1.1 Getting Started with Python 🐍

It's pretty easy to start with Python Language 🐍. We would be using Python >= 3.9 in this Repository as of now 🙂

1. Download Python
2. Build a Docker image by using the Dockerfile present in our Repository and run the container out of it which comes bundled with everything to run the code present in our repository 🚀.

Lets check the version of Python we are using. We have 2 ways to know this.

1. Open the cmd or terminal and execute **python --version**
2. Using Python's builtin sys module

In [1]:

```python
import sys

print(sys.version)
```

```
3.9.4 (default, Apr 10 2021, 15:31:19)
[GCC 8.3.0]
```

1.2 Creating variables and assigning values

Python is a Dynamically typed language. It means based on the value we assign to a variable, it sets the datatype to it.

Now the question is "How do we assign a value to a variable?🤔". It's pretty easy.

```
<variable_name> = <value>
```

We have a big list of data types that come as builtins in Python.

- None
- bytes
- int
- bool
- float
- complex
- string
- tuple
- list
- set
- dict

Apart from the above prominent data types, we have a few other data types like namedtuple, frozensets, etc..

Let's create examples for the above data types, will be little bored in just seeing the examples. We would be covering in depth about these data types in upcoming chapters :)

Few things to know before getting into the examples:😉

1. `print` function is used to print the data on to the console. We used `f` inside the print function which is used to format the strings as `{}`, these are known as f-strings.
2. `type` function is used to find the type of the object or datatype.

In [1]:

```
# None
none_datatype = None
print(f"The type of none_datatype is {type(none_datatype)}")
```

```
The type of none_datatype is <class 'NoneType'>
```

In [2]:

```
# int
int_datatype = 13
print(f"The type of int_datatype is {type(int_datatype)}")
```

```
The type of int_datatype is <class 'int'>
```

In [3]:

```
# bytes
bytes_datatype = b"Hello Python!"
print(f"The type of bytes_datatype is {type(bytes_datatype)}")
```

```
The type of bytes_datatype is <class 'bytes'>
```

In [4]:

```
# bool
```

```python
# bool datatype can only have either True or False. Integer value of True is
bool_datatype = True
print(f"The type of bool_datatype is {type(bool_datatype)}")
```

```
The type of bool_datatype is <class 'bool'>
```

In [5]:

```python
# float
float_datatype = 3.14
print(f"The type of float_datatype is {type(float_datatype)}")
```

```
The type of float_datatype is <class 'float'>
```

In [6]:

```python
# complex
complex_datatype = 13 + 5j
print(f"The type of complex_datatype is {type(complex_datatype)}")
```

```
The type of complex_datatype is <class 'complex'>
```

In [7]:

```python
# str
str_datatype = "Hey! Welcome to Python."
print(f"The type of str_datatype is {type(str_datatype)}")
```

```
The type of str_datatype is <class 'str'>
```

In [8]:

```python
# tuple
tuple_datatype = (None, 13, True, 3.14, "Hey! Welcome to Python.")
print(f"The type of tuple_datatype is {type(tuple_datatype)}")
```

```
The type of tuple_datatype is <class 'tuple'>
```

In [9]:

```python
# list
list_datatype = [None, 13, True, 3.14, "Hey! Welcome to Python."]
print(f"The type of list_datatype is {type(list_datatype)}")
```

```
The type of list_datatype is <class 'list'>
```

In [10]:

```python
# set
set_datatype = {None, 13, True, 3.14, "Hey! Welcome to Python."}
print(f"The type of set_datatype is {type(set_datatype)}")
```

```
The type of set_datatype is <class 'set'>
```

In [11]:

```python
# dict
dict_datatype = {
    "language": "Python",
    "Inventor": "Guido Van Rossum",
    "release_year": 1991,
}
print(f"The type of dict_datatype is {type(dict_datatype)}")
```

```
The type of dict_datatype is <class 'dict'>
```

Tidbits

The thing which I Love and Hate the most about Python is the dynamic typing. We might not know what are the types of parameters we might pass to a function or method. If you pass any other type of object as a parameter, **boom** you might see Exceptions raised 👻. Let's remember that **With great power comes great responsibility** 🕷

To help the developers with this, from Python 3.6 we have Type Hints(PEP-484).

We will get through these in the coming chapters. Stay tuned 😇

1.3 Python Keywords and allowed Variable names

In [1]:

```
# To retrieve the python keyword list, we can use the keyword built-in packag
import keyword
```

Let's print the keywords present.

keyword.kwlist returns python's keywords in a list datatype.

We are using *(starred) expression to print the values returned by keyword.kwlist each separated by "\n"(newline).

In [2]:

```
print(*keyword.kwlist, sep="\n")
```

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

Variable Names

TLDR:

- Variable names shouldn't be same as that of built-in keywords.

- Variable name shouldn't start with a number or with a symbol(except "_", protected and private attributes are created using underscore, 🤔 it's better to say it as name mangling rather than protected or private. That's for a different notebook session 😀).

PS: Don't give a try naming the variable that starts with #, it would be a Python's comment, which would be neglected by the interpreter 😅.

Allowed Variable names

In [3]:

```
x = True
_x = False
x_y = "Hey Python geek!"
x9 = "alphabet_number"
# Python is a case sensitive language, so `x` is different from `X`. Let's gi
X = "one more variable"
print(f"x is equal to X:{x==X}")
```

```
x is equal to X:False
```

Invalid Variable names

We will be using `exec` within `try` - `except` to catch the syntax error. 🤔 But why? Syntax errors can't be catched, well it shouldn't for good 😉. so we are using `exec` to execute the code.

`exec` takes the string argument and interprets the string as a python code.

In [4]:

```
# variable name starting with number.
code_string = "9x=True"
try:
    exec(code_string)
except SyntaxError as exc:
    print(f"Ouch! In the exception: {exc}")
```

```
Ouch! In the exception: invalid syntax (<string>, line 1)
```

In [5]:

```
# variable name starting with a symbol(other than underscore"_").
code_string = "$g = 10"
try:
    exec(code_string)
except SyntaxError as exc:
    print(f"Ouch! In the exception: {exc}")
```

```
Ouch! In the exception: invalid syntax (<string>, line 1)
```

1.4 Data types

`Kiddo explanation 😇:`

We might use many materials like sand, bricks, concrete to construct a house. These are basic and essential needs to have the construction done and each of them have a specific role or usage.

Likewise, we need various data types like string, boolean, integer, dictionary etc.. for the development of a code. We need to know where to use a specific data type and it's functionality. 😊

We have various built-in data types that come out of the box 😎.

Data type	Mutable?
None	❌
bytes	❌
bool	❌
int	❌
float	❌
complex	❌
str	❌
tuple	❌
list	✅
set	✅
dictionary	✅

The First question we would be interested in is "What is Mutable?🤔". If a object can be altered after its creation, then it is Mutable, else Immutable.

None

None is a singleton object, which represents empty or null.

Example of None usage:

In this example, Let's try getting the environment variables 😉

We would be using the `os` module's `getenv` method to fetch the environment variable's value, if there isn't that environment variable, it would be returning `None`

In [1]:

```
import os

# let's set a env variable first
new_environment_variable_name: str = input("Enter the variable name: \n>>>")
new_environment_variable_value: str = input("Enter the variable's value: \n>>
os.environ[new_environment_variable_name] = new_environment_variable_value

# Now let's try to fetch a envrionment's variable value
env_variable_name: str = input("Enter the variable name to be searched: \n>>>
value = os.getenv(env_variable_name)
```

```
if value is None:
    print(f"There is no environment variable named {env_variable_name}")
else:
    print(
        f"The value assigned for the environment variable named {env_variable
    )
```

```
There is no environment variable named Golang
```

bytes

byte objects are the sequences of bytes, these are machine readable form and can be stored on the disk. Based on the encoding format, the bytes yield results.

bytes can be converted to string by decoding it, vice-versa is known as encoding.

bytes objects can be created by prefixing `b` before the string.

In [2]:

```
bytes_obj: bytes = b"Hello Python Enthusiast!"
print(bytes_obj)
```

```
b'Hello Python Enthusiast!'
```

We see that they are visually the same as string when printed. But actually they are ASCII values, for the convenience of the developer, we see them as human readable strings.

But how to see the actual representation of bytes object? 🤔 It's pretty simple 😉! We can typecast the bytes object to a list and we see each character as it's respective ASCII value.

In [3]:

```
print(list(bytes_obj))
```

```
[72, 101, 108, 108, 111, 32, 80, 121, 116, 104, 111, 110, 32, 69, 110, 116, 1
04, 117, 115, 105, 97, 115, 116, 33]
```

bool

bool objects have only two values: `True` ✅ and `False` ❌, integer equivalent of True is 1 and for False is 0

In [4]:

```
do_we_love_python = True
if do_we_love_python:
    print("🐍 Python too loves and takes care of you ❤")
else:
    print("🐍 Python still loves you ❤")
```

```
🐍 Python too loves and takes care of you ❤
```

PS: Boolean values in simple terms mean **Yes** for `True` and **No** for `False`

int

int objects are any mathematical Integers. pretty easy right 😎

In [5]:

```
# Integer values can be used for any integer arithmetics.
# A few simple operations are addition, subtraction, multiplication, divisio
operand_1 = int(input("Enter an integer value: \n>>>"))
operand_2 = int(input("Enter an integer value: \n>>>"))
print(operand_1 + operand_2)
```

```
8
```

float

float objects are any rational numbers.

In [6]:

```
# Like integer objects float objects are used for decimal arithmetics
# A few simple operations are addition, subtraction, multiplication, divisio
# We are typcasting integer or float value to float values explicitly.
operand_1 = float(input("Enter the integer/float value: \n>>>"))
operand_2 = float(input("Enter the integer/float value: \n>>>"))
print(operand_1 + operand_2)
```

```
11.620000000000001
```

complex

complex objects aren't so complex to understand 😉

complex objects hold a Real number and an imaginary number. While creating the complex object, we would be having a `j` beside the imaginary number.

In [7]:

```
operand_1 = 10 + 5j
operand_2 = 3 + 4j
print(operand_1 * operand_2)
```

```
(10+55j)
```

explanation for the above math: 😉

```
math
(3+4j)*(10+5j)
3(10+5j) + 4j(10+5j)
30 + 15j + 40j + 20(j*j)
30 + 15j + 40j + 20(-1)
30 + 15j + 40j - 20
30 - 20 + 15j + 40j
10 + 55j
```

str

string objects hold an sequence of characters.

In [8]:

```
my_string = "🐍 Python is cool"
print(my_string)
```

```
🐍 Python is cool
```

tuple

tuple object is an immutable datatype which can have any datatype objects inside it and is created by enclosing paranthesis `()` and objects are separated by a comma.

Once the tuple object is created, the tuple can't be modified, although if the objects in the tuple are mutable, they can be changed 😊

The objects in the tuple are ordered, So the objects in the tuple can be accessed by using its index ranging from 0 to (number of elements - 1).

In [9]:

```
# tuples are best suited for having data which doesn't change in it's lifeti

apple_and_its_colour = ("apple", "red")
watermelon_and_its_colour = ("watermelon", "green")

language_initial_release_year = ("Golang", 2012)
language_initial_release_year = ("Angular", 2010)
language_initial_release_year = ("Python", 1990)

# We can't add new data types objects, delete the existing datatype objects,
# of the existing objects.

# We can get the values by index.
print(
    f"{language_initial_release_year[0]} is released in {language_initial_rel
)
```

```
Python is released in 1990
```

list

list objects are similar to tuple, the differences are the list object is mutable, so we can add or remove objects in the list even after its creation. It is created by using `[]`.

In [10]:

```
about_python = [
    "interpreted",
    "object-oriented",
    "dynamically typed",
    "open source",
    "high level language",
    "🐍",
    1990,
]
print(about_python)
# We can add more values to the above list. append method of list object is
# let's give a try 🙃

about_python.append("Guido Van Rossum")
print(about_python)
```

```
['interpreted', 'object-oriented', 'dynamically typed', 'open source', 'high
level language', '🐍', 1990]
['interpreted', 'object-oriented', 'dynamically typed', 'open source', 'high
level language', '🐍', 1990, 'Guido Van Rossum']
```

set

set objects are unordered, unindexed, non repetitive collection of objects. Mathematical set theory operations can be applied using set datatype objects. 😊 it is created by using `{}`.

PS: `{}` denotes a dictionary, we need to use `set()` for creating an empty set, there won't be this issue when creating set objects containing objects, for example: `{1,"a"}`

set objects are good for having the mathematical set operations.

In [11]:

```
set_obj = {6, 4, 4, 3, 10, "Python", "Python", "Golang"}
# We see that we have created a set with 8 objects.
```

```
print(set_obj)
# But when printed, we see that only 6 are present because set doesn't allow
```

```
{'Golang', 3, 4, 10, 6, 'Python'}
```

dict

dictionary objects are used for creating key-value pairs, Here keys would be unique while values can be repeated.

The object assigned to a key can be fetched by using `<dict_obj>[key]` which raises a KeyError when no given key is found. The other way to fetch is by using `<dict_obj>.get(key)` which returns `None` by default if no key is found.

In [12]:

```
dict_datatype = {
    "language": "Python",
    "Inventor": "Guido Van Rossum",
    "release_year": 1991,
}
print(f"The programming language is: {dict_datatype['language']}")
# We could use get method to prevent KeyError if the given Key is not found.
result = dict_datatype.get("LatestRelease")
# Value of the result would be None as the key LatestRelease is not present i
print(f"The result is: {result}")
```

```
The programming language is: Python
The result is: None
```

1.5 Collection Types

We have many collection types in Python, `str`, `int` objects hold only value, but coming to collection types, we can have various objects stored in the collections.

The Collection Types we have in Python are:

- Tuple
- List
- Set
- Dictionary

Tuple

A Tuple is a ordered collection of objects and it is of fixed length and immutable, so the values in the tuple can not be changed nor added or removed.

Tuples are generally used for small collections which we are sure about them from right before such as IP addresses and port numbers. Tuples are represented with paranthesis `()`

Example:

In [1]:

```
ip_address_port = ("127.0.0.1", 8080)
```

A tuple with a single member needs to have a trailing comma, else the type of the variable would be the datatype of the member itself.

In [2]:

```
# Proper way to create a single member tuple.
single_member_tuple = ("one",)
print(type(single_member_tuple))
single_member_tuple = ("one",)
print(type(single_member_tuple))
```

```
<class 'tuple'>
<class 'tuple'>
```

In [3]:

```
# Improper way trying to create a single member tuple.
single_member_tuple = "one"
print(type(single_member_tuple))
```

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

List

List collection types are similar to tuples, the only difference would be that new objects can be created, removed or object's data can be modified 😉.

In [4]:

```
int_list = [1, 2, 3]
string_list = ["abc", "defghi"]
```

In [5]:

```
# A list can be empty:
empty_list = []
```

objects in the list are not restricted to be of a particular datatype. let's see an example 👇.

In [6]:

```
mixed_list = [1, "abc", True, 3.14, None]
```

list can contain lists as objects too. These are called nested lists.

In [7]:

```
nested_list = [[1, 2, 3], ["a", "b", "c"]]
```

The objects present in the list can be accessed by the index it is placed. The index starts from 0 👻.

In [8]:

```
my_list = ["Iron man", "Thor", "Wonder Woman", "Wolverine", "Naruto"]
```

In [9]:

```
print(my_list[0])
print(my_list[1])
```

```
Iron man
Thor
```

In the `my_list`, we have 5 strings in the list, but in the below example, let's give a try to get the 100th index element which is not present in the `my_list` 🙄.

As there is no 100th element, we would be seeing an `IndexError` exception.

In [10]:

```
try:
    print(my_list[100])
except IndexError as exc:
    print(f"👻 Ouch! we got into IndexError exception: {exc}")
```

```
👻 Ouch! we got into IndexError exception: list index out of range
```

The question I have is, how do I get the 2nd element from the last 🤔? Should I find the length of the list and access the <length - 2>? Yup, it works 😉.

But we have one good way to do it by negative index, example: `-2`

In [11]:

```
# Access the 2nd element from the last.
print(my_list[-2])
```

```
Wolverine
```

We have a few methods of list that we can give it a try now 😎

`append`

In [12]:

```
# Append a new item to the list.
# We use append method of the list.
my_list.append("Zoro")
print(my_list)
```

```
['Iron man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto', 'Zoro']
```

`remove`

In [13]:

```
# Remove the item present in the list.
# We use remove method of the list.
# If there's no object that we are trying to remove in the list, then ValueEr
try:
    my_list.remove("Zoro")
    print(my_list)
except ValueError as exc:
    print(f"Caught ValueError: {exc}")
```

```
['Iron man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto']
```

`insert`

In [14]:

```
# Insert a object at a particular index.
# We use insert method of the list.
my_list.insert(1, "Super Man")
print(my_list)
```

```
['Iron man', 'Super Man', 'Thor', 'Wonder Woman', 'Wolverine', 'Naruto']
```

`reverse`

In [15]:

```
# Reverse the objects in the list.
# we use reverse method of the list.
my_list.reverse()
print(my_list)

# revert to the actual order
my_list.reverse()

# We have one more method too for this 🙃
# The indexing of the list would be in the form of list[start: end: step]
# We will use step as -1 to get the elements in reverse order 😉
print(my_list[::-1])
```

```
['Naruto', 'Wolverine', 'Wonder Woman', 'Thor', 'Super Man', 'Iron man']
['Naruto', 'Wolverine', 'Wonder Woman', 'Thor', 'Super Man', 'Iron man']
```

`index`

In [16]:

```
# Index of an object in the list.
# we use index method of the list.
# raises a ValueError, if no given object is found in the list.
try:
    print(my_list.index("Naruto"))
except ValueError as exc:
    print(f"Caught ValueError: {exc}")
```

```
5
```

`pop`

In [17]:

```
# Pop is used to remove and return the element present at the last in the lis
# When index argument is passed, it would remove and return the element at th
# raises IndexError when no object is present at the given Index.
try:
    last_element = (
        my_list.pop()
    )  # can be passed index argument value, if required to pop at a specifi
    print(last_element)
except IndexError as exc:
    print(f"Caught IndexError: {exc}")
```

```
Naruto
```

set

A set is collection of unique items, the items does not follow insertion order.

Defining an set is pretty similar to a list or tuple, it is enclosed in `{}`

PS 🔔: If we need to have a empty set, `{}` won't create a set, it creates a empty dictionary instead. So we need to create a empty set by using `set()`

In [18]:

```
anime = {"Dragon ball", "One Piece", "Death Note", "Full Metal Alchemist", "N
print(anime)
```

```
{'Death Note', 'Dragon ball', 'One Piece', 'Full Metal Alchemist', 'Naruto'}
```

`add`

In [19]:

```
anime.add("Tokyo Ghoul")
print(anime)
```

```
{'Death Note', 'Dragon ball', 'One Piece', 'Full Metal Alchemist', 'Naruto',
'Tokyo Ghoul'}
```

`remove`

remove method of set can be used to remove a particular object from the set, if the object is not present, KeyError would be raised.

In [20]:

```
try:
    anime.remove("Tokyo Ghoul")
    print(anime)
except KeyError as exc:
    print(
        f"Caught KeyError as there's given anime series present in the anime
    )
```

```
{'Death Note', 'Dragon ball', 'One Piece', 'Full Metal Alchemist', 'Naruto'}
```

Dictionary

As in few other languages, we have hashmaps, Dictionaries in python are similar. It has unique Key - Value pairs.

The Key and Value can be of any object. Each Key-Value pair is separated by a `,`

In [21]:

```
anime_protagonist = {
    "Dragon Ball": "Goku",
    "One Piece": "Luffy",
    "Death Note": "Yagami Light",
    "Full Metal Alchemist": "Edward Elric",
    "Naruto": "Naruto",
}
print(anime_protagonist)
```

```
{'Dragon Ball': 'Goku', 'One Piece': 'Luffy', 'Death Note': 'Yagami Light',
'Full Metal Alchemist': 'Edward Elric', 'Naruto': 'Naruto'}
```

We can access the values of the dictionary by `<dictionary>[<key>]`. If there's no `<key>` in the dictionary, we would be seeing an KeyError 🔑❌

In [22]:

```python
try:
    print(anime_protagonist["Dragon Ball"])
except KeyError as exc:
    print(
        f"👻 Ouch, Keyerror has been raised as no given key is found in the d
    )
```

```
Goku
```

Iterate over keys, values and both in the dictionary 🐇

In [23]:

```python
# Keys
print("===Keys===")
for my_key in anime_protagonist.keys():
    print(my_key)

# Values
print("===Values===")
for my_value in anime_protagonist.values():
    print(my_value)

# Key-Values
print("===Key-Values===")
for my_key, my_value in anime_protagonist.items():
    print(f"{my_key} : {my_value}")
```

```
===Keys===
Dragon Ball
One Piece
Death Note
Full Metal Alchemist
Naruto
===Values===
Goku
Luffy
Yagami Light
Edward Elric
Naruto
===Key-Values===
Dragon Ball : Goku
One Piece : Luffy
Death Note : Yagami Light
Full Metal Alchemist : Edward Elric
Naruto : Naruto
```

PS 🔔: Are dictionaries ordered collection🤔?

From Python 3.7 dictionaries follow insertion order 😎

In python versions older than 3.7, the insertion of items is not ordered🙄. No problem 🙃, we still have OrderedDict(present in collections module) `from collections import OrderedDict` which does the same 😉

1.6 IDEs/Editors for Python

We have a lot of IDEs/Editors available for Python. Although we get **IDLE** abrevated as **I**ntegrated **D**evelopment and **L**earning **E**nvironment

IDLE gets installed automatically on Windows along with Python installation. On Mac or *nix operating systems we need install it manually

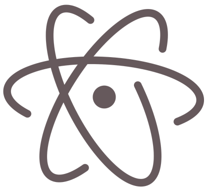
A few great IDEs/Editors for Python

PyCharm

Spyder

Visual Studio Code

Atom



Jupyter

Google Colab

This is my Personal Favourite when I need huge memory and GPU. We get those for free here 😎

PS 😉: I always say to prefer using basic text editor like notepad/gedit when learning a new language and use a good IDE if your Boss wants you to do the work quick 😜

1.7 User Input

`input` is a builtin function in Python, which prompts for the user to enter as standard input upto newline(`\n`).

`input` function always returns a string datatype, we need to typecast to respective datatype required.

Python 2.x's `input` is different from Python 3.x's `input` .

Python 2.x's `input` evaluates the string as a python command, like `eval(input())` .

In [1]:

```
user_entered = input("Hey Pythonist! Please enter anything: \n>>>")
print(user_entered)
```

```
Hello Pythoneer ♥
```

Let's try typecasting to integers we got from the user.

If the input is not a valid integer value, typecasting to integer raises `ValueError`

In [2]:

```
try:
    variable_1 = input("Enter variable 1 to be added: \n>>>")  # string
    variable_2 = input("Enter variable 2 to be added: \n>>>")  # string
    integer_1 = int(variable_1)  # Typecasting to integer
    integer_2 = int(variable_2)  # Typecasting to integer
    print(f"sum of {variable_1} and {variable_2} = {integer_1+integer_2}")
except ValueError as exc:
    print(f"👻 unable to typecast to integer: {exc}")
```

```
👻 unable to typecast to integer: invalid literal for int() with base 10: 'I
am not an Integer 😜'
```

1.8 Builtins

In [1]:

```
import builtins
```

We can see what all builtins does Python provide.

For our sake, we are traversing the complete list and printing the number and builtin attribute.

The function we are usign to traverse in `dir(builtins)` and get index and builtin attribute is `enumerate` which is also a bulitin 😉

In [2]:

```
for index, builtin_attribute in enumerate(dir(builtins)):
    print(f"{index}) {builtin_attribute}")
```

```
0) ArithmeticError
1) AssertionError
2) AttributeError
3) BaseException
4) BlockingIOError
5) BrokenPipeError
6) BufferError
7) BytesWarning
8) ChildProcessError
9) ConnectionAbortedError
10) ConnectionError
11) ConnectionRefusedError
12) ConnectionResetError
13) DeprecationWarning
14) EOFError
15) Ellipsis
16) EnvironmentError
17) Exception
18) False
19) FileExistsError
20) FileNotFoundError
21) FloatingPointError
22) FutureWarning
23) GeneratorExit
24) IOError
25) ImportError
26) ImportWarning
27) IndentationError
28) IndexError
29) InterruptedError
30) IsADirectoryError
31) KeyError
32) KeyboardInterrupt
33) LookupError
34) MemoryError
35) ModuleNotFoundError
36) NameError
37) None
38) NotADirectoryError
39) NotImplemented
40) NotImplementedError
41) OSError
42) OverflowError
43) PendingDeprecationWarning
44) PermissionError
45) ProcessLookupError
46) RecursionError
47) ReferenceError
```

```
48) ResourceWarning
49) RuntimeError
50) RuntimeWarning
51) StopAsyncIteration
52) StopIteration
53) SyntaxError
54) SyntaxWarning
55) SystemError
56) SystemExit
57) TabError
58) TimeoutError
59) True
60) TypeError
61) UnboundLocalError
62) UnicodeDecodeError
63) UnicodeEncodeError
64) UnicodeError
65) UnicodeTranslateError
66) UnicodeWarning
67) UserWarning
68) ValueError
69) Warning
70) ZeroDivisionError
71) __IPYTHON__
72) __build_class__
73) __debug__
74) __doc__
75) __import__
76) __loader__
77) __name__
78) __package__
79) __spec__
80) abs
81) all
82) any
83) ascii
84) bin
85) bool
86) breakpoint
87) bytearray
88) bytes
89) callable
90) chr
91) classmethod
92) compile
93) complex
94) copyright
95) credits
96) delattr
97) dict
98) dir
99) display
100) divmod
101) enumerate
102) eval
103) exec
104) filter
105) float
106) format
107) frozenset
108) get_ipython
109) getattr
110) globals
111) hasattr
112) hash
113) help
114) hex
115) id
116) input
```

```
117) int
118) isinstance
119) issubclass
120) iter
121) len
122) license
123) list
124) locals
125) map
126) max
127) memoryview
128) min
129) next
130) object
131) oct
132) open
133) ord
134) pow
135) print
136) property
137) range
138) repr
139) reversed
140) round
141) set
142) setattr
143) slice
144) sorted
145) staticmethod
146) str
147) sum
148) super
149) tuple
150) type
151) vars
152) zip
```

There's a difference between **Keywords** and **Builtins** 🤔. We can't assign a new object to the Keywords, if we try to do, we would be seeing an exception raised 🔴. But coming to builtins, we can assign any object to the builtin names, and Python won't have any issues, but it's not a good practice to do so 😇

1.9 Module

A module is a importable python file and can be created by creating a file with extension as `.py`

We can import the objects present in the module.

In the below 👇 example, we are importing `hello` function from `greet` module (greet.py)

`greet.py`

```python
"""Module to greet the user"""

import getpass


def hello():
    username: str = getpass.getuser().capitalize()
    print(f"Hello {username}. Have a great day :)")


if __name__ == "__main__":
    hello()
```

In [1]:

```python
from greet import hello
```

In [2]:

```python
hello()
```

```
Hello Root. Have a great day :)
```

let's have a look at the greet.py module. Well, we see the below `if` condition.

```python
if __name__ == "__main__":
    hello()
```

But why do we we need to have it🤔? We can just call the `hello` function at the end as

```python
hello()
```

Let's see the below👇 code to know why we use the first approach rather than the second.🙃

In [3]:

```python
import greet
```

🔍 The above code doesn't greet you 😢

In [4]:

```python
%run ./greet.py
```

```
Hello Root. Have a great day :)
```

But, this above code greets you😎.

The reason for this is, in the first snippet, we are importing a module called `greet`, so the actual code we are executing is in this REPL or Ipython shell.

Coming to second snippet, we are executing the `greet.py` directly.

Value of `__name__` would be "__main__" if we are executing a Python module directly. If we import a module(using the module indirectly) then value of `__name__` would be the relative path of the imported module. In the first example the `__name__` in the greet module would be "greet". As the "greet" is not equal to "__main__", that's the reason, we never went to the `if` condition when we imported greet module. 🙂

1.10 String representations of objects: str() vs repr()

`str()` and `repr()` are builtin functions used to represent the object in the form of string.

Suppose we have an object `x`.

`str(x)` would be calling the dunder (double underscore) `__str__` method of `x` as `x.__str__()`

`repr(x)` would be calling the dunder (double underscore) `__repr__` method of `x` as `x.__repr__()`

😑 Well, what all are these new terms `__str__` and `__repr__` 🤔?

As we know that Python is object oriented language, and so supports inheritance. In Python, all the classes would inherit from the base class `object`. `object` class has the methods `__str__`, `__repr__` and a lot more (which can be deepdived in someother notebook 😉). Hence every class would be having `__str__` and `__repr__` implicitly 😊

Python's official documentations states that `__str__` should be used to represent a object which is human readable(informal), whereas `__repr__` is used for official representation of an object.

In [1]:

```
from datetime import datetime

now = datetime.now()

print(f"The repr of now is: {repr(now)}")
print(f"The str of now is: {str(now)}")
```

```
The repr of now is: datetime.datetime(2021, 5, 28, 13, 19, 7, 751471)
The str of now is: 2021-05-28 13:19:07.751471
```

In [2]:

```
class ProgrammingLanguage:
    def __init__(self, language: str):
        self.language = language


language_obj = ProgrammingLanguage(language="Python")
print(f"The repr of language_obj is: {repr(language_obj)}")
print(f"The str of language_obj is: {str(language_obj)}")
```

```
The repr of language_obj is: <__main__.ProgrammingLanguage object at 0x7faa74
0420a0>
The str of language_obj is: <__main__.ProgrammingLanguage object at 0x7faa740
420a0>
```

In the above example we see that output to be something like:

```
The repr of language_obj is: <__main__.Language object at
0x7f1580c67190>
The str of language_obj is: <__main__.Language object at
0x7f1580c67190>
```

The address of the object might be different for everyone

Now let's try to override the `__str__` and `__repr__` methods and see how the representations work

In [3]:

```python
class Human:
    def __init__(self, name: str, age: int):
        self.name = name
        self.age = age

    # overriding __str__ method
    def __str__(self):
        return f"I am {self.name} of age {self.age}"

    # overriding __repr__ method
    def __repr__(self):
        return f"Human(name={self.name}, age={self.age}) object at {hex(id(se


human_obj = Human(name="IronMan", age=48)
print(f"The repr of human_obj is: {repr(human_obj)}")
print(f"The str of human_obj is: {str(human_obj)}")
```

```
The repr of human_obj is: Human(name=IronMan, age=48) object at 0x7faa74090be
0
The str of human_obj is: I am IronMan of age 48
```

We see that the result representations of the `human_obj` have been changed as we have overridden the `__str__` and `__repr__` methods 😊

1.11 Installing packages

Python has one of the largest programming community who build 3rd party packages and support community help ❤️.

That's pretty good, Now, how do we install the packages 🤔? We could use Python's package manager **PIP**.

Python's official 3rd party package repository is Python Package Index (PyPI) and its index url is https://pypi.org/simple

Here's how to use PIP in shell/terminal:

To search for a package:

```
pip search [package name]
```

To install a package: Install

```
pip install [package name]
```

Install a specific version

```
pip install [package name]==[version]
```

Install greater than a specific version

```
pip install [package name]>=[verion]
```

To uninstall a package

```
pip uninstall [package name]
```

Tidbits 🔔

There are modern ways of managing the dependencies using poetry, flit etc.. We will get to those soon...😊

1.12 Help Utility

Python has a builtin help utility which helps to know about the keywords, builtin functions, modules.

```
help()
```

You can pass keyword, bulitin function or Module to help function to know about the same.

In [1]:

```
import os
```

In [ ]:

```
# Help utility on the builtin module 'sys'
help(os)
```

snipped output:

```
Help on module os:

NAME
    os - OS routines for NT or Posix depending on what system we're
on.

MODULE REFERENCE
    https://docs.python.org/3.9/library/os

    The following documentation is automatically generated from the
Python
    source files.  It may be incomplete, incorrect or include
features that
    are considered implementation detail and may vary between Python
    implementations.  When in doubt, consult the module reference at
the
    location listed above.
```

In [3]:

```
# Help utility on getcwd function of sys module
help(os.getcwd)
```

```
Help on built-in function getcwd in module posix:

getcwd()
    Return a unicode string representing the current working directory.
```

🔔 Help function returns the docstrings associated with the respective Modules, Keywords or functions.