Title: Python Fundamentals

Author: Kornel Chromiński, Ľubomír Benko, Zenón José Hernández-Figueroa, José Daniel González-Domínguez, Juan Carlos Rodríguez-del-Pino, Jan Přichystal

Citation style: Chromiński Kornel, Benko Ľubomír, Hernández-Figueroa Zenón José, González-Domínguez José Daniel, Rodríguez-del-Pino Juan Carlos, Přichystal Jan. (2021). Python Fundamentals. Nitra : Constantine the Philosopher University in Nitra.

FITPED

Python fundamentals

Kornel Chromiński
Ľubomír Benko
Zenón José Hernández-Figueroa
José Daniel González-Domínguez
Juan Carlos Rodríguez-del-Pino
Jan Přichystal

www.fitped.eu

2021

Co-funded by the
Erasmus+ Programme
of the European Union

Work-Based Learning in Future
IT Professionals Education
(Grant. no. 2018-1-SK01-KA203-046382)

Python Fundamentals

Published on

November 2021

Authors

Kornel Chromiński | University of Silesia in Katowice, Poland

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

Zenón José Hernández-Figueroa | University of Las Palmas de Gran Canaria, Spain

José Daniel González-Domínguez | University of Las Palmas de Gran Canaria, Spain

Juan Carlos Rodríguez-del-Pino | University of Las Palmas de Gran Canaria, Spain

Jan Přichystal | Mendel University in Brno, Czech Republic

Reviewers

Jozef Kapusta | Pedagogical University of Cracow, Poland

Peter Švec | Teacher.sk, Slovakia

Eugenia Smyrnova-Trybulska | University of Silesia in Katowice, Poland

Piet Kommers | Helix5, Netherland

Graphics

Ľubomír Benko | Constantine the Philosopher University in Nitra, Slovakia

David Sabol | Constantine the Philosopher University in Nitra, Slovakia

Erasmus+ FITPED

Work-Based Learning in Future IT Professionals Education

Project 2018-1-SK01-KA203-046382

Co-funded by the
Erasmus+ Programme
of the European Union

ISBN 978-80-558-1783-5

Table of Contents

Python as Programming Language

Chapter 1

1.1 Python as a programming language

🕮 1.1.1

Python is a scripted, interpreted programming language. In Python, we write a ***script*** that is interpreted by ***an interpreter***.

Python was created in the 90s, the name of the language comes from the series "*Monty Python's Flying Circus*". Python is a ***free*** high-level programming language with open source code and an extensive standard library package. Its main idea is ***the clarity*** and ***readability*** of the code. Python supports various programming paradigms, such as object-oriented, imperative, and functional. It has a ***dynamic*** file system and automatic memory management.

Python is an easy-to-use language due to the use of ***keywords*** similar to words in English, and the programs written in it are shorter than their equivalent created in *C++* or *Java*. Programs written in Python are ***platform-independent***, work equally well on portable devices, home computers (regardless of what operating system we have) or supercomputers.

The current version of Python can be downloaded from the language page (http://python.org)

🕮 1.1.2

Python comes with an integrated ***IDLE*** programming environment. The development environment is a set of tools that make writing programs easier.

In addition to the built-in ***IDLE*** environment, there is also a large number of independent programming environments for writing programs in *Python*.

Examples include:

- *PyCharm,*
- *Spyder,*
- *PyScripter,*
- *Anaconda,*
- *Pydev.*

The choice of programming environment depends on the programmer's preferences.

📝 1.1.3

Python is a language:

- interpreted
- compiled

📖 1.1.4

To start writing in *Python*, we need to run one of the development environments, it can be built-in (*Python IDLE*).

The window that appears gives us access to *PythonShell*, we can ***directly enter and execute*** *Python* code in it (of course, you can also create your own scripts separately and run them as a whole).

We start to type commands at the prompt:

```
>>>
```

For example, enter the following at the prompt:

```
print("Hello in Python World")
```

And press enter. This is how you wrote the first *Python* program to display a greeting. The ***print*** command is used to display on the screen.

Note in Python, the size of letters is important, if we change the letter p to P, the program will not recognize the command, the result will be an error (can try to make changes).

📝 1.1.5

In *PythonShell*, we start to write after:

📝 1.1.6

In *Python*, the case of the letters in commands is not important.

- True
- False

1.2 Python as a programming language - programs

⌨ 1.2.1 First Program

Run a script with output

```
Hello world
```

⌨ 1.2.2 Hello

Complete the code so that the message "Hello + Your name" will be displayed

⌨ 1.2.3 Subtracting numbers

Complete the program so that the result of its operation returns the difference of the numbers given on the input.

E.g.:

```
Input : 9 5
Output: 4
```

Types

Chapter 2

2.1 Int, float, str values, type()

📖 2.1.1

Python is a language with dynamic type control, which means that there is no need to define data types in *Python*.

Python will determine what type of data it is dealing with based on what values the user or programmer will enter.

- 4 - will be interpreted as an *integer*
- 4.0 - will be interpreted as a *float*ing point number
- "4" - will be interpreted as a *string*

📖 2.1.2

The most commonly used data types are:

Numeric types:

- float - floating point numbers (equivalent to double from C or Java)
- integer - integers
- long integer - integers limited (to the range of numbers) to system resources

Sequential types:

- string - represented by single or double quotation marks (' or ") e.g. 'Python'
- list - a sequence of data, not necessarily of one type (equivalent to arrays, except that the data does not have to be of one type) e.g. [1, 2.3, "Python"]
- tuple - works like a letter, except that the data saved cannot be modified, e.g. (1, 2.3, "Python")
- dictionary - a list of elements indexed using keys (key-value pairs) e.g. {"first": "1", "second": "2"}

Logical Type

- bool - takes one of two values True or False

📝 2.1.3

Floating point numbers in Python are stored as a type:

- float

- double
- integer
- long

📝 2.1.4

Indicate the correct creation of dictionary entries:

- {"key1": 3, "key2": 4, 'key3': 5}
- ["key1": 3, "key2": 4, 'key3': 5]
- {"key1" - 3; "key2" - 4; 'key3' - 5}
- [{key1, key2,key3}:{1,2,3}]

📝 2.1.5

Complete the entry so that the data presented is a list:

```
_____ 2.35, 5.54, 8.98_____
```

📖 2.1.6

As part of ***character strings***, we also have special symbols preceded by **"\"**, they are used to insert special characters such as tabs or newlines. The most popular are:

- **** - displays one backslash
- **\'** - single quotation mark
- **\"** - double quotation marks
- ***\a*** - call the system bell (alarm)
- ***\n*** - newline character, moves the cursor to begin of newline
- ***\t*** - tab

For example:

```
>>> print("We start \nlearning \t\'Python\'")
We start
learning     'Python'
```

📝 2.1.7

To insert a tab into a string, you should type:

📝 2.1.8

Insert the appropriate symbols into the Python code so that you get the following text:

```
      I am a
'Python'
      programmer
print("_____ I am a_____ _____ Python
_____ _____ _____ programmer")
```

🕮 2.1.9

The **type()** function is one of the built-in Python functions, we can use it to return information about the type of value. For example, invoking a command:

```
>>>type(4)
```

Will return to us:

```
<class 'int'>
```

So the entered value is treated as an integer.

And the command

```
>>>type("Python")
```

Will return to us:

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

So the entered value is treated as a string

📝 2.1.10

What will return the command:

```
type(2.31)
```

- <|class 'float'>
- <|class 'double'>

- <|class 'integer'>
- <|class 'number'>

📝 2.1.11

Insert the appropriate operator so that the result returned by the code below is correct:

```
>>> 5 _____ 5
3125
```

2.2 Int, float, str - operations

🕮 2.2.1

Python allows you to perform arithmetic operations on both numeric and text values.

The following operations are available for numerical data:

- ***Adding two values (the '+' symbol)***

```
>>>3 + 4
7
>>>3.0 + 4.5
7.5
```

- ***Subtraction of values (symbol ' -')***

```
>>>4-3
1
>>>4.0-3.5
0.5
```

- ***Multiplying values (the '*' symbol)***

```
>>> 4*3
12
>>> 4.0 * 3.0
12.0
```

- ***Division of floating values (the symbol '/')***

```
>>> 4/2
2
>>> 2/4
0.5
```

- ***Division of integer values***

```
>>>4//2
2
>>> 2//4
0
```

- ***Modulo operation - the rest of the division (symbol '%')***

```
>>> 4%3
1
```

- ***Exponentiation (power, symbol '**')***

```
>>> 2**4
16
```

2.2.2

To get the remainder of dividing two numbers, use the operator

- %
- **
- /
- &

2.2.3

Enter a value that is the result of the Python code presented below

```
5 // 2
```

2.2.4

Insert the appropriate operator so that the result returned by the code below is correct:

```
>>> 5 _____ 5
```

```
3125
```

📖 2.2.5

Python as a calculator - an example:

```
>>> (((2*3)+6)/4)+8-6*(3%2)
5.0
```

📝 2.2.6

In Python, the result of dividing two integers is also an integer.

- False
- True

📖 2.2.7

Python lets you perform operations on a string type

You can use the operator ***'+'*** (string + string) to ***connect strings***

```
>>> "Learn" + "Python"
'Learn Python'
```

To ***multiply strings*** we will use the operator ***'*'*** (string * how many times)

```
>>> "Python" * 3
'PythonPythonPython'
```

📝 2.2.8

What will be the result of the code execution:

```
'a'*3+'b'
```

- 'aaab'
- Error
- 'a3b'
- 'abababab'

2.3 Types - programs

⌨ 2.3.1 Data type

Complete the code so that it displays the data type stored in variable *a*.

⌨ 2.3.2 Subtracting numbers

Complete the program so that the result of its operation returns the difference of the numbers given on the input.

For example:

```
Input : 9 5
Output: 4
```

⌨ 2.3.3 Raising to a power

Complete the program so that the power of the numbers given at the input is returned as a result of its operation.

E.g.

```
Input : 2 2
Output: 4
```

⌨ 2.3.4 Division

Complete the program so that the total value from division is returned as the result of its operation, and the rest from the division of the numbers given at the input.

E.g.

```
Input : 5 2
Output: 2 1
```

⌨ 2.3.5 Average speed

Write a program that calculates the average speed of the car based on the length of the route and travel time.

E.g.

```
Input : 300 3.0
Output: 100
```

⌨ 2.3.6 Average fuel consumption

Write a program that calculates the average fuel consumption per 100 km based on the given route length and total fuel consumption.

E.g.

```
Input : 100 5.0
Output: 5.0
```

⌨ 2.3.7 Triangle area

Write the code that calculates the area of the triangle based on the height and base length provided and writes it to the variable c

Variables

Chapter 3

3.1 Variables

📖 3.1.1

Variables are simply a separate space in the computer's memory where data can be stored.

Variables have their unique name set by the programmer.

In Python, you don't need to specify the type of data you want to store in a variable. **Creating** a variable is based on assigning a value to it.

📝 3.1.2

In Python, it is necessary to provide the type of data that will be stored in the variable at the stage of its declaration

- False
- True

📖 3.1.3

To **assign** a value to a variable use the operator '='.

On the left side of the operator is the element to which we assign the value, on the right the value.

The assigned value can be either a **direct value** (e.g. number), or the value of another variable, or the **result** of a mathematical or logical operation.

```
#creating a new variable
var_1 = 4
var_2 = 'Python'
var_3 = 3 + 4
var_5 = var_1 + var_3
```

📝 3.1.4

Which of the following is the correct version of the variable declaration in Python.

- var = 4.5
- var(4.5)

- var <|> 4.5
- var ~ 4.5

📝 3.1.5

Is the following variable declaration correct:

```
var_1 = 5
var_3 = 3
var = var_1 ** var_3
```

- Yes
- No

🕮 3.1.6

Variable names can be anything, but there are a few rules for the naming of variables:

- **the first character must be the letter of the alphabet** (lowercase or uppercase) or the underscore character '_'. Letters from alphabets of different languages may also be used (this is not recommended)

```
#it can be
number
_number
Number
# not allowed
9Number
```

- **the rest of the variable name** may consist of letters, underscore '_', and numbers

```
Number_1
number_Second
```

- variable names are **case sensitive**, the variables '*myvariable*' and '*Myvariable*' are two different variables

```
Number ≠  number
```

- the variable **cannot** exist without a value

```
>>>number = 4
```

- variable names **cannot** be the same as Python keywords

```
>>> True = 3
SyntaxError: can't assign to keyword
```

📝 3.1.7

Which of the following variable names are correct:

- _4num
- num
- num4
- n_u_m
- #num
- 5num

📝 3.1.8

In Python, the case of variable names is not important. "*NUMBER*" and *"number"* are the same variables.

- False
- True

🕮 3.1.9

Using the **type()** function, you can check what type of value is currently stored in the variable:

```
>>> number = 3
>>> type(number)
<class 'int'>
>>> string = 'Python'
>>> type(string)
<class 'str'>
```

📝 3.1.10

Calling the command *type(2)* returns:

- <|class 'int'>
- <|class 'num'>
- Error
- <|class 'float'>

🕮 3.1.11

📝 3.1.12

Which of the following variable names are correct:

- _4num
- num
- num4
- n_u_m
- #num
- 5num

3.2 Variables - programs

⌨ 3.2.1 Variables

Declare variable a and assign the value 3.14 to it.

⌨ 3.2.2 Variable error

Correct the code so that the program starts.

Input and Output

Chapter 4

4.1 The print() command

📖 4.1.1

The **print()** function is used to display the value given in bracket

```
>>>print(5)
5
>>>var = 'Python'
>>>print(var)
Python
```

We separate the next values to display with a comma

```
>>>string = 'apple'
>>>print('I have ', 3, string)
I have  3  apple
```

📝 4.1.2

Write the command displaying the value of variable *var*

```
>>>var = 5
>>>_____
```

📝 4.1.3

In the *print()* command, to display the values of several variables, separate them with the character:

- ,
- ;
- +
- &

📝 4.1.4

What will be the result of the following command:

```
print(3*'a')
```

- 'aaa'

- '3a'
- TypeError: unsupported operand
- 'a'

📖 4.1.5

In the *print()* function we can also put *mathematical operations*

```
>>>a=15
>>>b=10
>>>print(a+b)
25

>>>print((a*b)-a)
135
```

📝 4.1.6

Complete the code so that the displayed result will be correct:

```
>>> a = 6
>>> b = 7
>>> print((b _____ a)*_____)
2
```

📖 4.1.7

The *print* function can take additional parameters

*print(*values, sep=' ', end='\n')*

- ***sep*** - to set the separator between entered values to display (default separator is space),

```
>>>print('Python','is', 'the', 'best', sep=' ')
Python is the best

>>>print('Python', 'is', 'the', 'best', sep=',')
Python,is,the,best
```

- ***end*** - to set the end of display character (default - newline character)

```
print('Python', end=' ')
print('is the best', end = ' ')

Python is the best
```

📝 4.1.8

For the code below to display the string in the following way, what need we enter the in the print command:

```
>>> print('a','b','c = 10')
'a*b*c = 10'
```

- sep = '*'
- end = '*'
- sep = *
- end = *

📝 4.1.9

Complete the following code so that the result of its operation will be correct:

```
print('value of a ', _____)
print(33)
value of a =33
```

4.2 The input() command

🕮 4.2.1

The ***input()*** function is used to enter data by the user

```
>>>print('Enter the number:')
>>>a = input()
>>>print(a)
```

Inside the brackets of the ***input()*** function, we can put commands for the user

```
>>>a = input('Enter the number:')
```

The value read in using the ***input()*** function is treated as a string

```
>>>a = input('Enter the number:')
>>>type(a)
<class 'str'>
```

4.2.2

Indicate the correct version of *input()*

- a = input()
- input(a)
- input('value of ', a)

4.2.3

What type of value is read using the *input()* command

- string
- depends on the value we give
- type is set as the command parameter

4.3 Input and output - programs

4.3.1 Srea of a circle

Write the code that calculates an area of a circle for a radius entered by the user (take the value Pi as 3.14).

E.g. The area of a circle is to be displayed on the console.

```
Input : 2
Output: 12.56
```

4.3.2 User's age

Write the code that will calculate the user's age based on the year of birth and the current year given by the user

E.g.

```
Input : 1999 2020
Output: 21
```

4.3.3 Print separator

Write the code that displays 3 words entered by the user on one line separated by a comma.

```
Input : red green blue
Output: red,green,blue
```

Comments

Chapter 5

5.1 Comments

📖 5.1.1

Comments are part of the code that the Python interpreter omits. The comments can include a piece of information on what the code does, they are a kind of help for the programmer.

There are two types of comments in *Python*.

single-line comments are preceded by a *#* and continue until the end of the line, for example

```
#this is a comment
print ('text') # displaying a message
```

📝 5.1.2

Insert the appropriate symbol so that the command does not complete

```
_____print('value of variable a is ', 10)
```

📖 5.1.3

There are also comments in Python that can contain several lines, so-called **block comments**.

Block comments are preceded and ended with """

```
"""Is a block comment
        May contain several lines """
```

📝 5.1.4

To insert a multi-line comment use:

- """
- '''
- ###
- %

5.2 Comments - programs

⌨ 5.2.1 First Comment

Insert any code comment.

⌨ 5.2.2 Comments

Comment on the appropriate line of code so that the value of variable *b* remains unchanged.

Operators and Functions

Chapter 6

6.1 Type casting

📖 6.1.1

When using Python, it is sometimes necessary to convert values from one type to another. An example of this could be using the **input()** function if we want the user to give us a number. **Input()** takes the value as a string (*string*), to get the number you need to convert from string to number:

If we entered the code like this:

```
>>>a = input('Give a number')
>>>3 + a
```

We will receive information about the incorrect use of the + operator, we cannot add the string to the number:

```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

To make the above code work, we have to **change the type** of the entered value from the string to the number.

📝 6.1.2

What will return the following code:

```
>>> var = input('Enter a number')
>>> 10 + var
```

- TypeError: unsupported operand type(s) for +: 'int' and 'str'
- '10+entered value'
- 10+entered value

📖 6.1.3

In order to convert types (changes from one type to another), you can use the function:

- *int(a)* - an integer will be returned
- *float(a)* - a floating point number will be returned

```
>>> a = input ('Enter number')
>>> 3 + int(a) # will work correct
```

You can also convert the number to a string:

```
>>>str(5)
'5'
```

6.1.4

Enter the command to change the given string value to an integer:

```
a = '10'
b = _____a_____ + 3
```

6.1.5

What will be the result of the following code:

```
>>> str(5)*3
```

6.2 Assignment operators

6.2.1

Operators extended assignments are used to shortening the write operation in a situation where we perform an operation on a variable and assign a new value to the same variable, that is, instead of using the record:

```
>>> var = 2
>>> var = var * 5 # multiply the variable var by 5 times and
assign the new value to the variable var
>>> print(var)
10
```

We can use the shorthand:

```
>>>var = 2
>>>var *= 5
>>> print(var)
10
```

6.2.2

Which command will be equivalent to the command:

```
>>> var = var * 8
```

- var *= 8
- var ** 8
- var =* 8
- var * 8

📖 6.2.3

Examples of extended assignment operators:

```
 *=, eg. x*=5, equivalent x = x * 5
 x/, eg. x/=5, equivalent x = x / 5
 %=, eg. x%=5, equivalent x = x % 5
 +=, eg. x+=5, equivalent x = x + 5
 -=, eg. x-=5, equivalent x = x - 5
```

📝 6.2.4

Insert the appropriate symbols:

```
a _____ 3 #a = a / 3
b _____ 2 #b = b % 2
```

6.3 Standard functions

📖 6.3.1

Python has a number of built-in functions that we can use without the need to import additional modules.

Examples of functions can be included

- ***abs()*** - returns the absolute value of a number

```
>>> abs(-3)
3
```

- ***round()*** - rounds a float value to an integer value

```
>>> round(-1.5)
-2
```

```
>>> round(1.45)
1
```

- ***chr()*** - returns a string consisting of one character of ASCII code, given as a function parameter

```
>>> chr(97)
'a'
```

- ***len()*** **-** returns the length (number of elements) of the variable

```
>>> len('python')
6
```

- ***max()*** - returns the maximum value of the given values

```
>>> max(2, 5, 6, 8, 1, 3)
8
```

- ***min()*** - returns the minimum value from the given values

```
>>> min(2, 5, 6, 8, 1, 3)
1
```

- ***pow(x, y)*** returns *x* to the power of *y*

```
>>> pow(3, 2)
9
```

6.3.2

To round 1.89 to the total value use the command:

- round(1.89)
- abs(1.89)
- rd(1.89)
- int(1.89)

6.3.3

Insert the command to get the result:

```
a = -8
print(_____(a))
```

```
8
```

6.4 Multiple assignment

📖 6.4.1

In Python, you can **assign** values to several variables **at once**. Instead of assigning values to subsequent variables in each line, we can use a **simplified notation**:

```
>>>Var1, var2, var3 = 5, 2.9, 'Python'
>>>print(var1)
5
>>>print(var2)
2.9
>>>print(var3)
'Python'
>>>var4, var5 = 'ab'
>>>print(var4)
'a'
>>>print(var5)
'b'
```

We write the names of the variables to which we want to assign values (separate by comma), and then after the = sign, we give the values in the order corresponding to the order of the variables.

📝 6.4.2

Enter the value that will be assigned to the variable *b*

```
a, b, c = 2, 'd', 8
```

📝 6.4.3

The result of the following code will be:

```
a, b = '12'
print((a*2)+b)
```

- 112
- 1122
- 121212

- 12122

6.5 Operators and functions - programs

⌨ 6.5.1 Assignment operators

Write a program that doubles the variable value given by the input.

```
Input : 5
Output: 10
```

⌨ 6.5.2 Word length

Write the code that displays on the console the length of the word entered by the user in the format "Word x has y characters", where *x* is the word entered and *y* is its length.

```
Input : Python
Output: Word Python has 6 characters
```

⌨ 6.5.3 Max value

Write the code that displays the maximum value for 3 values given by the user.

```
Input : 2 8 4
Output: 8
```

Formatted Output

Chapter 7

7.1 Formatted output

📖 7.1.1

In *Python*, we also have the option of formatting the displayed text.

The older type of formatting is based on special characters preceded by the symbol **%**

```
>>>name = 'John'
>>>print('Welcome, %s!' %name)
Welcome John
>>>age = 23
>>>print('%s is %d year old.' %(name, age))
'John is 23 year old.'
```

📝 7.1.2

Complete the entry so that the code entered is correct

```
var = 'Sam'
print('Hello %s' _____(var))
```

📖 7.1.3

After the **%** sign, we put information on which type of data we want to display or in which formations, examples of specifiers include:

- ***%s*** - string
- ***%d*** - an integer
- ***%f*** - floating point number
- ***%.<x>*** - floating point number with a fixed number of digits after the dot (e.g.%. <2>, it will display only two digits after the dot)
- ***%10s*** - it means that the text will be displayed occupying a width of 10, aligned to the right and completed with blank characters to the left if it is shorter in length; if it is longer, it will be shown as is
- ***%.5s*** shorten the string to 5 characters

E.g.:

```
>>>price = 2.32456
>>>product_name = 'apples'
>>>print('1 kg of %s cost %.2'%(product_name, price))
'1 kg of apples cost 2.32'
```

📝 7.1.4

Is this statement is correct:

```
price  = 9.99
print('Price of product is %s ' %price)
```

- No
- Yes

📝 7.1.5

Complete the following command to match the displayed result

```
var = 'String'
print('We shortened the String to _____' %var)
We shortened String to Str
```

🕮 7.1.6

A newer way to text formatting is the function ***format()***

```
>>>name = John
>>>print('Welcome, {}'.format(name))
'Welcome John'
>>> age = 23
>>>print('{} is {} year old.'.format(name, age))
'John is 23 year old.'
```

The newer format also allows you to set the order in which the values in brackets are displayed

```
>>>print('{1} is {0} year old.'.format(age, name))
'John is 23 year old.'
```

Moved the displayed text by 10 spaces

```
>>>'{:>10}'.format('text')
         Text
```

Filling the offset with a different character (other than space)

```
>>>'{:_>10}'.format('text')
__________text
```

Shortening the string

```
>>>'{:.3}'.format('Python')
Pyt
```

7.1.7

The correct statement will be:

- print('Welcome, {}'.format(name))
- print('Welcome, %'.format(name))
- print('Welcome, {name}'.format())
- print('Welcome, {}'.format{name})

7.1.8

Complete the code so that it works correctly

```
>>> age = 20
>>> name = 'John'
>>> print('{_____} is _____ years old.'.format(age, name))
'John is 20 years old.'
```

7.2 Formatted Output - programs

7.2.1 Hello

For the name and age entered by the user, display the following text using the formatted Output:

```
Input : John 23
Output: Hello John. You are 23 years old.
```

7.2.2 Rectangle area

Write the code that calculates the area of the rectangle, based on the side lengths entered into the program. The result of the action is to be displayed in the following format:

```
Input : 2 5
Output: The area of a rectangle with sides 2 and 5 is 10.
```

Logic Expression

Chapter 8

8.1 Bool, logic expressions

📖 8.1.1

There is a ***bool*** data type in Python that has two values *True* and *False*

A **bool** value is a result of comparing values.

In *Python*, to compare two values with each other, use the operator '**==**'

```
>>> 2==2
True
>>> 'Python' == 'Python'
True
>>> 'Python' == 'Java'
False
```

To check if two values are different from each other, the operator '**!=**' Is used

```
>>> 2 != 3
True
>>> 2 != 2
False
```

Operators "**>**" and "**<**" will be used to check if something is bigger or smaller.

```
>>> 2>3
True
>>> 2<3
False
```

📝 8.1.2

The operator _____ is used to check if two values are different

📝 8.1.3

The result of the comparison

```
2 == 2
```

will be:

📖 8.1.4

You can also check if something is greater than or equal to, or less than or equal to. The operators "**>=**" and "**<=**" are used for this.

```
>>> 2>=2
True
>>>3 <= 2
False
```

📝 8.1.5

Which entry is correct:

- 2 >= 4
- 2 => 4
- 4 => 2
- 4 <|> 2

📖 8.1.6

It is also possible to check several conditions at the same time by connecting them with the appropriate operator

- and returns *True* if both conditions are true, otherwise, it returns *False*

```
>>> 2 > 1 and 3 < 4
True
```

- or returns *True* if at least one of the conditions is true

```
>>> 2 > 1 or 3 < 4
True
>>> 2 > 1 or 3 > 4
True
```

- not is used to negate the condition if the checked condition is set to *True* as a result of the not operator, it will have a *False* value, if the checked condition is *False*, then in combination with not, it will be set to *True*

```
>>> not 2 > 3
True
>>> not 2 < 3
False
```

8.1.7

Complete the record to get *True* as a result

```
>>> _____ 45 >= 60
True
```

8.1.8

Complete the record to get *True* as a result

```
>>> 3 > 2 _____ 1 < 4
True
>>> 2 == 4 _____ 3 > 2
True
```

8.1.9

In Python, there is also an operator that checks if two variables refer to the same place in memory, this is the operator '**is**', and its version negated '**is not**'

```
>>> a = 3
>>> b = 3
>>> c = a
>>>a is b
False
>>> a is c
True
>>>a is not c
False
```

8.2 Logic Expression - programs

8.2.1 Smaller number

Write the code to check if the first of the two numbers given is smaller than the second.

```
Input : 2 4
Output: True
```

```
Input : 5 2
```

```
Output: False
```

⌨ 8.2.2 Number from the range

Write the code that checks if the given number is between 0 and 20.

```
Input : 5
Output: True
```

```
Input : 25
Output: False
```

If Command

Chapter 9

9.1 If - else

📖 9.1.1

Sometimes there is a need to separate the code into one that will be executed if the condition is *True* and one that will be executed if the condition is *False*.

The ***if - else*** statement is used for this

If condition:

Instructions when the condition is true

else:

Instructions when the condition is false

Example:

```
>>> a = input ('Enter a number')
>>> b = input ('Enter the second number')
>>> if a == b:
  print ('the numbers given are equal')
else:
  print ('the numbers given are different')
```

📝 9.1.2

What will be the result of the following code?

```
a = 4
b = 'Python'
if a%2 == 0:
  print(b)
else:
  print(a)
```

- 'Python'
- 4
- 2
- 'Python' 4

9.1.3

Complete the code so that the statement inside if executes

```
a = 5
b = 6
if a _____ b_____
  print(a+b)
```

9.2 If - elif - else

9.2.1

Sometimes the if - else statement is not sufficient, for example, if we have more than one condition. There is a version of the *if* statement in *Python* that lets you check several conditions: ***if -elif - else***.

```
if condition1:
                Instructions executed when the condition 1 is
true
elif condition2:
                 Instructions executed when the condition 2 is
true
else:
                Instructions executed when condition1 and
condition 2 are false
```

The number of additional conditions is unlimited.

```
>>> if a == b:
  print ('the numbers given are equal')
elif a> b:
  print ('a is greater than b')
else:
  print ('a is less than b')
```

9.2.2

Complete the following code:

```
if a == b:
  print ('the numbers given are equal')
```

```
_____ a> b:
  print ('a is greater than b')
_____:
  print ('a is less than b')
```

📝 9.2.3

In Python, you can use the elif statement to split the code into more than two cases as part of an if statement.

- True
- False

9.3 If

🕮 9.3.1

The ***if*** statement allows you to limit the execution of a part of the program to a situation when we have some condition (or conditions)

```
If condition:
        Statement_1
Statement_2
```

Statement_1 will be executed only if the condition is true, *Statement_2* will be executed regardless of the condition

```
>>> a = 5
>>> b = 6
>>> if a == b:
  print ('numbers are equal')
>>> print ('value a is {} and value of b is {}'. format (a,
b))
  'value a is 5 and value b is 6'
```

```
>>> a = 5
>>> b = 5
>>> if a == b:
  print ('numbers are equal')
>>> print ('value a is {} and value of b is {}'. format (a,
b))
```

```
  'numbers are equal'
  'value a is 5 and value b is 6'
```

📝 9.3.2

Complete the code:

```
_____ a%2 == 0 _____
  print(a)
```

🕮 9.3.3

Within one ***if*** statement we can ***check several conditions*** using logical operators:

- and both conditions must be true

```
If a> b and b> c:
       statement_1
```

- or - one of the conditions must be true

```
If a> b or b> c:
        statement_1
```

- not - the second condition cannot be true

```
If a> b and not a == 4:
        statement_1
```

Example:

```
if a> b and a> 0:
print ('a greater than b and a greater than 0')
```

📝 9.3.4

Complete the code so that the command inside the *if* statement executes

```
a = 8
b = 6
if a%2==0 _____ b_____a:
  a = a*b
```

9.3.5

Will the print command execute in the following code?

```
a = 5
b = 0
if a == 5 and not b == 0:
  print(a*b)
```

- No
- Yes

9.4 If command - programs

9.4.1 Bigger number

Write the code that will display the bigger of two numbers given by the user

```
Input : 2 4
Output: 4
```

9.4.2 BMI

Write the code that calculates the body mass index *BMI*. *BMI* is calculated as *weight[kg] / (height [m]* ^{2}*)*.

- BMI in the range 19 - 25 means *normal weight*
- BMI below 19 means *underweight*
- BMI over 25 means *overweight*

The program is to display an appropriate message depending on the BMI value (*correct weight, overweight, underweight*)

```
Input : 75 1.80
Output: correct weight
```

9.4.3 Interval

Write the code to see if the given number is from a given interval. Ensure that the upper and lower bounds of the interval are correctly entered. The first number given is always the searched one and then are the interval boundaries followed.

```
Input : 10 2 13
Output: true
```

```
Input : 5 15 7
Output: false
```

9.4.4 Similar numbers

Write the code that will decide whether the three given numbers are similar (use only one condition).

```
Input : 75 175 48
Output: False
```

```
Input : 8 8 8
Output: True
```

9.4.5 Roots of Quadratic Functions

Write the code that will calculate for given factors of the quadratic equation the roots of the equation.

The program is to display one or two solutions or information about the *no solution*

```
Input : 2 5 3
Output: -1.5 -1
```

```
Input : 2 4 2
Output: -1
```

```
Input : 2 4 3
Output: no solutions
```

9.4.6 Even number

Write the code to check if the given number is an even number.

```
Input : 8
Output: True
```

```
Input : 7
Output: False
```

⌨ 9.4.7 Mathematical operations

Write the code that based on the given math operator *(+, -, *, /)* will return the sum, difference, product or division between two given numbers.

```
Input : + 1 2
Output: 3
```

```
Input : * 3 5
Output: 15
```

Range

Chapter 10

10.1 The range() function

📖 10.1.1

The ***range()*** function in *Python* is used to generate a list of numbers.

The simplest ***range()*** call is to specify one parameter - number of integers (how many numbers do we want to get from zero):

```
range (3) #give a list [0, 1, 2]
```

Calling range alone will not do anything, only an iterable object will be created

Example of using range ():

```
print ([a for a in range (3)])
# this is equivalent to print ([0, 1, 2])
```

The list does not have to be iterated from zero, we can call *range (start, stop)*, where:

Start - An integer value indicating the position from which to start - 0 by default

Stop - means on what position to end

```
print([a for a in range(10, 15)])
give
[10, 11, 12, 13, 14]
```

📝 10.1.2

The first element of the list created with the *range (10)* command will be

📝 10.1.3

The last element of the list created with the *range (10, 20)* command will be

📖 10.1.4

In the ***range ()*** function we can also define the step by which the values in the list will change (***step*** parameter):

range (start, stop step)

Example:

```
print ([a for a in range (0, 11, 2)])
returns even numbers in the range 0-10:
[0, 2, 4, 6, 8, 10]
```

The ***step*** parameter can ***also take negative values***, then we can create a list with decreasing values, for example:

```
print ([a for a in range (10, -1, -2)])
[10, 8, 6, 4, 2, 0]
```

📝 10.1.5

The second element of the list created using the *range (10, 20, 3)* command will be:

10.2 The reversed() function

🕮 10.2.1

The ***reversed ()*** function returns the reverse list, given as a parameter ***reversed (seq).***

Example:

```
x=[1, 2, 3, 4 , 5, 6, 7, 8 , 9, 10]
list(reversed(x))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

The ***reversed()*** function can be used on the *tuple, list, string, range.*

```
# string
text = 'Python'
print(list(reversed(text)))

# tuple
exTuple = ('P', 'y', 't', 'h', 'o', 'n')
print(list(reversed(exTuple)))

# range
exRange = range(1, 6)
```

```
print(list(reversed(exRange)))

# list
exList = [0, 1, 2, 4, 3, 5]
print(list(reversed(exList)))

['n', 'o', 'h', 't', 'y', 'P']
['n', 'o', 'h', 't', 'y', 'P']
[5, 4, 3, 2, 1]
[5, 3, 4, 2, 1, 0]
```

📝 10.2.2

What will be the first element of the list created using the following code:

```
text = 'Python'
print(list(reversed(text)))
```

Loops

Chapter 11

11.1 For - in range()

📖 11.1.1

Sometimes when writing a program we have to repeat the execution of a piece of code a certain number of times. To avoid having to copy the same code multiple times, you can use ***loops***. There are several types of *loops* in Python.

The for ***loop*** allows you to repeat certain code. The for *loop* repeats a portion of the program based on a sequence - an ordered list of certain elements.

The overall *loop* record is as follows:

```
For iteratingVariable in sequance:
                instructions
```

The ***for*** *loop* repeats the instruction string inside the loop for each sequence element in turn. When the end of the sequence is reached, the loop ends.

Instructions executed inside a *for* loop are indented relative to the *for* statement itself.

📝 11.1.2

The correct entry of a *for* loop in Python is:

- for variable in sequence:
- for sequence:
- for (){}
- for variable:sequence:

📖 11.1.3

You can use the ***range()*** function to create a sequence for the *for* loop

```
for i in range(1,6):
   print(i)
1
2
3
4
5
```

Example:

```
1.    i=1;
2.    num = int(input("Enter a number:"));
3.    for i in range(1,11):
4.        print("%d X %d = %d"%(num,i,num*i));

Enter a number:5
5 X 1 = 5
5 X 2 = 10
5 X 3 = 15
5 X 4 = 20
5 X 5 = 25
5 X 6 = 30
5 X 7 = 35
5 X 8 = 40
5 X 9 = 45
5 X 10 = 50
```

📝 11.1.4

Complete the following *for* loops:

```
_____ i _____ range(10)_____
      print(i)
```

📝 11.1.5

How many times the following loop will be executed:

```
for i in range(5):
      print(i)
```

11.2 For - in values

🕮 11.2.1

In a *for* loop, you can use the values in the loop separated by the comma

```
for i in 'cat', 'dog', 'fish', 'hamster':
    print('I have a', i)
I have a cat
```

```
I have a dog
I have a fish
I have a hamster
```

Or

```
for i in 2 , 3, 4:
    print('I have', i, 'dogs')

I have 2 dogs
I have 3 dogs
I have 4 dogs
```

The sequence for the *for* loop can also be ***defined before***:

```
animals = ['cat', 'dog', 'fish', 'hamster']
for i in animals:
    print('I have a', i)
I have a cat
I have a dog
I have a fish
I have a hamster
```

Or

```
num = [2,3,4]
for i in num:
    print('I have', i, 'dogs')

I have 2 dogs
I have 3 dogs
I have 4 dogs
```

11.2.2

Is the following *for* loop statement correct?

```
for i in [2,3,4]:
      print(i)
```

- Yes
- No

📝 11.2.3

What will appear after executing the following code:

```
animals = ['cat', 'dog', 'fish', 'hamster']
for i in animals:
    if i == 'fish':
          print('I have a', i)
```

📝 11.2.4

How many times will the following loop be performed?

```
for i in 2, 4, 0, 10, 20, 13:
    print(i)
```

11.3 For - in string

🕮 11.3.1

Using a *for* loop it is also possible to iterate over characters into a string:

```
for i in 'Python':
    print(i)
P
y
t
h
o
n
```

Or

```
text = 'Python'
for i In text:
    print(i, end=" ")
P y t h o n
```

📝 11.3.2

Is the following entry correct?

```
text = 'for loop'
```

```
for i in text:
    print(i, end="-")
```

- Yes
- No

11.4 Cycles - programs

⌨ 11.4.1 Displaying range of numbers

Write the code that displays numbers from 1 to the value given by the user.

```
Input : 3
Output: 1 2 3
```

⌨ 11.4.2 Even numbers in the range

Write a program that displays even numbers between 2 and the value given by the user.

```
Input : 9
Output:
2
4
6
8
```

⌨ 11.4.3 Word by character

Write the code that displays the word given by the user, character by character:

```
Input : Python
Output: P
y
t
h
o
n
```

⌨ 11.4.4 Sum of numbers from interval

Write the code that will return the sum of all numbers from a given interval.

```
Input : 5 10
Output: 45
```

```
Input : 8 1
Output: 36
```

⌨ 11.4.5 Product of numbers from interval

Write the code that will return the product of all numbers from a given interval.

```
Input : 5 10
Output: 151200
```

```
Input : -4 -1
Output: 24
```

⌨ 11.4.6 Number of divisible numbers

Write the code that will return for a given number the count of divisible numbers in a given interval. Ensure that the upper and lower bounds of the interval are correctly entered. The first number given is always the searched one and then are the interval boundaries followed.

```
Input : 5 1 25
Output: 5
```

```
Input : 2 6 17
Output: 6
```

⌨ 11.4.7 Word without 'a'

Write the code that will display the word given by the user omitting the letters 'a'

```
Input : Banana
Output: Bnn
```

Modules

Chapter 12

12.1 Import

📖 12.1.1

There are a large number of **additional modules** in Python. Modules are just files with the *.py* extension that contain a **set of functions**. To be able to use the module, you must import it into our program, use the **import** command

```
import module_name
```

The module is attached to our script as if we had written it ourselves. Thanks to the use of ready-made modules, we do not have to create part of the function

📝 12.1.2

To import a module, use the command:

📖 12.1.3

One of the sample modules is the math module, which contains **mathematical functions** defined in the C standard

Within the module we have many mathematical functions available, below are some of the most popular:

```
import math
x=2.34
print(math.ceil(x)) # Return the smallest integer value
greater than or equal to x
print(math.floor(x))# Return the largest integer value greater
than or equal to x
print(math.modf(x)) # Return the fractional and integer parts
of x

x = -3
print(math.fabs(x))# Return the absolute value of x

x = 4

print(math.log(x, 2)) # Return the logarithm of x to the given
base
print(math.log1p(x)) # Return the natural logarithm (base e)
```

```
print(math.log10(x)) # Return the base-10 logarithm of x

y = 2
print(math.pow(x, y)) # Return x raised to the power y
print(math.sqrt(x)) # Return the square root of x

print(math.sin(x)) # Return the sine of x radians
print(math.cos(x)) # Return the cosine of x radians
print(math.tan(x)) # Return the tangent of x radians

print(math.degrees(x)) # Convert angle x from radians to
degrees
print(math.radians(x)) # Convert angle x from degrees to
radians

print(math.pi) # The mathematical constant π = 3.141...
print(math.e) # The mathematical constant e = 2.718...
```

Output:

```
3
2
(0.33999999999999986, 2.0)
3.0
2.0
1.6094379124341003
0.6020599913279624
16.0
2.0
```

12.1.4

To import the math module, use the command

12.1.5

To carry 3 to the power of 4, use the _____(3,4) command.

12.1.6

The ***math.sin(x)*** command takes the angle value in degrees as the *x* parameter

- False
- True

12.1.7

To see what the **module contains**, we can display the functions that have been included in it:

```
Import module
dir(module)
[list of function in module]
```

Example

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

To get help on a specific function, use the **help** command

```
help(function_name)
```

Example

```
Help(math.cos)
Help on built-in function cos in module math:
cos(...)
   cos(x)
   Return the cosine of x (measured in radians).
```

Creating your own modules

In Python, any file with the *.py* extension can be a module, which means it can be imported into another script.

📝 12.1.8

To call help for a command from a module, you can use the command:

- help()
- ?operation
- help - operation
- operation?

12.2 Random

🕮 12.2.1

The ***random module*** is also a popular module, in which pseudo-random methods for various types of data have been implemented.

```
Module Import
import random
```

The basic functions contained in the module can be included

```
>>> random.random()         # Random float x, 0.0 <= x < 1.0
0.37444887175646646
>>> random.uniform(1, 10)   # Random float x, 1.0 <= x < 10.0
1.1800146073117523
>>> random.randint(1, 10)   # Integer from 1 to 10, endpoints
included
10
>>> random.randrange(0, 101, 2)   # Even integer from 0 to 100
26
>>> random.choice('abcdefghij')   # Choose a random element
'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)    # Randomly changes the order of
items in the list
>>> items
```

```
[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5],  3)  # Choose 3 elements
[4, 1, 5]
```

12.2.2

Complete the command to generate a random integer from 10-20

```
random._____(10, 21)
```

12.2.3

To change the order of items in the list, use the command

- random.shuffle()
- random.reverse()
- random.random()
- random.reorder()

12.3 Modules - programs

12.3.1 Circumference of the circle

Write the code that calculates the circumference of a circle for the given radius value (use the math module for *PI* values). Round up the result to integer values.

```
Input : 4
Output: 26
```

12.3.2 Degrees to radians

Write the code converting the angle value in degrees to radians. The result should be rounded to three decimal places

```
Input : 80
Output: 1.396
```

Loops II.

Chapter 13

13.1 Break and continue

📖 13.1.1

Loops often need additional commands when you want to stop the loop when a condition is met or skip one loop. Two additional commands ***break*** and ***continue*** are used

The ***break*** command terminates the loop

Eg.

```
for i in range(1,10):
    if (i == 5):
         break
      print(i)
1
2
3
4
```

The command ***continue*** terminates the current iteration of the loop and goes to the next iteration

Eg.

```
for i in 'Python':
     if (i == 'h'):
         continue
     print(i, end = " " )
```

Output:

```
P y t o n
```

📝 13.1.2

How many times the following loop will be executed:

```
for i in range(1,10):
    if( i == 5):
           break
      print(i)
```

📝 13.1.3

How many times the print command will be executed in the following loop

```
for i in range(1,10):
    if( i % 2 == 0):
            continue
      print(i)
```

13.2 Nested cycles

🕮 13.2.1

A loop can contain one or more other loops - we can **create loops inside** the loop

For example, try displaying the multiplication table for numbers 1 to 5:

```
for i in range(1,6):
    for j in range(1,6):
        print(i, 'x', j, '=', i*j, end="; ")
    print('')
```

Output:

```
1 x 1 = 1; 1 x 2 = 2; 1 x 3 = 3; 1 x 4 = 4; 1 x 5 = 5;
2 x 1 = 2; 2 x 2 = 4; 2 x 3 = 6; 2 x 4 = 8; 2 x 5 = 10;
3 x 1 = 3; 3 x 2 = 6; 3 x 3 = 9; 3 x 4 = 12; 3 x 5 = 15;
4 x 1 = 4; 4 x 2 = 8; 4 x 3 = 12; 4 x 4 = 16; 4 x 5 = 20;
5 x 1 = 5; 5 x 2 = 10; 5 x 3 = 15; 5 x 4 = 20; 5 x 5 = 25;
```

🕮 13.2.2

Example two

```
lists = [['apple', 'banana', 'orange'],[0, 1, 2],[1.1, 2.2,
3.3]]

for list in lists:
    for item in list:
        print(item)
```

Output:

```
apple
```

```
banana
orange
0
1
2
1.1
2.2
3.3
```

13.3 Cycles II - programs

13.3.1 Stars

Write the code that will for the given values *m* and *n* return *m* rows where the interior will be empty and the stars will only be perpendicular. In the code replace the stars with the character *a*.

```
Input : 4 3
Output:
***
* *
* *
***
```

```
Input : 5 5
Output:
*****
*   *
*   *
*   *
*****
```

13.3.2 Sequence of digits

Write the code that will return the given number in the following sequence:

```
Input : 3
Output:
1
12
123
```

```
Input : 7
```

```
Output:
1
12
123
1234
12345
123456
1234567
```

13.3.3 Sum of numbers

Write the code that calculates the sum of two times the numbers in the range provided by the user. When the sum will be greater than 100 the program should stop. The program should omit calculations number 13.

```
Input : 2 7
Output: 54
```

```
Input : 1 120
Output: 100
```

13.3.4 Multiplication table

Write the code that will return the multiplication table (from 1x1 to 10x10).

```
Input :
Output:
1 2 3 4 5 6 7 8 9 10
2 4 6 8 10 12 14 16 18 20
3 6 9 12 15 18 21 24 27 30
...
```

While

Chapter 14

14.1 While

📖 14.1.1

In Python, there is a ***while*** loop in addition to the for a loop. The loop syntax is:

while condition:

    instructions

The structure of the while loop resembles an *if* statement, it differs from it in that the instructions contained within the while loop are repeated ***as long as*** the condition is met

An example of a ***while*** loop displaying numbers from 1 to 10:

```
>>>x = 1
>>> while x < 11:
        print(x)
        x+=1
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

In the above example, remember to ***increase*** the value of the variable x, otherwise, we would get an infinite loop, the operation of which would not end, the condition would always be met.

📝 14.1.2

The correct entry for a *while* loop in Python is:

- while condition:
- while (condition){}

- while {condition}
- while condition do

📝 14.1.3

How many times the code inside *while* loop will be executed

```
>>> x = 8
>>> while x < 11:
        print(x)
        x+=1
```

📖 14.1.4

The ***while*** loop can also work on ***lists***

```
>>> x = ['cat', 'dog', 'fish']
>>> while x:
      print(x.pop(-1)) #.pop(-1) removed last element, when
the list is empty, x is false, and the loop terminates
```

Output:

```
fish
dog
cat
```

14.2 While true

📖 14.2.1

Sometimes it is more convenient to use an ***infinite*** loop, i.e. ***without a condition***. This loop should be served with a ***break*** statement, you should never use it if you don't need to use it;

Infinite loop:

>>> while True:

Instruction

Example:

```
>>> while True:
        Print('Python')
Python
Python
….
Python
```

This loop will not interrupt its operation, the stop condition is missing.

```
>>> i=1
>>> while True:
        print('Python')
        i += 1
          if (i>10):
              break
```

This loop will display Python 10 times, after which its execution will be stopped with the break statement

14.2.2

What command should stop the infinite loop:

```
while True:
```

14.2.3

Will the following loop work properly:

```
i=0
while True:
  print(i*2)
  i=i+1
  if (terms > 10)
    break
```

- No
- Yes

14.3 While - programs

⌨ 14.3.1 Square

Write the code that reads a positive integer n from the user and then displays on the screen all the powers of 2 not greater than the number given.

```
Input : 71
Output: 1
2
4
8
16
32
64
```

⌨ 14.3.2 Divisible by 7

Write the code displaying numbers divisible by 7 from the range 7 to the given number

```
Input : 30
Output: 7
14
21
28
```

Functions

Chapter 15

15.1 Functions without parameters

📖 15.1.1

The general term *function* is used to describe a traditional, stateless function that is invoked without the context of a particular class or an instance of that class. We use the more specific term *method* to describe a member function that is invoked upon a specific object using an object-oriented message passing syntax.

A *function* is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing. Python gives you many built-in functions for use but you can also create your own functions to fit your specific need.

A function in Python is a logical unit of code containing a sequence of statements indented under a name given using the "***def***" keyword. Functions allow you to create a logical division of a big project into smaller modules. They make your code more manageable and extensible. While programming, it prevents you from adding duplicate code and promotes reusability.

📖 15.1.2

The keyword ***def*** is a statement for defining a function in Python. You have to start every function with the **def** keyword and specify the function name followed by brackets and a colon symbol "**:**".

```
def my_function():
  statement
```

The ***def*** call creates a function object and assigns it to the given name. It is possible to further re-assign the same function object to other names. The brackets can contain parameters but we will discuss the parameters later. This is called a function without parameters.

📝 15.1.3

What is the keyword used to define a function?

📖 15.1.4

As was told in the previous lesson to define the function we use the keyword ***def***. Let us now focus on the function body. The statements that form the function body starts at the ***next line*** and must be ***indented***. When the function is called the code in the function body is run.

```
def print_hello():
      print('hello')
```

Then the output would be following when calling the function *print_hello()* :

```
hello
```

📝 15.1.5

Fill in the code to define a function:

```
_____ hello_world()_____
  print('hello world!')
```

📖 15.1.6

Using the function definition we learned how to create a function. We have got a blueprint that has a name and body with valid Python statements. The next step is to execute it. This can be done by calling it from the Python script or inside a function or directly from the Python shell.

```
def print_hello():
      print('hello')
```

To call a function, you need to specify the function name.

```
print_hello()
```

This will result in the following Output:

```
hello
```

📝 15.1.7

Fill in the code to call the created function:

```
def hello_world():
```

```
  print('hello world!')
```

```
>> _____() #call of the function
```

15.2 Functions without parameters (programs)

⌨ 15.2.1 Hello World

Write the code that will use a function without parameters and print a standard programming greeting.

```
Input :
Output: Hello World
```

⌨ 15.2.2 Print numbers

Write the code that will use a function without parameters and print the numbers from 1 to 10 in separate rows.

```
Input :
Output: 1
2
3
4
5
6
7
8
9
10
```

⌨ 15.2.3 Sum of 100

Write the code that will use a function without parameters and print the sum of numbers from 1 to 100.

```
Input :
Output: 5050
```

⌨ 15.2.4 Draw Rectangle

Write the code that will use a function without parameters and print a rectangle of 4x4 consisting of the characters *"o"*.

```
Input :
Output:
oooo
oooo
oooo
oooo
```

⌨ 15.2.5 Draw Triangle

Write the code that will use a function without parameters and print a triangle of height 5 consisting of the characters *"o"*.

```
Input :
Output:
o
oo
ooo
oooo
ooooo
```

⌨ 15.2.6 Draw an Empty Rectangle

Write the code that will use a function without parameters and print an empty rectangle of 10x10 consisting of the characters *"o"*.

```
Input :
Output:
oooooooooo
o        o
o        o
o        o
o        o
o        o
o        o
o        o
o        o
oooooooooo
```

⌨ 15.2.7 Print Name from Input

Write the code that will use a function without parameters and print a greeting for the name given at the input (read the input inside the function).

```
Input : John
Output: Hello John
```

⌨ 15.2.8 Multiplication Table

Write the code that will use a function without parameters and print a multiplication table for the given number (read the input inside the function).

```
Input : 7
Output: 1x7=7
2x7=14
3x7=21
4x7=27
5x7=35
6x7=42
7x7=49
8x7=56
9x7=63
10x7=70
```

15.3 Local and global variables with functions

🕮 15.3.1

When you declare variables inside a function definition, they are not related in any way to other variables with the same names used outside the function i.e. variable names are ***local*** to the function. This is called the ***scope*** of the variable. All variables have the *scope* of the block they are declared in starting from the point of definition of the name.

In the function where we use the defined variable for the first time (we declare it in the function), the variable stays available ***only for the function.*** That means that if we break the indent of the statements, then the variable cannot be used further. On the other hand, each function evaluation creates a local namespace that is manipulated at any level within the function. As a result, variables can be initially defined at a seemingly lower level of scope than they are eventually used.

📖 15.3.2

Let's look at an example of using local variables inside a function. There are not many cases that you need a function with only local variables. E.g. define a function that will print the current date.

```
import datetime         # we need the datetime module

def print_today():
  today = datetime.datetime.now().date
  print(today)

print_today()
```

The output will be the date when you call this function. It can be *2019-12-12* or any other day.

📝 15.3.3

What will be the output of the following code?

```
x = 25

def function():
  x = 0
  print('x is ', x)

function()
print('x is ', x)
```

- x is 0; x is 25;
- x is 25; x is 25;
- x is 0; x is 0;

📖 15.3.4

Let's use the last example of printing the current date and let's try to print the variable today outside of the function.

```
import datetime         # we need the datetime module

def print_today():
```

```
  today = datetime.datetime.now().date
  print(today)

print_today()
print(today)
```

In this example, we try to access a local variable defined in the function outside of the function body. The result of this call will be a *NameError*:

```
>> NameError: name 'today' is not defined
```

The variable *today* will not hold the value outside of the function. In this case, the variable is even not defined. Despite that, the names used inside a ***def*** do not conflict with variables outside the ***def*** even if there are used the same variable names elsewhere.

📝 15.3.5

What will be the output of the following code? The *random()* function will generate a random number from the interval 0-1.

```
from random import random

def print_random():
  r = random()
  print(r)

print(r)
```

- NameError: name 'r' is not defined
- 0.3011016848492

🕮 15.3.6

In *Python*, the variables assignment can occur at three different places:

- inside a ***def:*** the variable is local to the function;
- in an enclosing ***def***: the variable is nonlocal to the nested functions;
- outside all ***def***(s): the variable is global to the entire file.

The ***global*** keyword is a statement in *Python*. It makes it possible for the variables (names) to retain changes that live outside of a ***def*** at the top level of a module life. In a single *global* statement can be specified one or more names separated by commas. All the listed names attach to the enclosing module's scope when assigned or referenced within the function body.

```
x = 10
y = 20
def fn() :
    global x
    x = 100
    y = 200
      # a local variable 'y' is assigned and created here
      # whereas, the variable 'x' refers to the global name
fn()
print(x, y)
```

The resulting output will be following:

```
100 20
```

In the above code, *x* is a global variable that will retain any change in its value made in the function. Another variable *y* has local scope and won't carry forward the change.

15.3.7

What will be the value of the variable "*n*" after calling both functions?

```
n = 0

def function1() :
    n = 1

def function2() :
    global n
    n = 2
```

```
>> function1()
>> function2()
>> print(n)
```

- 2
- 1

- 0

15.4 Functions with parameters

📖 15.4.1

A function can take parameters which are values you supply to the function so that the function can do something utilising those values. These parameters are just like variables except that the values of these variables are defined when we call the function and are already assigned values when the function runs. Parameters are used when we want to pass data into our function.

Parameters are specified within the pair of parentheses in the function definition, separated by commas. When we call the function, we supply the values in the same way. We often use the terms ***parameters*** and ***arguments*** interchangeably. However, there is a slight difference between them. ***Parameters*** are the variables used in the function definition whereas ***arguments*** are the values we pass to the function parameters.

```
def print_text(text):
  print(text)

print("Hello World")
```

📝 15.4.2

Fill the following code with the correct parameter. The function will print the power of the given argument.

```
def print_power(_____):
  print(num*num)

print_power(2)
```

📖 15.4.3

When parameters are defined for a function you have to pass an argument. Otherwise the program will return a *TypeError*.

```
def print_text(text):
  print(text)
```

```
print_text()
```

This will result in the following error because when we call the function name we have to pass also the argument with the value.

```
TypeError: print_text() missing 1 required positional
argument: 'text'
```

📖 15.4.4

When declaring a function, we can add as many parameters as we want, we just need to separate them with ***commas***. In many cases, we need more than just one parameter. The order in which the arguments are passed corresponds to the order of the parameters in our function definition.

Also, you have to remember that all of the arguments get assigned to *local* variable names once passed to the function. Changing the value of an argument inside a function does not affect the caller.

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

addition(4,5)
```

This is an example of a function with more than one parameter.

📝 15.4.5

Fill in the code so that the function will print the number only if it is even.

```
def even(_____):
  if(n%2==0):
    print(_____)


even(5)
even(_____)
```

Where the output will be the following:

```
>> 6
```

📖 15.4.6

All parameters (arguments) in the *Python* language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Example:

```
# define the function
def changenames(nameslist):
   nameslist.append(["John","Peter","Carol"]);
   print("Values inside the function: ", nameslist)
   return

# now you can call changenames function
nameslist = ["Jack","Jill"];
changenames(nameslist);
print("Values outside the function: ", nameslist)
```

Here we are maintaining the reference of the passed object and appending values in the same object. So, this would produce the following result:

```
>> Values inside the function:  ["Jack", "Jill" ["John",
"Peter", "Carol"]]
>> Values outside the function:  ["Jack", "Jill" ["John",
"Peter", "Carol"]]
```

📖 15.4.7

There is one more example where the argument is being passed by reference and the reference is being overwritten inside the called function.

```
# define the function
def changenames(nameslist):
   nameslist=["John","Peter","Carol"];
   print("Values inside the function: ", nameslist)
   return

# now you can call changenames function
nameslist = ["Jack","Jill"];
changenames(nameslist);
print("Values outside the function: ", nameslist)
```

The parameter *nameslist* is *local* to the function changenames. Changing *nameslist* within the function does not affect *nameslist*. The function accomplishes nothing and finally, this would produce the following result:

```
>> Values inside the function:  ["John", "Peter", "Carol"]
>> Values outside the function:  ["Jack", "Jill"]
```

15.4.8

Fill in the code so that the function does not change the content of the *numlist*:

```
def changenumbers(numlist):
    _____[2,3,4]_____
    print("Values inside the function: ", numlist)
    return

numlist = [1,2];
changenumbers(numlist);
print("Values outside the function: ", numlist)
```

The code will produce the following result:

```
>> Values inside the function:  _____
>> Values outside the function:  _____
```

15.5 Functions with parameters (programs)

15.5.1 Even Digits

Write the code that will use a function with a parameter that contains a number given by the input and print all even digits that are contained in the given number (read the input outside the function and use it as an argument).

```
Input : 123456
Output: 2
4
6
```

15.5.2 Odd Digits

Write the code that will use a function with a parameter that contains a number given by the input and print all odd digits that are contained in the given number (read the input outside the function and use it as an argument).

```
Input : 123456
Output: 1
3
5
```

⌨ 15.5.3 Sequence

Write the code that will use a function with a parameter that contains a number given by the input and print all numbers in the following sequence (read the input outside the function and use it as an argument).

```
Input : 3
Output: 1
12
123
```

```
Input : 5
Output: 1
12
123
1234
12345
```

⌨ 15.5.4 Factorial

Write the code that will use a function with a parameter that contains a number given by the input and print the factorial of the given number (read the input outside the function and use it as an argument). Factorial is calculated following: *n! = 1*2*3*...*(n-2)*(n-1)*n*. The factorial of 0 is 1 and the factorial of a negative number does not exist.

```
Input : 5
Output: 120
```

```
Input : -5
Output: does not exist
```

⌨ 15.5.5 Maximum of three numbers

Write the code that will use a function with a parameter that contains numbers given by the input and print the maximum of the given numbers (read the input outside the function and use it as an argument).

```
Input : 5 3 6
Output: 6
```

```
Input : -5 -11 -7
Output: -5
```

15.5.6 Product of numbers

Write the code that will use a function with a parameter that contains numbers given by the input and print the product of the given numbers (read the input outside the function and use it as an argument).

```
Input : 5 3
Output: 15
```

```
Input : -5 -11
Output: 55
```

15.5.7 Difference of numbers

Write the code that will use a function with a parameter that contains numbers given by the input and print the difference of the given numbers (read the input outside the function and use it as an argument).

```
Input : 5 3
Output: 2
```

```
Input : -5 -11
Output: 6
```

15.5.8 Rectangle

Write the code that will use a function with a parameter that contains numbers given by the input and print the perimeter and area of the rectangle of the given numbers that are the sides of the rectangle (read the input outside the function and use it as an argument).

```
Input : 5 8
Output: Perimeter is 26 and area is 40
```

```
Input : 15 5
Output: Perimeter is 40 and area is 75
```

⌨ 15.5.9 Triangle

Write the code that will use a function with a parameter that contains numbers given by the input and print the perimeter of the triangle of the given numbers that are the sides of the triangle (read the input outside the function and use it as an argument).

```
Input : 5 8 5
Output: 18
```

```
Input : 1 2 3
Output: 6
```

⌨ 15.5.10 Surface Area and Volume of a Rectangular Prism

Write the code that will use a function with a parameter that contains numbers given by the input and print the surface area and volume of a rectangular prism from given sides (read the input outside the function and use it as an argument).

```
Input : 5 8 6
Output: The surface area is 236 and volume is 240
```

```
Input : 15 10 12
Output: The surface area is 900 and volume is 1800
```

⌨ 15.5.11 Chessboard

Write the code that will use a function with a parameter that contains string given by the input and prints a grid with the given dimensions *m* x *n* and mark them as a chessboard (*x*, *o*) (read the input outside the function and use it as an argument).

```
Input : 4 4
Output: xoxo
oxox
xoxo
oxox
```

```
Input : 2 3
Output: xox
oxo
```

15.6 Function arguments

📖 15.6.1

You can call a function by using the following types of formal arguments:

- required arguments;
- keyword arguments;
- default arguments.

📖 15.6.2

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

```
def greeting(name):
  print("Hello ",name)

greeting()
```

If you want to call a function that requires an argument, you definitely have to pass one otherwise you will get an error:

```
TypeError: greeting() takes exactly 1 argument (0 given)
```

📝 15.6.3

What will be the output of the following code?

```
def greeting(name):
  print("Hello ",name)

greeting("John")
```

- Hello John
- TypeError: greeting() takes exactly 1 argument (0 given)

📖 15.6.4

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *greeting()* function in the following ways:

```
def greeting(name):
  print("Hello ",name)

greeting(name="Jack")
```

When the above code is executed, it produces the following result:

```
>> Hello Jack
```

The following example gives a more clear picture. Note that the order of parameters does not matter.

```
def greetings(name, surname):
  print("Hello ",name," ",surname)

greetings(surname="Silver", name="John")
```

When the above code is executed, it produces the following result:

```
>> Hello John Silver
```

📝 15.6.5

Fill in the code to print the subtraction of the two given number parameters.

```
def difference(a,b):
  print(_____)

difference(_____7,b=_____)
```

```
>> -2
```

📖 15.6.6

A ***default argument*** is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default work position if it is not passed:

```
def employee(name, position="administrative"):
  print("Mr./Mrs. ",name," works at position: ",position)

employee(name="Smith")
employee(position="director",name="Jackson")
```

When the above code is executed, it produces the following result:

```
>> Mr./Mrs. Smith works at position: administrative
>> Mr./Mrs. Jackson works at position: director
```

📝 15.6.7

What will be the output of the following code:

```
def first_year(name, age=18):
  print("Student ",name," is ",age," years old.")

first_year(name="Pond")
```

- Student Pond is 18 years old.
- Student Pond is years old.
- Student Pond is 0 years old.
- Student Pond is None years old.
- Student Pond is NULL years old.

15.7 Functions - the return statement

📖 15.7.1

The statement ***return [expression]*** exits within a function, optionally passing back an expression to the caller. A ***return*** statement with no arguments is the same as ***return None***. All the examples used till now are not returning any value. In *Python* functions, you can add the *"**return**"* statement to return a value.

```
def sum(a,b):
```

```
  res = a+b
  return res

sum(3,5)
```

```
>> 8
```

15.7.2

Fill in the code to return the product of two numbers from the function:

```
def prod(a,b):
  res = _____
  _____

prod(5,10)
```

```
>> _____
```

15.7.3

As you have already learned the variable by *Python* is not needed to be defined as for other languages (e.g. *Java*, *Pascal*, ...). This means that you can return any type of the variable from the function: *integer*, *decimal* numbers (*float*), *boolean* (true/false), *string*.

Usually, the functions return a single value. But if required, *Python* allows returning multiple values by using the collection types such as using a tuple or list. This feature works like the call-by-reference by returning tuples and assigning the results back to the original argument names in the caller.

```
def compare(a,b):
  if(a>b):
     res = "a is bigger"
  elif(b>a):
     res = "b is bigger"
  else:
     res = "both numbers are equal"
  return res

compare(3,5)
```

```
>> b is bigger
```

15.7.4

Fill in the code to return the information which of the three number parameters is the smallest from the function:

```
def mini(a,b,c):
  min = a
  if(b_____min):
    min = _____
  if(c<min):
    _____
  return _____

mini(5,10,2)
```

```
>> _____
```

15.7.5

What will be the output of the following function?

```
def underage(age):
    if(age<18):
        res = True
    else:
        res = False
    return res

print(underage(17))
```

- True
- False

15.7.6

In the previous lessons, we worked with functions that did not return any result or value. Despite that, we could have used the keyword ***return*** to end the function or to return no value in case this happened. So when do we use *return* ***None***, ***return*** and no return? And is there any difference between these three notations? On the actual behavior, there is no difference. They all return ***None*** and that's it. However, there is a time and place for all of these.

📖 15.7.7

Using return None

This tells that the function is indeed meant to return a value for later use, and in this case, it returns *None*. This value *None* can then be used elsewhere. ***return None*** is never used if there are no other possible return values from the function.

In the following example, we return *a person*'s *mother* if the *person* given is a human. If it's not a human, we return *None* since the *person* doesn't have a *mother* (let's suppose it's not an animal or something).

```
def get_mother(person):
    if is_human(person):
        return person.mother
    else:
        return None
```

📖 15.7.8

Using return

This is used for the same reason as *break* in loops. The return value doesn't matter and you only want to exit the whole function. It's extremely useful in some places, even though you don't need it that often.

We have got 10 numbers from 80-90 and we know that one of them is divisible by 7. We loop through each number one by one to check if the division remainder of the number is 0. If we hit the correct number, we can just exit the function because we know there's only one number in the interval of 10 numbers that can be divided by 7. We do not have to check the rest of the numbers. If we do not find the number divided by 7 we print the info. This could be done in many different ways and this one is probably no the best way but it's an example of how to use the ***return*** to exit a function.

```
def find_seven(numbers):
    for num in numbers:
        if(num%7==0):
            print(num)
            return # no need to check rest of the numbers
    print("No such number here")
```

Note: You should never do *a = find_seven(numbers)* since the return value is not meant to be caught.

📖 15.7.9

Using no return at all

This will also return *None* but that value is not meant to be used or caught. It simply means that the function ended successfully. It's basically the same as a *return* in *void* functions in languages such as *C++* or *Java*.

In the following example, we set the person's mother's name and then the function exits after completing successfully.

```
def set_mother(person, mother):
    if is_human(person):
        person.mother = mother
```

Note: You should never do *a=set_mother(my_person, my_mother)* since the return value is not meant to be caught.

15.8 Functions - the return statement (programs)

⌨ 15.8.1 BMI

Write the code that will create a function with a parameter that contains numbers given by the input and return the BMI index and return information about your weight. BMI is calculated as a division of weight (in kg) and the height squared (in m), where BMI < 18.5 underweight, 18.5 <= BMI < 25 healthy, 25 <= BMI < 30 overweight, BMI > 30 obese. (read the input outside the function and use it as an argument)

```
Input : 75 175
Output: healthy
```

```
Input : 87 164
Output: obese
```

⌨ 15.8.2 Absolute value

Write the code that will create a function with a parameter that contains the number given by the input and return the absolute value of the given number. (read the input outside the function and use it as an argument)

```
Input : 75
Output: 75
```

```
Input : -7
Output: 7
```

15.8.3 Similar numbers

Write the code that will create a function with a parameter that contains numbers given by the input and return whether the three given numbers are similar. (read the input outside the function and use it as an argument)

```
Input : 75 175 48
Output: False
```

```
Input : 8 8 8
Output: True
```

15.8.4 Prime numbers

Write the code that will create a function with a parameter that contains numbers given by the input and return all prime numbers to the given number. (read the input outside the function and use it as an argument)

```
Input : 5
Output: 2 3 5
```

```
Input : 14
Output: 2 3 5 7 11 13
```

15.8.5 Days of the month

Write the code that will create a function with a parameter that contains numbers given by the input and for the given month return the count of its days (assume it is NOT a gap year). (read the input outside the function and use it as an argument)

```
Input : 2
Output: 28
```

```
Input : 14
Output: wrong month
```

15.8.6 Math operations

Write the code that will create a function with a parameter that contains numbers given by the input and return based on the given math operator (+, -, *, /) the sum, difference, product or division between two given numbers. (read the input outside the function and use it as an argument)

```
Input : + 1 2
Output: 3
```

```
Input : * 3 5
Output: 15
```

15.8.7 Month name

Write the code that will create a function with a parameter that contains a string given by the input and return the number of days of a given month name. (read the input outside the function and use it as an argument)

```
Input : february
Output: 28
```

```
Input : december
Output: 31
```

15.8.8 Seasons

Write the code that will create a function with a parameter that contains the number given by the input and return the season based on the given month number.

If the number is from the interval <3,5> = "*spring*"

If the number is from the interval <6,8> = *"summer"*

If the number is from the interval <9,11> = "*autumn*"

If the number is from the interval <1,2> or 12 = "*winter*"

Else return wrong month. (read the input outside the function and use it as an argument)

```
Input : 2
Output: winter
```

```
Input : 15
```

```
Output: wrong month
```

⌨ 15.8.9 Aircraft range

Write the code that will create a function with a parameter that contains numbers given by the input and return the range of the aircraft from the given *velocity* and *flight length* in hours. (read the input outside the function and use it as an argument)

```
Input : 850 4.5
Output: 3825
```

```
Input : 900 10
Output: 9000
```

⌨ 15.8.10 Comparison of two numbers

Write the code that will create a function with a parameter that contains numbers given by the input and return the bigger number from two given numbers. Make sure that in case the numbers are equal the program outputs that information. (read the input outside the function and use it as an argument)

```
Input : 5 4
Output: Bigger is the number 5
```

```
Input : 10 10
Output: Both numbers are equal
```

15.9 Recursive function

🕮 15.9.1

Recursion is a method of programming or coding a problem, in which a function calls itself one or more times in its body. Usually, it is returning the return value of this function call. If a function definition satisfies the condition of recursion, we call this function a recursive function.

Termination condition: A recursive function has to fulfil an important condition to be used in a program: it has to terminate. A recursive function terminates, if with every recursive call the solution of the problem is downsized and moves towards a base case. A base case is a case, where the problem can be solved without further recursion. A recursion can end up in an infinite loop if the base case is not met in the calls.

Example:

```
4! = 4 * 3!
3! = 3 * 2!
2! = 2 * 1
```

Replacing the calculated values gives us the following expression:

```
4! = 4 * 3 * 2 * 1
```

Generally, we can say: Recursion in computer science is a method where the solution to a problem is based on solving smaller instances of the same problem.

📖 15.9.2

Now we come to implement the most typical recursion example: the factorial. It's as easy and elegant as the mathematical definition.

Example:

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

We can track how the function works:

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for  2  * factorial( 1 ):  2
intermediate result for  3  * factorial( 2 ):  6
intermediate result for  4  * factorial( 3 ):  24
intermediate result for  5  * factorial( 4 ):  120
120
```

Let's have a look at an iterative version of the factorial function.

```
def it_factorial(n):
    result = 1
    for i in range(2,n+1):
```

```
        result *= i
    return result
```

It is common practice to extend the factorial function for 0 as an argument. It makes sense to define 0! to be 1 because there is exactly one permutation of zero objects, i.e. if nothing is to permute, "everything" is left in place. Another reason is that the number of ways to choose *n* elements among a set of *n* is calculated as *n!* divided by the product of *n!* and 0!.

All we have to do to implement this is to change the condition of the if statement:

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

15.9.3

Fill in the code so the function will return the *power* of *n*-th of the given number:

```
def powers(x,n):
  if(n==1):
    return _____
  else:
    return x*_____
```

15.9.4

The *Fibonacci numbers* are the numbers of the following sequence of integer values:

```
0,1,1,2,3,5,8,13,21,34,55,89,...
```

The Fibonacci numbers are defined by:

$F_n = F_{n-1} + F_{n-2}$ where $F_0 = 0$ and $F_1 = 1$.

The *Fibonacci sequence* is named after the mathematician *Leonardo of Pisa*, who is better known as *Fibonacci*. In his book *"Liber Abaci"* (publishes 1202) he introduced the sequence as an exercise dealing with bunnies. His sequence of the *Fibonacci numbers* begins with $F_1 = 1$, while in modern mathematics the sequence starts with $F_0 = 0$. But this has no effect on the other members of the sequence.

The *Fibonacci numbers* are easy to write as a *Python* function. It's more or less a one to one mapping from the mathematical definition:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

15.9.5

Fill in the code so the function will return the *n*-th element of the Fibonacci sequence (the sequence is following: 1, 1, 2, 3, 5, 8, 13, ...):

```
def fibo(n):
  if(n==1):
    return _____
  elif(n==2):
    return _____
  else:
    return fibo(_____)+_____(n-2)
```

15.9.6 Sum of numbers

Write a code that will create a recursive function with a parameter that contains a number given by the input and return the sum of numbers from 1 to the given number. (read the input outside the function and use it as an argument)

```
Input : 5
Output: 15
```

```
Input : 10
Output: 55
```

15.9.7 Product of numbers

Write a code that will create a recursive function with a parameter that contains a number given by the input and return the product of numbers from 1 to the given number. (read the input outside the function and use it as an argument)

```
Input : 5
```

```
Output: 120
```

```
Input : 10
Output: 55
```

15.9.8 Fibonachi sequence

Write a code that will create a recursive function with a parameter that contains a number given by the input and return the Fibonacci sequence element. (read the input outside the function and use it as an argument). Fibonacci sequence is following: 0, 1, 1, 2, 3, 5, 8, 13, 21, ...

```
Input : 5
Output: 5
```

```
Input : 10
Output: 55
```

15.9.9 Sequence I.

Write a code that will create a recursive function with a parameter that contains a number given by the input and return the sum of the positive integers of *n+(n-2)+(n-4)... (until n-x =< 0)*. (read the input outside the function and use it as an argument).

```
Input : 10
Output: 30
```

```
Input : 5
Output: 9
```

15.9.10 Sequence II.

Write a code that will create a recursive function with a parameter that contains a number given by the input and return the sum of the positive integers of *(n-1)+(n-2)+(n-3)...* where $a_0=2$, $a_1=4$ and $a_2=7$. (read the input outside the function and use it as an argument).

```
Input : 10
Output: 927
```

```
Input : 5
Output: 44
```

⌨ 15.9.11 Prime number

Write a code that will create a recursive function with a parameter that contains a number given by the input and return information whether the given number is a prime number. (read the input outside the function and use it as an argument). Use a default argument for the base divisor to check as number 2.

```
Input : 10
Output: no
```

```
Input : 5
Output: yes
```

15.10 Docstrings

🕮 15.10.1

A documentation string, ***"docstring"***, is descriptive text that helps to better understand the functionality of a class, module, function, or method.

The document and the commentary are similar, but there are a few differences:

- comments are used to describe how the program works, the ***document*** describes ***what the program should do***
- the ***document*** is an improved, more logical and more useful version of the comments
- comments are mainly used to explain unusual parts of the code and can be useful for fixing bugs and tasks that need to be done

The ***document (docstring)*** should look like this:

- begins with a capital letter and ends with a period
- the first line is a short description
- if the document contains multiple lines, the second line is blank and visually separates the summary from the rest of the description
- the next lines use one or more paragraphs to describe a common call to an object, its side effects, and so on.

📝 15.10.2

What is the meaning of "docstring"?

- It is used to explain in brief, what a function does.
- It is used to explain in brief, what is Python.
- It is used to describe the programming task.
- It is used to heal the function of errors.

📖 15.10.3

Documents (docstring) are written using triple quotes *"""*This function greets the user*"""*, directly below the class, method, or function declaration. All functions should include a docstring that describes their functionality.

Documents are accessible from the ***(__doc__)*** attribute for any of the *Python* objects, as well as through the built-in ***help()*** function, which provides all the necessary information.

Example of using the ***docstring*** attribute and the ***help()*** function

```
def say_hello():
    """Function that returns hello to the user."""
    return "Hello"

print("Usage of __doc__:",say_hello.__doc__)

>> Usage of atribute__doc__: Function that returns hello to
the user.

print("Using help:")
help(say_hello)

>> Using of function help():
>> Help on function say_hello in module __main__:

say_hello()
    Function that returns hello to the user.
```

📝 15.10.4

What keyword or function do you have to use to obtain the docstring attribute of a function with the name *my_function*?

- docstring
- --doc--
- help(my_function)
- sos(my_function)
- __doc__
- my_function(help)

15.10.5

Fill in the code to list the docstring attribute.

```
def say_hi(name):
    """Function that prints a greeting with the name."""
    print("Good Morning ",name)

print("Usage of __doc__:",_____)

>> Usage of __doc__: _____
```

15.10.6

Fill in the code to list the information about the *docstring* attribute without the use of *"__doc__"* attribute.

```
def say_hi(name):
    """Function that prints a greeting with the name."""
    print("Good Morning ",name)


_____

>> _____ on function say_hi in module __main__:

say_hi(name)
    _____
```

15.10.7

One-line documentation

This is a description that is on just one line. The closing quotation marks are on the same line as the opening quotation marks. It is used in unambiguous cases, with simple functions.

```
def diff(a,b):
   """Returns the difference of the arg1 and arg2."""
   return a-b

print(diff.__doc__)

>> Returns the difference of the arg1 and arg2.
```

📖 15.10.8

Multiline documentation

Multi-line documentation consists of a summary line, just like single-line documentation, followed by a blank line for better clarity and finally a more detailed description of the function, method, etc. The summary line is on the same line as the opening quotes or on a new line.

Example of using multiline documentation

```
def my_function(arg1):
    """
    Summary line.

    Extended description of function.

    Parameters:
    arg1 (int): Description of arg1

    Returns:
    int: Description of return value

    """

    return arg1

print(my_function.__doc__)
```

Corresponding Output:

```
    Summary line.
    Extended description of function.
    Parameters:
    arg1 (int): Description of arg1

    Returns:
    int: Description of return value
```

📖 15.10.9

Indentation in documents

The entire *document (docstring)* is indented in the same way as the quotation marks on the first line. Documentation processing tools remove the same amount of

indentation from the second and subsequent lines of documentation, which is equal to the minimum indent of all non-empty lines after the first line. Any indentation in the first line of the *docstring* (ie up to the first newline) is negligible and will be removed. The relative indentation of later lines in the documentation is maintained.

15.10.10

What docstring types can be used for functions?

- one-line
- on-line
- multi-line
- empty-line
- tree-line

Namespaces

Chapter 16

16.1 Namespaces

📖 16.1.1

To better understand how local variables really work, you need to understand what they are and how a ***namespace*** works.

Namespaces are collections of identifiers that belong to a specific module, resp. function. Each module has its own ***namespace***, which means that two different modules can contain identifiers with the same name. Their distinction is solved by an entry, within which the module to which the given identifier belongs is unambiguously determined.

In short, ***namespace*** ensures that all names in the program are unique and can be used without conflicts.

In *Python*, all identifiers are one of three types:

- ***local namespace*** - valid within a function, created when the function is called and lasts only until the function returns a value
- ***global namespace*** - valid within the module (file), is created when the module is imported into the project and lasts until the script ends
- ***built-in namespace*** - identifiers predefined in Python, includes built-in functions and built-in exception names

When evaluating which identifier to use, local identifiers take precedence over global and global ones over built-ins.

📝 16.1.2

What does *namespace* mean?

- It is a system to secure that all the names in the program can be used without conflict and are unique.
- It is a system to secure that most of the names in the program can be used without conflict and are unique.
- It is a special type of function.
- It is a system to secure that the names in the program can be duplicate.

📖 16.1.3

Namespaces help uniquely identify all the ***names*** inside a program. However, this doesn't imply that we can use a variable name anywhere we want. A name also has

a scope that defines the parts of the program where you could use that name without using any prefix. Just like namespaces, there are also multiple scopes in a program. Here is a list of some scopes that can exist during the execution of a program.

- ***A local scope***, which is the innermost scope that contains a list of local names available in the current function.
- ***Scope of all the enclosing functions***. The search for a name starts from the nearest enclosing scope and moves outwards.
- ***A module-level scope*** that contains all the global names from the current module.
- ***The outermost scope*** contains a list of all the built-in names. This scope is searched last to find the name that you referenced.

In the next parts of the lesson, we will use a built-in Python function ***dir()*** that returns a list of names in the current local scope.

📝 16.1.4

Choose which of the following examples represent the namespace.

- Local namespace
- Focusal namespace
- Built-out namespace
- Global namespace
- Space namespace
- Built-in namespace

🕮 16.1.5

The search for a given name starts from the innermost function and then moves higher and higher until the program can map that name to an object. When no such name is found in any of the namespaces, the program raises a ***NameError*** exception.

Let's see what will happen if we type ***dir()*** into IDLE or any *Python* IDE:

```
dir()
```

The output will be the following:

```
['__builtins__','__doc__','__loader__','__name__','__package__
','__spec__']
```

All these names listed by ***dir()*** are available in every Python program.

Let's see the output of the ***dir()*** function when we create a variable inside a function:

```
a = 1
dir()
```

The output will be the following:

```
['__builtins__','__doc__','__loader__','__name__','__package__
','__spec__','a']
```

```
def func():
  b = 2
  print(dir())

func()
```

The output will be following:

```
['b']
```

And if we call the ***dir()*** function again we get this Output:

```
['__builtins__','__doc__','__loader__','__name__','__package__
','__spec__','a','func']
```

The ***dir()*** function only outputs the list of names inside the current scope. That's why inside the scope of **func()**, there is only one name called **b**. Calling ***dir()*** after defining **func()** adds it to the list of names available in the global namespace.

16.1.6

What will be the output of the following code?

```
def summa(x,y):
    print(dir())
    return x+y

res = summa(4,2)
```

- ['x', 'y']
- 6

- ['__builtins__','__doc__','__loader__','__name__','__package__','__spec__','res','summa']
- ['__builtins__','__doc__','__loader__','__name__','__package__','__spec__','x','y','res','summa']

📖 16.1.7

List of names inside some nested functions

The code in this block continues from the previous block.

```
def main_func():
    a = 1
    def nested_func():
        b = 2
        print(dir(), ': names in nested_func')
    c = 3
    nested_func()
    print(dir(), ': names in main_func')

main_func()
```

This will result in the following Output:

```
['b'] : names in nested_func
['a', 'c', 'nested_func'] : names in main_func
```

The above-defined function defines two variables and a function inside the scope of ***main_func()***. Inside ***nested_func()***, the ***dir()*** function only prints the name ***b***. This is correct as ***b*** is the only variable defined there.

Unless explicitly specified by using the ***global*** keyword, reassigning a global name inside a local *namespace* creates a new local variable with the same name.

📝 16.1.8

Select the correct outputs of the following code:

```
a = 1
b = 2

def main_func():
    global a
```

```
    a = 3
    b = 4
    def nested_func():
        global a
        a = 5
        b = 6
        print('a inside nested_func :', a)
        print('b inside nested_func :', b)
    nested_func()
    print('a inside main_func :', a)
    print('b inside main_func :', b)

main_func()
print('a outside all functions :', a)
print('b outside all functions :', b)
```

- a inside nested_func : 5
- a outside all functions : 1
- a inside main_func : 3
- a inside main_func : 5
- a outside all functions : 5
- a inside nested_func : 3
- b outside all functions : 2
- b outside all functions : 6
- b inside main_func : 3
- b inside main_func : 4
- b inside nested_func : 6

Strings

Chapter 17

17.1 Introduction to Strings

📖 17.1.1

Let's review what we learned about string in the previous course. A string is a sequence of characters. A character is simply a symbol. For example, the English language has 26 characters. Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0's and 1's. This conversion of a character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encoding used.

In Python, a string is a sequence of Unicode character. Unicode was introduced to include every character in all languages and bring uniformity in encoding.

Strings can be created by enclosing characters inside a single quote or double quotes. Even triple quotes can be used in Python but are generally used to represent multiline strings and docstrings.

```
my_name = "Peter"
# or it can be written as 'Peter', '''Peter'''
print(my_name)
```

Triple quotes string can even extend multiple lines:

```
my_string = """Hello
                world"""
```

📝 17.1.2

Fill in the code with correct string quotes:

```
my_string = _____This sentence
         will be longer than a normal
            sentence and you want to print it
             with this formating._____
print(my_string)
```

📖 17.1.3

Like many other popular programming languages, strings in Python are arrays of bytes representing Unicode characters. However, Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used **to access elements** of the string.

```
my_name = "Peter"
print(my_name[1])
```

The output will be following:

```
>> e
```

We can access individual characters using indexing and a range of characters using slicing. **Index** starts from 0. Trying to access a character out of the index range will raise an ***IndexError***. The index must be an integer. We can't use float or other types, this will result in ***TypeError***.

17.1.4

What will be the output of the following code?

```
my_str = "Informatics"
print(my_str[5])
```

- l
- n
- f
- o
- r
- m
- a
- t
- i
- c
- s

17.1.5

Python allows **negative indexing** for its sequences. The index of **-1** refers to the **last item**, -2 to the second last item and so on. We can access a range of items in a string by using the **slicing operator** (colon). Slicing can be best visualized by considering the index to be between the elements as shown below. If we want to access a range, we need the index that will slice the portion from the string.

```
print(my_name[-1])
>> r
print(my_name[1:5])
>> eter
```

```
print(my_name[2:-2])
>> t
```

17.1.6

What will be the output of the following code?

```
my_str = "Informatics"
print(my_str[-6])
```

- l
- n
- f
- o
- r
- m
- a
- t
- i
- c
- s

17.1.7

Fill in the code to get the following Output:

```
my_str = "Informatics"
print(_____)

>> format
```

17.1.8

Strings are **immutable**. This means that elements of a string cannot be changed once it has been assigned. We can simply reassign different strings to the same name.

```
my_str = "Informatics"
my_str[6] = '4'

>> TypeError: 'str' object does not support item assignment
```

We **cannot delete or remove** characters from a string. But deleting the string entirely is possible using the keyword ***del***.

```
my_str = "Informatics"
del my_str[6]

>> TypeError: 'str' object doesn't support item deletion

del my_str
my_str
>> NameError: name 'my_str' is not defined
```

17.2 Strings (programs)

⌨ 17.2.1 Second letter

Write the code that will use a function with a parameter that contains a string given by the input and returns the second letter of the given string (read the input outside the function and use it as an argument).

```
Input : Hello World
Output: e
```

```
Input : PYTHON
Output: Y
```

⌨ 17.2.2 Compare two strings

Write the code that will use a function with a parameter that contains strings given by the input and returns the information whether the two given strings are equal (read the input outside the function and use it as an argument).

```
Input : tree
tree
Output: yes
```

```
Input : PYTHON
python
Output: no
```

17.2.3 Palindrome

Write the code that will use a function with a parameter that contains a string given by the input and returns the information whether the given string is a palindrome (read the input outside the function and use it as an argument).

```
Input : tree
Output: no
```

```
Input : toot
Output: yes
```

17.2.4 Incorrect input

Write the code that will use a function with a parameter that contains numbers given by the input and returns the sum of two given numbers and will deal with an incorrect input (read the input outside the function and use it as an argument).

```
Input : 5 7
Output: 12
```

```
Input : 4 a
Output: incorrect input
```

17.2.5 Number of digits

Write the code that will use a function with a parameter that contains numbers given by the input and returns return how many times is a digit in a given number. (read the input outside the function and use it as an argument).

```
Input : 5 154265
Output: 2
```

```
Input : 1 11154231
Output: 4
```

17.2.6 Maximum digit

Write the code that will use a function with a parameter that contains numbers given by the input and returns the maximum digit from a given number. (read the input outside the function and use it as an argument).

```
Input : 1248556
```

```
Output: 8
```

```
Input : 111111
Output: 1
```

17.2.7 Minimum digit

Write a code that will use a function with a parameter that contains numbers given by the input and returns the minimum digit from a given number. (read the input outside the function and use it as an argument).

```
Input : 1248556
Output: 1
```

```
Input : 1101111
Output: 0
```

17.2.8 Mirror image

Write the code that will use a function with a parameter that contains a number given by the input and returns the mirror image of a given number. (read the input outside the function and use it as an argument).

```
Input : 1234
Output: 4321
```

```
Input : 110011
Output: 110011
```

17.2.9 Without first and last letter

Write the code that will use a function with a parameter that contains a string given by the input and returns the string without the first and last letter. (read the input outside the function and use it as an argument).

```
Input : alphabet
Output: lphabe
```

```
Input : 110011
Output: 1001
```

17.3 String operations

📖 17.3.1

Joining two or more strings into a single one is called **concatenation**. The **+** operator does this in *Python*. Simply writing two string literals together also concatenates them. The * operator can be used to **repeat the string** a given number of times.

```
name = "John"
surname = "Silver"
print('name + surname =',name+surname)
print('name * 5 =',name*5)
```

The output will be following:

```
name + surname = JohnSilver
name * 5 = JohnJohnJohnJohnJohn
```

📝 17.3.2

Fill in the code so to get the following Output:

```
str1 = "I like "
str2 = "Python"
print(_____)
print(_____)
```

The output will be following:

```
I like Python
PythonPythonPython
```

📖 17.3.3

If we want to **iterate** through a string we can use the **for** loop. Using the loop we can browse the string by characters or letters. This can be used also to browse a number to for e.g. calculate the digit sum.

```
count = 0
for let in 'informatics':
   if(let == 'i'):
       count += 1
print(count,' i letters found')
```

The code will output the following result:

```
2 i letters found
```

17.3.4

Fill in the code to calculate the digit sum of the given number:

```
num = "12345"
digsum = _____
for n in _____:
    digsum_____int(n)
print(_____)
```

```
>> 15
```

17.3.5

To check if a certain phrase or character is **present in a string**, we can use the keywords ***in*** or ***not in***. This can be used to browse through the given string without the use of a loop. It can be used only to compare two strings or better to say that if a substring is part of the browsed string.

```
txt = "Have a great day."
is_in = "av" in txt
print(is_in)
```

The return of the expression using the keyword ***in*** is a boolean. This means that the previous code will generate the following Output:

```
>> True
```

The ***not in*** keywords work the other way around. If the substring is contained in the string then it will return *False*.

```
txt = "Have a great day."
is_in = "av" not in txt
print(is_in)
```

```
>> False
```

17.3.6

Fill in the code so that you will get the following Output:

```
txt = "Informatics is cool"
res = "oo" _____ txt
print(res)
```

```
>> False
```

17.3.7

Python in contrast to other programming languages, e.g. Java, **does not support combinations** of string and numbers the **following way**:

```
age = 18
txt = "I'm "+age+" years old."
print(txt)
```

This will result in an error message:

```
>> TypeError: must be str, not int
```

In Python, we can **combine** strings and numbers using the ***format()*** method. The method takes the passed arguments, formats them and places them in the string where the placeholders ***{}*** are:

```
age = 18
txt = "I'm {} years old."
print(txt.format(age))
```

The output will be following:

```
>> I'm 18 years old.
```

17.3.8

The format() method takes an unlimited number of arguments, and are placed into the respective placeholders:

```
age = 26
address = 49
salary = 876
```

```
txt = "My name is John, I'm {} years old, I work at St. John's
{} and earn {} euros."
print(txt.format(age, address, salary))
```

The result will be following:

```
>> My name is John, I'm 26 years old, I work at St. John's 49
and earn 876 euro.
```

You can use index numbers ***{0}*** to be sure the arguments are placed in the correct placeholders:

```
salary = 876
age = 26
address = 49
txt = "My name is John, I'm {2} years old, I work at St.
John's {0} and earn {1} euros."
print(txt.format(address, salary, age))
```

The result will be following:

```
>> My name is John, I'm 26 years old, I work at St. John's 49
and earn 876 euro.
```

We can even format strings like the old ***sprintf()*** style used in the C programming language. We use the **%** operator to accomplish this.

```
age = 32
print("I am %d years old" %age)
```

17.3.9

Fill in the code with the correct index of the variables in the text format method:

```
quant = 2
item = 653
price = 4.99
order = "I want to pay _____ euros for _____ pieces of item
_____."
print(order._____(quant, price, item))
```

📖 17.3.10

To insert characters that are illegal in a string, use an **escape character**. An escape character is a backslash **** followed by the character you want to insert. An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

```
txt = "These so-called "experts" do not know anything."
```

This will result in an invalid syntax. To fix this issue we will input the backslash escape character:

```
txt = "These so-called \"experts\" do not know anything."
```

Other escape characters used in Python are following:

- single quotes: **\'**
- backslash: ****
- new line: **\n**
- carriage return: **\r**
- tab: **\t**
- backspace: **\b**
- form feed: **\f**

17.4 String operations (programs)

⌨ 17.4.1 Contains a string

Write the code that will use a function with a parameter that contains strings given by the input and returns the count of how many times the given string is contained in another given string. (read the input outside the function and use it as an argument).

```
Input : ab babababa
Output: 3
```

```
Input : dog dosdogsddosg
Output: 1
```

⌨ 17.4.2 String on the beginning or the end

Write the code that will use a function with a parameter that contains strings given by the input and returns whether the string is in the beginning or at the end of a given string. (read the input outside the function and use it as an argument).

```
Input : ab bababab
Output: end
```

```
Input : dog dogdosgsddosg
Output: beginning
```

```
Input : so dogdosgsddosg
Output: neither
```

⌨ 17.4.3 Replacing the string

Write the code that will use a function with a parameter that contains strings given by the input and returns a replaced given string with a given substring. (read the input outside the function and use it as an argument).

```
Input : babababa a b
Output: bbbbbbbb
```

```
Input : hello e 3
Output: h3llo
```

⌨ 17.4.4 The most common character

Write the code that will use a function with a parameter that contains a string given by the input and returns the most common character in the given string. (read the input outside the function and use it as an argument).

```
Input : popocatepetl
Output: p
```

```
Input : bratislava
Output: a
```

⌨ 17.4.5 Zero

Write the code that will use a function with a parameter that contains a string given by the input and returns whether the given string contains a zero. (read the input outside the function and use it as an argument).

```
Input : popocatepetl
Output: no zero
```

```
Input : 100
```

```
Output: has zero
```

⌨ 17.4.6 Odd numbers

Write the code that will use a function with a parameter that contains a string given by the input and returns the count of odd numbers in a given string and whether it contains a zero. (read the input outside the function and use it as an argument).

```
Input : 24168
Output: odd numbers: 1
        no zero
```

```
Input : 100
Output: odd numbers: 1
        has zero
```

17.5 String functions

🕮 17.5.1

Various built-in functions work with strings. Some of the commonly used ones are ***enumerate()*** and ***len()***. The ***enumerate()*** function returns an enumerate object. It contains the index and value of all the items in the string as pairs. This can be useful for iteration.

Similarly, ***len()*** returns the length (number of characters) of the string.

```
txt = "Informatics"
print(len(txt))
```

This will result in the following Output:

```
>> 11
```

📝 17.5.2

What will be the output of the following code:

```
txt = "My name is John "
print(len(txt))
```

- 16
- 12

- 15
- 10
- 0

📖 17.5.3

There are some functions that work with the case of the text in a string. Like we can change the lowercase to uppercase and vice versa. Also, we can capitalize the first letter in the string.

It does not matter what the letters are the ***capitalize()*** function will convert the first character to an uppercase letter and other characters will be lowercased. The string will be not changed in any other way.

```
txt = "hello World"
print(txt.capitalize())
```

```
>> Hello world
```

The string ***lower()*** method converts all uppercase characters in a string into lowercase characters and returns it. On the other hand, the string ***upper()*** method converts all lowercase characters in a string into uppercase characters and returns them.

```
txt = "Hello World"
print(txt.upper())
print(txt.lower())
```

```
>> HELLO WORLD
>> hello world
```

📝 17.5.4

What will be the output of the following code:

```
txt1 = "I AM CoDING PYTHON!"
txt2 = "I am CoDiNg Python!"

if(txt1.lower() == txt2.lower()):
    print("The strings are same.")
else:
    print("The strings are not same.")
```

17.5.5

What will be the output of the following code:

```
txt = "i am programming Python."
print(txt.capitalize())

txt2 = "* is an operator."
print(txt.capitalize())
```

- I am programming python.
- i am programming Python.
- I AM PROGRAMMING PYTHON.
- * is an operator.
- * Is an operator.
- * IS AN OPERATOR.
- i am programming python.

17.5.6

The other useful functions for string are the following. If you want to convert a string to a number then it is better first to check whether the string contains only numerical characters otherwise the conversion would lead to an error. For this purpose can be used the functions ***isdigit()*** or ***isdecimal()***.

The ***isdecimal()*** method returns *True* if all characters in a string are decimal characters. If not, it returns *False*.

```
txt = "1234"
print(txt.isdecimal())
>> True
txt = "H3110"
print(txt.isdecimal())
>> False
```

The ***isdigit()*** method returns *True* if all characters in a string are digits. If not, it returns *False*.

```
txt = "1234"
print(txt.isdigit())
>> True
txt = "H3110"
print(txt.isdigit())
>> False
```

17.5.7

Fill in the code so that you check whether the given string is a number.

```
txt1 = 'INF0RM4T1CS'
print(txt1._____)
>> False
txt2 = '3167'
print(_____)
>> _____
```

17.5.8

In the previous chapter, we dealt with the format of the strings and their concatenation with another string. This can be done also using a function ***join()*** and we can use another function ***split()*** to split a string into two separate strings.

The ***join()*** is a string function that returns a string concatenated with the elements of an iterable. It provides a flexible way to concatenate string. It concatenates each element of an iterable (such as list, string and tuple) to the string and returns the concatenated string.

The ***split()*** function breaks up a string at the specified separator and returns a list of strings. About the list, you will learn in the intermediate course. The ***split()*** function takes a maximum of 2 parameters:

- **separator** (optional): the string splits at the specified separator. If the separator is not specified, any whitespace (space, newline etc.) string is a separator.
- **maxsplit** (optional): the maxsplit defines the maximum number of splits. The default value of maxsplit is -1, meaning, no limit on the number of splits.

```
txt= 'I like Python' # splits at space
print(txt.split())
>> ['I', 'like', 'Python']

buy = 'Bread, Cereal, Ham, Cheese' # splits at ','
print(buy.split(', '))
>> ['Bread', 'Cereal', 'Ham', 'Cheese']
```

17.5.9

Fill in the code to generate the following Output:

```
keywords = 'java;python;pascal;c++;php'
```

```
print(keywords._____('_____'))
```

```
>> ['java', 'python', 'pascal', 'c++', 'php']
```

🕮 17.5.10

The strings can be browsed using a for loop that we showed in the previous chapters. But we can also use functions to find whether a substring is contained inside a string and we can even replace a part of the string with another one. For this purpose, we can use the function ***find()*** and ***replace()***.

The ***find()*** function returns the index of the first occurrence of the substring (if found). If not found, it returns -1. The find() method takes a maximum of three parameters:

- **sub:** It's the substring to be searched in the str string
- **start** and **end** (optional): substring is searched within *str[start:end]*

```
txt = 'I like Python, I like Java, I like PHP'

rsl = txt.find('I like')
print("Substring 'I like':", rsl)

>> Substring 'I like': 0

rsl = txt.find('pascal')
print("Substring 'pascal':", rsl)

>> Substring 'pascal': -1
```

The ***replace()*** function returns a copy of the string where all occurrences of a substring are replaced with another substring. The ***replace()*** function can take a maximum of 3 parameters:

- **old:** old substring you want to replace
- **new:** new substring which would replace the old substring
- **count** (optional): the number of times you want to replace the old substring with the new substring

If *count* is not specified, ***replace()*** function replaces all occurrences of the old substring with the new substring. The ***replace()*** function returns a copy of the string where the old substring is replaced with the new substring. The original string is unchanged. If the old substring is not found, it returns the copy of the original string.

```
txt = 'I like Python, I like Java, I like PHP'
```

```
print(txt.replace('PHP', 'Pascal'))

>> I like Python, I like Java, I like Pascal

txt = 'I like Python, I like Java, I like PHP'
print(txt.replace('like', 'love', 2))

>> I love Python, I love Java, I like PHP
```

17.5.11

Fill in the code to get the following result:

```
if  (quote.find('be,') != -1):
  print("Contains substring 'be,'")
else:
  print("Doesn't contain substring")
```

```
>>
```

17.5.12

Fill in the code to generate the following Output:

```
txt = 'Exam results: John got B, Marry got B, Peter got B,
Barry got B'
print(txt._____('B', 'A', 3))
```

```
>> Exam results: John got _____, Marry got _____, Peter got
_____, Barry got _____
```

17.5.13

At the most basic level, computers store all information as numbers. To ***represent*** character data, a translation scheme is used which maps each character to its representative number. The simplest scheme in common use is called ***ASCII***. It covers the common Latin characters you are probably most accustomed to working with.

The simplified coding ***ASCII*** table contains 255 base symbols (despite that nowadays are alphabets coded using ***Unicode/UTF8***). The first 32 symbols are controlling but the others are used often. The characters 'a' and 'A' are not the same (are not equal).

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
32	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
48		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
64	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
80	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
96	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
112	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
128	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Unicode is an ambitious standard that attempts to provide a numeric code for every possible character, in every possible language, on every possible platform. Python 3 supports *Unicode* extensively, including allowing *Unicode* characters within strings.

As long as you stay in the domain of the common characters, there is little practical difference between ASCII and Unicode. The ***chr()*** function returns a character (a string) from an integer (represents Unicode code point of the character). On the other hand, the ***ord()*** function returns an integer representing the Unicode character.

🕮 17.5.14

The ***chr()*** function returns a character (a string) from an integer value (represents Unicode code point of the character). The ***chr()*** function takes a single parameter, an integer value. The ***chr()*** returns a character (a string) whose Unicode code point is the integer value. If the integer value is outside the range, ***ValueError*** will be raised.

```
print(chr(68))
print(chr(87))
print(chr(103))
```

```
>> D
>> W
>> g
```

17.5.15

Fill in the corresponding char integer values to get the following Output:

```
print(_____)
```

```
>> H E L L O
```

17.5.16

The ***ord()*** function returns an integer representing the Unicode character. The ***ord()*** function takes a single parameter: **ch** - a Unicode character. By the way, the ***ord()*** function is the inverse of the Python ***chr()*** function.

```
print(ord('D'))
print(ord('a'))
print(ord('7'))
```

```
>> 68
>> 97
>> 55
```

17.5.17

Fill in the corresponding Unicode character values to get the following Output:

```
print(_____)
```

```
>> 35 112 121 116 104 111 110
```

17.6 String functions (programs)

17.6.1 String length

Write the code that will use a function with a parameter that contains a string given by the input and returns the length of the given string. (read the input outside the function and use it as an argument).

```
Input : alphabet
Output: 8
```

```
Input : 110011
Output: 6
```

⌨ 17.6.2 Lowercase

Write a code that will use a function with a parameter that contains a string given by the input and returns the given string in lowercase. (read the input outside the function and use it as an argument).

```
Input : PYTHON
Output: python
```

```
Input : 100
Output: 100
```

⌨ 17.6.3 Uppercase

Write the code that will use a function with a parameter that contains a string given by the input and returns the given string in uppercase. (read the input outside the function and use it as an argument).

```
Input : python
Output: PYTHON
```

```
Input : 100
Output: 100
```

⌨ 17.6.4 Number

Write the code that will use a function with a parameter that contains a string given by the input and returns information whether the given string is a number or not. (read the input outside the function and use it as an argument).

```
Input : pyth0n
Output: False
```

```
Input : 100
Output: True
```

⌨ 17.6.5 First occurence

Write the code that will use a function with a parameter that contains strings given by the input and returns information of the first occurrence of a given substring in the given string. (read the input outside the function and use it as an argument).

```
Input : I like Python
        like
Output: 2
```

```
Input : It is a trap
        a
Output: 6
```

⌨ 17.6.6 Characters I.

Write the code that will use a function with a parameter that contains a string given by the input and returns whether the given string is a letter, number or character. (read the input outside the function and use it as an argument).

```
Input : a
Output: letter
```

```
Input : 4
Output: number
```

```
Input : !
Output: character
```

⌨ 17.6.7 Characters II.

Write the code that will use a function with a parameter that contains a string given by the input and returns whether the given string is a uppercase or lowercase letter. If it is not a letter write NaN. (read the input outside the function and use it as an argument).

```
Input : a
Output: lowercase
```

```
Input : 4
Output: NaN
```

```
Input : F
Output: uppercase
```

17.6.8 Encoding

Write the code that will use a function with a parameter that contains a string given by the input and returns the encoded given string so that it will move each letter by 3 digits in the alphabet. For example, Hello will be: H-K, e-h, l-o, l-o, o-r. Make sure that in case of the end of the aplhabet it will start from the beginning, for example z-c. (read the input outside the function and use it as an argument).

```
Input : Hello
Output: Khoor
```

```
Input : World
Output: Zruog
```

17.6.9 Vowels

Write the code that will use a function with a parameter that contains a string given by the input and returns the vowels from the given string (a, e, i, o, u, y). (read the input outside the function and use it as an argument).

```
Input : Hello
Output: eo
```

```
Input : World
Output: o
```

17.6.10 Vowels

Write the code that will use a function with a parameter that contains a string given by the input and returns the information whether the word contains the letter "*y*". If yes, it will return the count and substitute it with the letter "*i*". (read the input outside the function and use it as an argument).

```
Input : fyto
Output: 1 time
        fito
```

```
Input : bit
Output: 0 time
        bit
```

PRISCILLA

priscilla.fitped.eu