starts from zero, so the first character in a string has index 0.

```
# indexing
mystr = 'Hello World!'   # this is a string
mystr[0]            # get the first character
```

```
## 'H'
```

```
mystr[1]            # get the second character
```

```
## 'e'
```

```
mystr[-1]           # get the last character, negative means "from the end"
```

```
## '!'
```

```
mystr[-2]           # get the second character from the end
```

```
## 'd'
```

Besides a single character we can extract multiple characters by saying where to start and where to finish. The finish of the selection is one step before provided index.

```
# slicing
mystr[0:3]          # start on index 0 and finish on index 2 (one step before 3)
```

```
## 'Hel'
```

```
mystr[3:]           # start on index 3 and go till the very end
```

```
## 'lo World!'
```

```
mystr[:4]           # start at the beginning and go till index 3 (one step before 4)
```

```
## 'Hell'
```

```
mystr[0:7:2]        # start on 0, finish on 6 (one step before 7) and take every second
```

```
## 'HloW'
```

```
mystr[-1:0:-2]      # go in reverse - from the end, finish on 1 and take every second
```

```
## '!lo le'
```

Here is a general approach to slicing:

```
string[start]           # get one character at index "start"
string[start:end]       # slice from "start" to "end-1"
string[start:end:step]  # slice from "start" to "end-1" with step size "step"
```

Operators with strings

String supports some arithmetic operators but in a different way to what you might expect.

```
str1 = "Hello"; str2 = "World!"
str1 + " " + str2       # it's called string concatenation, space is a character too
```

```
## 'Hello World!'
```

```
str1 * 3                # string multiplication
```

```
## 'HelloHelloHello'
```

Also, string supports comparison operators (in a "strange" way too). Strings are compared lexicographically – by ASCII value order. Try to google for "ASCII table".

```
str1 = "abc"; str2 = "abc"; str3 = "xyz"
str1 == str2 # is one string is the same as another string?
```

```
## True
```

```
str1 > str3  # "abc" comes before "xyz", that is, lower ASCII code
```

```
## False
```

```
"a" > "A"    # uppercase codes from 65 to 90, lowercase from 97 to 122, lowercase larger
```

```
## True
```

Important note: Strings are *immutable*. It means that you cannot change a string.

```
mystr = "Hello World!"
mystr[0]                # you can extract a character with index 1
mystr[0] = "G"          # you cannot change the character, the will be an error
```

```
# the only way is to create a new string (and destroy the old one on the way)
mystr = "G" + mystr[1:]   # concatenation of "G" and slice of the original string
print(mystr)
```

```
## Gello World!
```

In the example above, we don't change the string but create a new string and assign its value to the existing variable. The same way all numerical variables are *immutable* as well. It is impossible to change their value but assign a new value to the existing variable.

```
x = 1    # create variable "x" and assign value 1
x = 3    # assign a new value 3 to variable "x", value 1 remains in the memory but
         # we can not have access to it anymore - it is lost.
```

Iterations with strings

As strings are iterable objects, they can be used in loops. There are two ways to iterate a string: (1) by characters; and (2) by index

```
mystr = "Hello World!"          # get a string - a collection of characters

for i in mystr:        # iterate for every character in a string
  print(i)             # taking one element (character) at a time

str_length = len(mystr)       # get a length of the string
for i in range(str_length):   # create a sequence of numbers the same length as a string
  print(i)                    # iterate for every number in that sequence
  print(mystr[i])             # use that number is an index for the string
```

It is important that you use correct index for the string. For example, if the length of the string is 10 characters but you ask for index 100 – there will be an error as there is no element with index 100.

The above method of creating a sequence of numbers for the same length as a string ensures that you would not ask for the wrong index. Very often, for-loop uses the following syntax for string-based iterations:

```
for i in range(len(mystr)):    # two commands together to create a correct sequence
  pass                         # whatever code you need
```

Functions and methods on strings

Some Python built-in functions work with strings:

```
mystr = "Hello World!"
len(mystr)  # length of the string
```

```
## 12
```

```
min(mystr)  # character with lowest ASCII code value
```

```
## ' '
```

```
max(mystr)  # character with highest ASCII code value
```

```
## 'r'
```

I said before that we don't use object-oriented programming in out course, as in general we don't need it for data analysis. However, string is an object (as any variable in Python). As a result, string has a lot of very useful methods. Trick is the same as for checking a package content – use function `dir()`.

```
dir("")   # I use an empty string as an example - methods are the same for all strings
```

```
## ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '
    __format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '
    __hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '
    __mod__', '__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '
    __rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '
    capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
    'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit
    ', 'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper
    ', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'replace', 'rfind', '
    rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith',
     'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']
```

As before, ignore all names with underscore in the beginning – they are for Python itself and not for us as users. Remaining are methods we can use. You can think about them as functions from the package of your string – syntax will be the same. Here are some examples:

```
mystr = "Hello World!"
mystr.lower()       # create the same string but everything in lowercase

## 'hello world!'

mystr.upper()       # create the same string but everything in uppercase

## 'HELLO WORLD!'

mystr.split(" ")    # split the string by string " " (space)

## ['Hello', 'World!']

mystr.find("l")     # find string "l" inside my string - index of first appearance

## 2

mystr.find("l", 4)  # find string "l" inside my string - first appearance after index 4

## 9

mystr.count("l")    # count how many times string "l" appears in my string

## 3

mystr.replace("l", "L")  # replace string "l" by string "L"

## 'HeLLo WorLd!'

# getting help on any string method
help(mystr.count)
help("".count)      # I used empty string as a variable to get help on a method
```

Exercise

Start with any string, for a example:

```
message = "Hello World!"
```

Write a code to output the same string but with all vowels removed. Vowels in English language are *a, e, i, o, u*. Fun fact: American English includes in vowels also *y*; Latin (which is a base for English) also includes *w*. You are free to use any version of the language.

Improve your code to process correctly lower case and upper case letters. Try to create different versions of the same code.

String formatting and placeholders

It is a very common way to output the string on the screen by using a function `print()`. This function converts all variables into strings and then makes its own concatenation.

```
print("these", "are", 2, "strings")

## these are 2 strings
```

The result is one string where all elements combined using a space as a delimiter. You can change it.

```
print("these", "are", 2, "strings", sep = " = ")
```

```
## these = are = 2 = strings
```

String might include *escape* characters – special symbols that are not really a part of the text but instructions for the computer about how to show that text on the screen. The most popular escape characters are *new line* and *tabulation*.

```
print("I can also split the line here\nand then print \\n")          # new line

## I can also split the line here
## and then print \n

print("I'd like some more space\tthank you. I did that using \\t")   # tabulation

## I'd like some more space thank you. I did that using \t
```

The backslash `\` is an escape character, so it cannot be included in a string and printed directly. You should mark it as an escape character by another backslash.

```
print("I can print a backslash like this \\")    # backslash

## I can print a backslash like this \
```

There are several ways how you can include the result of your calculations in to a meaningful and good-looking screen output.

```
result = 3    # we assume this is a result to output

# make a string with the result - number should be converted to string
to_print = "My result is " + str(result) + ". This is a significant achievement!"
print(to_print)

## My result is 3. This is a significant achievement!

# let function print to do conversion and concatenation
print("My result is ", result, ". This is a significant achievement!", sep = "")

## My result is 3. This is a significant achievement!
```

The most flexible and powerful way is to use placeholders – special symbols that will be replaced by the desired values.

```
print("My result is %s. This is a significant achievement!" % result)

## My result is 3. This is a significant achievement!
```

Symbol `%s` is a placeholder that should be replaced by the value after the symbol `%` at the end of the function. There might be multiple placeholders of different type to accommodate multiple outputs.

```
a = 3; b = 4; name = "Tim" # results to include in the output
print("My name is %s and if I add %d to %d I get %d." % (name,a,b,a+b))

## My name is Tim and if I add 3 to 4 I get 7.

print("Alternatively, if I add %f to %e I get %G." % (a,b,a+b))

## Alternatively, if I add 3.000000 to 4.000000e+00 I get 7.
```

Every placeholder should be matched by the value to replace it. So, the should be the same number of values as the number of placeholders. Letters in the placeholder signify formatting of the value. The same number can be presented as an integer or a float or in scientific notation.

String method `format()` has its own version of placeholders that can do the same job as placeholders in `print()`. However, syntax is a bit different. The placeholder is defined by `{}`.

```
print("My name is {} and if I add {} to {} I get {}.".format(name,a,b,a+b))
```

```
## My name is Tim and if I add 3 to 4 I get 7.
```

```
# special formatting: we provide an integer but got a float with two decimal places
my_string = "You have ${:.2f} in your account".format(100)
print(my_string)
```

```
## You have $100.00 in your account
```

Try to google for "formatting and placeholders in python" to see more examples.

Data types – Tuple

Tuple is one-dimensional, immutable, fixed-length object. It means, that the tuple is a collection of some elements, which cannot be changed once it was created. Similar to strings, you can create another tuple from the elements of the original tuple but you cannot change the existing tuple.

Tuple assignment

Tuple is recognised by parentheses (or round brackets) and it is created by writing a set of objects separated by a comma.

```
simple_tuple = 4, 5, 6      # create a tuple
simple_tuple
```

```
## (4, 5, 6)
```

```
another_tuple = (4, 5, 6)   # explicitly use round brackets
another_tuple
```

```
## (4, 5, 6)
```

```
small_tuple = 4,            # tuple with one element only should use a comma anyway
small_tuple
```

```
## (4,)
```

Tuple can combine different data types as its elements, including other tuples

```
my_tuple = "test", (4,5,6), 7
my_tuple
```

```
## ('test', (4, 5, 6), 7)
```

```
len(my_tuple)    # check the length of the tuple
```

```
## 3
```

The tuple above has length 3, as there are only three elements or three objects. They are: string, tuple and integer.

You can convert any iterable object in to a tuple by the function `tuple()`.

```
test_string = "Hello World!"   # string is a collection of characters
tuple(test_string)             # tuple with a collection of the same characters
```

```
## ('H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd', '!')
```

```
test_list = [3, 5, 7, 11, 13]  # list - you will learn it in the next section
tuple(test_list)               # tuple with the same elements as the original list
```

```
## (3, 5, 7, 11, 13)
```

```
tuple(range(10))               # tuple with a sequence of integers from 0 to 9
```

```
## (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Indexing and slicing

We use indexing and slicing to access the elements of the tuple – the same way as it was with strings. Index starts from zero.

```
print(my_tuple)
# indexing
```

```
## ('test', (4, 5, 6), 7)
```

```
my_tuple[0]    # get very first element of the tuple
```

```
## 'test'
```

```
my_tuple[-1]   # get the last element of the tuple
```

```
## 7
```

```
my_tuple[1]    # get the element number 2

# slicing
```

```
## (4, 5, 6)
```

```
my_tuple[1:]   # start with element 2 and go till the end of the tuple
```

```
## ((4, 5, 6), 7)
```

```
my_tuple[1::2] # start with element 2 and go till the end of the tuple with step 2
```

```
## ((4, 5, 6),)
```

When you do indexing – you get an actual element: string or tuple or integer. When you do slicing – you always get a tuple as a result. Even if slicing can extract only one element, the result is a tuple with that element.

A reminder: you can use indexing to extract values but you cannot assign new values to the existing tuple as tuples are immutable.

```
my_tuple[1] = 99   # result is an error
```

If you extract any element from the tuple by indexing and that element is an iterable object, like string or other tuple, then you can apply indexing or slicing to that object too. It looks as following

```
my_tuple = "test", (4,5,6), 7     # this is a tuple with three elements
print(my_tuple)
```

```
## ('test', (4, 5, 6), 7)
```

```
my_tuple[0]      # extract element number 1, which happens to be a string
```

```
## 'test'
```

```
my_tuple[1][2]   # extract element 3 from the that string
```

```
## 6
```

```
my_tuple[1][1:]  # extract elements from 2 till the end from the that string
```

```
## (5, 6)
```

```
my_tuple[1]      # extract element number 2, which is a tuple too
```

```
## (4, 5, 6)
```

```
my_tuple[1][2]   # extract element 3 from that tuple
```

```
## 6
```

Tuple operators and functions

Tuples can use concatenation (operator `+`) and replication (operator `*`)

```
# tuple plus tuple results in a new tuple
(2, 4, 6) + (8, 10) + ("test", )
```

```
## (2, 4, 6, 8, 10, 'test')
```

```
# tuple "multipled" by the integer results in a new tuple integer times longer
(2, 4, 6) * 4
```

```
## (2, 4, 6, 2, 4, 6, 2, 4, 6, 2, 4, 6)
```

You cannot multiply a tuple by a non-integer value as it is a replication procedure and not a real multiplication.

There are not so many functions/methods to work with tuples. Tuple are very primitive objects. On the positive side, tuples are very quick to process.

```
dir((1,))    # check content of the tuple object

# there are only two functions we can use
```

```
## ['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '
    __format__', '__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '
    __hash__', '__init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '
    __mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__rmul__', '
    __setattr__', '__sizeof__', '__str__', '__subclasshook__', 'count', 'index']
```

```
my_tuple = 2, 3, 4, 4, 3, 3, 5, 3
my_tuple.count(3)    # count how many times some value appears in the tuple
```

```
## 4
```

```
my_tuple.index(3)    # find an index of the first appearance of the value
```

```
## 1
```

```
my_tuple.index(3, 2) # find an index of the first appearance of the value starting after 2
```

```
## 4
```

Unpacking tuple (and other iterable objects)

In Python it is possible to assign multiple variable at the same time if you work with iterable objects.

```
my_tuple = (4, 5, 6)      # tuple
a, b, c = my_tuple        # assign values to three variables simultaneously, check them

my_tuple = (4, 5, (8, 9))  # nested tuple
a, b, c = my_tuple         # assign values to three variables
a, b, (c, d) = my_tuple    # four variables to unpack nested tuple
                           # check the result of both commands to see the difference

my_string = "test"        # string as it works for any iterable object
a, b, c, d = my_string    # assign values to four variables
```

These unpacking procedures are very cool however dangerous. You need to be confident that your iterable object has the same number of values to unpack as the number of variables you try to assign to. If there is a mismatch, there will be an error message.

One more useful function from the standard library related to unpacking to multiple variables is `enumerate()`. It takes an iterable object and create a sequence of pairs – each pair contains an index and a corresponding value from the iterable object. It is very convenient to use them inside for-loops.

```
my_tuple = ("abc", "def", "ghi")    # this is a tuple
enumerate(my_tuple)            # it does not look exiting by itself
```

```
## <enumerate object at 0x0000022D32428DC0>
```

```
# unpacking each pair of enumerate object in to two variables
for counter, value in enumerate(my_tuple):
  print(counter, "=", value)  # we get both variables to use at the same time
```

```
## 0 = abc
## 1 = def
## 2 = ghi
```

```
# enumerate works for any iterable object, e.g. for string as a collection of characters
for counter, value in enumerate("test"):
  print(counter, "=", value)
```

```
## 0 = t
## 1 = e
## 2 = s
## 3 = t
```

Data type – List

List is the most popular data type for storing data in Python fundamentals. It is iterable, one-dimensional, mutable(!), variable-length(!) object. It means, that the list is a collection of some elements, which can be changed. The list itself can be changed as well. It is possible to add new elements, or delete elements, or change the order of elements in the list.

List assignment

List is defined by squared brackets. Elements of the list can be any objects including tuples and other lists.

```
x = [1, 2, 3]
print(x)
```

```
## [1, 2, 3]
```

```
y = ["abcd", 2, 3.14, ("tuple", 99), x]
print(y)
```

```
## ['abcd', 2, 3.14, ('tuple', 99), [1, 2, 3]]
```

```
len(y)
```

```
## 5
```

There is a function `list()` which works very similar to function `tuple()` – it takes an iterable object and convert it in to a list.

```
list("test")     # from string
```

```
## ['t', 'e', 's', 't']
```

```
list((4, 5, 6))  # from tuple
```

```
## [4, 5, 6]
```

Concatenation and replication work the same as for tuples

```
[1, 2, 3] + ["test", 99]
```

```
## [1, 2, 3, 'test', 99]
```

```
[1, 2, 3] * 3
```

```
## [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Indexing and slicing

List indexing and slicing work (again) the same as for tuples and strings. For indexing – you get an actual value for the given index. For slicing – you get a list even if it includes only one value.

```
my_list = [1, 'abc', 2, 'def', 3, 'ghi']
my_list[0]     # get first element of the list
```

```
## 1
```

```
my_list[0:1]   # get a list with the first element of the original list
```

```
## [1]
```

```
my_list[1:3]   # elements starting from 2 and finishing one step before 4 - that is 3
```

```
## ['abc', 2]
```

```
my_list[2:]    # elements starting from 3 and finishing at the end
```

```
## [2, 'def', 3, 'ghi']
```

```
my_list[1::2]  # every second elements starting from 2 and finishing at the end
```

```
## ['abc', 'def', 'ghi']
```

```
my_list[::-1]  # get all elements in reversed order: by step -1
```

```
## ['ghi', 3, 'def', 2, 'abc', 1]
```

```
my_list[::100] # slicing with step 100 - result is a list with a single value
```

```
## [1]
```

Lists are mutable! You can change list elements.

```
# indexing - you extract a value, you can replace it by another value
my_list = [1, 'abc', 2, 'def', 3, 'ghi']
my_list[0] = 99        # assign an integer
print(my_list)
```

```
## [99, 'abc', 2, 'def', 3, 'ghi']
```

```
# slicing - you extract a list, you can replace it by iterable object of any length
my_list = [1, 'abc', 2, 'def', 3, 'ghi']
print(my_list)
```

```
## [1, 'abc', 2, 'def', 3, 'ghi']
```

```
my_list[0:2] = [9999]         # assign a list with a single value
print(my_list)
```

```
## [9999, 2, 'def', 3, 'ghi']
```

```
my_list[0:1] = [77, 88, 99]   # assign a list with multiple values
print(my_list)
```

```
## [77, 88, 99, 2, 'def', 3, 'ghi']
```

Functions and methods for lists

Lists are way more complex objects than tuples, so they have more functionality. Again, we start with checking the content of the list object.

```
dir([])   # [] is an empty list, it has length zero
```

```
## ['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '
    __doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__',
     '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__', '__iter__', '
    __le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__', '__reduce__', '
    __reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__', '__setitem__', '
    __sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy', 'count', 'extend
    ', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

Some of the available functions/methods work the same as for other data types, for example `count()` will count how many times some element appears in the list. So, it returns an integer value.

```
x = [1,2,3,2,3,3,4,3,4,5,4,3,6,7]
result = x.count(3)   # count value 3 in a list
print(result)
```

```
## 5
```

```
result = x.index(3)   # find index for value 3 the same as function find() before
print(result)
```

```
## 2
```

Other list methods will do some modification to the original list and don't return anything. There is no point in trying to assign the result of running this methods to a new variable as there will be nothing to assign.

```
x = [1, 'abc', 2, 'def', 3, 'ghi']
x.append(99)        # add new element at the end of the list
print(x)            # list has changed
```

```
## [1, 'abc', 2, 'def', 3, 'ghi', 99]

x.append(["a", "b"])  # add a list at the end of the original list
print(x)              # length of the list increased by one

## [1, 'abc', 2, 'def', 3, 'ghi', 99, ['a', 'b']]

x.extend(["a", "b"])  # open up the list and add all elements one-by-one
print(x)              # length of the list increased by two

## [1, 'abc', 2, 'def', 3, 'ghi', 99, ['a', 'b'], 'a', 'b']

x.insert(1, 8888)  # add value 8888 at index 1, that moves all elements one step back
print(x)           # list has changed again while there was no any returns from command

## [1, 8888, 'abc', 2, 'def', 3, 'ghi', 99, ['a', 'b'], 'a', 'b']

x.remove(8888)     # remove value 8888 from the list
print(x)

## [1, 'abc', 2, 'def', 3, 'ghi', 99, ['a', 'b'], 'a', 'b']

x.reverse()        # reverse the order of elements in the list
print(x)

## ['b', 'a', ['a', 'b'], 99, 'ghi', 3, 'def', 2, 'abc', 1]

x.clear()          # clean up the list - remove all values
print(x)

## []
```

All methods above modify the original list but don't return anything. There is one method that is very unique in its behaviour. It modifies the original list and returns a value at the same time. It is `pop()` – it returns an element for a given index and removes this element from the list.

```
x = [1, 'abc', 2, 'def', 3, 'ghi']
n = x.pop(1)    # get value of element with index 1
print(n)        # here is a value

## abc

print(x)        # list has changed - above value is removed

## [1, 2, 'def', 3, 'ghi']
```

Obviously, you could do it without `pop()` but you will need two commands

```
x = [1, 'abc', 2, 'def', 3, 'ghi']
n = x[1]        # get value for the given index
del x[1]        # delete value for that index
print(n)        # here is a value

## abc

print(x)        # list was changed

## [1, 2, 'def', 3, 'ghi']
```

Above is an example of removing an element by its index. Method `remove()` deletes an element by its value – the first occurrence of this value. That might be a problem if there are several elements with the same value and you want to remove not the first one.

Some functions you have seen before will work for lists as well.

```
x = [1,3,6,2,4,5]   # list of numerical values only
max(x)    # maximal value
```

```
## 6
```

```
min(x)    # minimal value
```

```
## 1
```

```
len(x)    # length of the list
```

```
## 6
```

```
sum(x)    # summation of all values in the list
```

```
## 21
```

If you get a mix of different data types in a list, for example, numbers and strings, then `max()` and `min()` would not work. Summation `sum()` works if the list has numerical values only.

If you have two lists that are related to each other, then you can combine them pair wise.

```
seq1 = ['foo','bar','baz']    # two lists are assumed to be related to each other
seq2 = ['one','two','three']
zip(seq1, seq2)   # combine them pairwise as a list of tuples
```

```
## <zip object at 0x0000022D32445AC0>
```

```
# it is common to use zip() with enumerate and unpacking to multiple variables
for i,(a,b) in enumerate(zip(seq1, seq2)):
  print(i, "=", a, ",", b)
```

```
## 0 = foo , one
## 1 = bar , two
## 2 = baz , three
```

```
# however, you can get the same result without fancy functions
for i in range(len(seq1)):   # both lists are assumed to have the same length
  a = seq1[i]; b = seq2[i]
  print(i, "=", a, ",", b)
```

```
## 0 = foo , one
## 1 = bar , two
## 2 = baz , three
```

The opposite operation of unzipping might be useful if you already have a list of pair wise values and you want to split them apart.

```
# list of tuples representing first and last name of a person each
actors = [('Leo','DiCaprio'), ('George','Clooney'),('Daniel','Craig')]
first_names, last_names = zip(*actors)  # unzip and unpack
print(first_names)
```

```
## ('Leo', 'George', 'Daniel')
```

```
print(last_names)
```

```
## ('DiCaprio', 'Clooney', 'Craig')
```

Data types – Dictionary

Dictionary is an iterable object. It is mutable and variable size. One important difference between dictionary and list (or tuple) is that dictionary does not support indexing. Dictionary is a hash table type object used as an associative array.

List has its elements in a particular order. We can change the order, we can extract a first element by using index. Elements in a dictionary don't have any order, so we cannot use index to extract their values. We have to use elements names or **keys**. Elements of the dictionary, they are also called **items**, have format **key:value**.

Dictionary assignment

Dictionary is defined by curled brackets and a pair of a key and a corresponding value. To extract any value, we use a key in squared brackets – as in index with list.

```
my_dict = {"name":"John", "code":1234, "dept":"sales"}
print(my_dict)
```

```
## {'name': 'John', 'code': 1234, 'dept': 'sales'}
```

```
my_dict["name"]      # get a value for a key "name"
```

```
## 'John'
```

```
my_dict["code"]      # get a value for a key "code"
```

```
## 1234
```

```
my_dict.get("name")  # an alternative way to get a value for a key by using a method
```

```
## 'John'
```

If you try to ask for any index, e.g. `my_dict[1]`, you will get an error message.

Dictionary is mutable, so you can change it by using a **key** instead of index.

```
my_dict = {"name":"John", "code":1234, "dept":"sales"}
print(my_dict)
```

```
## {'name': 'John', 'code': 1234, 'dept': 'sales'}
```

```
my_dict["name"] = "Tim"   # set a value for a key "name"
print(my_dict)            # check the result
```

```
## {'name': 'Tim', 'code': 1234, 'dept': 'sales'}
```

```
my_dict["gender"] = "M"   # setting a value for a NEW key creates a new element
print(my_dict)
```

```
## {'name': 'Tim', 'code': 1234, 'dept': 'sales', 'gender': 'M'}
```

```
my_dict[1] = 999   # the same as before, you create a new element with key 1
print(my_dict)
```

```
## {'name': 'Tim', 'code': 1234, 'dept': 'sales', 'gender': 'M', 1: 999}
```

Number 1 in the last example is not an index but a key. Now you can run command `my_dict[1]`. You will get an element with **key** equal 1 but not the index 1 as there are no order, no indexes in the dictionary.

Dictionary elements might be any data types: strings or numbers as above, or more complex structures, like list or tuple or other dictionary.

Functions and methods for dictionary

As usual, we start with function `dir()` to see the list of all available methods (and ignore all names with underscore).

```
dir({})   # use an empty dictionary {}
```

```
## ['__class__', '__contains__', '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__
    ', '__format__', '__ge__', '__getattribute__', '__getitem__', '__gt__', '__hash__', '
    __init__', '__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__ne__', '
    __new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__setattr__', '
    __setitem__', '__sizeof__', '__str__', '__subclasshook__', 'clear', 'copy', 'fromkeys',
    'get', 'items', 'keys', 'pop', 'popitem', 'setdefault', 'update', 'values']
```

```
my_dict = {"name":"John", "code":1234, "dept":"sales"}    # create a dictionary
my_dict.keys()   # get a list of keys from the dictionary
```

```
## dict_keys(['name', 'code', 'dept'])
```

```
my_dict.values() # get a list of values from the dictionary
```

```
## dict_values(['John', 1234, 'sales'])
```

Two functions above result in a special data types: `dict_keys` and `dict_values`, which can be converted to list or tuple if you want.

```
tuple(my_dict.keys())    # to tuple
```

```
## ('name', 'code', 'dept')
```

```
list(my_dict.values())   # to list
```

```
## ['John', 1234, 'sales']
```

Method `items()` results another special data type: `dict_items`, which looks like a list of pair-wise tuples. It can be converted in to a nested tuple or a list of tuples.

```
my_dict.items()          # get a list of items from the dictionary
```

```
## dict_items([('name', 'John'), ('code', 1234), ('dept', 'sales')])
```

```
tuple(my_dict.items())   # convert it to tuple
```

```
## (('name', 'John'), ('code', 1234), ('dept', 'sales'))
```

```
list(my_dict.items())    # convert it to list
```

```
## [('name', 'John'), ('code', 1234), ('dept', 'sales')]
```

It is very common for many functions to return their results as some special data type you have never seen before. Most of these special data types behave similar as data types from Python fundamentals – tuple, list, dictionary – and can be converted in to them if required.

```
my_dict = {"name":"John", "code":1234, "dept":"sales"}
my_dict.pop("name")  # return a value and delete an item (key:value)
```

```
## 'John'
```

```
print(my_dict)
```

```
## {'code': 1234, 'dept': 'sales'}
```

```
my_dict.popitem()    # return an item as a tuple of key and value and delete that item
```

```
## ('dept', 'sales')
```

```
print(my_dict)
```

```
## {'code': 1234}
```

Method `popitem()` used to return a random item from the dictionary. In the latest versions of Python, this method returns an item that was entered last in the dictionary.

Some functions from the standard library work with dictionary too.

```
my_dict = {"name":"John", "code":1234, "dept":"sales"}
len(my_dict)    # get the length of the dictionary, that is the number of items
```

```
## 3
```

Exercise

Start with a list of words, like below and organise them in groups by their first letter.

```
words = ['apple', 'bat', 'bar', 'atom', 'book']
```

Dictionary would be a very good structure for such data, e.g. "a" as a key and a list of all words starting with "a" as a value. The result should be something as following.

```
## {'a': ['apple', 'atom'], 'b': ['bat', 'bar', 'book']}
```

Data types – Set

One more data to mention here is a set. Set is an ordered collection of unique elements. You can think about the set as keys of a dictionary – you cannot have tow elements in a dictionary with the same key, so every key is unique. Data type set supports mathematical set functions, like union or intersection.

Set is defined by function `set()` or by curled brackets.

```
# when we create a set, all repeated values disappear automatically
set([1,2,2,2,3,3,4,4,4,4])
```

```
## {1, 2, 3, 4}
```

```
{1,2,2,2,3,3,4,4,4,4}
```

```
## {1, 2, 3, 4}
```

Sets support *Bitwise operators*, which we did not mention before but you could see them by the same links: https://www.w3schools.com/python/python_operators.asp or https://www.programiz.com/python-programming/operators. Also, set has its own methods – use function `dir()`. Some methods duplicate *Bitwise operators* functionality.

```
a = {1, 2, 3, 4, 5}
b = {3, 4, 5, 6, 7, 8}

a | b               # union (or)
```

```
## {1, 2, 3, 4, 5, 6, 7, 8}
```

```
a.union(b)          # union as before
```

```
## {1, 2, 3, 4, 5, 6, 7, 8}
```

```
a & b               # intersection (and)

## {3, 4, 5}

a.intersection(b)  # intersection again

## {3, 4, 5}

a - b               # difference

## {1, 2}

a.difference(b)    # difference again

## {1, 2}
```

Some method works similar to methods for lists that modify the original object but not return anything.

```
a = {1, 2, 3, 4, 5}
a.add(99)           # add extra value to the set
print(a)

## {1, 2, 3, 4, 5, 99}

a.remove(99)        # remove a value from the set
print(a)

## {1, 2, 3, 4, 5}
```

User-defined Functions

Introduction and benefits

Some problems can be very complex. In such cases, it may be easier to separate the problem into smaller, simpler tasks. This process is called *decomposition*.

Alternatively, sometimes a calculation is very popular (e.g. calculating the mean) and is repeated often and in different parts of a program. We don't want to rewrite the code each time it is needed!

As a rule of thumb, you should remember that if you copy-paste the same code in different places (without any changes or with minor changes only), this is an indication that you do something wrong. This is a poor programming practice.

In all these situations, we use functions to help us out. Functions are the primary and most important method of code organisation and re-use in Python.

Function is a self-contained segment of code which implements a specific, well-defined task. Python programs typically combine user-defined functions with library functions available in the standard library and functions from external libraries loaded by the statement `import`.

A function is invoked by a call which specifies: (1) A function name; (2) Parenthesis or round brackets after the name; (3) Possibly parameters/arguments inside parenthesis. A function may also return a value.

Functions `input()` and `print()` are examples of standard library functions and are invoked in the following way:

```
print("Welcome to functions!")
num = input("Please enter number: ")
```

There are a number of advantages of using functions in your code: * **Program development is more manageable** The divide-and-conquer approach makes program development more manageable. Construct program from smaller pieces/modules. * **Simpler, streamlined code** Typically simpler and easier to understand when code is broken down into functions. * **Software reusability** Use existing functions as building blocks to create new programs. * **Avoid repeating code** Reduce the duplication of code within a program. * **Better testing** Testing and debugging becomes simpler when each task within a program is contained in its own function. Test each function in a program individually to determine whether it correctly performs its operation. Easier to isolate and fix errors.

Function definition

You already know how to use builtin functions. Now we focus on *custom* functions or *user-defined* functions.

Functions are declared using a statement `def`. It is common to use a statement `return` to exit the function and return some value resulted from running the function. Absolute majority of custom functions would have `return` or even multiple `return` statements. However, this is not compulsory. There might be functions without `return`.

```
# general format of the function
def functionName(parameters):
  <function-commands>
  return expression
```

The keyword `def` introduces a function definition. The `functionName` is a name of the function and it follows the same rules as variable names.

The `parameters` are idetifier names that will be used as variable names *inside the function* for the objects passed to the function as inputs. Parameters are optional. If there is more than one parameter, then parameters are written as a sequence of comma-separated identifiers.

`<function-commands>` is a block of Python code. Similar to blocks of code inside selection or repetition structures, function commands should be indented. `<function-commands>` contains definitions and statements – just normal Python code.

`expression` is a variable you want to pass out of the function. Typically, it is a result of `<function-commands>` execution over provided `parameters`.

When the function code reaches a `return` statement, the function stops and `expression` value is passed out of the function. If there is no `expression` after `return`, then the function returns `None`. `None` is a special value than means *nothing*. It is not a number or a string or an iterable object – it is just *nothing*. If there is no `return` statement in the function, then the function returns `None` as well. So, every function returns a value – even if it is a special value `None`, which means *nothing*.

```
# Example 1: write a function which will take two numbers as parameters,
# sum them, and return the result.

def addNumbers(num1, num2):
  total = num1 + num2
  return total

# test the function
addNumbers(4, 5)
```

```
## 9
```

```
# Example 2: write a function which will take two numbers as parameters, compare them,
# if first number is larger than return the product of these numbers;
# if second number is larger than return the summation of these numbers.

def compareNumbers(num1, num2):
  if num1 > num2:
```

```
    return num1 * num2
  else:
    return num1 + num2


# test the function
compareNumbers(4, 5)
```

```
## 9
```

```
compareNumbers(5, 4)
```

```
## 20
```

Function return

By definition across most (all???) programming languages including Python, a function is designed to return a single value or object. At the same time, there are no limitations on the nature of that object – it can be simple as an integer or it can be really complex, like a list or dictionary or anything else you will learn later.

It is impossible to return two (or more) values out of the function but it is possible to return an iterable object, which is a collection of multiple values.

```
# create a function that assign two values
def fun(x):
  a = x * 10
  b = x / 10
  return a, b

# test the function
fun(15)   # result is not two numbers but one tuple; comma creates a tuple
```

```
## (150, 1.5)
```

```
# an alternative function with explicit definition of the return object
def fun(x):
  a = x * 10
  b = x / 10
  return {"a":a, "b":b}

# test the function
fun(15)   # result is a dictionary - a single object with multiple values
```

```
## {'a': 150, 'b': 1.5}
```

A function might return any number of values as far as they are packed in one object. You get an object from the function and unpack it if necessary.

Namespace and scope

Namespace is the name we use to describe a variable scope in Python. Functions can access variables in two different scopes: **global** and **local**.

Variables assigned within a function are assigned to the **local** namespace. Local namespace: created when the function is called and immediately populated by the function's arguments (that is, parameters' values. After the function is finished, the local namespace is destroyed. Once the function has finished execution, we no longer have access to the variables inside of the function.

Variables that have **global** scope can be accessed anywhere within the Python code -– useful, but constitute bad programming style!

When you run the function it looks for required variables in the local namespace. If it cannot find them, then it looks in the global namespace.

```
# Example 1:

x = 3                   # this is a variable in global namespace

def testFunction(z):    # local namespace with variable "z"
  res = z ** x          # this is a mix of variables from global and local namespace
  return res            # it works but it is a poor practice

testFunction(2)         # test the function
```

```
## 8
```

```
# Example 2:

x = 3                   # this is a variable in global namespace

def testFunction(z):    # local namespace with variable "z"
  x = 2                 # this is a variable in local namespace, it is unrelated to
  res = z ** x          # the variable "x" from global namespace
  return res

testFunction(2)         # test the function
```

```
## 4
```

If you try to call for variables `z` or `res` after running those functions, you get an error message. All variables assigned (or created) inside the function are destroyed after the function finishes its execution.

If you check the variable `x`, its value is `3`. Variable `x` was created in global namespace, and our function cannot change it as the function works with local namespace only.

It is possible to change global space variables from inside a custom function. Sometimes this functionality might be useful.

```
# Example 3:

x = 3                   # this is a variable in global namespace

def testFunction(z):    # local namespace with variable "z"
  global x              # explicit reference to "x" in global namespace
  x = 2                 # we change the value of "x" in global namespace
  res = z ** x
  return res

testFunction(2)         # test the function
```

```
## 4
```

```
print(x)                # check the value of "x"
```

```
## 2
```

Now the value of `x` is `2`. Whatever you do with `x` in global namespace, its value would become `2` each time you run the function, as the function changes it.

This is an advanced functionality and there are not so many situations when you need it. It is a better practice to work only with local namespace inside the function. If you want you can google and read more about statements `global` and `nonlocal`, e.g. https://www.programiz.com/python-programming/global-local-nonlocal-variables or https://python-course.eu/python-tutorial/global-local-variables-namespaces.php

Using global variables makes debugging difficult – many locations in the code could be causing a wrong variable value. Functions that use global variables are usually dependent on those variables. It makes harder to transfer functions to another program. Using global variables makes a program hard to understand.

Follow the good programming style: *Whatever was created inside the function, should die inside the function.* Don't mix-up global and local namespaces.

Exercise

Try to work out the output from the below code without entering it in Python. You can check yourself later by running the code.

```
j = 1
k = 2

def function1():
  j = 3
  k = 4
  print('j is:', j, '; k is', k)

def function2():
  j = 6
  function1()
  print('j is:', j, '; k is', k)

k = 7
function1()
print('j is:', j, '; k is', k)

j = 8
function2()
print('j is:', j, '; k is', k)
```

Lambda function

Python supports *anonymous* or *lambda* functions. These are simple functions consisting of a single statement, the result of which is the return value. They are defined using the `lambda` keyword. This signals to Python that we are declaring an anonymous function.

```
# using "normal" user-defined function
def by_2(x):
  return x * 2

# test the function
by_2(3)
```

```
## 6
```

```
# the same job by using lambda function
by_2 = lambda x: x * 2

# test the function
by_2(3)
```

```
## 6
```

Both functions above are the same, they do the same job. If the function would be more complex, if it had more lines of code – it would be impossible to use `lambda` function.

Very often, we create and use `lambda` functions even without a name. So, it can be used only once in the moment of definition. Then it will be destroyed and would not occupy any memory.

```
# function to apply a given function to every element of the list
# some_list: a list of some values
# fun: function that can be used over elements of the list
def apply_to_list(some_list, fun):
```

```
  return [fun(x) for x in some_list]

# test the function
import math
numbers = [2, 4, 0, 9, 5]
apply_to_list(numbers, math.sqrt)
```

```
## [1.4142135623730951, 2.0, 0.0, 3.0, 2.23606797749979]
```

We applied a function of a square root and got a list of square roots for each value in the original list. If we want to apply some "fancy" function not available in Python, we can create a user-defined function.

```
# create user-defined function
def fancy_function(x):
  return x ** 2 / 3

apply_to_list(numbers, fancy_function)
```

```
## [1.3333333333333333, 5.333333333333333, 0.0, 27.0, 8.333333333333334]
```

It works fine as a user-defined function is the same as any other function in Python. However, our `fancy_function()` is not so fancy at all. It has only one statement and we are not really going to use this function again in our code. So, it is a good case for using a `lambda` function.

```
# using lambda function
apply_to_list(numbers, lambda x: x ** 2 / 3)
```

```
## [1.3333333333333333, 5.333333333333333, 0.0, 27.0, 8.333333333333334]
```

Obviously, result is the same. Code takes less space and less memory. This use of `lambda` functions if very popular for large data manipulations, e.g. data cleaning. You will see more examples later.

Function arguments and parameters

Very often terms *arguments* and *parameters* are used interchangeably and mean the same thing – information that are passed into a function. However, there are some differences. Here is an example of the function:

```
def addNumbers(num1, num2):
  total = num1 + num2
  return total

# test the function
addNumbers(4, 5)

x = 4; y = 5
addNumbers(x, y)
```

Function *parameters* are variables listed in the round brackets in the function definition. They have no values (yet!) as you just define the function and have not used it. In the example above, `num1` and `num2` are parameters. They are variables defined in the local namespace of the function and they will be destroyed after function finishes its work.

Function *arguments* are values or variables that are passed into the function. *Parameters* will assume these values when we run the function. In the example above, numbers `4` and `5`, and then variables `x` and `y` are function *arguments*.

As you have seen, a function may have multiple *parameters*; or, a function may accept multiple *arguments*.

Parameters are a list of items separated by a comma. Arguments are passed by position to the corresponding parameters. For this reason arguments must be passed in the exact order in which they are defined as they are positional arguments.

```
def power(num1, num2):
  return num1 ** num2

power(2, 3)
```

```
## 8
```

```
power(3, 2)
```

```
## 9
```

Above function has two parameters `num1` and `num2` and it returns a value of `num1` taken in power `num2`. Values 2 and 3 are arguments. If we change the order of arguments, then they are assigned to different parameters and the result is different as well. This is *positional arguments*.

At the same time, we can use *keyword arguments*. In this case, we must provide the name of the parameter as a keyword to have our arguments match up explicitly. *Keyword arguments* allow out-of-order parameters.

```
power(num1 = 2, num2 = 3)
```

```
## 8
```

```
power(num2 = 3, num1 = 2)
```

```
## 8
```

Results are the same as in both cases arguments are matched to the same parameters.

It is possible to combine positional and keyword arguments, however you cannot mix them at random – there should be an order: positional arguments first, keyword arguments after that.

```
def example_function(a, b, c, d):
  print("result: a =", a, ", b =", b, ", c =", c, ", d =", d)

example_function(1, 2, 3, 4) # positional arguments, order is important
```

```
## result: a = 1 , b = 2 , c = 3 , d = 4
```

```
example_function(b = 2, a = 1, d = 4, c = 3) # keyword arguments, order is not important
```

```
## result: a = 1 , b = 2 , c = 3 , d = 4
```

```
# a mix of positional and keyword arguments, order is important for positional part
# but not important for keyword part. Also, you cannot use keywords for parameters
# "a" and "b" as they are already defined by position
example_function(1, 2, d = 4, c = 3)
```

```
## result: a = 1 , b = 2 , c = 3 , d = 4
```

Functions may have *default arguments*, that is, arguments/values declared for some parameters during the functions definition. In this case, it is not compulsory to provide an argument for such parameter. If you skip that argument, then the function will use a default value.

Parameters with default arguments must come after all other parameters[!!!].

```
# function with two parameters "c" and "d" having default arguments
def example_function(a, b, c = 88, d = 99):
  print("result: a =", a, ", b =", b, ", c =", c, ", d =", d)

example_function(1, 2, 3, 4) # we provide all arguments, so default values are ignored
```

```
## result: a = 1 , b = 2 , c = 3 , d = 4
```

```
example_function(1, 2, 3) # we skip one argument, so missing argument uses default value
```

```
## result: a = 1 , b = 2 , c = 3 , d = 99
```

```
example_function(1, 2) # we provide only compulsory arguments and skip optional ones
```

```
## result: a = 1 , b = 2 , c = 88 , d = 99
```

```
example_function(1, 2, d = 4) # use keyword arguments, so order is not important
```

```
## result: a = 1 , b = 2 , c = 88 , d = 4
```

Reminder: Parameters are function local variables, they are defined in the local namespace. So, any changes you might be doing to parameters inside the function do not affect the arguments you pass into the function.

There is an exception from the above rule if arguments are mutable and non-primitive values. See a section "Extra comments on mutability" below.

Extra comments on mutable/immutable variables

Some objects in Python are immutable, that is, they cannot be change. For example: numbers, strings, tuples. You cannot change immutable but you can create a new object with the same name.

```
x = 2         # create a variable referencing integer value 2
x = x + 1     # create a new variable with the same name, referencing value 3
              # value 2 remains in memory but we cannot access it, it is lost
```

Other objects are mutable, that is, we can change them. The most interesting example is a list. In fact, list is just a container – a collection of references on different places in computer memory where stored all values included in the list. Check the following examples

```
a = [1,2,3]
b = a
a
```

```
## [1, 2, 3]
```

```
b
```

```
## [1, 2, 3]
```

Lists `a` and `b` are the same. Now we change list `a`.

```
a[0] = "a"
a
```

```
## ['a', 2, 3]
```

```
b
```

```
## ['a', 2, 3]
```

List `b` has changed as well. It happens because lists don't store actual values but only references on values in computer memory. List `b` got references on the same objects/values as list `a`. Then we change a referenced value, and it changes at the same time in both lists.

This features allows to minimise usage of computer memory as we don't store the same information twice. Also, it allows to change function arguments without any tricks with global variables

```
def fun(x):
  x[0] = 99    # the function has no return, so it returns "None" - nothing

data = ["a", "b", "c"]   # variable to use as an argument
fun(data)                # run the function - no return, as expected
print(data)              # variable has changed
```

```
## [99, 'b', 'c']
```

Mutability and values referencing are powerful programming tools. However, sometimes this connection between two (or more) objects is not desirable. We want to change one list but leave the other one in the original state. For that we need not a reference but a copy.

```
a = [1,2,3]
b = a.copy()   # we do not an assignment but a copy of the list

# check the result - now each list references a different set of values
a
```

```
## [1, 2, 3]
```

```
b
```

```
## [1, 2, 3]
```

```
a[0] = "a"     # try to change
a   # this one is different
```

```
## ['a', 2, 3]
```

```
b   # this one is the same
```

```
## [1, 2, 3]
```

Now we try a more complex example

```
a = [1,2,3]
b = ['a', 'b', 'c']
a.append(b)

# check results
a
```

```
## [1, 2, 3, ['a', 'b', 'c']]
```

```
b
```

```
## ['a', 'b', 'c']
```

We try to change list `b`

```
del b[0]    # delete first element from "b"
b           # both lists have changed as you already should expect
```

```
## ['b', 'c']
```

```
a
```

```
## [1, 2, 3, ['b', 'c']]
```

Next, we create a copy of list `a` and then change the original list `a`

```
c = a.copy()
c
```

```
## [1, 2, 3, ['b', 'c']]
```

```
a[0] = 99
a    # list is different
```

```
## [99, 2, 3, ['b', 'c']]
```

```
c    # list is the same as before, as we got a copy instead of a reference
```

```
## [1, 2, 3, ['b', 'c']]
```

Now we change list `b`, which was also included in `a` and then in `c`

```
b[0] = 'z'
b
```

```
## ['z', 'c']
```

```
a
```

```
## [99, 2, 3, ['z', 'c']]
```

```
c
```

```
## [1, 2, 3, ['z', 'c']]
```

All lists have changed despite list `c` being a copy of list `a`. The reason is in method `copy()` it make a copy only for the top level values in the list, while elements from the list `b` are inside a nested structure – list inside a list – and they remained as references rather than copies.

What method `copy()` does is called a *shallow copy*. To make a full copy of the list with all references replaced by the copies regardless the nested structure, you need a *deep copy*. There is a special package for that.

```
import copy              # load a package
c = copy.deepcopy(a)     # make a deep copy of list "a"
b[0] = 'x'               # change list "b" as we tried before
b  # list has changed
```

```
## ['x', 'c']
```

```
a  # list has changed too as expected - "b" is included in "a"
```

```
## [99, 2, 3, ['x', 'c']]
```

```
c  # list has not change as it has no connection at all to "a" and "b"
```

```
## [99, 2, 3, ['z', 'c']]
```