Python Course

Github	https://github.com/Pierian-Data/Complete-Python-3-Bootcamp
Google Drive link	https://drive.google.com/drive/folders/1CKqOQzst1cGURXGiRVivi2Xsc0n-X8CR
Status	In progress

Python is a high-level, interpreted programming language created by Guido van Rossum and first released in 1991. It emphasizes code readability with its notable use of significant whitespace and simple, clear syntax.

Key characteristics of Python:

- It's an interpreted language, meaning code is executed line by line rather than being compiled first
- Features dynamic typing, where variable types are determined at runtime
- Supports multiple programming paradigms including procedural, object-oriented, and functional programming
- Has a large standard library that provides rich functionality for various tasks
- Known for its "batteries included" philosophy, providing comprehensive built-in modules and functions
- mainly used for Machine learning and Data science & Python can be used in Web development and perform backend with frameworks like Django and flasks.
- Can create interactive dashboards for large data analysis.

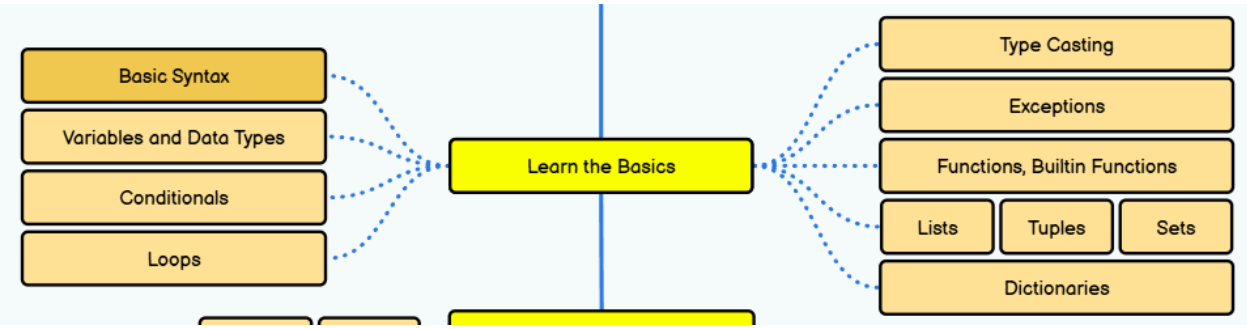
Compiler vs interpreter:

▼ The difference is here ⬇️

Compiler	Interpreter
Steps of Programming: • Program Creation. • Analysis of language by the compiler and throws errors in case of any incorrect statement. • In case of no error, the Compiler converts the source code to Machine Code. • Linking of various code files into a runnable program. • Finally runs a Program.	**Steps of Programming:** • Program Creation. • Linking of files or generation of Machine Code is not required by Interpreter. • Execution of source statements one by one.
The compiler saves the Machine Language in form of Machine Code on disks.	The Interpreter does not save the Machine Language.
Compiled codes run faster than Interpreter.	Interpreted codes run slower than Compiler.

Linking-Loading Model is the basic working model of the Compiler.	The Interpretation Model is the basic working model of the Interpreter.
The compiler generates an output in the form of (.exe).	The interpreter does not generate any output.
Any change in the source program after the compilation requires recompiling the entire code.	Any change in the source program during the translation does not require retranslation of the entire code.
Errors are displayed in Compiler after Compiling together at the current time.	Errors are displayed in every single line.
The compiler can see code upfront which helps in running the code faster because of performing Optimization.	The Interpreter works by line working of Code, that's why Optimization is a little slower compared to Compilers.
It does not require source code for later execution.	It requires source code for later execution.
Execution of the program takes place only after the whole program is compiled.	Execution of the program happens after every line is checked or evaluated.
Compilers more often take a large amount of time for analyzing the source code.	In comparison, Interpreters take less time for analyzing the source code.
CPU utilization is more in the case of a Compiler.	CPU utilization is less in the case of a Interpreter.
The use of Compilers mostly happens in Production Environment.	The use of Interpreters is mostly in Programming and Development Environments.
Object code is permanently saved for future use.	No object code is saved for future use.
C, **C++**, **C#**, etc are programming languages that are compiler-based.	**Python**, **Ruby**, **Perl**, SNOBOL, **MATLAB**, etc are programming languages that are interpreter-based.

Introduction - Basics



Data Types

▼ There are 8 Data types in the python.

Complete Python 3 Bootcamp

Name	Type	Description
Integers	int	Whole numbers, such as: **3 300 200**
Floating point	float	Numbers with a decimal point: **2.3 4.6 100.0**
Strings	str	Ordered sequence of characters: **"hello" 'Sammy' "2000" "楽しい"**
Lists	list	Ordered sequence of objects: **[10,"hello",200.3]**
Dictionaries	dict	Unordered Key:Value pairs: **{"mykey" : "value" , "name" : "Frankie"}**
Tuples	tup	Ordered immutable sequence of objects: **(10,"hello",200.3)**
Sets	set	Unordered collection of unique objects: **{"a","b"}**
Booleans	bool	Logical value indicating **True** or **False**

Int, float & String:

```
# INTEGER - int
number = 10
type(number) # int

# FLOAT - float
pi = 3.14
type(pi) # float
```

```
# STRING - string
text = "python" # For string - "This" or 'This' both same, Quotes doesn't matter.
type(text) # string

# Find length
print(len(text)) # 6
# Find text index
print("output :", text[2])  # output : t
print("output :", text[-2]) # Negative indexing, output : o
# String Slice
# text[start:stop:jump]
print(text[2:]) # thon
print(text[0:2]) # py
print(text[::2])  # pto
print(text[::-1])  # nohtyp - Trick to reverse the string

# METHODS
# String Format Method
print("the {} {} {}".format("python", "class", "over"))  # the python class over
print("the {2} {1} {0}".format("python", "class", "over")) # the over class python
print("the {o} {p} {c}".format(p="python", c="class", o="over")) # the over pytho
print("The Precision is : {val:1.3f}".format(val=100.123456789)) # value:width.Pr
print(f"the {text} is good language") # f - infront of string will also format

# Methods of Strings
print(text.upper())  # PYTHON
print(text.islower())  # True
print(text.split("y"))  # ["p", "thon"]
print("hello world".capitalize())  # "Hello world" ~~ Capitalize the first word
print("HELLo WorLd".casefold())  # "hello world" ~~ Lower case but powerful th



# we can include escape sequence like \n -  new line, \t - tab in string
# We can do string multiplication eg : "Ha" * 3 results in "HaHaHa" but not in JS
```

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isascii()	Returns True if all characters in the string are ascii characters
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case
join()	Converts the elements of an iterable into a string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations

Method	Description
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

LISTS VS. TUPLES VS. SETS VS. DICTIONARIES

	Lists	Tuples	Sets	Dictionaries
Ordering	Ordered	Ordered	Unordered	Ordered Unordered before Python 3.7
Indexing	Indexed	Indexed	Not Indexed	Keyed
Mutability	Mutable	Immutable	Mutable Only Adding and Removing	Mutable
Duplicates Allowed	Yes	Yes	No	Yes Only in values and not in keys
Types Allowed	Mutable and Immutable	Mutable and Immutable	Only Immutable	Only Immutable In keys

List:

```
myList = ["Apple", "Meta", "Alphabets"]
addList = ["Microsoft", "Tesla"]

print(myList + addList)  # ['Apple', 'Meta', 'Alphabets', 'Microsoft', 'Tesla']
print(myList, addList)  # ['Apple', 'Meta', 'Alphabets'] ['Microsoft', 'Tesla']
```

Dictionaries:

```
myDict = {"key1": 100, "key2": 200, "key3": 300}
print(myDict["key3"])  # 300
print(myDict.keys())  # ['key1', 'key2', 'key3']
print(myDict.values())  # [100, 200, 300]
```

```
print(myDict.items())  # [('key1', 100), ('key2', 200), ('key3', 300)]
print(myDict.pop("key3"))  # {'key1': 100, 'key2': 200}
print(myDict.clear())  # None
```

Tuples:

```
myTuples = ("Hello", "World", 100, 100, 200, "Hello")
print(myTuples.count("Hello"))
print(myTuples.index("Hello"))
```

```
# 💡Find each data type methods
def get_method():
    i: int = 0
    for method in dir(str):
        if '__' not in method:
            i += 1
            print(i, method, sep=": ")


get_method()
```

Set:

```
myList = ["Apple", "Meta", "Alphabets", "Apple", "META", "A", "B", "C"]
newList = ["Apple", "Meta", "Alphabets", "Apple", "META", "X", "Y", "Z"]
mySet = set(myList)
newSet = set(newList)

# methods for sets
# Add
mySet.add("I added")  # it adds to set

# Discard & Remove
mySet.discard("I added")  # It removes from set
mySet.remove("I added")  # ERROR - It throw error when mentioned item is not i
```

```
mySet.pop()  # It removes random element from set

# Clear entire set
print(mySet.clear())  # it empties the set

# Copy
copy_set = mySet.copy()  # copy one set to another
print(copy_set)  # {"Apple", "Meta", "Alphabets", "Apple", "META", "A", "B", "C"}

# difference - -=
print(newSet.difference(mySet))  # {'X', 'Z', 'Y'}
# This also written as myset - newset
print(mySet.difference_update(newSet))
# This also written as myset -= newset
print(mySet)  # {'A', 'B', 'C'}

# intersection & &=
print(newSet.intersection(mySet))  # {'Alphabets', 'Meta', 'Apple', 'META'}
print(newSet.intersection_update(mySet))  # None
print(newSet)  # {'Alphabets', 'Meta', 'Apple', 'META'}

# symmetric_difference ^ ^=
print(newSet.symmetric_difference(mySet))  # {'B', 'A', 'Z', 'C', 'Y', 'X'}
print(newSet.symmetric_difference_update(mySet))  # None
print(newSet)  # {'B', 'Z', 'A', 'Y', 'X', 'C'}

# Union and Update | |=
print(newSet.union(mySet))  # {'C', 'META', 'Z', 'Apple', 'A', 'Y', 'B', 'Meta', 'X', 'Al
print(newSet.update(mySet))  # None
print(newSet)  # {'C', 'META', 'Z', 'Apple', 'A', 'Y', 'B', 'Meta', 'X', 'Alphabets'}

#Disjoint
print(newSet.isdisjoint({1, 2, 3}))  # True
print(newSet.isdisjoint(mySet))  # False

# Subset <= and Superset >=
set1 = {1, 2, 3}
```

```
set2 = {1, 2, 3, 4, 5}
print(set1 <= set2)  # True
print(set1 >= set2)  # False
```

“Dunder” Variable :

“Dunder” variables (officially called "special attributes") ***define a contract between Python programmers and the Python interpreter***.

💡 `__name__` is a special built-in variable in Python.

The construct `if __name__ == '__main__':` is commonly used to ensure that a block of code is only executed when the script is run directly, and not when it is imported as a module into another script.

```
# script.py
# __name__
def main():
    print("Hello, world!")

if __name__ == '__main__':
    main()
```