Table of Contents:

- Duck Typing
- MultiThreading

What is Python

1. It is a high-level, general-purpose programming language
2. It is used for: web development (server-side), software development, mathematics, data science, system scripting.
3. It's design philosophy is to enhance code-readability with the use of indentation
4. It is dynamically typed & auto garbage-collected language
5. It works on diferent operating system
6. It runs on an interpreter system, which means each line of code can be executed as soon as its written, i.e. supports quick prototyping
7. It is a multi-paradigm programming language, which means python supports Object Oriented Programming, Procedural Programming and Structured Programming

Python Packages

1. Package in Python is a folder & Module is a python file.
2. Python Package that can contains single or multiple sub-folders called as sub-modules, single or multiple python files.
3. Python module can contains several classes, functions, variables, etc.
4. On installation of a package it resides in the system and only when we import it, it gets loaded in RAM to avoid wastage of RAM.
5. Advantages of using packages are it reduces coding and increases code re-usability.

In [1]:

```
# Here 'math' is a module name

from math import sqrt  # Here we are importing the specific method under math module
print(sqrt(81))

import math  # Here we are importing the math module itself. And to call any method, we n
eed to use module name with it
print(math.sqrt(81))
```

```
9.0
9.0
```

Keywords in Python

1. Keywords are special reserved words in Python that have specific meanings and purposes
2. They can't be used for anything but its own defined purpose only.
3. They are in-built and need not to be imported.
4. There are 36 keywords in Python3.9

1	2	3	4	5	6	7	8	9
False	None	True	and	as	assert	async	pass	raise
await	break	class	continue	def	del	elif	return	try
else	except	finally	for	from	global	if	while	with
import	in	is	lambda	nonlocal	not	or	yield	**peg_parser**

In [2]:

```
# One way to get Python Keywords is using 'keyword' package
import keyword
kw = keyword.kwlist  # its an attribute and and not a function
print(kw)
print(f"Total numbers of Keywords are {len(kw)}")

# Another way to get Python Keywords is by using 'help()' function and passing 'keywords'
as arg.
help("keywords")
```

```
['False', 'None', 'True', '__peg_parser__', 'and', 'as', 'assert', 'async', 'await', 'bre
ak', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from
', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'rais
e', 'return', 'try', 'while', 'with', 'yield']
Total numbers of Keywords are 36

Here is a list of the Python keywords.  Enter any keyword to get more help.

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

Case sensitive

Python is a case sensitive language that means captial & small letters are treated differently

In [3]:

```
#None = 10  # cannot assign to None is the syntax error
none = 10
print(none)
```

```
10
```

In [4]:

```
marks1 = 80
Marks1 = 90

# total = marks1 + marks1
sum1 = marks1 + Marks1

print(sum1)
```

```
170
```

Identifiers

1. **Its the name we give to identify a variable, function, class, module or other object.**
2. **That means, whenever we want to give an entity a name, that's called 'identifier'**
3. **It can start with Alphanumeric character & underscore**
4. **It cannot start with a Digit or Any special characters**
5. **It cannot be a reserved keyword**

In [5]:

```
# Identifiers

_age = 10  # You can start identifier/variable name with underscore
Age = 10  # You can start identifier/variable name with Captial Alpha letter
age = 10  # You can start identifier/variable name with small Alpha letter
_123age = 10  # You can start identifier/variable name with underscore followed by any al
phanumeric characters
#123age = 10  # invalid syntax as it started with numeric value
```

```
#@123age = 10  # invalid syntax as it started with numeric value
```

Variables

1. **Definition:** In Python, a variable is a named location used to store data in memory.
2. **Declaration and Assignment:** Variables are declared by writing the variable name and assigning it a value using the equals sign (`=`). For example:

In [6]:

```
a = 5   # 'a' is a variable name
st = "hello"   # 'st' is a variable name
```

Comments

1. **Definition:** A comment in Python is a piece of text un your vode that is not executed. Its typically used to explain what the code is doing or leave notes fort developers who will be reading ot maintaing the code
2. **Single Line Comments**: Python uses a has symbol (#) to denote a comment. Anything written after Hash will be ignored by Python interpreter
3. **Multi-Line Comments:** Multi-Line strings using triple quotes(''' or """) can be used as multi-line comments, as any strings written within will be ignored by Python interpreter

In [7]:

```
# Add to numbers
a=4 # value of a is 4
b=5 # value of b is 5
c = a+b   # sum is in c
print(c)
```

```
9
```

In [8]:

```
def add_number(a,b):
    """
    Parameters:
    a (int): First variable
    b (int): Second variable

    Return:
    int: sum of variables a & b
    """
    return a+b
```

In [9]:

```
print(add_number.__doc__)   # to print the doc string
```

```
    Parameters:
    a (int): First variable
    b (int): Second variable

    Return:
    int: sum of variables a & b
```

In [10]:

```
# Multi Line Comment: We use Triple Quotes e.g. """ <text content> """

text = """Jose Marcial Portilla has a BS and MS in Mechanical Engineering from Santa Clar
a University and
years of experience as a professional instructor and trainer for Data Science, Machine Le
arning and
```

```
Python Programming. He has publications and patents in various fields such as microfluidi
cs,
materials science, and data science."""

print(text)
```

```
Jose Marcial Portilla has a BS and MS in Mechanical Engineering from Santa Clara Universi
ty and
years of experience as a professional instructor and trainer for Data Science, Machine Le
arning and
Python Programming. He has publications and patents in various fields such as microfluidi
cs,
materials science, and data science.
```

Indentation

1. **Importance:** **In Python, identation is not just for readability. Its part of the syntax and is used to indicate a block of code**
2. **Space Usage:** **Python uses identation to define the scope of loops, functions, classes, etc. Its Four Spaces or a Tab**
3. **Colon:** **Usually, a colon(:) at the end of a line is followed by an intended block of code.**
4. **Without properly indenting the Python code, you will end up seeing 'IndentationError' and the code will not get compiled.**
5. **Python Indentation is a way of telling a Python interpreter that the group of statements belongs to a particular block of code**

In [11]:

```
if True:
#print("This is True")   # IndentationError: expected an indented block
    print("This is True")
```

```
This is True
```

Statements

1. **Definition:** **A statement in Python is a logical instruction that the Python interpreter can read and execute.**
2. **In general, Any instruction written in the source code and executed by the Python interpreter is called a statement.**
3. **Types:** **Python includes several types of statements viz, assignment statements, conditional statement, looping statement, etc**

In [12]:

```
# Conditional statement

x = 2
if x > 0:
    print("{} is a positive number".format(x))


# Looping Statement

for i in range(3):
    print("I am {}".format(i+1))
```

```
2 is a positive number
I am 1
I am 2
I am 3
```

Multi-Line Statement

In Python, end of statement is identified by a newline character i.e '\n'. But we can make a statement extend

over multiple line.

1. **continuation character '\'**
2. **paranthetis ()**
3. **brackets []**
4. **braces {}**
5. **semi-colon ";"**

In [13]:

```
# continuation character.
g = "Geeks \
for \
Geeks"
print(g)
```

```
Geeks for Geeks
```

In [14]:

```
# Paranthesis

X = (1+2+4+5+6+7+1+2+4+5+6+7+
     1+2+4+5+6+7+1+2+4+5+6+7+1+
     2+4+5+6+7+1+2+4+5+6+7+1+2+
     4+5+6+7+1+2+4+5+6)
print(X)

Y = {1+2+4+5+6+7+1+2+4+5+6+7+
     1+2+4+5+6+7+1+2+4+5+6+7+1+
     2+4+5+6+7+1+2+4+5+6+7+1+2+
     4+5+6+7+1+2+4+5+6}
print(Y)

Z = [1+2+4+5+6+7+1+2+4+5+6+7+
     1+2+4+5+6+7+1+2+4+5+6+7+1+
     2+4+5+6+7+1+2+4+5+6+7+1+2+
     4+5+6+7+1+2+4+5+6]
print(Z)
```

```
193
{193}
[193]
```

Dynamic Typing

Python is dynamically typed, which means that you dont have to declare the type of a variable when you create one. You can chage the type of data held by a variable at any time throughout the life of program

In [15]:

```
#Dynamic Typing

a = 2  # 'a' contains integer
print("Datatype of 'a' ", type(a))
a = 'Abhi'  # now 'a' contains string and overrides previously stored integer
print("Datatype of 'a' ", type(a))
a = 4.34  # now 'a' contains float and overrides previously stored string
print("Datatype of 'a' ", type(a))
```

```
Datatype of 'a'  <class 'int'>
Datatype of 'a'  <class 'str'>
Datatype of 'a'  <class 'float'>
```

Data Types

Primitive Data Types:

1. **Integers:** Integers are whole numbers, without a fractional component. They can be positive or negative.
2. **Floats:** Floats represent real numbers and are written with a decimal point.
3. **Complex:** Complex data type consists of two values, the first one is the real part and the second is imaginary part. Its denoted as `3 + 7j`
4. **Strings:** Strings in Python are sequences of character data. They are created by enclosing characters in quotes.
5. **Boolean:** Boolean in Python is used to represent the truth value of an expression. True & False are the two types of it

Advanced Data Types:

1. **List:** Ordered collection, Mutuable, Hetergoenous data, supports Indexing & Slicing
2. **Tuple:** Ordered collection, Immutuable, Hetergoenous data, supports Indexing & Slicing
3. **Dictionary:** Ordered collection, Mutable, Key-Value pair, Values can be Hetergoenous data, Keys are immutable
4. **Set:** Unordered collection, Mutable, Contains Unique value only, Hetergoenous data

In [16]:

```
a = 10   #'int' datatype
print(f"Data Type is {type(a)}")

b = 10.10   #'float' datatype
print(f"Data Type is {type(b)}")

c = "Hello 123"   #'str' datatype
print(f"Data Type is {type(c)}")

d = True #'bool' datatype
print(f"Data Type is {type(d)}")
```

```
Data Type is <class 'int'>
Data Type is <class 'float'>
Data Type is <class 'str'>
Data Type is <class 'bool'>
```

Lists

Definition: A list in Python is an ordered collection (also known as a sequence) of items. Its Heterogenous in nature.

Mutable: Lists are mutable, which means their elements can be changed (added, modified, or deleted) after they are created.

Creation: A list is created by placing items (elements) inside square brackets [], separated by commas.

Indexing and Slicing: Lists support indexing and slicing to access and modify their elements. Python list indices start at 0.

In [17]:

```
d = [1, 2.2, "data", [3, "data1"]] # Here 'd' is of 'list' datatype | Ordered | Mutable
|
print(type(d))
print("List: ", d)

print("Indexing: Data in list at position 1 is ", d[1])
print("Slicing: Data in list at between position 1 to 4 is ", d[1: 5])
```

```
<class 'list'>
List:  [1, 2.2, 'data', [3, 'data1']]
Indexing: Data in list at position 1 is  2.2
Slicing: Data in list at between position 1 to 4 is  [2.2, 'data', [3, 'data1']]
```

Tuples

Definition: A tuple in Python is similar to a list. It's an ordered collection of items. Heterogenous in nature

Immutable: The major difference from lists is that tuples are immutable, which means their elements cannot be changed (no addition, modification, or deletion) after they are created.

Creation: A tuple is created by placing items (elements) inside parentheses `()`, separated by commas.

Indexing and Slicing: Lists support indexing and slicing to access and modify their elements. Python list indices start at 0.

In [18]:

```
e = (1, 2.2, "data", [3, "data1"]) # Here 'e' is of 'tuple' datatype | Ordered | Immutab
le | Heterogenous in nature
print(type(e))
print("Tuple: ", e)


print("Indexing: Data in list at position 1 is ", e[1])
print("Slicing: Data in list at between position 1 to 4 is ", e[1: 5])
```

```
<class 'tuple'>
Tuple:  (1, 2.2, 'data', [3, 'data1'])
Indexing: Data in list at position 1 is  2.2
Slicing: Data in list at between position 1 to 4 is  (2.2, 'data', [3, 'data1'])
```

Dictionaries

Definition: A dictionary in Python is an unordered collection of items. (from 3.7+, its a ordered collection of items) Each item stored in a dictionary has a key and value, making it a key-value pair.

Mutable: Dictionaries are mutable, which means you can change their elements. You can add, modify, or delete key-value pairs from a dictionary.

Creation: A dictionary is created by placing items (key-value pairs) inside curly braces `{}`, separated by commas. Each item is a pair made up of a key and a value, separated by a colon `:`

Ordered | Immutable Keys | Mutable Values | Heterogenous in nature

Unique Keys: Each key in a dictionary should be unique. If a dictionary is created with duplicate keys, the last assignment will overwrite the previous ones.

Accessing Values: You can access a value in a dictionary by providing the corresponding key inside square brackets `[]`.

Updating Values: You can update the value for a particular key by using the assignment operator `=`.

Adding and Deleting Key-Value Pairs: You can add a new key-value pair simply by assigning a value to a new key. You can delete a key-value pair using the `del` keyword.

In [19]:

```
# Here 'f' is of 'dictionary' datatype | Key-Value pair | Unordered | Immutable Keys | Mu
table Values | Heterogenous in nature
f = {1: "data1", 2.2: 3.4, "key1": "value1",(4,5): [6, 7, 8]}
print(type(f))
print("Dictionary: ", f)

t = {'fruit1': 'apple', 'fruit2': 'guava', 'fruit3': 'banana', 'fruit2': 'pineapple'}
print("Dictionary keys should be unique. If not, the last assignment will overwrite the p
revious ones.", t)

print("Accessing dictionary using key 'fruit1', who's value is: ", t.get('fruit1'))

t['fruit3'] = 'papaya'
print("Updated dictionary: ", t)

t['fruit4'] = "sugarcane"
print("Added new key-value to dictionary: ", t)
t['fruit5'] = "kiwi"
```

```
print("Added new key-value to dictionary: ", t)

# Deleting key-value pair from the dictionary
del t['fruit4']
print("Dicitonary: ", t)
```

```
<class 'dict'>
Dictionary:  {1: 'data1', 2.2: 3.4, 'key1': 'value1', (4, 5): [6, 7, 8]}
Dictionary keys should be unique. If not, the last assignment will overwrite the previous
ones. {'fruit1': 'apple', 'fruit2': 'pineapple', 'fruit3': 'banana'}
Accessing dictionary using key 'fruit1', who's value is:  apple
Updated dictionary:  {'fruit1': 'apple', 'fruit2': 'pineapple', 'fruit3': 'papaya'}
Added new key-value to dictionary:  {'fruit1': 'apple', 'fruit2': 'pineapple', 'fruit3':
'papaya', 'fruit4': 'sugarcane'}
Added new key-value to dictionary:  {'fruit1': 'apple', 'fruit2': 'pineapple', 'fruit3':
'papaya', 'fruit4': 'sugarcane', 'fruit5': 'kiwi'}
Dicitonary:  {'fruit1': 'apple', 'fruit2': 'pineapple', 'fruit3': 'papaya', 'fruit5': 'ki
wi'}
```

Sets: A set is an unordered collection of items where every element is unique.

In [20]:

```
g = {1, 3, 4, 5, 6, 4, 3, 9, 1, "abc", "gef", "abc"}  # Here 'g' is of 'set' datatype |
Unordered | Mutable | Unique values | Heterogenous in nature
print(type(g))
print("Set: ", g)
```

```
<class 'set'>
Set:  {1, 3, 4, 5, 6, 'abc', 'gef', 9}
```

Advanced Set Operations

In [13]:

```
# difference
art = {"Bob", "Rolf", "Anne", "Charlie"}
science = {"Bob", "Jane", "Adam", "Anne"}

only_science = science.difference(art)  # print(science - art)
print(only_science)

# union
all_science_art = science.union(art) # print(science | art)
print(all_science_art)

# intersection
both = science.intersection(art) # print(science & art)
print(both)
```

```
{'Adam', 'Jane'}
{'Bob', 'Rolf', 'Jane', 'Anne', 'Adam', 'Charlie'}
{'Anne', 'Bob'}
```

List Comprehension

In [21]:

```
l = [1, 2, 3, 4, 5, 6]
lc = [i**2 for i in l]
print(lc)
```

```
[1, 4, 9, 16, 25, 36]
```

Dictionary Comprehension

In [22]:

```
d = {'A': 1, 'B': 2, 'C': 3, 'D': 4, 'E': 5, 'F': 6}
dc = {k: v**3 for k, v in d.items()}
print(dc)
```

```
{'A': 1, 'B': 8, 'C': 27, 'D': 64, 'E': 125, 'F': 216}
```

Standard Output

print() is a standard output function which displays anything passed as an argument to the function

Standard Input

input() is a standard input function which accepts anything passed as an argument to the function in string format

In [23]:

```
name = input("Enter your name ")
print("Datatype of input data ", type(name))
print(f"How are you, {name}")
```

Operators

Arthmetic / Mathametical operators

+, -, /, *, //, %, **

In [ ]:

```
a = 10
b = 3

print("a + b: ", a+b)
print("a - b: ", a-b)
print("a * b: ", a*b)
print("a / b: ", a/b)
print("a // b: ", a//b)
print("a % b: ", a%b)
print("a ** b: ", a**b)
```

```
a + b:  13
a - b:  7
a * b:  30
a / b:  3.3333333333333335
a // b:  3
a % b:  1
a ** b:  1000
```

Logical Operators

and, or, not

In [ ]:

```
print("AND operator")
print(True and True)
print(True and False)
print(False and True)
print(False and False)

print("OR operator")
print(True or True)
print(True or False)
print(False or True)
print(False or False)

print("NOT operator")
print(not True)
print(not False)
```

```
AND operator
True
False
False
False
OR operator
True
True
True
False
NOT operator
False
True
```

Bitwise Operator

&, |, ^, >>, <<

In [ ]:

```
a = 10  # 1010
b = 4  # 0100

# Bitwise AND
print("a & b ", a&b)  # 1010 and 0100 --> 0000 --> 0 (apply 'and' operator bit against b
it)
# 10 = 1 0 1 0
# 4  = 0 1 0 0
# =============
#      0 0 0 0 = 0

# Bitwise OR
print("a | b ", a|b)  # 1010 or 0100 --> 1110  --> 14(apply 'or' operator bit against bi
t)
# 10 = 1 0 1 0
# 4  = 0 1 0 0
# =============
#      1 1 1 0 = 14


# Bitwise XOR --> compares each bit of 1st operand with the 2nd
# If one of the bit is 1 (but not both) , the corresponding result bit is set to 1, other
wise set to 0
print("a ^ b ", a^b)

# 10 = 1 0 1 0
# 4  = 0 1 0 0
# =============
#      1 1 1 0 = 14

# 10 = 1 0 1 0
# 4  = 1 1 0 0
# =============
#      0 1 1 0 = 6


# Bitwise Right Shift
a = 10
print("a >> 2 ", a >> 2)
# 10 = 1010
# >>= 0010 --> 2


# Bitwise Left Shift
```

```
a = 10
print("a << 2 ", a << 2)
# 10 = 1010
# <<= 101000 --> 40  i.e 32+0+8+0+0+0   i.e 2^5 + 2^4 + 2^3 + 2^2 + 2^1 +2^0

a = 10
print("a << 3 ", a << 3)
# 10  =     010
# 80  = 1010000
```

```
a & b  0
a | b  14
a ^ b  14
a >> 2  2
a << 2  40
a << 3  80
```

Trick: To get the binary value of an integer

In [ ]:

```
bin(10)
```

Out[ ]:

```
'0b1010'
```

Control Flow Statements

if-else

The if/else statement executes a block of code if a specified condition is true. If the condition is false, another block of code will be executed.

In [5]:

```
x = 2  # x is assigned a value of 2
y = 4  # y is assigned a value of 4
if x < y:
    print(f"x: {x} is less than y: {y}")
else:
    print(f"x: {x} is greater than y: {y}")
```

```
x: 2 is less than y: 4
```

while loop

A while loop is used to execute a block of statements repeatedly until a given condition is satisfied.

And when the condition becomes false, the line immediately after the loop in the program is executed.

In [9]:

```
counter = 0

while counter < 5:
    print(f"{counter}: I enjoy coding!")
    counter += 1
else:  # The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.
    print("While loop ended")
```

```
0: I enjoy coding!
1: I enjoy coding!
2: I enjoy coding!
3: I enjoy coding!
4: I enjoy coding!
While loop ended
```

for loop

For loops are used for sequential traversal

In [10]:

```
for counter in range(5):
    print(f"{counter}: I enjoy coding!")
```

```
0: I enjoy coding!
1: I enjoy coding!
2: I enjoy coding!
3: I enjoy coding!
4: I enjoy coding!
```

USECASE of WHILE and FOR loops

Both `for` loop and `while` loop is used to execute the statements repeatedly while the program runs.

The major difference between `for` loop and the `while` loop is that

`for` loop is used when the number of iterations is known, whereas execution is done in the `while` loop until the statement in the program is proved wrong.

break and continue

1	2
Break statement stops the entire process of the loop	**Continue statement only stops the current iteration of the loop**
Break also terminates the remaining iterations	**Continue doesn't terminate the next iterations; it resumes with the successive iterations**

In [13]:

```
# break

for i in range(5):
    print(f"Iteration count: {i}")
    if i >= 3:
        print("Breaking for loop")
        break
```

```
Iteration count: 0
Iteration count: 1
Iteration count: 2
Iteration count: 3
Breaking for loop
```

In [20]:

```
# continue

for i in range(5):
    if i == 3:
        print("Continuing with next iteration")
        continue # skips the current iteration
    print(f"{i}: Some more operation...")
```

```
0: Some more operation...
1: Some more operation...
2: Some more operation...
Continuing with next iteration
4: Some more operation...
```

Functions

Advantags of having functions:

1. Makes programming fast
2. Reusability

Built-In Functions

In [ ]:

```
# len
l = len('abc')   # 3

# print
print(l)   # prints the given input
```

```
3
```

User Defined Functions

Function with no arguments

In [ ]:

```
def introduction():
    print("Hello, this is my first function")

introduction()
```

```
Hello, this is my first function
```

Function with positional arguments

In [ ]:

```
def function1(firstname, lastname):
    print(f"First Name: {firstname}")
    print(f"Last Name: {lastname}")
```

In [ ]:

```
# function1("Abhijit")   # TypeError: function1() missing 1 required positional argument:
'lastname'
function1("Abhijit", "Paul")
```

```
First Name: Abhijit
Last Name: Paul
```

Functions with *args

***args specifies the number of non-keyword arguments that can be passed to the function.**

During the function call, user can pass 'n' number of arguments and while perform the operations all of them will be considered

In [ ]:

```
def calculate_sum(*args) -> int:
    """This function is capable to calculate sum of 'n' number of arguments provided

    Returns:
        int: sum value
    """
    print(f"Number of arguments: {len(args)}")
    return f"Sum: {sum(args)}"

print(calculate_sum(2,3))
print(calculate_sum(2,3,5))
```

```
Number of arguments: 2
Sum: 5
Number of arguments: 3
Sum: 10
```

Functions with **kwargs (Keyword Arguments)

****kwargs allows us to pass any number of keyword arguments.**

In [24]:

```
def makeSentence(**kwargs):
    sentence=''
    for word in kwargs.values():
        sentence+=word
    return sentence

print('Sentence:', makeSentence(a='Hi ',b='there!'))
print('Sentence:', makeSentence(a='Kwargs ',b='are ', c='awesome!'))
print('Sentence:', makeSentence(**{'a': "Dictionaries ", "b": "can also ", "c": "be used
"}))
```

```
Sentence: Hi there!
Sentence: Kwargs are awesome!
Sentence: Dictionaries can also be used
```

Recursion Function

A function that calls itself is said to be recursive, and the technique of employing a recursive function is called recursion.

When we call a function in Python, the interpreter creates a new local namespace so that variables defined within that function don't collide with identical variable defined elsewhere.

The same holds true if multiple instances of the same function are running concurrently.

In [ ]:

```
def function():
    x = 10
    function()
```

When function() executes the first time, Python creates a namespace and assigns x the value 10 in that namespace. Then function() calls itself recursively.

The second time function() runs, the interpreter creates a second namespace and assigns 10 to x there as well.

These two instances of the name x are distinct from each other and can coexist without clashing because they are in separate namespaces.

In [ ]:

```
# But the main problem with recursion is that we need to set some condition for the recur
sion to get over or stop,
# else it will throw `RecursionError: maximum recursion depth exceeded`

# Here on calling function(), it keep calling itself until the threshold limit of recursi
on is hit

function()
```

```
---------------------------------------------------------------------------
RecursionError                            Traceback (most recent call last)
```

```
Cell In[52], line 1
----> 1 function()

Cell In[48], line 3, in function()
      1 def function():
      2     x = 10
----> 3     function()

Cell In[48], line 3, in function()
      1 def function():
      2     x = 10
----> 3     function()

    [... skipping similar frames: function at line 3 (2970 times)]

Cell In[48], line 3, in function()
      1 def function():
      2     x = 10
----> 3     function()

RecursionError: maximum recursion depth exceeded
```

In [ ]:

```
# Decreasing Counter
def countdown(n):
    print(n)
    if n == 0:
        return
    else:
        countdown(n-1)

countdown(5)
```

```
5
4
3
2
1
0
```

In [ ]:

```
# Factorial using recursion
def factorial(n: int):
    if n < 2:
        return 1
    else:
        return n * factorial(n-1)

print(factorial(4))
```

```
24
```

In [ ]:

```
# Sum of Fibonacci series for a given 'n' position
# 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
def fibonacci(n: int):
    if n <= 1:
        return n
    return fibonacci(n-1) + fibonacci(n-2) + 1

print(fibonacci(7))
```

```
33
```

Get & Set Recursion Limit

In [ ]:

```
from sys import getrecursionlimit
print(getrecursionlimit())
```

```
3000
```

In [ ]:

```
from sys import setrecursionlimit
setrecursionlimit(2000)
print(getrecursionlimit())
```

```
2000
```

Functional Programming

- **Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style.**
- **It is a declarative type of programming style.**
- **Its main focus is on "what to solve" in contrast to an imperative style where the main focus is "how to solve".**
- **It uses expressions instead of statements. An expression is evaluated to produce a value whereas a statement is executed to assign variables.**

To support functional programming, it's useful if a function in a given programming language has two abilities:

- **To take another function as an argument**
- **To return another function to its caller**

Different types of special functions

- **lambda**
- **map**
- **filter**
- **reduce**
- **sorted**
- **Iterator**
- **Generator**

Lambda Function

is a small anonymous function which can take any number of arguments, but can only have one expression. e.g. lambda argument(s) : expression

- **Regular function can have more than one expressions whereas Lambda function can have only one.**
- **Regular function has a function name, whereas Lambda function doesnt.**

Lambda function can be useful when we need to create a function which has only one expression irrespective of number of arguments and has no requirement of repeatability

In [ ]:

```
# lambda function with one argument
even_or_odd = lambda x: 'even' if x%2==0 else 'odd'
print(even_or_odd(4))
print(even_or_odd(11))
```

```
even
odd
```

In [ ]:

```
# lambda function with two argument
max_num = lambda x,y: x if x>y else y
print(max_num(4,9))
```

```
9
```

map()

- It is a built-in function that allows us to process and transform all the items in an iterable without using an explicit for loop.
- This technique is known as mapping.
- It is useful when we need to apply a transformation function to each item in an iterable and transform them into a new iterable.
- Two arguments: map(function, iterable)

In [ ]:

```
def square(num):
    return num ** 2

numbers = [2,4,6,8]
print(list(map(square, numbers)))
```

```
[4, 16, 36, 64]
```

Using lambda with map

In [ ]:

```
l1 = [1,2,3]
l2 = [9,6,5]
print(list(map(lambda x,y: x-y, l2, l1)))
```

```
[8, 4, 2]
```

filter()

- Its a built in function that yields the items of the input iterable for which function returns True.
- It means that filter() will check the truth value of each item in iterable and filter out all of the items that are falsy
- Two arguments: filter(function, iterable)

In [ ]:

```
# Suppose we need to filter out (-ve) value from the list of mixed values before getting
the squared root of them
import math

data = [49, 36, -9, 81]

# Here we are filtering out the (-ve) or unwanted values first
print(list(filter(lambda x: x>0, data)))

# As a next step we can now calculate the squared root using map()
print(list(map(math.sqrt, filter(lambda x: x>0, data))))
```

```
[49, 36, 81]
[7.0, 6.0, 9.0]
```

In [ ]:

```
numbers = [1,2,3,4,5,6,7,8]

def even(a):
    return a%2==0

result = filter(even, numbers)
print(list(result))
```

```
#alternative way
result = filter(lambda a: a%2==0, range(20))
print(list(result))
```

```
[2, 4, 6, 8]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

reduce()

- **It performs functional computation by taking a function and an iterable (e.g., list, tuple, dictionary, etc.) as arguments**
- **It doesn't return multiple values or any iterator, it just returns a single value as output which is the result of the whole iterable getting reduced to only a single integer or string or boolean.**

In [ ]:

```
# importing functools for reduce()
import functools

# initializing list
l = [1, 3, 5, 6, 2]

# using reduce to compute sum of list
print("The sum of the list elements is : ", end="")
print(functools.reduce(lambda a, b: a+b, l))

# using reduce to compute maximum element from list
print("The maximum element of the list is : ", end="")
print(functools.reduce(lambda a, b: a if a > b else b, l))
```

```
The sum of the list elements is : 17
The maximum element of the list is : 6
```

sorted()

- **It returns a sorted list of the specified iterable object (e.g list, tuple)**

In [ ]:

```
# Here you have a list of dictionaries,
# and you need to sort in reverse order the list on the basis of 'price' in the dictionar
y
products = [{'name': 'Phone', 'price': 10000, 'location': 'Pune'},
            {'name': 'Printer', 'price': 20000, 'location': 'Mumbai'},
            {'name': 'Laptop', 'price': 40000, 'location': 'Ahmedabad'},
            {'name': 'Headphone', 'price': 5000, 'location': 'Gujrat'}]

for item in (sorted(products, key = lambda x: x['price'], reverse=True)):
    print(item)
```

```
{'name': 'Laptop', 'price': 40000, 'location': 'Ahmedabad'}
{'name': 'Printer', 'price': 20000, 'location': 'Mumbai'}
{'name': 'Phone', 'price': 10000, 'location': 'Pune'}
{'name': 'Headphone', 'price': 5000, 'location': 'Gujrat'}
```

In [ ]:

```
number = [1, 2, 3, 4, 5, 6, 7, 8]
print(list(filter(lambda x: x%2==0, number)))
```

```
[2, 4, 6, 8]
```

iter() --> Iterators

- **This method returns an iterator for the given argument which is a iterable**
- **One argument: iter(iterable)**

In [1]:

```
numbers = [1,2,3,4,5,6,7,8]

iteration = iter(numbers)

# Practical use case of iter()
for _ in range(100):
    try:
        print(next(iteration))   # using next() would give first available value in itera
tor object queue. Once the value is read, its delete from the queue.
    except StopIteration:   # Once all the values are read from iterator queue, accessing
next() would throw 'StopIteration' exception
        break
```

```
1
2
3
4
```

Generator

- **Its an elegant and simple way to create custom iterators in Python**
- **A generator is a function that returns an iterator that produces a sequence of values when iterated over.**
- **It is useful when we want to produce a large sequence of values, but we don't want to store all of them in memory at once. (Memory Efficient approach)**
- **Its defined as a normal function with `def`, but instead of `return` statement it has `yield`**
- **The yield keyword is used to produce a value from the generator and pause the generator function's execution until the next value is requested.**

In [9]:

```
#Generating data stream upto maximum value without storing in memory and use as much as r
equired
def even_generator(max=2):
    n = 2
    while n <= max:
        yield n
        n += 2
result = even_generator(4)

print(next(result))  # --> max=2 -> yields n=2 --> prints 2 and n=2+2
print(next(result))  # --> max=2 -> yields n=4 --> prints 4 and n=4+2
print(next(result))  # --> max=2 -> yields n=6 --> n <= max condition becomes False since
n = 6 and max = 4, so StopIteration exception is raised
```

```
2
4

---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Cell In[9], line 11
      9 print(next(result)) # --> max=2 -> yields n=2 --> prints 2 and n=2+2
     10 print(next(result)) # --> max=2 -> yields n=4 --> prints 4 and n=4+2
---> 11 print(next(result)) # --> max=2 -> yields n=6 --> n <= max condition becomes Fals
e since n = 6 and max = 4, so StopIteration exception is raised

StopIteration:
```

In [ ]:

```
# Generating Fibonacci series with `generators`

def generate_fibonacci():
    n1 = 0
    n2 = 1
    while True:
        yield n1
        n1 , n2 = n2, n1 + n2

seq = generate_fibonacci()
print(next(seq))
print(next(seq))
print(next(seq))
print(next(seq))
print(next(seq))
```

```
0
1
1
2
3
```

Exception Handling

Some of the most common types of exceptions are:

- `ZeroDivisionError`: Raised when the second argument of a division or modulo operation is zero.
- `TypeError`: Raised when an operation or function is applied to an object of inappropriate type.
- `ValueError`: Raised when a built-in operation or function receives an argument that has the right type but an inappropriate value.
- `IndexError`: Raised when a sequence subscript is out of range.
- `KeyError`: Raised when a dictionary key is not found.
- `FileNotFoundError`: Raised when a file or directory is requested but doesn't exist.
- `IOError`: Raised when an I/O operation (such as a print statement, the built-in open() function or a method of a file object) fails for an I/O-related reason.
- `ImportError`: Raised when an `import` statement fails to find the module definition or when a `from ... import` fails to find a name that is to be imported.
- `MemoryError`: Raised when an operation runs out of memory.
- `OverflowError`: Raised when the result of an arithmetic operation is too large to be expressed by the normal number format.
- `AttributeError`: Raised when an attribute reference or assignment fails.
- `SyntaxError`: Raised when the parser encounters a syntax error.
- `IndentationError`: Raised when there is incorrect indentation.
- `NameError`: Raised when a local or global name is not found.

```
Role of Try and Except:
```

- `try` **block:** The code within the `try` block contains the statements that may potentially raise an exception. It allows you to specify the section of code that you want to monitor for exceptions.
- `except` **block:** If an exception occurs within the `try` block, the corresponding `except` block(s) are executed. The `except` block allows you to define the actions or code that should be executed when a specific exception is raised. You can have multiple `except` blocks to handle different types of exceptions.
- The **else** block allows you run code without errors.
- The **finally** block executes code regardless of the try-and-except blocks.
- Use the **raise** keyword to throw (or raise) an exception.

In [ ]:

```
# try raise an exception bec x is not defined
try:
  print(x)
except:
  print("Some issue with x")
```

```
2
```

In [ ]:

```
"""You can use the "else" keyword to specify a block
   of code that will be performed if no errors are raised:"""
try:
  print("Good morning today is 17th June")
except:
  print("Some issue")
else:
  print("No issues")
```

```
Good morning today is 17th June
No issues
```

In [ ]:

```
"""If the "finally" block is supplied,
   it will be executed whether or not the try block raises an error."""

try:
  x = 2
  print(x)
except:
  print("There is no X")
finally:
  print("The 'try except' executed")
```

```
2
The 'try except' executed
```

In [ ]:

```
# Use the "raise" keyword to throw an exception.

x = 2

if x < 10:
  raise Exception("There is a problem: X is below zero")
```

```
---------------------------------------------------------------------------

Exception                                 Traceback (most recent call last)

Cell In[2], line 6

      3 x = 2

      5 if x < 10:

----> 6   raise Exception("There is a problem: X is below zero")



Exception: There is a problem: X is below zero
```

ZeroDivisionError

In [ ]:

```
n = int(input("Please enter the numerator: "))
d = int(input("Please enter the denominator: "))

result = n / d
print("Result:", result)
```

```
---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)
```

```
Cell In[3], line 4

      1 n = int(input("Please enter the numerator: "))

      2 d = int(input("Please enter the denominator: "))

----> 4 result = n / d

      5 print("Result:", result)


ZeroDivisionError: division by zero
```

In [ ]:

```
try:
    n = int(input("Please enter the numerator: "))
    d = int(input("Please enter the denominator: "))

    result = n / d
    print("Result:", result)

except ZeroDivisionError:
    print("There is an Error: Division by zero is not allowed.")
```

```
There is an Error: Division by zero is not allowed.
```

ValueError

In [ ]:

```
"""Raised when a built-in operation or function receives an
   argument that has the right type but an inappropriate value."""

n = int(input("Please enter the numerator: "))
d = int(input("Please enter the denominator: "))

result = n / d
print("Result:", result)
```

```
Result: 3.0
```

In [ ]:

```
try:
    n = int(input("Please enter the numerator: "))
    d = int(input("Please enter the denominator: "))

    result = n / d
    print("Result:", result)

except ValueError:
    print("Please enter valid integers for the numerator and denominator.")
```

```
Please enter valid integers for the numerator and denominator.
```

In [ ]:

```
# Multiple exceptions

try:
    n = int(input("Please enter the numerator: "))
    d = int(input("Please enter the denominator: "))

    result = n / d
    print("Result:", result)

except ValueError:
    print("Please enter valid integers for the numerator and denominator.")
```

```
except ZeroDivisionError:
    print("Division by zero is not allowed.")
```

```
Please enter the numerator: 2
Please enter the denominator: 2
Result: 1.0
```

TypeError

In [ ]:

```
try:
    x = "10"  # Assigning a string value to the variable x
    y = 2

    z = x + y  # Attempt addition between a string and an integer

    print("Result:", z)

except TypeError:
    print("Error: TypeError occurred.")
```

```
Error: TypeError occurred.
```

IndexError

In [ ]:

```
try:
    list1 = [1, 2, 3]

    print(list1[3])  # Trying to reach an index that is out of range

except IndexError:
    print("IndexError error occurred.")
```

```
IndexError error occurred.
```

KeyError

In [ ]:

```
try:
    my_dictionary = {"name": "John", "age": 30}

    print(my_dictionary["city"])  # Accessing a KEY which does not exist

except KeyError:
    print("KeyError error occurred.")
```

```
KeyError error occurred.
```

FileNotFoundError

In [ ]:

```
try:
    file_location = "my_file.txt"

    with open(file_location, "r") as file:
        contents = file.read()

except FileNotFoundError:
    print(f"File '{file_location}' not found.")
```

```
File 'my_file.txt' not found.
```

IOError

In [ ]:

```
try:
    file_location = "file123.txt"

    with open(file_location, "r") as file:
        file.write("This is my file")

except IOError:
    print(f"Unable to write to file '{file_location}'.")
```

```
Unable to write to file 'file123.txt'.
```

ImportError

In [ ]:

```
try:
    import library1234

except ImportError:
    print("Unsuccessful to import module 'library1234'.")
```

```
Unsuccessful to import module 'library1234'.
```

MemoryError

In [ ]:

```
# Creating a large list that consumes a significant amount of memory
try:
    large_list = [1] * (10 ** 12)

    """The phrase [1] * (10 ** 12) generates a
    list by repeatedly repeating the element [1]."""

except MemoryError:
    print("Insufficient memory for the list.")
```

```
Cannot execute code, session has been disposed. Please try restarting the Kernel.

The Kernel crashed while executing code in the the current cell or a previous cell. Pleas
e review the code in the cell(s) to identify a possible cause of the failure. Click <a hr
ef='https://aka.ms/vscodeJupyterKernelCrash'>here</a> for more info. View Jupyter <a href
='command:jupyter.viewOutput'>log</a> for further details.

Cannot execute code, session has been disposed. Please try restarting the Kernel.

The Kernel crashed while executing code in the the current cell or a previous cell. Pleas
e review the code in the cell(s) to identify a possible cause of the failure. Click <a hr
ef='https://aka.ms/vscodeJupyterKernelCrash'>here</a> for more info. View Jupyter <a href
='command:jupyter.viewOutput'>log</a> for further details.
```

OverflowError

In [ ]:

```
"""Raised when the result of an arithmetic operation is too large
    to be expressed by the normal number format."""
```

```
try:
    result = 5000 ** 100  # Attempting to calculate an extremely large number

except OverflowError:
    print("Error: Calculation resulted in an overflow.")
except ValueError:
    print("Exceeds the limit (4300) for integer string conversion; ")
else:
    print(result)
```

```
78886090522101180541172856528278622967320643510902300477027893066406250000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000
```

AttributeError

In [ ]:

```
try:

    age = 20

    # Using append method
    age.append(12)

except AttributeError:
    print("'age' object has no attribute.")
else:
    print(result)
```

```
'age' object has no attribute.
```

SyntaxError

In [ ]:

```
try:
    print("John age is:"
    age = 20

except SyntaxError:
    print("Error: Invalid syntax.")
```

```
  Cell In[14], line 3

    age = 20

    ^

SyntaxError: invalid syntax
```

IndentationError

In [ ]:

```
try:
    name = "John"
        age = 50

except IndentationError:
    print("There is an IndentationError")
```

```
  Cell In[15], line 3

    age = 50
    ^

IndentationError: unexpected indent
```

NameError

In [ ]:

```python
try:
    side = "4"
    print(area)   # Attempting to access an undefined variable

except NameError:
    print("NameError is there")
```

```
NameError is there
```

In [ ]:

```python

```

User-defined Exceptions

By deriving a new class from the default Exception class in Python, we can define our own exception types.

In [ ]:

```python
class MyCustomError(Exception):
    pass
```

In the above code, `MyCustomError` is derived from the built-in `Exception` class. You can use this in your code by using the `raise` statement.

In [ ]:

```python
raise MyCustomError("This is a custom error")
```

```
---------------------------------------------------------------------------

MyCustomError                             Traceback (most recent call last)

Cell In[3], line 1

----> 1 raise MyCustomError("This is a custom error")



MyCustomError: This is a custom error
```

Custom Exception

In [26]:

```python
# define user-defined exceptions
class WrongAge(Exception):
    "Raised when the input value is less than 100"
    pass
```

In [27]:

```python
# you need to guess this number
n = 18
```

```
try:
    input_num = 16
    if input_num < n:
        raise WrongAge # calling your custom exception
    else:
        print("You can work")
except WrongAge:
    print("Invalid Age: You are not allowed to work")
```

```
Invalid Age: You are not allowed to work
```

Logging

- **Logging is a technique for monitoring events that take place when some software is in use.**
- **For the creation, operation, and debugging of software, logging is crucial.**
- **There are very little odds that you would find the source of the issue if your programme fails and you don't have any logging records.**
- **Additionally, it will take a lot of time to identify the cause.**

In [1]:

```
# first import the logging library
import logging

""" In the code above, we first import the logging module, then we call the
    basicConfig method to configure the logging.

    We set the level to DEBUG, which means that all logs of level
    DEBUG and above will be tracked."""

logging.basicConfig(level=logging.DEBUG)

# Logging Level: severity of the event being logged
# Least severe to most severe  DEBUG > INFO > WARNING > ERROR > CRITICAL
logging.debug('This is a debug message')
logging.info('This is an info message')
logging.warning('This is a warning message')
logging.error('This is an error message')
logging.critical('This is a critical message')
```

```
DEBUG:root:This is a debug message
INFO:root:This is an info message
WARNING:root:This is a warning message
ERROR:root:This is an error message
CRITICAL:root:This is a critical message
```

- **Some programmers utilise the idea of "Printing" the statements to check whether they were correctly performed or if an error had occurred.**
- **However, printing is not a smart move. For basic scripts, it might be the answer to your problems, however the printing solution will fall short for complex programmes.**
- **A built-in Python package called logging enables publishing status messages to files or other output streams. The file may provide details about which portion of the code is run and any issues that have come up.**

- **Here are the different log levels in increasing order of severity:**
  - **DEBUG: Detailed information, typically of interest only when diagnosing problems.**
  - **INFO: Confirmation that things are working as expected.**
  - **WARNING: An indication that something unexpected happened, or may happen in the future (e.g. ‘disk space low’). The software is still working as expected.**
  - **ERROR: More serious problem that prevented the software from performing a function.**
  - **CRITICAL: A very serious error, indicating that the program itself may be unable to continue running.**

Debug

Debug

In [ ]:

```python
import logging

logging.basicConfig(level=logging.DEBUG)

def add(x, y):
    logging.debug('Variables are %s and %s', x, y)
    return x + y

add(1, 2)
```

```
DEBUG:root:Variables are 1 and 2
```

```
3
```

Info

In [ ]:

```python
import logging

logging.basicConfig(level=logging.INFO)

def login(user):
    logging.info('User %s logged in', user)

login('Admin User')
```

```
INFO:root:User Admin User logged in
```

Warning

In [ ]:

```python
import logging

logging.basicConfig(level=logging.WARNING)

def MyBalance(amount):
    if amount < 40000:
        logging.warning('Sorry you have Low balance: %s', amount)

MyBalance(10000)
```

```
WARNING:root:Sorry you have Low balance: 10000
```

Error

In [ ]:

```python
import logging

logging.basicConfig(level=logging.ERROR)

def LetUsDivide(n, d):
    try:
        result = n / d
    except ZeroDivisionError:
        logging.error('You are trying to divide by zero, which is not allowed')
    else:
        return result

LetUsDivide(4, 0)
```

```
[illegible]
```

```
ERROR:root:You are trying to divide by zero, which is not allowed
```

Critical Errors

In [ ]:

```python
import logging

logging.basicConfig(level=logging.CRITICAL)

def LetUsCheckSystem(sys):
    if sys != 'OK':
        logging.critical('System failure: %s', sys)

LetUsCheckSystem('You need to handle the issue now')
```

```
CRITICAL:root:System failure: You need to handle the issue now
```

Save to a file | Working with Files

In [ ]:

```python
import os

# Specify the directory and file
dir_path = r'C:\Users\Dell\Desktop\June\Latest\iNeuron\Sessions\17_18June2023'
log_file = 'system.txt'
full_path = os.path.join(dir_path, log_file)

# Check if the directory exists and create it if necessary
os.makedirs(dir_path, exist_ok=True)

# Try writing a test message to the file
with open(full_path, 'w') as f:
    f.write('This is a test message')
```

In [ ]:

```python
import os
print(os.getcwd())
```

```
C:\Users\Dell\Desktop\June\Latest\iNeuron\Sessions\17_18June2023
```

In [ ]:

```python
import os
from os.path import dirname, join
import logging

# Specify the directory and file
dir_path = "/Users/abhijitpaul/Documents/Personal/Learning/iNeuron/Classes"
log_file = 'system.txt'
full_path = join(dir_path, log_file)

# Check if the directory exists and create it if necessary
os.makedirs(dir_path, exist_ok=True)

# Set up logging
# Get a logger instance (this will fetch the root logger)
logger = logging.getLogger()

# Set the level of the logger to CRITICAL
# This means it will handle events of level CRITICAL and above
logger.setLevel(logging.CRITICAL)

# Create a FileHandler instance to write logs to a file
handler = logging.FileHandler(full_path)
```

```
# Set the format of the logs using a Formatter
# This format includes the log timestamp, log level and log message
handler.setFormatter(logging.Formatter('%(asctime)s:%(levelname)s:%(message)s'))

# Add the handler to the logger
# This connects the logger to the handler so that logs get written to the file
logger.addHandler(handler)


def LetUsCheckSystem(sys):
    if sys != 'OK':
        logging.critical('System failure: %s', sys)

LetUsCheckSystem('You need to handle the issue now')
handler.close()
```

```
CRITICAL:root:System failure: You need to handle the issue now
```

In [ ]:

```
import pdb

def addition(a, b):
    pdb.set_trace()  # Set a breakpoint here
    result = a + b
    return result

print(addition(5, 7))
```

```
> <ipython-input-39-7ea871499ca9>(5)addition()
      3 def addition(a, b):
      4     pdb.set_trace()  # Set a breakpoint here
----> 5     result = a + b
      6     return result
      7

ipdb> 2
2
ipdb> 2
2
ipdb> 2
2
```

OOPs - Object Oriented Programming

In Python, everything is an object. It is a programming paradigm that uses objects and classes in programming.

CLASS

- **It is a blueprint of an object or you can say its the design of an object**

e.g Bank Account --> Class

Attributes

- **Acct No.**
- **Acct Holder Name**
- **Balance**
- **Account Type**
- **IFSC**
- **MObile No**

Methods

- **Deposit()**
- **Withdrawal()**
- **Transfer()**

OBJECT

- **Creating object of a class which will allocate memory in the system to load the class members and utilize it**

`self` **--> it holds the memory reference of the object and is initialized inside the init() method. Using the** `self`**, we can access variables & methods written inside the class**

In [ ]:

```
# Empty Class definition
class Person:
    pass

# Creating object of Person class which will allocate
# a memory in the system to load the class and utilize it
p = Person()

print(f"Type of object is {type(p)}")
```

```
Type of object is <class '__main__.Person'>
```

In [ ]:

```
# Adding attribute in runtime is not a good practice
p.name = "Johnny"
p.name
```

Out[ ]:

```
'Johnny'
```

Dunder Method or Magic Methods

- **Dunder methods are methods that allow instances of a class to interact with the built-in functions and operators of the class itself provided by Python Language**
- **In other words, all the operations we do on a Python object is implemented using these dunder methods**
- **The two underscores are there just to prevent name collision with other methods implemented by unsuspecting programmers.**

`dir(<class instance>)` **: Prints out list of all dunder methods available**

Few of the useful dunder methods are described below

`__init__`

- **Its a constructor or initializer method provided in a class**
- **Its called every time an object is created from a class**
- **Its purpose it to initialize the object's attributes**
- **Its only used within class**

`self`

- **Its denotes the current instance of the class.**
- **It holds the memory reference of the object and is initialized inside the** `__init__()` **method.**
- **Using the** `self`**, we can access variables & methods written inside the class**

In [ ]:

```
# Will redefine the class
class Person:
    def __init__(self, first_name, last_name) -> None:
        """
        "__init__ is the dunder method that INITialises the instance.

        To create a Person, we need to know the first name and the last name,
        so that will be passed as an argument later, e.g. with Person("Abhijit", "Paul).
        To make sure the instance of the class when created knows its own first_name & la
st_name,
        we save it with self.first_name = first_name & self.last_name = last_name

        """
        self.first_name = first_name
        self.last_name = last_name

p = Person("Abhijit", "Paul")  # These two arguments will be initialized in `__init__()`
print(f"First Name: {p.first_name}")
print(f"Last Name: {p.last_name}")
```

```
First Name: Abhijit
Last Name: Paul
```

`__repr__`

Computes "official" string representation of class

In [ ]:

```
class Person:
    def __repr__(self) -> None:
        return f"This is a Person class without customized representation"

p = Person()
print(f"Print Person instance: {p}")
```

```
Print Person instance: This is a Person class without customized representation
```

`__str__`

When we print an object (class instance), it calls the `__str__()` function and prints the contents written within

In [ ]:

```
class Person:
    def __str__(self) -> None:
        return f"This is a Person class without customized representation"

p = Person()
print(f"Print Person instance: {p}")
```

```
Print Person instance: This is a Person class without customized representation
```

`__getattr__`

Get value of the attribute defined in a class

In [ ]:

```
class Person:
    def __init__(self, first_name, last_name) -> None:
        self.first_name = first_name
        self.last_name = last_name

p = Person("Abhijit", "Paul")  # These two arguments will be initialized in `__init__()`
print(f"Get Attribute: {p.__getattribute__('first_name')}")
```

```
Get Attribute: Abhijit
```

`__setattr__`

Set value of the attribute defined in a class

In [10]:

```
class Person:
    def __init__(self, first_name, last_name) -> None:
        self.first_name = first_name
        self.last_name = last_name

p = Person("Abhijit", "Paul")  # These two arguments will be initialized in `__init__()`

p.__setattr__('first_name', 'Mohan')  # Setting attribute on the runtime using `__setattr
__()`

print(f"Get Attribute: {p.__getattribute__('first_name')}")
```

```
Get Attribute: Mohan
```

We will create a Class called 'ToDo' which will keep a track of to-do list

In [ ]:

```
class ToDo:
    def __init__(self, owner: str) -> None:
        self.owner = owner
        self.tasks = []

    def add_task(self, task: str) -> None:
        self.tasks.append(task)

    def complete_task(self, task: str) -> None:
        if task in self.tasks:
            self.tasks.remove(task)
            print(f"Task '{task}' completed")
        else:
            print(f"Task is not found")

    def display_tasks(self) -> None:
        print(f"The Todo task of {self.owner}:")
        for task in self.tasks:
            print(task)
```

In [ ]:

```
todo = ToDo("Abhijit")
todo.add_task("I will wake up at 8am")
todo.add_task("I will attend iNeuron Live Session")

todo.display_tasks()

todo.complete_task("I will wake up at 8am")

todo.display_tasks()
```

```
The Todo task of Abhijit:
I will wake up at 8am
I will attend iNeuron Live Session
Task 'I will wake up at 8am' completed
The Todo task of Abhijit:
I will attend iNeuron Live Session
```

4-Pillars of Object-Oriented Programming (OOPs)

1. **Inheritance --> Provides reusability**
2. **Encapsulation --> Provides security | Binding data with methods | No exposing data directly by allowing data accessibility by using those methods**
3. **Abstraction --> Hides the backend implementation**
4. **Polymorphism --> Provides many forms**

Inheritance

- **Inheritance is one of the characteristics of Object Oriented Programming paradigm.**
- **It facilitates the passing of attributes and methods of a class (base) to another class (derived)**
- **A child class acquires the properties and can access all the data members and functions defined in the parent class.**
- **Moreover, a child class can also provide its specific implementation to the functions of the parent class.**

***Usages*:**

- **Its primary usage is code reusability because we can inherit all attributes and methods related to a particular functionality which is already written in a class instead of writing the functionality from scratch.**

***Types*:**

- **Single Inheritance**
- **Multilevel Inheritance**
- **Multiple Inheritance**

Single Inheritance

In Single Inheritance, one child inherits from one parent i.e. there will be a single base class and a single derived class

In [ ]:

```
# E.g. Single Inheritance
class Human:   # Base Class
    def __init__(self, name) -> None:
        self.name = name

    def info(self):
        print(f"Name: {self.name}")

class Male(Human):   # Derived Class from a Single Base Class
    def __init__(self, name, gender) -> None:
        super().__init__(name)
        self.gender = gender

    def info(self):
        super().info()   # Reusing the info() method of base class
        print(f"Gender: {self.gender}")

male = Male("Akshay", "Male")
male.info()
```

```
Name: Akshay
Gender: Male
```

Multilevel Inheritance

In Multilevel Inheritance, one child class inherits attributes and methods from a parent class and subsequently becomes the parent class of another child class.

In [ ]:

```
class Parent:
    def __init__(self,name):
        self.name = name

    def getName(self):
        return self.name

class Child(Parent):
    def __init__(self,name,age):
        super().__init__(name)
        self.age = age

    def getAge(self):
        return self.age

class Grandchild(Child):
    def __init__(self,name,age,location):
        super().__init__(name,age)
        self.location=location

    def getLocation(self):
        return self.location

gc = Grandchild("Srinivas",24,"Assam")
print(gc.getName(), gc.getAge(), gc.getLocation())
```

```
Srinivas 24 Assam
```

Multiple Inheritance

In Multiple Inheritance, one child class can inherit attributes and methods from multiple parent classes i.e there will be multiple base class and a single derived class

In [ ]:

```
# E.g. Multiple Inheritance
class Person:   # Base Class
    def __init__(self, name, age) -> None:
        self.name = name
        self.age = age

    def person_info(self):
        print(f"Name: {self.name} & Age: {self.age}")

class Company:   # Base Class
    def __init__(self, name, domain) -> None:
        self.name = name
        self.domain = domain

    def company_info(self):
        print(f"Company Name: {self.name} & Domain: {self.domain}")

class Employee(Person, Company):   # Derived Class from a Single Base Class
    def __init__(self, name, age, company_name, domain, salary):
        Person.__init__(self, name, age)
        Company.__init__(self, company_name, domain)
        self.salary = salary

    def employee_info(self):
        super().person_info()
        super().company_info()
        print(f"Salary: {self.salary}")


person1 = Employee("Abhijit", 30, "TT", "Trading Platform", 10000)
person1.employee_info()
```

```
Name: TT & Age: 30
Company Name: TT & Domain: Trading Platform
Salary: 10000
```

MRO (Method Resolution Order)

--> Its very important to under MRO to grasp Multiple Inheritance

- **It is the order in which Python looks for a method in a hierarchy of classes.**
- **It plays vital role in the context of multiple inheritance as a single method may be found in multiple super classes**
- **Python constructs the order in which it will look for a method in the hierarchy of classes.**

- **It uses this order, known as MRO, to determine which method it actually calls.**
- **It is possible to see MRO of a class using mro() method of the class.**

IMP RULE TO REMEMBER: A Parent/Base/Super Class cannot be accessed before a Child/Derived/Subclass

In [ ]:

```
#CASE1
class A:
    def process(self):
        print('A process()')

class B:
    pass

class C(A, B):
    pass

obj = C()
obj.process()
print(C.mro())   # C -> A -> B -> ObjectCLass
```

```
A process()
[<class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>, <class 'object'>]
```

From MRO of class C, we get to know that Python looks for a method first in class C. Then it goes to A and then to B.

So, first it goes to super class given first in the list then second super class, from left to right order. Then finally Object class, which is a super class for all classes.

In [ ]:

```
#CASE2
class A:
    def process(self):
        print('A process()')


class B:
    def process(self):
        print('B process()')


class C(A, B):
    def process(self):
        print('C process()')


class D(C,B):
    pass


obj = D()
obj.process()

print(D.mro()) # D -> C -> A -> B -> ObjectCLass
```

```
C process()
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class '__main__.B'>,
<class 'object'>]
```

In [ ]:

```
#CASE3 There are cases when Python cannot construct MRO owing to complexity of hierarchy.
In such cases it will throw an error.
class A:
    def process(self):
        print('A process()')
```

```
class B(A):
    def process(self):
        print('B process()')


class C(A, B):
    pass


obj = C()
obj.process()   # C -> A -> B -> objectClass (A cannot be before B --> leading to MRO fai
lure)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[39], line 12
      8     def process(self):
      9         print('B process()')
---> 12 class C(A, B):
     13     pass
     16 obj = C()

TypeError: Cannot create a consistent method resolution
order (MRO) for bases A, B
```

The problem comes from the fact that class A is a super class for both C and B. If you construct MRO then it should be like this: C -> A -> B -> A

super() function & Multiple Inheritance

In [11]:

```
# Scenario 1
class A:
    def __init__(self) -> None:
        print("A")

class B:
    def __init__(self) -> None:
        print("B")

class C(A, B):
    def __init__(self) -> None:
        super().__init__()

obj = C()    # Here as well MRO came into picture. super() method calls the __init__() of
its first superclass i.e. A in this case since init() is available else it would then loo
k for init() in B class
```

```
A
```

In [ ]:

```
# Scenario 2: What if I want to access the __init__() method of both the super class in M
ultiple Inheritance?

class A:
    def __init__(self) -> None:
        print(f"Inside A. Type of self: {type(self)}")   # But this will show that the 'se
lf' if of type C
        print("A")

class B:
    def __init__(self) -> None:
        print(f"Inside B. Type of self: {type(self)}")   # But this will show that the 'se
lf' if of type C
        print("B")
```

```
class C(A, B):
    def __init__(self) -> None:
        print(f"Inside C. Type of self: {type(self)}")
        A.__init__(self)   # In this scenario, we can not use super() anymore.
        B.__init__(self)   # Here we need to call the constructor of superclass is by Clas
s Name directly
obj = C()    # Here as well MRO came into picture. super() method calls the __init__() of
its first superclass i.e. A in this case

# Above is a limitation of using Multiple Inheritance in Python and we have to be cautiou
s while using it
```

```
Inside C. Type of self: <class '__main__.C'>
Inside A. Type of self: <class '__main__.C'>
A
Inside B. Type of self: <class '__main__.C'>
B
```

Encapsulation:

- **It provides security by binding data with methods ensuring no data is not exposed directly and allowing data accessibility by using the methods only.**
- **We bind the data with methods**
- **Its bundling of data along with the methods that operates on that data, into a single unit. e.g. Class, Containers**
- **Its a way to restrict the direct access to both attributes and methods of an object, so that unauthorised users cannot access it.**

Access Modifiers | Format Specifier

To implement encapsulation, we have different types of access modifiers.

- **Public: Attributes and methods declared as Public are easily accessible from any part of the program i.e. inside as well as outside the class. All data members and member functions of a class are public by default**

- **Protected: Attributes and methods declared as Protected are accessible to only subclass or derived class of it. (Denoted by `single underscore as prefix`)**
- **Private: Attributes and methods declared as Private are only accessible within the class and not outside of it (Denoted by `double underscore as prefix`)**

In [ ]:

```
# define parent class Company
class Company:
    # constructor
    def __init__(self, name, proj):
        self.name = name       # name(name of company) is public
        self._proj = proj      # proj(current project) is protected defined with single un
derscore

    # public function to show the details
    def show(self):
        print("The code of the company")

# define child class Emp
class Emp(Company):
    # constructor
    def __init__(self, eName, sal, cName, proj):
        # calling parent class constructor
        super().__init__(cName, proj)
        self.name = eName    # public member variable
        self.__sal = sal     # private member variable defined with double underscore

    # public function to show salary details
    def show_sal(self):
        print(f"The salary of {self.name} is {self.__sal}")
```

```
# creating instance of Company class
c = Company("Stark Industries", "Mark 4")
# creating instance of Employee class
e = Emp("Steve", 9999999, c.name, c._proj)

print("Welcome to ", c.name)
print("Here ", e.name," is working on ",e._proj)

# only the instance itself can change the __sal variable
# and to show the value we have created a public function show_sal()
e.show_sal()
```

```
Welcome to  Stark Industries
Here  Steve  is working on  Mark 4
The salary of Steve is 9999999
```

Abstraction:

- **Its a process of handling complexity by hiding unnecessary information from the user.**
- **It enables the user to implement more complex logic on top of the provided abstraction without understanding or even thinking about all the hidden complexities**

Achieving Abstraction in Python

1. **Decorators**
2. **Abstract Method & Abstract Classes**

In Python, abstraction can be achieved by using `abstract classes` and `abstract methods`

Abstract Method: Its a method that is declared but doesn't contain the implementation.

An abstract method in a base class identifies the functionality that should be implemented by all its subclasses. However, since the implementation of an abstract method would differ from one subclass to another, often the method body comprises just a pass statement. Every subclass of the base class will override this method with its implementation.

Abstract Class A class containing abstract methods (atleast one abstract method) is called abstract class.

In [ ]:

```
# Note:
# 1. A class is defined as an Abstract Class by inheriting from `ABC` class
# 2. A method is defined as an Abstract Method by preceding its definition with @abstract
method (function decorator).

from abc import ABC, abstractmethod

class Shape(ABC):

    def __init__(self, shapeType) -> None:
        self.shapeType = shapeType

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

class Rectangle(Shape):
    def __init__(self, length, breadth) -> None:
        self.length = length
        self.breadth = breadth
        super().__init__("Rectangle")

    # If the abstract methods are not implement, then while instantiating Rectangle class
```

```
, error will be thrown
    # `Can't instantiate abstract class Rectangle with abstract methods area, perimeter`

    def area(self):  # Implemented the abstract method from base class
        return self.length * self.breadth

    def perimeter(self):  # Implemented the abstract method from base class
        return 2 * (self.length + self.breadth)

print(f"Is `Rectangle` a subclass of `Shape`?: {issubclass(Rectangle, Shape)}")

rectangle = Rectangle(30, 15)
print(f"Area of rectangle: {rectangle.area()}")
print(f"Perimeter of rectangle: {rectangle.perimeter()}")
```

```
Is `Rectangle` a subclass of `Shape`?: True
Area of rectangle: 450
Perimeter of rectangle: 90
```

Polymorphism

- **Polymorphism means** `having different forms`
- **In Python, polymorphism refers to a function having same name but being used in different ways and different scenarios.**

Types

There are two different ways we can achieve Polymorphism.

- **Compile Time Polymorphism (e.g. Method Overloading)**
- **Run Time Polymorphism (e.g. Method Overriding)**

Method Overloading

In this, more than one method in same class shares the same method name but different signatures.

Note: Python doesn't support method overloading, as the interpreter only detects the last method and ignores the previous ones.

Workaround: By Using Multiple Dispatch Decorator (Install `pip3 install multipledispatch`**) we can implement the feature of Method Overloading**

In [1]:

```
from multipledispatch import dispatch

@dispatch(int, int)
def product(first: int, second: int):
    return first * second

@dispatch(int, int, int)
def product(first: int, second: int, third: int):
    return first * second * third

@dispatch(float, float, float)
def product(first: float, second: float, third: float):
    return first * second * third


print(product(2,3))
print(product(3,4,5))
print(product(1.1,2.2, 3.3))
```

```
6
60
7.986000000000001
```

Method Overriding

- **Its an OOPs concept and related to `Inheritance`.**
- **When child class method overrides (inducing its own implementation) its parent class method having same name, parameters and return type, its called Method Overriding**

In [ ]:

```
# Single Inheritance

class Rectangle:
    def area(self, length, breadth):
        return length * breadth

class Triangle(Rectangle):
    def area(self, base, height):
        return 0.5 * base * height

rec = Rectangle()
tri = Triangle()

print(rec.area(4, 8))
print(tri.area(4, 8))
```

```
32
16.0
```

In [ ]:

```
import logging

logging.basicConfig(level=logging.INFO, filename='app.log', filemode='a', format='%(asct
ime)s - %(levelname)s - %(message)s')
logging.debug("This is a debug message and should not be printed")
logging.info("Second Hello World!")
```

Operator Overloading

In [ ]:

```
class Vegetables:
    def __init__(self, carrot: int, beans: int) -> None:
        self.carrot = carrot
        self.beans = beans

    def __add__(self, other):  # So here we have implemented the code to overload the `+`
operator
        carrot = self.carrot + other.carrot
        beans = self.beans + other.beans
        return Vegetables(carrot, beans)

v1 = Vegetables(4, 8)
v2 = Vegetables(9, 3)

# without operator (+) overloading, we get error `TypeError: unsupported operand type(s)
for +: 'Vegetables' and 'Vegetables'`
v3 = v1 + v2
print("Sum of carrots of both object: ", v3.carrot)
print("Sum of beans of both object: ",v3.beans)
```

```
Sum of carrots of both object:  13
Sum of beans of both object:  11
```

Class Variable vs Instance Variable

1 2 3
1 2 3

Comparison Basis|Class Variables|Instance Variables Definition|Class variables are defined within the class but outside of any class methods.|Instance variables are defined within class methods, typically the constructor| Scope|Changes made to the class variable affect all instances of that class.|Changes made to the instance variable does not affect all instances.| Initialization|Class variables can be initialized either inside the class definition or outside the class definition.|Instance variables are typically initialized in the constructor of the class.| Access|Class variables are accessed using the class name, followed by the variable name.|Instance variables are accessed using the instance name, followed by the variable name.| Usage|Class variables are useful for storing data that is shared among all instances of a class, such as constants or default values.|Instance variables are used to store data that is unique to each instance of a class, such as object properties.|

In [ ]:

```
class Employee:

    company_name = "ABC Technologies"  # Class Variable

    def __init__(self, first_name, last_name):
        self.first_name = first_name  # Instance Variable
        self.last_name = last_name  # Instance Variable

    def display(self):
        print(f"First Name: {self.first_name} | Last Name: {self.last_name}")

emp1 = Employee("Abhijit", "Paul")
emp1.display()
print(emp1.company_name)  # We can access class variable using the instance of the class
print(Employee.company_name)  # We can access class variable using the Class too


emp2 = Employee("Mousumi", "Paul")
emp2.display()
print(emp2.company_name)  # We can access class variable using the instance of the class
print(Employee.company_name)  # We can access class variable using the Class too
```

```
First Name: Abhijit | Last Name: Paul
ABC Technologies
ABC Technologies
First Name: Mousumi | Last Name: Paul
ABC Technologies
ABC Technologies
```

Local Variable vs Global Variable

1 2 3

Comparison Basis|Global Variable|Local Variable Definition|Global variables are declared outside the functions|Local variables are declared within the functions Lifetime|Global variables are created as execution of the program begins and are lost when the program is ended|Local variables are created when the function starts its execution and are lost when the function ends Data Sharing|Global Variables Offers Data Sharing|Local Variables doesn't offers Data Sharing Scope|Accessible throughout the code|Accessible inside the function Storage|Global variables are kept in a fixed location selected by the compiler|Local variables are kept on the stack Parameter Passing|For global variables, parameter passing is not necessary|For local variables, parameter passing is necessary Changes in a variable value|Changes in a global variable is reflected throughout the code|Changes in a local variable doesn't affect other functions of the program

NOTE

In Python programming, both local and global variables are crucial when writing programs. However, many global variables could take up a lot of memory. It gets harder to spot an undesireable change in global variables.

As a result, it is wise to avoid declaring pointless global variables and using local variables for passing and

manipulating data. It is generally good practice to make use of local variables in Python.

In [ ]:

```python
# Local Variable
a = 20
def func1():
    a = 10  # This variable `a` is local variable and is scope inside the function only
    print(f"Local Variable value inside the function is {a}")

func1()
print(f"Variable value outside the function is {a}")

# Explanation of Global Variable
b = 10  # Global variable
def func2():
    global b  # Accessing global variable inside the function using `global` keyword
    b += 5
    print(f"Global Variable value inside the function is {b}")
func2()
print(f"Global Variable value outside the function is {b}")
```

```
Local Variable value inside the function is 10
Variable value outside the function is 20
Global Variable value inside the function is 15
Global Variable value outside the function is 15
```

Decorators

Decorators are functions (or classes) that provide enhanced functionality to the original function (or class) without the programmer having to modify their structure.

1. **Function level decorator**
2. **Class level decorator**
3. **Property (Setter | Getter | Deleter)**

1. Function Level Decorator

In [2]:

```python
# Sequence of execution
def hello_decorator(func): #3 func = sum_two_numbers
    def inner(a,b): #6
        print("Before execution") #7
        value = func(a,b) #8  Here sum_two_number function will be called and gets 50 ba
ck from the func execution and stored in `value` variable
        print("After execution") #12
        return f"Computed value: {value}" #13  End of execution
    return inner  #4 It returns the inner() function definition which accepts two argumen
ts

@hello_decorator #2 Decorator function is called
def sum_two_number(a, b): #9 gets called after step 8
    print("Inside the function") #10
    return a+b #11 : 10+40 = 50 is returned

sum_two_number(10,40) #1 #5   It invokes inner() function and passes two parameters 10,20
i.e. Assume sum_two_number(10,40) = inner(10, 40)
```

```
Before execution
Inside the function
After execution
```

Out[2]:

```
'Computed value: 50'
```

In [ ]:

```
def grade_decorator(func):
    def wrapper(score, total):
        percent = func(score, total)

        grades = {
            5: 'A+',
            4: 'A',
            3: 'B',
            2: 'C',
            1: 'D'}
        return percent, grades[percent // 20]
    return wrapper

class Student:
    def __init__(self, name, score, total):
        self.name = name
        self.__score = score
        self.total = total

    @staticmethod
    @grade_decorator
    def get_percent(score, total):
        return score/total * 100

    def get_record(self):
        percent, grade = Student.get_percent(self.__score, self.total)
        return f"Name: {self.name} | Percentage scored: {percent} % | Grade: {grade}"

    def __str__(self):
        return f"Name: {self.name} | Score: {self.__score} | Total: {self.total}"

stu = Student("Abhijit", 25, 100)
stu.get_record()
```

Out[ ]:

```
'Name: Abhijit | Percentage scored: 25.0 % | Grade: D'
```

In [3]:

```
import functools

def make_secure(func):

    @functools.wraps(func)
    def secure_function():
        if user["access_level"] == "admin":
            return func()
        else:
            return f"No admin permission for {user['username']}"
    return secure_function

@make_secure
def get_admin_password():
    return "1234"

# user = {"username": "jose", "access_level": "guest"}
user = {"username": "jose", "access_level": "admin"}

print(get_admin_password())
print(get_admin_password.__name__)  # It prints `secure_function` as function name, witho
ut `functool.wraps()` becase decorator overrides the caller function name
```

```
1234
get_admin_password
```

In [4]:

```
# Decorators with parameters

import functools
```

```
def make_secure(func):

    @functools.wraps(func)
    def secure_function(*args, **kwargs):
        if user["access_level"] == "admin":
            return func(*args, **kwargs)
        else:
            return f"No admin permission for {user['username']}"
    return secure_function

@make_secure
def get_password(panel):
    if panel == "admin":
        return "1234"
    else:
        return "super secret password"

# user = {"username": "jose", "access_level": "guest"}
user = {"username": "jose", "access_level": "admin"}

print(get_password("admin"))
print(get_password("billing"))
```

```
1234
super secret password
```

In [50]:

```
# Decorators with parameters

import functools

def make_secure(access_level):  # this is a factory method which helps in creating the de
corator
    def decorator(func):  # this is the decorator function
        @functools.wraps(func)
        def secure_function(*args, **kwargs):
            if user["access_level"] >= 1:  # guest = 0, user = 1, admin = 2
                return func(*args, **kwargs)
            else:
                return f"No {access_level} permission for {user['username']}"
        return secure_function
    return decorator

@make_secure("admin")
def get_admin_password():
    return "1234"

@make_secure("user")
def get_dashboard_password():
    return "super secret password"

user = {"username": "jose", "access_level": 0}

print(get_admin_password())
print(get_dashboard_password())

user = {"username": "jose", "access_level": 2}

print(get_admin_password())
print(get_dashboard_password())
```

```
No admin permission for jose
No user permission for jose
1234
super secret password
```

2. Class level decorator

Instance Method, Class Method, Static Method

Instance Method:

- It is the method that takes the first parameter, `self`, which points to an instance of the class when the method is called.
- Through the `self` parameter, instance methods can freely access attributes and other methods on the same object.
- Not only can they modify object state, instance methods can also access the class itself through the self.**class** attribute. This means instance methods can also modify class state.

Class Method:

- A class method is decorated with @classmethod decorator.
- Instead of accepting a `self` parameter, class methods take a `cls` parameter that points to the class—and not the object instance—when the method is called.
- Because the class method only has access to this `cls` argument, it can't modify object instance state. However, class methods can still modify class state that applies across all instances of the class.

Static Method:

- A static method is decorated with @staticmethod decorator.
- This type of method takes neither a `self` nor a `cls` parameter (but of course it's free to accept an arbitrary number of other parameters).
- Therefore a static method can neither modify object state nor class state.
- Static methods are restricted in what data they can access - and they're primarily a way to namespace your methods.

In [ ]:

```
class MyClass:
    data = 10  # class variable
    def method(self):
        # Instance methods can also access the class itself through the self.__class__.da
ta.
        return 'instance method called', self

    @classmethod
    def classmethod(cls):
        return 'class method called', cls

    @staticmethod
    def staticmethod():
        return 'static method called'

obj = MyClass()

# This confirms that the instance method has access to the object instance (printed as <M
yClass instance>) via the self argument.
print(obj.method())

# When the method is called, Python replaces the self argument with the instance object,
obj.
# We could ignore the syntactic sugar of the dot-call syntax (obj.method()) and pass the
instance object manually to get the same result
# print(MyClass.method(obj))

# Calling classmethod() showed us it doesn't have access to the <MyClass instance> object
, but only to the <class MyClass> object, representing the class itself.
print(obj.classmethod())


# Behind the scenes Python simply enforces the access restrictions by not passing in the
`self` or the `cls` argument when a static method gets called using the dot syntax.
# This confirms that static methods can neither access the object instance state nor the
class state. They work like regular functions but belong to the class's (and every instan
ce's) namespace.
print(obj.staticmethod())
```

```
('instance method called', <__main__.MyClass object at 0x7fb1e88b6730>)
('class method called', <class '__main__.MyClass'>)
static method called
```

We can call the `classmethod` & `staticmethod` methods on the class itself - without creating an object instance beforehand

In [ ]:

```python
print(MyClass.classmethod())
print(MyClass.staticmethod())
```

```
('class method called', <class '__main__.MyClass'>)
static method called
```

In [26]:

```python
# Class Methods are mostly used as Factory Methods

class Book:
    TYPES = ("Hardcover", "Paperback")  # Class variable which shows that book can be of only two types

    def __init__(self, name: str, book_type: str, weight: int):
        self.name = name
        self.book_type = book_type
        self.weight = weight

    def __repr__(self) -> str:
        return f"<Book {self.name}, {self.book_type}, weighing {self.weight}g>"

    @classmethod
    def hardcover(cls, name: str, page_weight: int) -> "Book":
        return cls(name, cls.TYPES[0], page_weight+100)

    @classmethod
    def paperback(cls, name: str, page_weight: int) -> "Book":
        return cls(name, cls.TYPES[1], page_weight)


book1 = Book.hardcover("Harry Potter", 1500)
book2 = Book.paperback("Python 101", 700)
print(book1)
print(book2)
```

```
<Book Harry Potter, Hardcover, weighing 1600g>
<Book Python 101, Paperback, weighing 700g>
```

3. Property (getter | setter | deleter)

The property decorator is very useful when defining methods for data validation, like when deciding if a value to be assigned is valid and won't lead to issues later in the code.

- `getter` **and** `setter` **methods are used to access and modify (respectively) a private instance.**
- `deleter` **method lets you delete a protected or private attribute using the del function.**

In [ ]:

```python
class Person:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        print('Getting name')
        return self.__name
    @name.setter
    def name(self, value):
        print('Setting name to ' + value)
        self.__name = value

    @name.deleter
    def name(self):
        print('Deleting name')
        del self.__name

# Now, whenever we use p.name, it internally calls the appropriate getter, setter, and de
leter as shown by the printed output present inside the method.

p = Person('Adam')
print('The name is:', p.name)
p.name = 'John'
del p.name
```

```
Getting name
The name is: Adam
Setting name to John
Deleting name
```

Constructors & Destructors

- **A `Constructor` is called when an class instance is created**
- **A `Destructor` is called when an object is deleted or destroyed**
  - **The `__del__` method is called for any object when the reference count for that object becomes zero.**
  - **The reference count for that object becomes zero when the application ends, or we delete all references manually using the del keyword.**
  - **The destructor will not invoke when we delete object reference. It will only invoke when all references to the objects get deleted.**

In [ ]:

```
class Employee:

    # constructor
    def __init__(self, name):
        print('Inside Constructor')
        self.name = name
        print('Object initialized')

    def show(self):
        print('Hello, my name is', self.name)

    # destructor
    def __del__(self):
        print('Inside destructor')
        print('Object destroyed')

# create object
s1 = Employee('Emma')
```

```
Inside Constructor
Object initialized
```

In [ ]:

```
s1.show()
```

```
Hello, my name is Emma
```

In [ ]:

```
# delete object
del s1
```

```
Inside destructor
```

```
Object destroyed
```

In [52]:

```python
from typing import List, Optional

Optional[List[int]]
```

Out[52]:

```
typing.Optional[typing.List[int]]
```

Duck Typing

- **Duck Typing is a way of programming in which an object passed into a function or method supports all method signatures and attributes expected of that object at run time.**
- **The object's type itself is not important. Rather, the object should support all methods/attributes called on it.**

In [4]:

```python
class Duck:
    def sound(self):
        print("quack quack")

class Dog:
    def sound(self):
        print("woof woof")

class Cat:
    def __init__(self):
        self.sound = "meow meow"

def all_sounds(obj):
    if hasattr(obj, 'sound') and callable(obj.sound):
        obj.sound()
du = Duck()
do = Dog()
ca = Cat()
for obj in [du, do, ca]:
    all_sounds(obj)
```

```
quack quack
woof woof
```

Multi-Threading

Process in Python

In computing, a process is an instance of a computer program that is being executed. Any process has 3 basic components:

- **An executable program.**
- **The associated data needed by the program (variables, workspace, buffers, etc.)**
- **The execution context of the program (State of the process)**

Thread:

- **Entity within a process that can be scheduled for execution**
- **Smallest unit of execution**
- **Multiple threads can exist within one process where:**
  - **Each thread contains its own register set and local variables (stored in the stack).**
  - **All threads of a process share global variables (stored in heap) and the program code.**
- **A thread is simply a subset of a process**
- **A thread contains all this information in a Thread Control Block (TCB)**

- A thread contains all this information in a Thread Control Block (TCB)

Thread Control Block (TCB)

- **Thread Identifier:** Unique id (TID) is assigned to every new thread
- **Stack pointer:** Points to the thread's stack in the process. The stack contains the local variables under the thread's scope.
- **Program counter:** a register that stores the address of the instruction currently being executed by a thread.
- **Thread state:** can be running, ready, waiting, starting, or done.
- **Thread's register set:** registers assigned to thread for computations.
- **Parent process Pointer:** A pointer to the Process control block (PCB) of the process that the thread lives on.

Process Control Block (PCB)

- Process Identifier (PID)
- Process State (waiting, running, exiting, etc)
- Process counter (gives the address of process in memory)
- Process Registers
- Memory Limits
- List of open files

Multithreading in Python

- Defined as the ability of a processor to execute multiple threads concurrently.
- In a simple, single-core CPU, it is achieved using frequent switching between threads. This is termed context switching.
- In context switching, the state of a thread is saved and the state of another thread is loaded whenever any interrupt (due to I/O or manually set) takes place.
- Context switching takes place so frequently that all the threads appear to be running parallelly (this is termed multitasking).

In [9]:

```python
import threading
import os

def task1():
    print("Task 1 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 1: {}".format(os.getpid()))

def task2():
    print("Task 2 assigned to thread: {}".format(threading.current_thread().name))
    print("ID of process running task 2: {}".format(os.getpid()))
```

In [11]:

```python
# print ID of current process
print("ID of process running main program: {}".format(os.getpid()))

# print name of main thread
print("Main thread name: {}".format(threading.current_thread().name))

# creating threads
t1 = threading.Thread(target=task1, name='t1')
t2 = threading.Thread(target=task2, name='t2')

# starting threads
t1.start()
t2.start()

# wait until all threads finish
t1.join()
t2.join()
```

```
ID of process running main program: 68450
```

```
Main thread name: MainThread
Task 1 assigned to thread: t1
ID of process running task 1: 68450
Task 2 assigned to thread: t2
ID of process running task 2: 68450
```