INTRODUCTION TO PYTHON

Compilation vs. interpretation - advantages and disadvantages

	COMPILATION	INTERPRETATION
ADVANTAGES	• the execution of the translated code is usually faster; • only the user has to have the compiler - the end-user may use the code without it; • the translated code is stored using machine language - as it is very hard to understand it, your own inventions and programming tricks are likely to remain your secret.	• you can run the code as soon as you complete it - there are no additional phases of translation; • the code is stored using programming language, not the machine one - this means that it can be run on computers using different machine languages; you don't compile your code separately for each different architecture.
DISADVANTAGES	• the compilation itself may be a very time-consuming process - you may not be able to run your code immediately after any amendment; • you have to have as many compilers as hardware platforms you want your code to be run on.	• don't expect that interpretation will ramp your code to high speed - your code will share the computer's power with the interpreter, so it can't be really fast; • both you and the end user have to have the interpreter to run your code.

- Python is an **interpreted language**. This means that it inherits all the described advantages and disadvantages. Of course, it adds some of its unique features to both sets.
- If you want to program in Python, you'll need the **Python interpreter**. You won't be able to run your code without it. Fortunately, **Python is free**. This is one of its most important advantages.
- Due to historical reasons, languages designed to be utilized in the interpretation manner are often called **scripting languages**, while the source programs encoded using them are called **scripts**.

What is Python?

- Python is a widely-used, interpreted, object-oriented, and high-level programming language with dynamic semantics, used for general-purpose programming.
- Python was created by **Guido van Rossum**, born in 1956 in Haarlem, the Netherlands.
- **Python goals:**

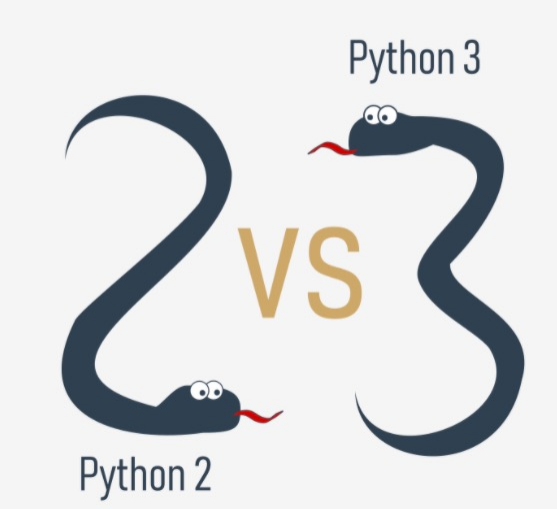
In 1999, Guido van Rossum defined his goals for Python:

  - an **easy and intuitive** language just as powerful as those of the major competitors;
  - **open source**, so anyone can contribute to its development;
  - code that is as **understandable** as plain English;
  - **suitable for everyday tasks**, allowing for short development times.

What makes Python special?

Python 2 vs. Python 3

- There are two main kinds of Python, called Python 2 and Python 3.
- These two versions of Python aren't compatible with each other. Python 2 scripts won't run in a Python 3 environment and vice versa, so if you want the old Python 2 code to be run by a Python 3 interpreter, the only possible solution is to rewrite it, not from scratch, of course, as large parts of the code may remain untouched, but you do have to revise all the code to find all possible incompatibilities. Unfortunately, this process cannot be fully automatized.



Using your Python Interpreter

How to…	Command
Access the Python Interactive Shell	$ **python**
Running a Python script	$ **python** *script.py*

Basic Data Types

Python type()	Values (examples)
int	-128, 0, 42
float	-1.12, 0, 3.14159
bool	True, False
str	“Hello” Can use ‘’, “”, and “””””””
bytes	b”Hello \xf0\x9f\x98\x8e”

```
>>> type(3)
<class ‘int’>

>>> type(1.4)
<class ‘float’>

>>> type(True)
<class ’bool’>

>>> type("Hello")
<class ’str’>

>>> type(b"Hello")
<class ‘bytes’>
```

Numerical Operators

Math Operations

Addition:	+
Subtraction:	-
Multiplication:	*
Division:	/
Floor Division:	//
Modulo:	%
Power:	**

```
>>> 5 + 2
7
>>> 9 * 12
108
>>> 13 / 4
3.25
>>> 13 // 4
3
>>> 13 % 4
1
>>> 2 ** 10
1024
```

Variables

Names

- Cannot start with a number [0-9]
- Cannot conflict with a language keyword
- Can contain: [A-Za-z0-9_-]
- Recommendations for naming (variables, classes, functions, etc.) can be found in PEP8

Created with the = assignment operator

Can see list of variables in the current scope with dir()

```
>>> b = 7
>>> c = 3
>>> a = b + c
>>> a
10

>>> string_one = "Foo"
>>> string_two = "Bar"
>>> new_string = string_one + string_two
>>> new_string
'FooBar'
```

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

Name convention

The PEP 8 -- Style Guide for Python Code recommends the following naming convention for variables and functions in Python:

- #### variable names should be lowercase, with words separated by underscores to improve readability (e.g., var, my_variable)
- #### function names follow the same convention as variable names (e.g., fun, my_function)
- #### it's also possible to use mixed case (e.g., myVariable), but only in contexts where that's already the prevailing style, to retain backwards compatibility with the adopted convention.

Working with Strings

String Operations

Concatenation: **+**

Multiplication: *****

Some Useful String Methods

Composition: **“{}”.format()**

Splitting: **“”.split()**

Joining: **“”.join()**

```
>>> "One" + "Two"
'OneTwo'

>>> "Abc" * 3
'AbcAbcAbc'

>>> "Hi, my name is {}!".format("Chris")
'Hi, my name is Chris!'

>>> "a b c".split(" ")
['a', 'b', 'c']

>>> ",".join(['a', 'b', 'c'])
'a,b,c'
```

String Escape Characters

```
txt = "We are the so-called \"Vikings\" from the north.”
print(txt)
print("\"I\'m\"\n\"\"learning\"\"\n\"\"\"Python\"\"\"")
```

Advanced Data Types (Collections)

Name **type()**	Notes	Example
list	• Ordered list of items • Items can be different data types • Can contain duplicate items • Mutable (can be changed after created)	**['a', 1, 18.2]**
tuple	• Just like a list; except: • Immutable (cannot be changed)	**('a', 1, 18.2)**
dict (dictionary)	• Unordered key-value pairs • Keys are unique; must be immutable • Keys don't have to be the same data type • Values may be any data type	**{"apples": 5,** **"pears": 2,** **"oranges": 9}**

Working with Collections

Name type()	Creating	Accessing Indexing	Updating	Useful methods

list	l = ['a', 1, 18.2]
	l2 = [53, 1, 67]
	l3 = [[1, 2], ['a', 'b']]
		>>> l[2]
		18.2
			>>> l[2] = 20.4
			>>> l
			['a', 1, 20.4]
				>>> concat_l = l2 + l3
				>>> concat_l
				[53, 1, 67, [1, 2], ['a', 'b']]
				
				>>> l2.sort()
				>>> l2
				[1, 53, 67]

tuple	t = ('a', 1, 18.2)
	t2 = (1, 3, 4)
		>>> t[0]
		'a'
			You cannot update tuples after they have been created.	>>> concat_t = t + t2
				>>> concat_t
				('a', 1, 18.2, 1, 3, 4)

dict	d = {"apples": 5,
	"pears": 2,
	"oranges": 9}
	d2 = {1: 15,
	5: 'grapes',
	9: [1, 2, 3]}
		>>> d["oranges"]
		9
		
		>>> d.get("pears")
		2
			>>> d["pears"] = 6
			>>> d
			{'apples': 5, 'pears': 2, 'oranges': 9}
				>>> d.items()
				dict_items([('apples', 5), ('pears', 2), ('oranges', 9)])
				
				>>> d.keys()
				dict_keys(['apples', 'pears', 'oranges'])
				
				>>> d.values()
				dict_values([5, 2, 9])

List Comprehension

List comprehension allows you to create new lists from existing ones in a concise and elegant way.

The syntax of a list comprehension looks as follows:

[expression for element in list if conditional]

```
lst = [i for i in range(101) if i % 2 == 0]
lst = [i ** i for i in range (1,10)]
```

Conditionals

Syntax:

```
if expression1:
    statements...
elif expression2:
    statements...
else:
    statements...
```

- ✓ Indentation is important!
- ✓ 4 spaces indent recommended
- ✓ You can nest if statements

Comparison Operators:

Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Equal	==
Not Equal	!=
Contains element	in

Combine expressions with: **and**, **or**

Negate with: **not**

Loops

Iterative Loops

for *individual_item* **in** *iterator*:

statements...

```
>>> names = ["chris", "iftach", "jay"]
>>> for name in names:
...     print(name)
...
chris
iftach
jay
```

Conditional Loops

while *logical_expression*:

statements...

```
>>> i = 0
>>> while i < 5:
...     print(i)
...     i += 1
...
0
1
2
3
4
```

Functions | Don’t Repeat Yourself

Modularize your code

- Defining your own Functions
- (optionally) Receive arguments
- (optionally) Return a value

Syntax:

```
def function_name(arg_names):
    statements…
    return value
…
function_name(arg_values)
```

```
>>> def add(num1, num2):
...     result = num1 + num2
...     return result
...
>>>
>>> add(3, 5)
8


>>> def say_hello():
...     print("Hello!")
>>>
>>> say_hello()
Hello!
```

Importing and Using Packages & Modules

- Module is a file containing Python definitions and statements, which can be later imported and used when necessary.
- Syntax

  Import module

  From module import thing

  Import module as alias_name

```
In [2]: import math, sys

In [3]: print(math.pi)
        3.141592653589793

In [4]: print(math.sin(math.pi/2))
        1.0

In [5]: from math import pi
        print(pi)
        3.141592653589793

In [6]: from math import sin, pi
        print(sin(pi/2))
        1.0

In [20]: import math as m
         print(m.sin(m.pi/2))
         1.0

In [9]: from math import pi as PI, sin as sine

        print(sine(PI/2))
        1.0
```

Importing and Using Packages & Modules

- Import user module in same directory

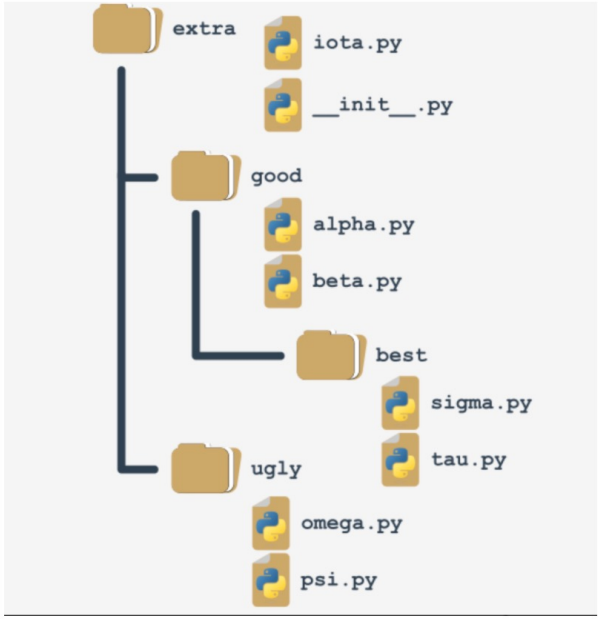
```
import module1 as m1
```

- Import user module from different directories

```
import os
from sys import path
module_path = ”path to module directory”
path.append(module_path)
import module2 as m2
```

Packages

the presence of the **init.py** file finally makes up the package.




```
import os
from sys import path


package_path = os.getcwd() + "/extra”
// package_path = os.getcwd() + "extrapack.zip"
path.append(package_path)
import extra.good.best.sigma as sig
import extra.good.alpha as alp
```

Python Package Installer (PIP)

- Uses the open PyPI repository
- Installs packages and their dependencies
- You can post your packages to PyPI

```
$ sudo apt install python3-pip

$ pip --version
pip 20.3.4 from /usr/lib/python3/dist-packages/pip (python 3.9)

$ pip3 --version
pip 20.3.4 from /usr/lib/python3/dist-packages/pip (python 3.9)
```

```
$ pip help
$ pip list
$ pip show package_name
$ pip search anystring
$ pip install package_name
$ pip uninstall package_name
```

Processing files

- A file needs to open before it can be processed by a program, and it should be closed when the processing is finished.
- Open a file

```
f = open("demofile.txt", "r")
```

- Read Files

```
print(f.read())
print(f.readline())
data = f.readlines();
for line in data:
        print(line)
```

- Close a file

```
f.close()
```

- Open & close files by with...as

```
with open("demofile.txt", "r") as data:
  print(data.read())
```

Processing files

Write to an Existing File

To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

```
f = open("demofile.txt", "a")
f.write("\nFourth line added by Python")
f.close()
```

```
f = open("demofile.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()
```

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

"a" - Append - will create a file if the specified file does not exist

"w" - Write - will create a file if the specified file does not exist

```
f = open("myfile.txt", "x")
```

```
f = open("myfile.txt", "w")
```