PYTHON'S HISTORY, PROFILE AND USE

PYTHON'S HISTORY

- Created by Guido van Rossum.
- Published officially for the first time in February 1991.
- On January 26, 1994, the first major version was released.
- Python 3.0 was released on December 8, 2008.
- Python 3.0 is currently the only officially updated and supported version of Python.

LANGUAGE PROFILE

- Simple and clear
- Sensitive to spaces
- High-level
- Interpreted
- Dynamically typed
- Object oriented
- A large standard library
- It is free

MAIN USE OF PYTHON

- Artificial intelligence and machine learning
- Creating websites
- Testing
- Task automation
- DevOps practice
- Data analysis

THE FIRST PROGRAM: HELLO, WORLD!

BEFORE WE START...

Make sure you have installed:

- Python version 3.7.
- PyCharm Community.

Installation instructions can be found in the appropriate document in the introduction module visible in Gitlab.

HELLO, WORLD!

```
    print("Hello, World!")
```

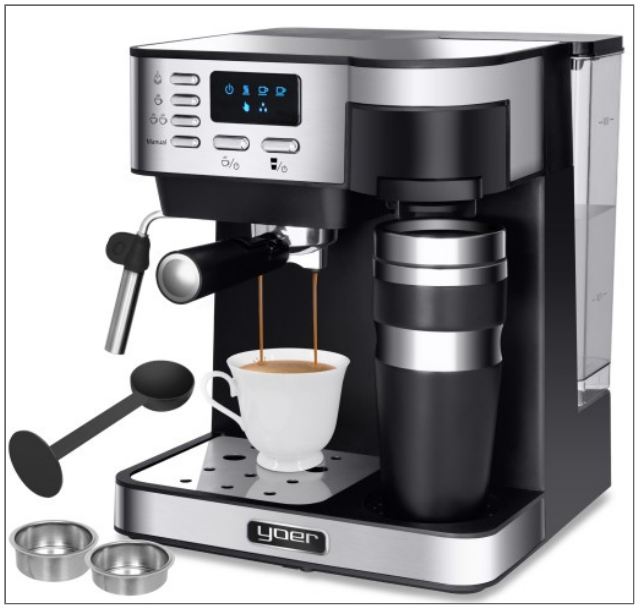
HOW THHIS PROGRAM WORKS

```
print("Hello, World!")
```

- The **Hello, World!** is displayed on the screen.
- The built-in function **print()** was used.
- The function has the parameter in brackets - it is what we want to write and display on the screen.

FUNCTION - THE EXPLANATION

For now, let's say a function is a certain **behavior**: something must be done depending on the given parameters. A coffee machine is a good example of such a function.
The right coffee type is prepared (**function**) depending on what the user chooses (**parameter**).



DATA TYPES

DATA TYPES IN PYTHON

Several data types are built into Python:

- **int** - integers.
- **float** - floating-point numbers (real numbers).
- **complex** - complex numbers.
- **str** and **bytes** - text sequences.
- **bool** - **boolean** data type: true / false.
- **NoneType** - special, **undefined** type of non-existent values.

DATA TYPES IN PYTHON: EXAMPLES

Data type	Sample values
int	0, 1, -3, 128, 4567654324567, 0b111, 0x18C
float	0.123, 256.2, -3.14, 1e10
complex	(1-2j), (30+15j), 3j
str	"text", "", "also!@#$%^text", 'single quotes'
bytes	b"bytes sequence", b'123456'
bool	True, False
NoneType	None

PRINTING VARIOUS TYPES OF DATA ON THE SCREEN

```
print("Text to print.")
print(-17)
print(123.4)
print(False)
print(None)
```

CHECKING THE DATA TYPE

In Python, we can check the data type by using the built-in **type()** function:

```
type("What is the data type of this data?")
```

CHECKING THE DATA TYPE - THE TYPE() FUNCTION

- The **type()** function takes the parameter the same way as the **print**() function.
- The **type()** returns a text value that is the type name of its parameter.
- We can write the data type by combining both functions:

```
    print(type("What is the data type of this
data?"))
    print(type(10.2))
    print(type(True))
```

VARIABLES AND OPERATORS

VARIABLES

Variables are containers for storing data. Think of variables as boxes where we can store something i.e. a value (implicitly).
A variable definition in Python:

```
number = 10
word = "text"
big_number = 99999.999
truth = True
```

VARIABLES

A variable:

- has a name,
- has a value,
- has its place in the computer's memory.

A VARIABLE NAMING CONVENTION

- You can change the value of a variable by assigning a new value.
- The variable name can only contain letters: a-z, A-Z, numbers 0-9 and the _ symbol.
- It is case sensitivite!
- The variable name cannot start with a digit!
- In Python, variables are named in the *snake case* style: each word is separated by an underscore (`very_big_number`) and in the CamelCase style: a new word starts with a capital letter (`VeryLongString`) - the latter is reserved for classes.

VARIABLE NAMING CONVENTION

Allowed solutions:

```
SUPER_VARIABLE = 1
bestVariableEver123 = "OK"
_my_name = "Alice"
another1_variable2 = 5.5
_____ = True
```

Not allowed:

```
123variable = 2
wrong$name%123 = False
!@AlmOStGoODone!@ = None
12345 = "bad_name"
VARIABLE-1 = 0
```

TYPES

Python allows you to define the type of a variable. This is optional and is only a suggestion.

```
number: int = 5
```

TYPES

The fact that Python types are only a suggestion makes the following code work without any problems:

```
number: int = 'i am a string'

print(type(number))  # prints <class 'str'>
```

PRINTING VARIABLES ON THE SCREEN

Variables store values so they can be used as function parameters.

```
number = 10
new_word = "new string"

print(number)
print(new_word)

print(type(number))
print(type(new_word))
```

A COMMENT IN PYTHON

```
    # Declares the variable and assign it to a
value
    number = 10
    print(number) # Prints 10
```

The # character starts the comment. Everything that follows it until the end of the line is ignored.

KEYWORDS

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

OPERATORS

Arithmetic operators/td>	+, -, *, **, /, //, %
Comparison operators	==, !=, <, >, <=, >=
Assignment operators	=, +=, -=, *=, **=, /=, //=, %=, &=, ^=, |=, <<=, >>=, |=
Identity operators	is, is not
Logical operators	and, or, not
Membership operators	in, not in
Bit operators	& AND, | OR, ^ XOR, ~ NOT, << left shift, >> right shift

ARITHMETIC OPERATORS

They are used for mathematical operations - addition, subtraction, etc.

```
# Arithmetic operators
print(1 + 2 + 5 - (2 * 2))
print(501.0 - 99.9999)
print(2 ** 3)
print(10.0 / 4.0)
print(10.0 // 4.0)
print(5 % 2)
```

```
# Text concatenation
name = "John"
greeting = "Hello, " + name
print(greeting)
joe = "Joe"
jane = "Jane"
print(joe + " and " + jane)
```

```
# Repetition of text
message = "Hi"
print(message * 2)
new_message = "Hi" * 5
print(new_message)
number = 3
message = message * number
print(message)
```

ERRORS IN THE PROGRAM - EXCEPTIONS

Python cannot be fooled. When we try to add a number and a string, it will return an error (also called the **exception**):

```
    print(1 + "a")

    Traceback (most recent call last):
      File "main.py", line 1, in <module>
        1 + "a"
    TypeError: unsupported operand type(s) for
+: 'int' and 'str'
```

EXAMPLES OF EXCEPTIONS

- TypeError
- ValueError
- NameError
- ZeroDivisionError
- SyntaxError

ASSIGNMENT OPERATORS

They are used to assign / set values to variables.

```
num = 3
num = num + 4
print(num)
num += 2
print(num)
num -= 1
print(num)
num = num * 3
print(num)
num /= 2
num **= 3
num = num // 2
print(num)
```

COMPARISON OPERATORS

They are used in operations where two values are compared.

```
john_1 = "John"
john_2 = "John"
print(john_1 == john_2)
print(1 != 1)
print(99 < 1.1)
print(99 > 1.1)
print(-32 >= -33)
print(123 <= 123)
```

LOGICAL OPERATORS

Operators used in Boolean algebra operations.

```
print(True or False)
print(False and False and True)
print(not False)
is_greater = 40 > 30
print(not is_greater)
```

MEMBERSHIP OPERATORS

They are used to test for assigning membership in a sequence, such as strings, lists, or tuples.

```
print("fox" not in "cow, dog, cat")
print("great" in "Python is great!!!")
```

COMPATIBILITY WITH UTF-8

ASCII

- It is a character encoding system.
- It has 7 bits.
- It can assign: latin letters, numbers, punctuation marks and other symbols to numbers from 0 to 127.
- An example: the letter "A" corresponds to the number 65.
- It was created in 1963.
- You cannot use this coding to write letters from, for instance, the Polish or Chinese alphabet.

UTF-8

- UTF-8 (**Unicode Transformation Format – 8-bit**) - a Unicode encoding system.
- It is the default character encoding system.
- It uses 1 to 4 bytes to encode a single character.
- It is ASCII compatible.
- Currently it encodes over one million characters.
- Character codes have the "U +" prefix.

UTF-8 AND PYTHON

- Python 3.X is encoded in UTF-8 by default.
- This means that diacritics can be used in strings.
- It is also possible to use e.g. Polish characters in the names of variables, functions or classes, but this is not recommended (people of other nationalities may participate in the project).

UTF-8 AND PYTHON - AN EXAMPLE

This is fine:

```
    text_en = "Hi, young programmers!"
    text_fr = "Voiÿ ambiguë d'un cœur qui, au
zéphĀr, préfère les jattes de kiwis."
    text_cn = "你好, 世界"
    print(text_pl)
    print(text_fr)
    print(text_cn)
```

This is not recommended though it works:

```
    整数 = 7
    turtle = "turtle"
    vérité = True
    print(整数)
    print(tortoise)
    print(vérité)
```

STRING FORMATTING

PRINTING STRINGS

There are four ways to print data:

1. Without any formatting.
2. The printf formatting from the C language - an older solution.
3. The str.format() formatting - a newer solution.
4. The f-string interpolation - the latest solution.

THE PRINT() FUNCTION

We have already talked about the print() function. In addition to what we already know, it offers some interesting solutions. It can:

- Display several values at once.
- Define the separator between the values (parameters) given to it.
- Add a string after the last value.

THE PRINT() FUNCTION - MULTIPLE VALUES AT ONCE

```
# Displaying multiple strings at once
print("What", "a", "beautiful", "day", ".")
print("1", "2", 3, 4, 5)
fruit = "orange"
print("apple", "banana", fruit)
```

THE PRINT() FUNCTION - SEPARATOR

```
    # Displaying multiple strings with a
separator simultaneously
    print("What", "a", "beautiful", "day", ".",
sep="-")
    print("1", "2", 3, 4, 5, sep=" < ")
    fruit = "orange"
    print("apple", "banana", fruit, sep=" + ")
```

THE PRINT() FUNCTION - LAST STRING

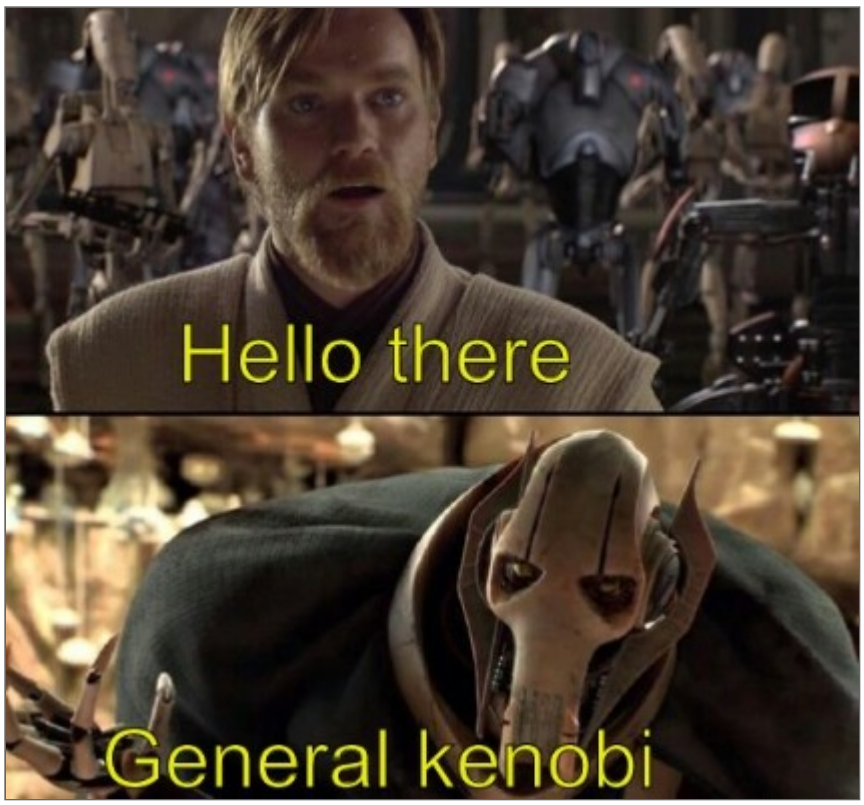
```
    # Displaying multiple strings
simultaneously with a separator and final
string
    print("What", "for", "beautiful", "day",
".", sep="-", end="! \n")
    print("1", "2", 3, 4, 5, sep="<", end="<...
\n")
    fruit = "orange"
    print("apple", "banana", fruit, sep="+",
end="= yummy \n")
```

By default, the **end** parameter uses the newline character ("\n").

FORMATTING STRINGS IN THE PRINTF STYLE

The printf style uses the % sign. It substitutes values in the order defined by the programmer.

```
# Format and display (older solution)
title = "General"
name = "Kenobi"
print("Hello there, %s %s" % (title, name))
```



FORMATTING STRINGS IN THE STR.FORMAT() STYLE

- It gets rid of the % character.
- Formatting is done by calling the **format** () function on the string.

```
    # Format and display (newer solution)
    title = "General"
    name = "Kenobi"
    print("Hello there, {} {}".format(title,
name))
```

FORMATTING STRINGS IN THE STR.FORMAT() STYLE

By using the **format()** function, you can add values to the string in any order.

```
    # Format and display (most recent solution)
    title = "General"
    name = "Kenobi"
    print("Hello there, {name}
{title}".format(name=name, title=title))
```

```
    # Format and display (most recent solution)
    title = "General"
    name = "Kenobi"
    print("Hello there, {1} {0}".format(title,
name))
```

INTERPOLATION OF F-STRING

- Available from Python 3.6.
- Gets rid of the format() function.
- Strings must be preceded by the letter f: f"...".

```
# Format and display (latest method)
title = "General"
name = "Kenobi"
print(f"Hello there, {title} {name}")
```

INTERPOLATION OF F-STRING

The f-string formatting can calculate the value of an expression (e.g. a string of mathematical operations) in runtime:

```
    # Format and display (latest method)
    a = 2
    b = 7
    print(f"{a} times {b} to the power of 2 is
{(a * b) ** 2}.")
```

INTERPOLATION OF F-STRING

Formatting also allows us to add a specific number of spaces to the left or to the right of the string:

```
header1 = "Name"
header2 = "Age"
name = "John"
age = 22

print(f"| {header1} | {header2} |")
print("-" * 27)
print(f"| {name} | {age} |")
```

INTERPOLATION OF F-STRING

You can also define how many digits after the decimal point should be displayed or display the number as a percentage:

```
    # Changing the way the variable is
displayed
    n = 109.2345654324
    print(f"{n: .3f}") # will display 109.234

    percent = 0.71
    print(f"{percent: .1%}") # will display
71.0%
```

BASIC OPERATIONS ON STRINGS

WHAT ARE PYTHON STRINGS?

- A sequence of characters in a specific order.
- Sequences are indexed from 0.

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

THE LEN () FUNCTION

The **len()** function returns the number of characters in a string.

```
    # Prints the number of characters in the
sequence
    sentence = "Lorem ipsum dolor sit amet..."
    print(len(sentence))  # Prints 29
```

THE .INDEX () FUNCTION

The **.index()** function searches for a given element from the start of the list and returns the lowest index where the element appears:

```
    # Displays the index of the first
occurrence of the letter 'o' in the sequence
    hello = "Hello, World!"
    print(hello.index('o'))  # Prints 4
```

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

THE .COUNT() FUNCTION

The **.count()** function returns the number of occurrences of the substring in the given string.

```
    # Displays the number of occurrences of the
letter 'o' in the sequence
    hello = "Hello, World!"
    print(hello.count('o'))  # Prints 2
```

EXTRACTING A SINGLE CHARACTER FROM THE STRING

The operator [] is used to refer to individual characters in the string.

```
    # Displays the eight character of the
string, counting from 0!
    hello = "Hello, World!"
    print(hello[7]) # Prints W
```

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

STRING SLICING

The [] operator can also be used to extract several characters at once.

```
# Extracts substring from the string
hello = "Hello, World!"
print(hello[7:12]) # Prints World
```

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

STRING SLICING WITHOUT SOME CHARACTERS

```
    # Extracts substring from the string,
omitting every second character
    hello = "Hello, World!"
    print(hello[7:12:2]) # Prints "Wrd"
```

H	e	l	l	o	,		W	o	r	l	d	!
0	1	2	3	4	5	6	7	8	9	10	11	12

REVERSING THE STRING

Using a special command with the [] operator, we can read the string from the end.

```
    # Prints characters in reverse order
    hello = "Hello, World!"
    print(hello[::-1]) # Prints "! dlroW,
olleH"
```

THE .UPPER() FUNCTION

The **.upper()** function returns the same string it is called for, changing all lowercase characters to uppercase ones.

```
    # Displays the inscription in capital
letters
    hello = "Hello, World!"
    print(hello.upper()) # Prints HELLO, WORLD!
```

THE .LOWER() FUNCTION

The **.lower()** function returns the same string it is called for, changing all uppercase characters to lowercase ones.

```
# Displays the inscription in lowercase
hello = "Hello, World!"
print(hello.lower()) # Prints hello, world!
```

COLLECTIONS

WHAT ARE COLLECTIONS?

Collections are containers (variables of a complex type) that can aggregate more than one value.

- list (list),
- dictionary (dict),
- tuple,
- set.

LIST

- It can store multiple values.
- Values can be of different types.
- [and] are used to create the list.

```
    # Declare and initialize the letter
variable
    alphabet = [] # Declares an empty list
    # Prints the number of elements in the list
    print (f"Current length of the alphabet
variable: {len(alphabet)}")
```

LIST.APPEND ()

The **.append()** function is used to add items to the end of the list.

```
    # Let's add some letters to the variable
from the previous slide
    alphabet.append("a")
    alphabet.append("b")
    alphabet.append("c")
    # Prints the content and length of the list
    print (f"Alphabet: {alphabet} (length:
{len(alphabet)})")
```

INDEXING LISTS

Similarly to characters in strings, items in the list are indexed (counted) from 0!!

```
    # Indexing
    print(f"The first letter of the alphabet is
'{alphabet[0]}'.")
```

LIST.EXTEND()

The **.extend()** function allows you to add multiple items to the list at the same time.

```
    # Let's add several elements at once
    alphabet.extend(["f", "d", "g", "e"])
    print(f"Alphabet (confused): {alphabet}
(length: {len(alphabet)})")
```

LIST.SORT()

We use the **.sort()** function when we want to sort items in a given list.

```
    alphabet.sort()
    print(f"Alphabet (sorted): {alphabet}
(length: {len(alphabet)})")
```

THE SORTED() FUNCTION

We use the **sorted()** function when we want to sort elements in a given list without changing the list itself - a new, sorted list will be returned, while the old one will remain intact.

```
    # Sorting list using the sorted() function
    sorted_alphabet = sorted(alphabet)
    print(f"Alphabet (sorted):
{sorted_alphabet}, alphabet (unsorted):
{alphabet})")
```

OTHER LIST FUNCTIONS

To view all list functions, enter the **help(list)** command in the terminal.

- .count(x)
- .index(x)
- .insert(index, x)
- .pop(index)
- .pop()
- .remove(x)
- .clear()
- .reverse()

SLICING LISTS

```
users = ["Alice", "Bob", "Chris", "John"]
print(users)
print(users[0:3])
print(users[1:2])
print(users[2])
print(users[1:])
```

DICTIONARY

- It is similar to the list, but instead of indexes, a key-value pair is used.
- Any value stored in the dictionary can be extracted by entering the key.
- { and } are needed to create the dictionary.

DICTIONARY

```
# Create an empty dictionary
phonebook = {}

# Add two items
phonebook["John"] = 111111111
phonebook["Jack"] = 222222222

print(phonebook)
print(phonebook["Jack"])

# Definition of the finished dictionary
phonebook2 = {
    "John": 111111111,
    "Jack": 222222222
}
```

DELETING ITEMS FROM THE DICTIONARY

To remove items from the dictionary, use the:

- **.pop()** function,
- **del** keyword.

```
    phonebook = {"John": 111111111, "Jack":
222222222}

    # We remove items
    del phonebook["John"]
    phonebook.pop("Jack")

    print(phonebook)
```

DICT.GET()

The **.get()** function of the dictionary returns the value under the key given as a parameter.

```
letters = {"a": 1, "b": 2}
print(letters.get("c")) # Prints None
```

We can specify the second argument. It will be the value that should be returned if the key does not exist in the dictionary.

```
print(letters.get("c", 0)) # Prints 0
```

TUPLE

- A list that cannot be modified after its creation.
- (and) can be used to define it.
- Alternatively, a tuple can also be defined by entering a comma.

TUPLE (TUPLE)

```
# Declaration of an empty tuple
tuple_1 = ()

# Tuple empty declaration
tuple_2 = ("dog", "cat", 2000, 5.0, True)
tuple_3 = ("a", 2, "c", [1, 2, 3])
```

SLICING TUPLES

```
# Accessing tuple item (s)
tuple_4 = (1, 10.5, False, None, "string")
print(tuple_4[2]) # Prints False
print(tuple_4[1:3]) # Prints (10.5, False)
```

SET

- A non-indexable and unordered collection.
- Written with { and } characters.
- An empty set is created by using the **set()** command.
- Set values are not repeated.
- There is no access to elements, set elements do not have indexes.

SET

```
    # Create a set
    animals = {"dog", "cat", "elephant"}

    # Add a new item
    animals.add("mouse")

    # Add several items at once
    animals.update(["bird", "horse"])

    # Add the same item again
    animals.add("mouse")
    print(animals)

    # Remove an item, Python will throw an
error if it is not in the set
    animals.remove("cat")

    # Remove an item, Python will NOT throw an
error if it is not in the set
    animals.discard("cat")
```

MODIFIABLE (MUTABLE) AND NON-MODIFIABLE (IMMUTABLE) TYPES

Mutable

- list,
- collection,
- dictionary.

Immutable

- int
- float,
- bool,
- str.
- tuple
- frozenset.

USER INPUT

DATA INPUT

All applications operate on data. One such source is the data collected from the user. They can be downloaded by:

- various controls, including text boxes, drop-down lists and checkboxes,
- command line parameters.

THE INPUT() FUNCTION

- Allows the user to enter certain data.
- Python stops the program until the user confirms the entered data.
- The parameter is the message that appears on the screen when the user is asked to enter data.

```
    # Ask the user to enter data and write it
out
    print("Welcome.")
    user_name = input("Enter your name:")
    print(f"Hello, {user_name}!")
```

COMMAND LINE PARAMETERS

Parameters are given explicitly by the user after the file name when starting the program.

```
python my-program.py 1 2 secretkey
```

COMMAND LINE PARAMETERS

- In order to capture parameters from the command line, attach a code (library) to the program. That will help the program to read the parameters.
- The code will extract the parameters in the form of a string list.
- The first element of the list is always an absolute path. It describes how to access a given file or directory starting from the root of the file system.
- The next elements are the parameters passed to the program when it is started from the terminal.

COMMAND LINE PARAMETERS

```
python my-program.py first-arg 2
```

```
# Add a sys library to the program to help you read the parameters you specify
import sys

print(f"Program: {sys.argv[0]}")
print(f"First argument: {sys.argv[1]}")
print(f"Second argument: {sys.argv[2]}")
```

CONTROL STATEMENTS

PROGRAM CONTROL

Python supports instructions that can change the order in which the code is executed (usually it goes from top to bottom).
These instructions are:

- **conditional statements** (if, elif, else),
- loops (for, while).

CONDITIONAL STATEMENTS

In the real world, we often have to make some choices.
If it rains **then** I will take an umbrella with me.
In programming, the **if** statement allows us to make various decisions in the code, depending on the given condition.

THE IF STATEMENT

Here's how decisions are made in programs:

```
if condition:
    instructions
```

- The value <condition> must be translatable into a Boolean value (True / False).
- If the <condition> is true (it is translated into True), Python will execute the <instructions>.
- An indentation is required!

THE IF STATEMENT - AN EXAMPLE

```
    x = 0
    y = 3

    if x > y: # This will be translated to
False because 0 is not greater than 3
        print(f"{x} is greater than {y}") #
This will not be displayed

    if x < y: # This will be translated to True
because 3 is greater than 0
        print(f"{x} is less than {y}") # This
will be displayed
```

INDENTATIONS

- Python's recognizable feature.
- In order to execute more than one instruction in the **if** block, all instructions must be indented in the code.

```
if condition:
    instruction_1
    instruction_2
    instruction_3
    ...
    instruction_n
Next_instructions_after_if_block
```

Indentation is used in Python to create blocks of code or compound statements.

THE ELSE STATEMENT

As in real life, programming allows you to make another choice if a certain condition is not met. The **else** clause is used for this.

```
if condition:
    instructions
else:
    other_instructions
```

The **else** statement is an optional statement and there could be at most only one **else** statement following **if** statement.

THE ELSE STATEMENT - AN EXAMPLE

```
    x = 0
    y = 3

    if x > y: # This will be translated to
False because 0 is not greater than 3
        print(f"{x} is greater than {y}") #
This will not be displayed
    else: # This will be translated to True
because 3 is greater than 0
        print(f"{x} is less than {y}") # This
will be displayed
```

THE ELIF STATEMENT

There is also a way to make one of many choices depending on which of the available conditions will be met first.

```
if condition:
    instructions
elif other_condition:
    other_instructions
elif even_other_condition:
    even_other_instructions
else:
    even_more_other_instructions
```

THE ELIF STATEMENT

- Any number of **elif** clauses can be implemented in the conditional statement.
- The **elif** clause is optional.
- If no condition (neither for **if** nor for any of the **elif** statements) is met, the instructions in the **else** block (if added) will be followed.

THE ELIF STATEMENT

```
    x = 0
    y = 3

    if x > y: # This will be translated to
False because 0 is not greater than 3
        print(f"{x} is greater than {y}") # It
will not be displayed
    elif x == 3: # This will be translated to
False because 0 is not equal to 3
        print(f"{x} is equal {y}")
    else: # This will be translated to True
because 3 is greater than 0
        print(f"{x} is less than {y}") # It
will be displayed
```

LOOPS

ITERATIONS

An **iteration** is a repeated execution of a set of statements. Programming structures that implement iterations are called **loops**.
In the **infinite iteration**, the number of loop executions is not specified in advance. A given code block is executed many times, as long as a certain condition is met.
In the **defined iteration**, the code block will be repeated a specified number of times.

THE WHILE LOOP

- The loop is executed as long as the <condition> is true.
- It is checked if the <condition> value is True. If so, the <instructions> are executed. If not - Python skips the loop block and executes the statements outside of it.
- After executing the while loop block, the <condition> is checked again. If it is still true, the loop is executed again.

```
while condition:
    <instruction_1>
    <instruction_2>
    ...
    <instruction_n>
```

THE WHILE LOOP - EXAMPLE

This program will write the numbers 1, 2, 3, 4, 5 - each in a new line.

```
    # Make loops as long as n is less than 5
    n = 0
    while n < 5:
        n += 1 # increment n with each loop
loop
        print(n)
```

LOOP TERMINATION

The **break** statement:

- Immediately stops the current iteration and the loop itself.
- The program exits the loop block and continues to execute instructions outside of it.

The **continue** statement:

- Immediately stops the current iteration and continues with the next one.
- Before starting the next cycle, the <condition> is checked again. This determines whether the next loop should happen or not.

THE WHILE LOOP - EXAMPLE 2

This program will write numbers 2 and 3, each in a new line.

```
    # Make loops as long as n is less than 5
    n = 0
    while n < 5:
        n += 1 # increment n with each loop
loop
        if n == 4: # if n is 4, end the loop
            break
        if n == 1: # if n is 1, start a new
iteration
            continue
        print(n)
```

THE FOR LOOP

```
for var in iterable:
    instructions
```

- An <iterable> is a collection of variables / values after which we can iterate - for instance, a list.
- Indentations will be needed to create the loop block.
- With the for loop we can execute a set of statements once for each item in a given: list, tuple, set etc.
- The <var> variable takes the value of each element in the <iterable> collection and is available in the loop.

THE FOR LOOP - AN EXAMPLE

The program will print all items in the list.

```
    animals = ["Dog", "Cat", "Fish"]

    # List all animals from the animals list
    for animal in animals:
        print(animal) # Lists one animal in
turn
```

LOOP TERMINATION - FOR

The **break** and **continue** commands are fully supported in the for loop.

THE RANGE() FUNCTION

range(start, stop, step)

- The **range()** function returns an iterable object containing numbers from 0 to **start** if only the number **start** is given as an argument.
- The **range()** function returns an iterable object containing numbers from **start** to **stop** excluding the number **stop**, if both **start** and **stop** are given.
- Optionally, you can include the **step** parameter specifying how many elements between values should be skipped.

THE RANGE() FUNCTION - EXAMPLES

```
# Will print 0, 1, 2 in new lines
for i in range(3):
    print(i)

# Will print -3, -2, -1, 0 in new lines
for i in range(-3, 1):
    print(i)
```

```
# Will print 3, 5, 7, 9 in new lines
for number in range(3, 11, 2):
    print(number)

# Will print -1, -2, -3 in new lines
for number in range(-1, -4, -1):
    print(number)
```

THE ENUMERATE() FUNCTION

- A lot of times when dealing with iterators, we also want to know the current count of iterations.
- The **enumerate()** function accepts the collection as a parameter and returns a tuple with two values: an element index and the currently considered element.

```
    fruits = ["apple", "banana", "lemon"]

    for index, fruit in enumerate(fruits):
        print(f"Fruit: {fruit}, under the
index: {index}.")
```

LIST COMPREHENSION

- Imagine a situation where we want to create a list of one thousand numbers from 0 to 999.
- The list is too large to enter values manually.
- It can be populated with values using the for loop or created using the list comprehension mechanism.

LIST COMPREHENSION - AN EXAMPLE

```
# List in loop for
numbers = []
for i in range(1000):
    numbers.append(i)

print(len(numbers)) # Prints 1000

# Folded list
numbers = [i for i in range (1000)]
print(len(numbers)) # Prints 1000
```

DICT COMPREHENSION

Similarly, you can use the dictionary submission mechanism to initialize the dictionary.

```
    keys_and_values = [(1, 'a'), (2, 'b'), (3,
'c')]
    dictionary = {number: letter for (number,
letter) in keys_and_values}
```

INTRODUCTION TO FUNCTIONS

WHAT ARE FUNCTIONS?

- A way to divide the code into useful blocks.
- It helps organize the code.
- It makes the code clearer.
- Instead of repeating the same lines over and over again, you can use a function that includes these instructions.
- It has input arguments.
- It calculates and generates the result based on the given arguments.
- It must be defined before it is used for the first time in the code!

FUNCTIONS

Functions in Python are defined by entering the **def** keyword, the name of the function, its possible parameters in brackets, and writing the necessary instructions in its block (remember to indent!).

```
    def function_name_1():
        instructions

    def function_name_2(arg_1, arg_2, ...,
arg_n):
        instructions
```

FUNCTIONS - EXAMPLE 1

```
    # Definition of the function named
print_hello_world
    def print_hello_world():
        print("Hello world from inside the
function!")
    # Calling print_hello_world()
    print_hello_world()
```

FEATURES - EXAMPLE 2

```
    # Function definition of greet_by_name
(name)
    def greet_by_name(name):
        print(f"Hello, {name}")
    # Call function greet_by_name (name) with
"John" as the name argument
    greet_by_name("John")
```

FUNCTION PARAMETERS

Function parameters can be:

- required (mandatory)
- optional (named parameters).

Arguments for mandatory parameters are usually passed without giving their names. Arguments for optional parameters are usually passed with their names when the function is called.

FUNCTION PARAMETERS - AN EXAMPLE

```
    # Function for printing the name and
surname
    def print_full_name(name, surname):
        print (f"{name} {surname}")

    # Calling a function without specifying thr
parameter names
    print_full_name("Jon", "Snow")

    # Function call with names of all
parameters
    print_full_name(name="Jon", surname="Snow")

    # Calling the function with the names of
the last parameter
    print_full_name("Jon", surname="Snow")
```

TYPES IN FUNCTIONS

Python gives you the ability to specify types of arguments and return types. The syntax is similar to the one we learned when creating variables:

```
def print_hello(text: str) -> None:
    print(f"Hello {text}")


print_hello("world")
```

FUNCTION PARAMETERS - DEFAULT PARAMETERS

Default arguments are values that are provided while defining functions. These parameters become optional during function calls. If we provide a value to the default arguments during function calls, it overrides the default value.

```
    # The definition of the function
greet_by_name (name) with the default value of
the name
    def greet_by_name(name="World!"):
        print(f"Hello, {name}")

    # Calling the function greet_by_name (name)
without an argument
    greet_by_name() # Prints "Hello, World!"
    # Calling the function greet_by_name (name)
with "John" as the name argument
    greet_by_name("John") # Prints 'Hello,
John'
    greet_by_name(name="John") # Prints 'Hello,
John'
```

FUNCTIONS - RETURN VALUES

- Python functions can return calculated values by using the **return** keyword.
- If **return** is not used in the function, then the function returns the **None** value.
- The function always returns something!

```
def calculate_square(a):
    return a * a

square = calculate_square(5)
print(square) # Prints 25
```

FUNCTIONS - RETURNING TYPES

We can specify the type of returning values. To do so, we can use the "->" sign and the colon. An example might look like this:

```
def get_hello(text: str) -> str:
    return f"Hello {text}"


print(get_hello("world"))
```

FUNCTIONS WITH ANY NUMBER OF ARGUMENTS

```
# Add two numbers
def add(a, b):
    return a + b

# Add three numbers
def add(a, b, c):
    return a + b + c

# Add four numbers
def add(a, b, c, d):
    return a + b + c + d
```

What if the user wants to add 10 numbers together?

FUNCTIONS WITH ANY NUMBER OF ARGUMENTS - ARGS

- Instead of creating functions with a large number of positional arguments, you can add the ***args*** parameter.
- User-supplied arguments will enter the ***args*** list and will be available from the function itself.

```
    # Add any number of numbers
    def add(*args):
        result = 0
        for arg in args:
            result += arg
        return result

    print (add (1,2,3,4,5)) # Prints 15

    # Prints the name and what the user gives
    def print_name_and_something(name, *
strings):
        print (f"First name: {name}")
        for string in strings:
            print (string)
```

FUNCTIONS WITH ANY NUMBER OF ARGUMENTS - KWARGS

- Instead of creating functions with a huge number of named arguments, you can add the ****kwargs** parameter.
- Named arguments given by the user will go to the dictionary named kwargs and will be available there in the function.

```
    # Add any number of ingredients
    def add_ingredients(**kwargs):
        result = 0
        for key in kwargs:
            result += kwargs [key]
        return result

    print(add_ingredients(eggs=3, spam=5,
cheese=2)) # Will print 10
```

FUNCTIONS WITH ANY NUMBER OF ARGUMENTS - ARGS AND KWARGS

Any number of non keyword (*args) and keyword (**kwargs) arguments can be combined into one function.

```
    # Add any number of ingredients
    def add_ingredients(*args, **kwargs):
        result = 0
        for arg in args:
            result += arg
        for key in kwargs:
            result += kwargs[key]
        return result

    print(add_ingredients(1, 2, 3, eggs=3,
spam=5, cheese=2)) # Will print 16
```

Basic operations and methods

Task 1: Which pizza has the best price/quantity ratio?

Write a program (or function) that will compare the area/price ratio between two pizzas. In order to calculate the area of a circle *P* at a given radius *r* - use this formula - Formula.

Find a restaurant in your area, enter the appropriate data and answer the question asked in the recommendation.

Important

You can use the math standard library to get the exact value of pi, but it is not required.

Hint

It's worth creating a function that computes the whole so that it doesn't repeat itself twice.

Task 2: Prime numbers (what if they are second?)

Write a program that checks if a given number is preceded by a prime number.

Important

When checking if n is prime, you don't need to check potential divisors from 2 to n. You can dramatically reduce the number of comparisons by only checking from 2 to √ (n) (root of n).

Example:

Let's try to find all the divisors of 100 and list them in the form of a table:

```
2  x  50 = 100
4  x  25 = 100
5  x  20 = 100
10  x  10 = 100 <-- √(100)
20  x  5  = 100
25  x  4  = 100
50  x  2  = 100
```

It can be seen that by reaching √ (100) - all divisors have already been found. This property applies to any value of *n*.

Hint

It's best to start by checking if the number you are checking is two, one, or divisible by 2.

There are many possible solutions when you search for a prime number on the Internet. Try to implement an additional one.

Task 3: Dancing parabolas

Write a function (or program) that will calculate the zeros of the given square function. For this purpose, you can use the formulas presented here.

NOTE

We assume movement only in the space of real numbers, complex solutions are not required.

Hint

In order to accomplish the task, it is best to create a function that will accept 3 arguments being the coefficients of the equation of the quadratic function. The math library for the square root calculation will also be useful.

Text formatting

Task: 1 Alice - The cat mom.

Write a function (or program) that will correctly display the sentence "Alice has x cats" depending on the number of cats. That is it can show: Alice has 1 *cat*, Alice has 2 *cats*, Alice has 10 *cats*.

Hint

The variation of the word "cat" depends on the remainder by dividing the number of cats by 10.

Task 2: HP - printer or programming wizard?

The one-whose-name-could-be-not-be-pronounced could talk to snakes. It's time for him to use Python to relieve himself in the course of his punitive work.

Write a program that will display the given sentence. Every third one will be capitalized and every fourth one will have an exclamation mark at the end. (Just don't tell lies!;)

Hint

It will be a good idea to create an additional string. t will be a copy of the repeated sentence, which, depending on the situation, will receive an additional character at the end, or it will be written in capital letters.

Task 3: Aaaaaa - that means 6.

Write a function (or program) that will determine the number of vowels in a given string.

Important

Try to use Counter in your task.

Hint

It is worth writing down the vowels you are looking for in the form of, for example, a set of vowels.

This solution can also be used with other data structures. Try using a dictionary.