Python Programming – My Self Notes

Basic Concepts

- Complex Data Type cannot be converted to any other number data type.
- Isinstance() function is used to return True if the Data Type is correct and false if not. It takes 2 arguments: *isinstance(**object , DataType**)*
- id() is used to return the directory where the data type is stored in RAM.
- type() function returns the type of data a variable holds.
- Float can also be scientific numbers with an "e" to indicate the power of 10. Examples: x = 35e3, y = 12E4, z = -87.7e100
- The dir() function returns all properties and methods of the specified object, without the values.
- If Statements are used for logical computations, Booleans and/or decision making.
- Unlike other programming languages, Python uses if , elif and else instead of else if.
- Almost any value is evaluated to True if it has some sort of content.
- Any string is True, except empty strings
- Any number is True, except 0.
- Any list, tuple, set, and dictionary are True, except empty ones.
- In fact, there are not many values that evaluate to False, except empty values, such as (), [], {}, "", the number 0, and the value None. And of course the value False evaluates to False.
- One more value, or object in this case, evaluates to False, and that is if you have an object that is made from a class with a __len__ function that returns 0 or False
- Python does not have built-in support for Arrays, but Python Lists can be used instead.
- A variable created inside a function belongs to the local scope of that function, and can only be used inside that function.
- A variable created in the main body of the Python code is a global variable and belongs to the global scope.
- Global variables are available from within any scope, global and local.
- If you operate with the same variable name inside and outside of a function, Python will treat them as two separate variables, one available in the global scope (outside the function) and one available in the local scope (inside the function). If you need to create a global variable, but are stuck in the local scope, you can use the global keyword. The global keyword makes the variable global.
- The enumerate() function takes a collection (e.g. a tuple) and returns it as an enumerate object.
- The enumerate() function adds a counter as the key of the enumerate object. The *syntax* is: ***Enumerate(iterable, start)*** iterable is an iterable object whereas start is to define the starting number; Default is 0.

Loops

Loop in simple terms means iteration or repetition. A loop in programming is an instruction that is repeated multiple times unless that specified condition is met. In python programming, there are two ways to loop a code:

1. For Loop

A *for* loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). With the *for* loop we can execute a set of statements, once for each item in a list, tuple, set etc.

2. While Loop

The *while* Loop With the while loop we can execute a set of statements as long as a condition is true . Remember to increment/decrement as the case maybe, or else the loop will continue forever.

Loop Controls

We can also use **controls** to handle the above mentioned loops. These Controls are optional.

1. *Break*

A *break* statement stops the execution of the loop as soon as the condition is met. When a desired condition is met, the loop is broken and no further action is taken.

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

2. *Continue*

A *continue* statement skips the iteration when the condition reaches the requirement.

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

3. *Pass*

A *pass* is a keyword that tells the interpreter not to give error and we will come back later. With this keyword, an error is not encountered. It is usually used with Functions.

4. *Else*

With the *else* statement we can run a block of code once when the condition no longer is true:

```
i = 1
while i < 6:
  print(i)
  i += 1
else:
  print("i is no longer less than 6")
```

Nested Loops

Loop inside a loop is called nested loop.

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

The Range () Function

To loop through a set of code a specified number of times, we can use the range() function,

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

The `range()` function defaults to 0 as a starting value, however it is possible to specify the starting value by adding a parameter: `range(2, 6)`, which means values from 2 to 6 (but not including 6)

Enumerate Function

The enumerate function is used to iterate over an iterable while keeping a track of the index of current element. It returns pair of index elements, at **Tuples**.

```
fruits = ['apple','banana','mango','grapes']
for index, fruits in enumerate(fruits):
    print(f'Index {index}: {fruits}')
```

Things to Remember in Loops

It is important to note that when using any loop, make sure you are clear of:

- Counter (Body)
- Logic of the code
- Increment/Decrement

Zip() Function

Zip Function has following attributes:

- ✶ Used to combine multiple Iterable Objects into a Single Iterable Object
- ✶ The Index of Element of One Iterable Object is matched with the First Index of Second Iterable Object and so on.
- ✶ It takes TWO OR MORE inputs
- ✶ Output result is either a List or any other Sequence Data Type as we desire.
- ✶ If input iterable is of different length, zip() stops creating pairs when shortest iterable is exhausted. However it is necessary to note that, If we dont explicitly convert the result of zip() in to list() or dict(), we still have a valid iterator we can use in for loop or other constructs. But in that case, we wont be able to access specific elements like we can use with list() or dict().

Unzip()

- ✶ Now there is no function in python specific to "unzip()"
- ✶ However, zipped values can be easily unpacked by adding a * before the object.
- ✶ Unpacking is done when we want to retrieve individual values that were previously combined using zip()
- ✶ Unzipping separates values from an iterable such as tuple, lists, dictionaries etc.

```
zippedData = [("A",1),("B",47),("C",7)] # List of Tuples
numbers, letters = zip(*zippedData)
print(numbers)
print(letters)
o You can also unpack values from list or tuples individually. Each variable on the left side of the assignment receives a value from
the data list based on its position.

data = ["Talaal", 27, "Karachi"]
Name, Age, City = data
print(Name)
print(Age)
print(City)
```

- Unpack is often used with for loops to iterate over iterable elements

```
pairs = [(1, 'apple'), (2, 'banana'), (3, 'cherry')]
for number, fruit in pairs:
print(f'Number: {number}, Fruit: {fruit}')
```

Unzipping or unpacking is a useful technique when working with complex data structures, iterating over data, or processing results from functions that return multiple values. It allows you to work with individual elements easily and efficiently.

Functions

- Functions are set of instructions that tells interpreter what to do. That are called by the caller.
- Functions are denoted by def keyword. (User Defined Function)
- Syntax of User Defined Function is as follows:

```
def custom_name(parameter1,parameter2,...parameterN):
Statement1
Statement2
...
...
...
StatementN
custom_name() #Function Call
```

Functions in Python are of two types:

1. Pre-Defined Functions:

- These are the functions that are defined already and injected to python interpreter.
- The few examples of pre defined functions are: print() , findall() , input() , id() etc. These functions are pre built in.
- Predefined functions, also known as built-in functions, are functions that are already available in Python. These functions are provided by the Python language itself and can be used directly without having to create them.

```
text = "Hello, world!"
length = len(text) # Using the built-in len() function to calculate the length of the string
print(length) # Output: 13
```

2. User Defined Functions:

- User defuned Functions are functions that are built separately by users.
- User Defined Functions are denoted by def() keyword followed by Function Name
- User-defined functions are functions that you create yourself to perform specific tasks. You give these functions a name, define what they should do, and then you can use them in your code whenever needed.

```
def calculate_square(number):
square = number * number
return square
result = calculate_square(5) # Calling our user-defined function with the argument 5
print(result) # Output: 25
```

Common Properties: The common properties of the above 2 functions are:

1. return
2. non-return
3. required Parameters
4. optional parameters
5. Default parameters
6. Positional Arguments
7. Key Value Parameters
8. pass unlimited optional arguments
9. pass unlimited key=value

Return Function:

- Return function is a function that can be assigned to a variable.
- A return function is a type of function that computes a value and sends it back as the result of the function call.
- The return statement is used to specify the value that the function should return.

```
def add(a, b):
result = a + b
return result
sum_result = add(3, 5) # The function returns 8, which is assigned to sum_result
```

Non Return Function:

- Non Return Function is not assigned to a vairable.
- A non-return function is a type of function that performs a certain task or set of tasks but doesn't produce a value that can be captured or used elsewhere in the program.
- These functions are primarily used for their side effects, such as printing output or modifying data.

```
def greet(name):
print(f"Hello, {name}!")
greet("Alice") # This function prints "Hello, Alice!"
```

The greet function doesn't return a value; it only prints a message to the console.

Difference between Return and Print

- The return statement is used within a function to specify the value that should be sent back to the caller. This value can be assigned to a variable or used in further computations.
- The print statement is used to display information to the console. It doesn't affect the flow of the program or provide a value that can be captured.

Parameters vs Arguments

Parameters:

- Parameters are placeholders or variables listed in the function definition.
- They act as local variables within the function and serve as receptacles for the values that are passed into the function when it's called.
- In simpler words, parameters are like empty slots in the function that you expect to be filled with values when the function is used.

```
def greet(name):
print(f"Hello, {name}!")
# 'name' is the parameter of the 'greet' function
```

Here, name is the parameter of the greet function. It's a placeholder for the actual name that will be passed to the function when it's called.

Arguments:

- Arguments are the actual values that are passed to a function when it is called.
- They are the real data that fills in the placeholders (parameters) defined in the function. In simpler words, arguments are the values you provide to a function when you use it.

```
def add(a, b):
result = a + b
return result
# 3 and 5 are the arguments passed to the 'add' function
sum_result = add(3, 5)
```

Remember:

- You define parameters in the function definition.
- You provide arguments when you call the function.

Optional vs Required Parameters

Required Parameters

- Required parameters are parameters that must be provided with a value when calling a function.
- These parameters are essential for the function to work correctly, and not providing a value will result in an error.

```
def greet(name):
print(f"Hello, {name}!")
greet("Alice")
```

In this example, the name parameter in the greet function is required. When you call the function with greet("Alice"), you provide a value for the required parameter name.

Optional Parameters

- Optional parameters (also known as default parameters) are parameters for which a default value is specified in the function definition.
- If you don't provide a value for an optional parameter when calling the function, it will use the default value.
- However, you can still provide a value to override the default if needed.

```
def greet(name="Guest"):
print(f"Hello, {name}!")
greet() # Output: Hello, Guest! (Using the default value)
greet("Alice") # Output: Hello, Alice! (Providing a value to override the default)
```

In this example, the name parameter in the greet function is optional because it has a default value of "Guest". If you don't provide a value, it will use the default. But if you do provide a value, it will use that value instead.

Can there be cases where parameters are require but we dont pass arguments OR its necessary for arguments to be simultaneously be inline with parameters?

Yes, there can be cases where parameters are required, but you don't pass arguments when calling a function. However, it's important to understand that in such cases, not providing arguments will result in an error.

```
def multiply(a, b):
result = a * b
return result
# Uncommenting the line below will result in an error
# product = multiply()
```

In this example, the multiply function requires two parameters a and b to be provided with values. If you try to call the function without providing arguments like multiply(), you will get a TypeError because the function expects two arguments but receives none.
If you intend to make certain parameters required and ensure that the function won't work without them, you should provide appropriate arguments when calling the function.
On the other hand, for optional parameters (parameters with default values), you can call the function without providing arguments, and it will use the default values:

```
def greet(name="Guest"):
print(f"Hello, {name}!")
greet() # Output: Hello, Guest!
```

In this case, the name parameter is optional, so if you don't provide an argument, the default value "Guest" will be used.

In summary:

- For required parameters, you must provide arguments with values when calling the function, otherwise, you'll encounter an error.
- For optional parameters, you can omit providing arguments, and the function will use the default values specified in the parameter list.

Positional Arguments

- Positional arguments in Python refer to the arguments that are passed to a function in the order in which they are defined in the function's parameter list.
- When you call a function and provide values for its parameters without explicitly specifying the parameter names, Python assigns these values based on their position.

```
def add(a, b):
return a + b
result = add(3, 5)
print(result) # Output: 8
```

In this example above, 3 is assigned to the parameter a, and 5 is assigned to the parameter b because of their respective positions.

- Positional arguments are straightforward and intuitive to use, but it's important to remember that their order matters.
- If you change the order of arguments in the function call, the values will be assigned to different parameters

```
result = add(5, 3)
print(result) # Output: 8
```

However, using positional arguments only will lead to confusion and this is where Named Arguments come in to play.

Keyword Arguments (key=value)

- In Python, "named arguments" and "keyword arguments" refer to the same concept. Both terms are often used interchangeably to describe a way of passing arguments to a function by explicitly mentioning the parameter names along with the values.
- When you use named or keyword arguments, you provide the parameter names followed by the values you want to assign to those parameters.
- This approach allows you to pass arguments in any order, making your code more readable and reducing the chances of confusion, especially when dealing with functions that have many parameters.

```
def greet(name, age):
print(f"Hello, {name}! You are {age} years old.")
greet(name="Alice", age=30)
```

In this example, the function greet takes two parameters: name and age. When calling the function, you provide the values for these parameters using their names as keywords. This way, the order in which you provide the arguments doesn't matter.

- Named/keyword arguments are particularly useful when a function has many parameters, and you want to make your code more self-explanatory.
- When using named or keyword arguments, position does not matter at all:

```
def example_function(a, b, c):
print(f"a: {a}, b: {b}, c: {c}")
example_function(b=2, a=1, c=3)
```

- However, both keyword and positional arguments have their place in Python programming, and the choice should be based on the specific needs of your code.

Arbitrary Arguments

- Arbitrary arguments, often referred to as "varargs" (variable number of arguments), allow you to pass a variable number of arguments to a function in Python.
- This can be useful when you're uncertain about the number of arguments you need to provide or when you want to create functions that can handle a dynamic number of inputs.
- Python provides two ways to implement arbitrary arguments in function definitions: *args and **kwargs.

*Arbitrary Positional Arguments(*args):*

- The *args syntax allows you to pass a variable number of positional arguments to a function.
- These arguments are collected into a tuple within the function. You can use any name you like for *args, but the asterisk () is required to unpack the arguments.

```
def print_args(*args):
for arg in args:
print(arg)
print_args(1, 2, 3) # Output: 1 2 3
print_args('a', 'b') # Output: a b
```

*Arbitrary Keyword Arguments(**kwargs):*

- The **kwargs syntax allows you to pass a variable number of keyword arguments to a function.
- These arguments are collected into a dictionary within the function
- Similar to *args, the double asterisks () are required to unpack the arguments.

```
def print_kwargs(**kwargs):
for key, value in kwargs.items():
print(f"{key}: {value}")
print_kwargs(name='Alice', age=30) # Output: name: Alice, age: 30
print_kwargs(city='New York') # Output: city: New York
```

You can also use both *args and **kwargs in the same function definition, but *args should appear before **kwargs:

```
def combined_example(arg1, *args, kwarg1=None, **kwargs):
print(f"arg1: {arg1}")
print(f"args: {args}")
print(f"kwarg1: {kwarg1}")
print(f"kwargs: {kwargs}")
combined_example(1, 2, 3, kwarg1='value', name='Alice', age=30)
```

In this example, arg1 is a required positional argument, *args captures additional positional arguments, kwarg1 is a keyword argument with a default value, and **kwargs captures additional keyword arguments.

- Arbitrary arguments provide flexibility, making it possible to create functions that can accept a varying number of inputs without needing to specify each argument explicitly.

NOTE: When you use the **kwargs syntax in a function definition, the arguments that are passed as keyword arguments are collected into a dictionary within the function. This dictionary is then accessible within the function, and it contains the keyword argument names as keys and their corresponding values.

```
def process_kwargs(**kwargs):
print(kwargs)
print(type(kwargs))
process_kwargs(name='Alice', age=30, city='New York')
# output
{'name': 'Alice', 'age': 30, 'city': 'New York'}
<class 'dict'>
```

You can then use this dictionary to perform various operations within the function, such as iterating over the keys and values, accessing specific values using their keys, and so on. This feature is particularly useful when you want to create functions that are highly customizable and can accept a variable number of named arguments.

Lambda Functions

Lambda is:

- One Line Function
- Without Name (Anonymous name)
- It didnt use before nor it'll be used afterwards This is called Lambda Functions
- A lambda function can take any number of arguments, but can only have one expression.
- *Syntax* of Lambda Function is:

```
lambda arguments : expression
```

- lambda is the keyword o arguments may be any for example x,y
- then any expression for example x+y separated by colon :

```
a = lambda x,y,z : x+y+x
print(a(5,5,5))
```

- Lambda function is returned by default. No need to return it separately.
- Use lambda function when you require an anonymous function for a Short Period of Time
- However, it is important to note that *lambda* function must be assigned to a variable or be used inside a function. If just used *lambda*, it will give an error.

Why Use Lambda Function?

- Lambda functions are efficient whenever you want to create a function that will only contain simple expressions
- that is, expressions that are usually a single line of a statement.
- They're also useful when you want to use the function once.

```
data = [[1,'X','E'],
[3,'Y','G'],
[2,'Z','F']]
print(sorted(data, key=lambda x:x[0]))
print(sorted(data, key=lambda x:x[1]))
print(sorted(data, key=lambda x:x[2]))
```

In the above example, x on the first instance is the matrix of data and other x is the column of the matrix and [ ] passes the index of the columns.

- Lambda functions are useful for creating functions on-the-fly and are often employed with functions like map(), filter(), and sorted() for efficient code.

Error Handling

- Error handling, in simple terms, is the process of dealing with unexpected or exceptional situations that can occur during the execution of a program.
- It involves identifying when something goes wrong in your code and taking appropriate actions to prevent program crashes or undesirable outcomes.

Error handling is a crucial skill for any programmer because it allows you to gracefully handle unexpected situations that may arise during the execution of your code. There are several mechanisms in python that facilitates Error Handling:

1. Try
2. Except
3. Else
4. Finally

In programming, errors can come in various forms, such as invalid inputs, file not found, division by zero, and many others. Error handling allows you to address these issues so that your programs can handle adversity and continue functioning as intended.

Types of Errors

In python, we can encounter the following types of errors:

Development Time Error are the errors encountered during developing a program. It is easy to locate and identify. These types of errors occur when we write codes for example non variable declaration, type error, zero division error etc.

Logical Error is hard for interpreter to identify. As of now, there is no tool to detect a logical error. The only best option available is to test the program. This type of error is given to seniors or Quality Assurance Team to perform the Testing. Testing is done through Dry Runs (Step by step)

```
def add_two_numbers(a,b):
print(7,7, type(a),type(b)) # thats how dry testing is done like see if the values are passed to a and b and if the data types are same.
return a - b
print(add_two_numbers(7,7)) # Create case 7 + 7 = 14, False
```

Production/Runtime Error can cause great loss. This error occurs during the running of an application. In businesses, this can cause a huge loss to an entity. The Application
should be capabale of running 24/7. This error can also be caused by a user who puts an invalid function or pressure to The server.

Try, Except, Else and Finally

- Try or try is used to prevent an error occured from the user side.
- Try is used from a user perspective.
- This is where you enclose code that you expect might raise an exception or error.
- Try lets us test the block of code
- Except block lets us handle the error
- It is a good practice to place the name of an error that may be encountered after an *except* keyword. For Example, *except* NameError or *except* (NameError, ZeroDivisibleError etc)
- The else block lets you execute code when there is no error.
- The finally block lets you execute code, regardless of the result of the try- and except blocks.

Syntax

```
try: # We can pass classes
**logic** # Maybe error occur here.
except (errorclass1, errorclass2):
**action** # What can be performed to prevent the error
else:
# run if error not occur
finally:
# line will always run
```

Example 1:

```
print("Talaal")
print("Amna")
try:
  print(a)
except NameError: # NameError is the class we defined that will encounter due to non declaration of object
  print("Variable Not Defined")
print(75)
print("Yaaay!")
```

Example 2:

```
num1 = 7
num2 = 0
b = 20
try:
  print(20)
  print(num1 / num2) # Error Occured
  print("Hello") # No print
  print("World") # No Print
except (NameError, ZeroDivisionError):
  print("Invalid Division") # After error, will directly come here
print("Pakistan") # This will be printed
print("Zindabad") # This will be printed too
```

If python encounters error in any line of Try Block, it will directly go to except block and then the cursor will move ahead not backwards. To solve this issue, we can individually *perform exception handling*

```
num1 = 24
num2 = 0
b = 25
print("Talaal")
print("Yousoof")
try:
  print(num1/num2)
except ZeroDivisionError:
  print("Zero Cannot Be Divided")
try:
  print(d)
except NameError:
  print("Variable d is not defined")
print("Pakistan")
print("Zindabad")
```

Exception Handling

- Exception handling is a programming concept and a set of techniques used to manage and respond to exceptional or unexpected events
- Exception handling is not specific to Python but is a common practice in many programming languages.
- It allows you to gracefully handle errors and unexpected situations, preventing program crashes and ensuring that your code continues to run smoothly.

Key Components of Exception Handling are:

1. Throwing or Raising Exceptions Exceptions can be raised by raise keyword.
2. To handle exceptions, we can use combination of try, excetp, else and finally.
3. Handling Logics Inside the except block, we can define logics to respond to exceptions
4. You can include an else block, which runs when no exceptions occur in the try block
5. The finally block is optional but runs regardless of whether an exception occurs or not. It's often used for cleanup tasks, like closing files or releasing resources.

```
num1 = 100
num2 = 10
b = 10
list1 = [1,2,3,4,5,6,7,8,9]
print("Talaal")
print("Yousoof")
try:
  print(b)
  print(num2*num1)
  print(list1[20])
  print("Hey Bro")
except Exception as e:
  print(f'Something is Wrong! {e}')
print("Pakistan")
print("Zindabad")
```

We can also create our own errors.

```
class StudentCard():
    def __init__(self, roll, name, age):
        if age < 18 or age > 70:
            raise Exception("Not Allowed")
self.roll = roll
self.name = name
self.age = age
s1 = StudentCard(1, "Talaal", 27)
print(s1.name)
print(s1.roll)
print(s1.age)
try:
 s1 = StudentCard(1, "Mehwish", 17)
except Exception as e:
 print(e)
```

```
try:
 num1 = int(input("Enter a number: "))
 num2 = int(input("Enter another number: "))
 result = num1 / num2
 print("Result:", result)
except ZeroDivisionError:
 print("You cannot divide by zero.")
except ValueError:
 print("Invalid input. Please enter a valid number.")
except Exception as e:
 print(f"An unexpected error occurred: {e}")
else:
 print("No exceptions were raised.")
finally:
 print("This will always be executed.")
```

Except Exception as e

- we might want sometimes to use the expression except Exception as e which is breakdown as under:
- ***except*** keyword: It signals the beginning of an exception-handling block.
- ***Exception*** is a base class for all exceptions in Python. By specifying Exception, you're indicating that you want to catch any exception that inherits from the Exception class.
- ***as e***: This part allows you to assign the caught exception object to a variable named
- ***e***. You can use this variable e to access information about the exception, such as its type or the error message it contains.

The use of except Exception as e is a common and widely accepted convention for handling exceptions in Python, but it's not a strict requirement. You can use different variable names instead of e if you prefer.

Hot Takes

- When we give a class name of error to except block, this approach is known to catch a specific exception by their class names. For example if we print(p), a NameError will arise because p is not defined.
- It's recommended to give class names to except blocks when you know the specific types of exceptions that your code might encounter and when you have a clear plan for handling those exceptions differently.
- This approach improves code clarity and maintainability, making it easier to understand how your program responds to different types of errors.

File Handling

File Management System

- File management system refers to handling of multiple files such as text files, audio files, video files etc.
- *Append* mode keeps old data as it it is and helps to create a new dat as well
- *open* function is used to open a file.
- If you're working in a terminal and you want to know which directory you're currently in, you can simply type ***pwd*** (Print Working Directory) and press Enter. The terminal will then display the full path to your current working directory, like /home/username/Documents or C:\Users\username\Documents, depending on your operating system.
- **pwd** provides a full path to directory and particularly useful when you need to reference or work with files and directories using relative paths, as it helps you understand your current position within the directory structure.

There are two types of paths in File Management System:

Absolute Path:

- An absolute path starts from the very beginning, at the root of your computer's file system, and gives the full, exact location of a file or folder.* Example: C:\Users\YourName\Documents\file.txt (for Windows) or /home/yourname/Documents/file.txt (for Linux).
- In most of the cases, it is a practice to give absolute path as a major path to open a file.
- Absolute path acts like a GPS giving complete full address; county, city, town, street, house

Relative Path

- Imagine a relative path as giving directions from your current location, like saying "go two blocks down and turn left."
- A relative path is described based on your current location within the file system. It's relative to where your program is running.
- Example: ../../folder/file.txt means "go up two levels in the directory structure, then find the 'folder' and 'file.txt' inside it." The Identifiers in Relative Paths:
- ./ Current Location
- ./xyz/aa/xyz.txt ../ the double dots before forward slash means we have moved one step behind.
- ../../ means we have moved one folder behind the current.
- It does not require 100% path.

Difference Between Absolute and Relative Path

- Absolute paths start from the root and provide the complete location. They are fixed and unchanging.
- Relative paths are described based on the current location and adapt as you move around the file system.
- The main difference is in how they specify file locations, with absolute paths being more explicit and relative paths being more adaptable.

File Modes in Python

- UTF-8 is a mode which has almost 14 million characters and are mapped in ASCII Codes. Best example is having multiple charachters of different languages together.
- UTF-8 is a standard mode There are several modes in python:
- *r* means to read file. Reads every data of file. It is a default mode.
- *r+* means read and write
- *w* means write only.
- *w+* means Write and Read
- *a* means append mode. Adds after last element. Append creates a file if no file is created previously. Prior data is not changed
- *a+* means read and add in the same system. Prior data is not changed
- *x* creates a new file if not existing. If ile already exists, it will generate an error.

```
f = open("README")
print(f.readline()) # Reads the first line
print(f.readlines()) # Returns comma separated lines in the form of elements
print(f.read()) # Reads every Data
print(f) # Returns the file location and type
```

close() function is always recommended to include when reading a file if NOT used with "**with**" block.

With Block

✶ The **with:** block in Python is versatile tool for making our code more readable and helps prevent resource leaks.
✶ Without the **with:** block, we will be needing to explicitly *open* and *close* the file, which can lead to mistakes if we forget to close it or if an exception occurs before the file is closed.
✶ It reduces the risk of resource leaks and maintains the integrity of resources preventing data loss or corruption
✶ When we use **with** block, we don't manually need to *close()* the file at the end of the block but if we do not use with block then we have to manually close it otherwise dangers can occur.

```
# Without With Block

file = open("Example.txt", "r")
try:
    data = file.read()
finally:
    file.close()

# With "with" block

with open("Example.txt", "r") as file:
    data = file.read()
# File automatically closes when with block is exits
```

Regular Expressions

Regular Expressions (RegEx)

1. A Regular Expression or RE or Regex is a sequence of characters that matches the character sequence with search pattern.
2. Functions in the module let us check if the specified string matches given RE.
3. Regex can contain both Special Characters and Ordinary Characters. Most Ordinary Characters like A or a or 0 are simplest Regular Expressions.
4. Regular Expressions in Python are powerful tool for complex strings and text data.
5. RegEx are always imported through re module. Regular Expressions are even though complex but makes easy to search characters in strings rather using long codes of FOR, PARSE etc
6. *Syntax* of Regular Expressions is as follows:

```
re.findall(pattern, string, flag)
pattern = the defined criteria of what we want to find or search. Pattern can be either Ordinary Characters or
Special Characters.
string = the place from where we want to find.
flag = Flag is if there is specific search criteria. Default is "False" which is 0. Flag = 1 is True
# Please note that at this stage, we only covered findall() function.
```

7. Regular Expressions are outputted in Lists.
8. We cannot pass 2 Flags in the same function but in different we can:

```
re.findall(pattern, string, re.Multiline)
re.findall(pattern, string, re.IGNORECASE)
```

9. REs allow us to manipulate strings on the pattern we define making it easy to perform Operations.

Patterns

In Regular Expressions (RegEx), a pattern refers to the sequence of characters that defines a specific search criteria. Patterns are the heart of regular expressions; they allow you to describe a set of strings that you want to match or manipulate within a larger text. Patterns can include a combination of ordinary characters, special characters, and metacharacters that have special meanings within the context of RegEx.

1. Combination of Characters: Patterns are built by combining different characters and metacharacters. For example, abc is a simple pattern that matches the sequence "abc" in a string.
2. Special Characters: Special characters have predefined meanings in RegEx. For example, . matches any character except a newline, * matches zero or more occurrences, and + matches one or more occurrences.
3. Characters Classes: Character classes allow you to define a set of characters that can match at a certain position. For instance, [aeiou] matches any vowel.
4. Quantifiers: Quantifiers determine how many times an element should occur. For example, {2,5} means the preceding element should occur between 2 to 5 times.
5. Anchors: Anchors define positions within the text. ^ and $ anchor to the start and end of a line (or string with re.MULTILINE).

6. Alternation: The pipe symbol | allows you to create alternatives within a pattern. For example, cat|dog matches either "cat" or "dog".
7. Groups and Captures: Parentheses () define groups and capture portions of the matched text. These can be used for extracting specific parts of the match.
8. Escape Characters: Some characters have special meanings in RegEx. To match them literally, you need to escape them with a backslash \.
9. Modifiers(Flags): Flags modify how a pattern is applied, like making it case-insensitive or enabling multiline matching.

Here's a simple example of a pattern: Suppose you want to find all email addresses in a text. The pattern for a basic email address might look like \w+@\w+\.\w+, where
o \w matches any word character and
o "+" indicates one or more occurrences.

Flags

In the context of Regular Expressions (RegEx), a flag is an optional modifier that you can apply to a regular expression pattern. Flags control how the pattern is matched against the input string. They provide additional instructions to the RegEx engine. Some common flags include:
1. **re.IGNORECASE**: Makes the pattern case-insensitive, so it matches both uppercase and lowercase characters.
2. **re.MULTILINE**: Changes the behavior of ^ and $ anchors to match the beginning and end of lines within a multiline string.
3. **re.DOTALL**: Makes the dot (.) match any character, including newline characters.
4. r**e.VERBOSE**: Allows you to write more readable and organized patterns using whitespace and comments.

Special Characters

In Regular Expressions, special characters are symbols that have special meanings and are used to define patterns for searching and manipulating text. These characters are used to represent certain classes of characters or to specify how many times a certain element should repeat

The Dot (.) Character

Dot (.) matches any character except backslash n or \n that is new line character.

```
# Dot . Character
import re
string = "Talaal Yousoof is currently Studiyng Data Sciences"
pattern = '.'
dotmatch = re.findall(pattern, string)
print(dotmatch) # since there is only one dot it matches only one character except \n had there been multiple dots, it would have matched the correspondent characters as per dots defined.
# Dot with preceding character
pattern2 = '.a'
dotmatch2 = re.findall(pattern2, string)
print(dotmatch2) # . will match each character preceding a
['Ta', 'la', 'Da', 'ta']
# Dot with Proceding Character
pattern3 = 'a..'
dotmatch3 = re.findall(pattern3, string)
print(dotmatch3)
# Output : ['ala', 'al ', 'ata']
```

Caret (^)

Matches the start of the String. In MULTILINE mode also matches immediately after new line.
If you use the caret (^) at the beginning of your RegEx pattern, it will look for matches that occur at the very beginning of a line or a piece of text. For example, if you have the RegEx pattern ^Hello, it will only match instances of "Hello" that appear at the very beginning of a line or string, and not if "Hello" appears later in the text.

```
# Caret Character
string2 = "Hello I am Talaal \n Hello I am Omer \n Hello World"
pattern_caret = "^Hello"
caretmatch = re.findall(pattern_caret, string2)
print(caretmatch) # Output ['Hello']
```

This is single output because its not MULTILINE. Since in the example, Hello is appearing thrice, we can use MULTILINE Flag:

```
string2 = "Hello I am Talaal \nHello I am Omer \n Hello World"
pattern_caret = "^Hello"
caretmatch = re.findall(pattern_caret, string2, re.MULTILINE)
print(caretmatch)
# Output => ['Hello' , 'Hello']
```

Why wasn't the third "Hello" matched? Notice the space after new line (\n), thats a character.

```
CaretString = """Python was invented by Goidum Van Rossum
He is from Netherlands"""
caretpattern = "^P..+"
matchCaret = re.findall(caretpattern, CaretString)
print(matchCaret)
# Output => ['Python was invented by Goidum Van Rossum']
```

Plus (+)

Plus + causes resulting REs to match 1 or more repetitions of preceding REs. ab+ will match "a" followed by non-zero numbers of "b's"; it will not just match "a". Character should appear at least once.

```
text = """Pakistan Zindabad! Pakistan is in Asia
Pakistan was Founded by Quaid-e-Azam"""
pluspattern = ".+"
plusmatch = re.findall(pluspattern, text)
print(plusmatch) #It printed whole sentence until line break. + matches the character that should appear at least once. In this case, dot matches every character individually since there is a single dot, and plus concatenates them.
```

Note: Had thre been no element before plus OR element after plus, it would have resulted in an error because it matches preceding character occuring at least once and there is either nothing to repeat at position 0.

Dollar ($)

Matches end of the string or just before the new line at end of string and in MULTILINE also matches before new line. In simpler terms, if you use the dollar sign ($) at the end of your RegEx pattern, it will look for matches that occur right at the very end of a line or a piece of text. It's like a marker that tells the RegEx engine to focus on finding patterns only at the end of the input.
For example, if you have the RegEx pattern Hello$, it will only match instances of "Hello" that appear right at the very end of a lin or string, and not if "Hello" appears earlier in the text.

```
import re
text = "Hello, this is a sample text. Hello"
# Using $ to match "Hello" at the end of lines
pattern = 'Hello$'
matches = re.findall(pattern, text, re.MULTILINE)
print(matches)
# Output => ['Hello']
```

In this example, the text contains two instances of "Hello." However, only the last "Hello" at the end of the text is matched by the pattern with the dollar sign ($) at the end.

Asterisk (*)

Causes resultant REs to match 0 or more repetitions of preceding REs, as many repititions are possible. ab* will match 'a', 'ab' or 'a' followed by any number of "b's". REs can appear 0 or as many unlike + that has to be at least 1. If empty text, * will return empty [].

Question Mark (?)

Will match 0 or 1 of preceding REs. ab will match either a or b. In short, it means optional.
1. It indicates that the preceding element in the pattern is optional.
2. This means that the preceding element can occur either zero times or exactly once.
3. In simpler terms, if you use the question mark (?) after a character or group of characters in your RegEx pattern, it means that the RegEx engine will consider that character or group as optional. It can be there, or it might not be there at all, and the pattern will still be considered a match.
4. For example, if you have the RegEx pattern colou?r, it will match both "color" and "colour." The question mark makes the "u" in "colour" optional, so the pattern will match regardless of whether the "u" is present or not.

```
import re
text = "The color of the sky is blue. The colour of the ocean is also blue."
# Using ? to make "u" optional in the pattern
pattern = r'colou?r'
matches = re.findall(pattern, text)
print(matches)
```

In this example, the pattern colou?r matches both "color" and "colour" in the text. The question mark allows the "u" to be there or not, making the pattern flexible enough to match both variations.

Other Characters

1. **[]** is used to indicate set of characters. In a Set:
   - Characters can be listed Individually eg [aml] will match a, m or l.
   - Ranges of characters can be indicated by giving two characters and separate them by -, like [a-z] will match any lowercase ASCII letter.
   - Special characters lose their speciality inside Brackets []. Meaning that any special character will be treated as a normal character inside [].
2. **Curly Braces {m,n}** causes Resulting REs to match from "m" to "n" attempting to match as many as possible.
3. **\d** Matches unicode decimal digit [0-9]
4. **\w** matches [A-z] Rule to convert any special character in to normal character, use \ followed by a special character. Example:

pattern = "." # any character without \n
pattern = "\." # Ordinary dot character.
pattern = "\+" ordinary plus character

Word Boundary Concept

The word boundary concept in Regular Expressions allows you to match patterns that are at the edges of words or surrounded by non-word characters. A word boundary \b is a zero-width assertion that doesn't consume any characters but asserts a specific position in the string. It's often used to match whole words.

\b: Matches a position where a word starts or ends. It's often used as ***\bword\b*** to match the entire word "word."

1. Word Boundary is denoted by \b. This concept checks the character boundary as defined.
2. Matches the empty string, but only at the beginning or end of a word.
3. A word is defined as a sequence of word characters.
4. To apply Boundary concept, use r before pattern pattern = r'STRING'
5. Use the r'\bword\b' pattern to match the whole word

```
import re
# To find all occurences of a word
text = "Hello Everyone. Hello, Sir. Hello Mam. Hello Teammates. Talaal, Hello"
pattern = r"\bHello"
word = re.findall(pattern, text)
print(word)
# To find whole words
pattern2 = r"\bhello\b"
word2 = re.findall(pattern2, text, re.IGNORECASE)
print(word2)
# Matching a Word at End of String only
pattern3 = r"Hello\b"
word3 = re.findall(pattern3, text)
unique_match = list(set(word3)) # Convert to set to get unique matches
print(unique_match) # To match  hello at end of the string only
```

A string has the following positions that qualify as word boundaries:

1. Before the first character in the string if the first character is a word character (\w).
2. Between two characters in the string if the first character is a word character (\w) and the other is not (\W - inverse character set of the word character \w).
3. After the last character in a string if the last character is the word character (\w)

In the example Python 3! has four words:

1. Before the letter P (criteria #1)
2. After the letter N (criteria #2)
3. Before the digit 3 (criteria #2)
4. After the digit 3 (criteria #2)

Word boundaries are important for ensuring precise matching of words within a larger text. They help avoid partial matches and ensure that you're matching complete words rather than substrings within words.

Zero Width Assertion

A zero-width assertion is a concept in regular expressions that refers to a condition that must be satisfied at a specific position in the string, but it doesn't consume (match) any characters from the string. There are 4 types of Zero-Width Assertions:

1. **Positive Lookahead Assertion (?=...)**:

Positive look ahead is a zero-width assertion that checks if a certain pattern is followed by another pattern without consuming any characters. It's denoted by (?=...).

```
# Suppose you want to find all occurrences of "apple" that are followed by "pie".
import re
text = "apple pie is delicious, apple is good, apple tart is amazing"
pattern = r'apple(?= pie)'
matches = re.findall(pattern, text)
print(matches) # Output: ['apple']
```

In this example, the pattern apple(?= pie) matches only the occurrences of "apple" that are followed by a space and then "pie". The lookahead assertion (?= pie) ensures that "pie" is present after "apple" without actually including "pie" in the match.

2. **Negative Lookahead (?!..)**:

Negative lookahead is a zero-width assertion that checks if a certain pattern is not followed by another pattern. It's denoted by (?!...).

```
# Suppose you want to find all occurrences of "apple" that are not followed by "pie"
import re
text = "apple pie is delicious, apple is good, apple tart is amazing"
pattern = r'apple(?! pie)'
matches = re.findall(pattern, text)
print(matches) # Output: ['apple', 'apple', 'apple']
```

In this example, the pattern apple(?! pie) matches all occurrences of "apple" that are not followed by a space and then "pie". The negative lookahead (?! pie) ensures that "pie" is not present after "apple" without actually including "pie" in the match.

3. **Positive Lookbehind (?<=...>):**

Positive lookbehind is a zero-width assertion that checks if a certain pattern precedes the current position in the string. It's denoted by (?<=...).

```
# Suppose you want to find all occurrences of "pie" that are preceded by "apple".
import re
text = "apple pie is delicious, pineapple tart is amazing, blueberry pie is sweet"
pattern = r'(?<=apple )pie'
matches = re.findall(pattern, text)
print(matches) # Output: ['pie']
```

In this example, the pattern (?<=apple)pie matches the occurrences of "pie" that are preceded by "apple " (with a space). The positive lookbehind (?<=apple) ensures that "apple " appears before "pie" without including "apple " in the match.

4. Negative Lookbehind (?<!...):

Negative lookbehind is a zero-width assertion that checks if a certain pattern is not preceding the current position in the string. It's denoted by (?<!...).

```
# Suppose you want to find all occurrences of "pie" that are not preceded by "blueberry".
import re
text = "apple pie is delicious, pineapple tart is amazing, blueberry pie is sweet"
pattern = r'(?<!blueberry )pie'
matches = re.findall(pattern, text)
print(matches) # Output: ['pie', 'pie']
```

In this example, the pattern (?<!blueberry)pie matches occurrences of "pie" that are not preceded by "blueberry " (with a space). The negative lookbehind (?<!blueberry) ensures that "blueberry " is not present before "pie" without including "blueberry " in the match.

Example 1 - Match Dates

```
datestring = """I was Born in 19/10/1195
Sister 1 in 19*08*1996
Sister 2 in 9.3.2000
"""
patterndate = "\d{1,2}[*/,.]\d{1,2}[*/,.]\d{2,4}"
matching = re.findall(patterndate, datestring)
print(matching)
```

I know that \d is used for decimal digits 0-9 and {1,2} in first 2 instances means digit can be either 1 or 2 similarly 2 or 4 (in last curly braces). [[*/,.]] means that the pattern can have any of the character in the string

The Parenthesis - ()

- If we need any desired data from pattern, use parenthesis. For example if we want date, name and roll number from pattern, we place them between ().
- Pattern is necessary for Parenthesis
- By placing part of a regular expression inside round brackets or parentheses, you can group that part of the regular expression together.

Example:

I = "12:05:45 From Talaal Yousoof to Everyone : DDOS-4578965"
lets use `()` to modify the code
modify = "(\d{2}:\d{2}:\d{2} From (.*) To Everyone : (DDOS-?\d{5,7})"

For Complete guide on Metacharacters, Sets, RegEx Functions and Special Sequences: See **Appendix A**

Object-Oriented Programming (OOP)

Introduction to Object-Oriented Programming

In python, there are mainly two approaches that are used to write programs:

1. Procedural Programming/Structured Programming
2. Object-Oriented Programming (OOP)

Till now, we have followed Procedural Programming for example defining function, making logical statements using If-Else, Looping etc. We can write structured/procedural code using functions and modules.

Procedural Programming is about writing procedures of functions that perform operations on data, while Object- Oriented Programming is about creating objects that contains both, data and function.

- OOP is a programming paradigm that organizes code into objects, which are instances of classes.
- Its significance lies in promoting modularity, reusability, and a clear structure for managing and manipulating data.
- It helps model real-world concepts more naturally in code.
- Imagine you're creating a program to manage a library. You can use OOP to model the objects in your program, such as Book, Library, and Member, making your code more organized and intuitive.
- Basic concept of OOP in Python is to use classes and objects to represent real-world concepts and entities.
- Python is known for strong support for OOP with features like classes, objects, inheritance and encapsulation.

Linear Programming is not a programming paradigm as Structured and OOP, instead it is a technique that is used to solve mathematical models with linear relationships. Poplar libraries are ***SciPy*** and ***PuLP***

Modular Programming is a software design and development approach that emphasizes breaking code in to small and reusable modules or components. Each module focuses on specific task. Python encourages modular programming through use of modules and packages. It is not a programming Paradigm.

Class

- A class is like a blueprint or template for creating ***objects***.
- It defines the properties (attributes) and methods that a class will have.
- **Methods** are actions that can be performed by an object whereas, **Attributes** are traits of that object.
- To create a class, use the class keyword.
- Example: A class Fruit can contain objects like *Apple, Banana, Mango* etc.
- So a class is a template of object and an object is an instance of class.
- When individual objects are created, they inherit all variable and functions from that class.

Example 1:

```
class Mycar:
    def __init__(self,make,car,model,displacement):
        self.make = make
        self.car = car
        self.model = model
        self.displacement = displacement
# When creating an instance of the class, you should pass the values as
arguments to the constructor,
not directly in the class definition.
Car1 = Mycar(2023,"Mercedes","Benz 7","2800cc")
print(Mycar)
print(f'''The car I got for myself is {Car1.car}. It has an Engine
Displacement of {Car1.displacement}. It was made
in {Car1.make} and the model is {Car1.model}''')
```

Example 2:

```
class Mycar:
    def __init__(self, make, car, model, displacement):
        # Assigning values to instance attributes
        self.make = make
        self.car = car
        self.model = model
        self.displacement = displacement

    def get_car_info(self):
        # Accessing instance attributes within a method
        return f"The car I got is {self.car}. It is {self.model} which
has an Engine Displacement of {self.displacement}. It is made in
{self.make}"

# Create an instance of Mycar with the desired values
car1 = Mycar(2023, "Mercedes Benz Rappot", "Sedan", "2800cc")

# Access the attributes and use them in a method
car_info = car1.get_car_info()
print(car_info)
```

Example 3:

```
class Phone:
    def __init__(self,model,year,battery,camera,dimensions):
        self.model = model
        self.year = year
        self.battery = battery
        self.camera = camera
        self.dimensions = dimensions
    def myPhone(self):
        return f'''I purchased {self.model} back in {self.year}. At that time, It
had a battery life of
{self.battery}%. It comes with {self.camera} megapixels and its Screen size is
{self.dimensions} inches'''
# Create Instance of Phone
IPhone = Phone("Iphone XS Max", "2022","90",16,5.5)
Phone_Info = IPhone.myPhone()
print(Phone_Info)
```

Note : In Python, it's a convention (not a rule) to name classes using CamelCase, which means starting with an uppercase letter.

Reasons of When to Create a class

1. **Abstraction**: Classes allow you to abstract complex real-world concepts into code. For example, you can create a Car class to model cars in your program, encapsulating their attributes and behaviors.

```
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start_engine(self):
        print(f"{self.make} {self.model}'s engine started.")

    def stop_engine(self):
        print(f"{self.make} {self.model}'s engine stopped.")
```

2. **Code Reusability**: Classes enable you to define reusable templates for objects. You can create multiple `Car` objects with the same attributes and methods.

```
car1 = Car("Toyota", "Camry", 2022)
car2 = Car("Honda", "Civic", 2023)
```

3. **Encapsulation***: Classes encapsulate data (attributes) and behavior (methods) into a single unit, making it easier to manage and understand your code.
4. **Organization**: Classes help organize your code by grouping related data and functions together. This improves code readability and maintainability.
5. **Inheritance**: You can use classes to create hierarchies and relationships between objects. For example, you can have a base class `Vehicle` with subclasses like `Car`, `Motorcycle`, and `Truck`, inheriting common properties and behaviors.

When not to Use class

Classes should be avoided when:

1. Our codes are very **simple**
2. We are performing **Procedural Programming**
3. Creating unnecessary classes can cause overheads which hinders **Performance**.
4. Avoid creating classes just for the sake of OOP if it does not make code clear and more readable. Avoid **Overengineering**

Create Object

- After a class has been named, we can use it to create an object.

In Continuation of Below Example

```
# Creating objects of the class
dog1 = Dog("Buddy", "Golden Retriever")
dog2 = Dog("Rex", "German Shepherd")
```

Dunders (Double Underscores __)

- Double underscores, also known as "dunders," are used to create special methods and attributes in Python classes.
- They are not limited to classes but are often associated with them. For example, **__init__** is a special method used for initializing instance of classes.

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

    def __str__(self):
        return f"{self.name} is a {self.breed}."

dog1 = Dog("Buddy", "Golden Retriever")
print(str(dog1))  # Outputs: "Buddy is a Golden Retriever."
```

The Self Parameter

- The `self` parameter is used to access and modify object attributes within the class methods.

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

    def bark(self):
        return f"{self.name} barks!"

dog1 = Dog("Buddy", "Golden Retriever")
print(dog1.bark())  # Outputs: "Buddy barks!"
```

- However, it does not have be named self, you can call it by whatever name BUT **has to be first parameter of any function**.

```
class Person:
  def __init__(mysillyobject, name, age):
    mysillyobject.name = name
    mysillyobject.age = age

  def myfunc(abc):
    print("Hello my name is " + abc.name)

p1 = Person("John", 36)
p1.myfunc()
```

Special Methods

Python supports special methods which are:

1. **__ init __()**: This is the constructor method used to initialize object attributes.
    - **__ init __()** is a built-in function and all classes have a function called __ init __
    - This is always executed when a class is created.
    - Use the **init**() function to assign values to object properties, or other operations that are necessary to do when the object is being created

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

p1 = Person("John", 36)

print(p1.name)
print(p1.age)
```

Note: The **init**() function is called automatically every time the class is being used to create a new object.

2. **__ str __()**:

- This function controls what should be returned when the class object is represented with string
- If __ str__() function is not set, the string representation of object is returned. String representation without __ str__() function:

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

p1 = Person("John", 36)

print(p1)
```

String representation WITH __ str__() function:

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

  def __str__(self):
    return f"{self.name}({self.age})"

p1 = Person("John", 36)

print(p1)
```

3. **__ main__()**:
    - This is not a commonly used special method in classes
    - Typically it is found **if __ name __ == "__ main__:"** in scripts to see if the script is being run directly

These described methods are the commonly used methods. To see a full list, check **Appendix B** at the end of this chapter.

Constructors

- Constructors like `__init__()` are used to initialize object attributes when an object is created.
- They are called automatically when you create object of class.
- You can create your own constructors by creating custom **init()** methods in your class.
    - In the examples above, The **init** method in the `Dog` class serves as a constructor, initializing the `name` and `breed` attributes when a new `Dog` object is created.
    - Constructors cannot be defined outside the class as they are integral part of OOP and are used to set initial state of objects

STEP BY STEP GUIDE ON HOW TO ***CREATE OBJECT WITH CLASS***

Step 1: *Make a Class with* "class" keyword and assign it a name:

```
class Fruits():
```

Step 2: Create a Constructor with "__init__" using "def" Keyword and Pass Arguments

```
def __init(self,arg1,arg2...argN):
```

Step 3: Assign Values to Instance Attributes

```
self.arg1 = arg1
self.arg2 = arg2
self.argN = argN
```

Step 4: Access Instance Attributes within a Method

```
def myFruits(self):
  STATEMENTS/BODY
```

Step 5: Create Instance with Desired Values

```
Variable_Name = Fruits(value1,value2... valueN)
```

Step 6: Access Attributes and Use them in a Method

```
PrintableVariable = Variable_Name.myFruits()
print(PrintableVariable)
```

Illustration 1: Create a class "Honda" and objectify its Specifications of its different Variants

```
# Step 1
class Honda():
# Step 2
    def __init__(self,model,year,color,engine,speed):
# Step 3
        self.model = model
        self.year = year
        self.color = color
        self.engine = engine
        self.speed = speed
# Step 4
    def HondaCars(self):
     return f"""Company: Honda
               Model : {self.model}
               Year of Launch = {self.year}
               Colors Available in: {self.color}
               Engine Displacement" {self.engine}
               Maximum Speed = {self.speed}"""
# Step 5
civic = Honda("Civic X","2023","Metallic Gray/Black/White/Silver","1800CC", "260KM/h")
city = Honda("City IDSI",2021,"Black/Maroon/Blue","1500CC","180KM/h")
accord = Honda("Accord Evolution",2022,"Gray/Navy Blue,Metallic Silver","2400CC","180mp/h")

# Step 6
civic_car = civic.HondaCars()
city_car = city.HondaCars()
accord_car = accord.HondaCars()

print(civic_car)
print(city_car)
print(accord_car)
```

Tips and Tricks

This is how we create an object with the help of a class:

1. Define class and assign a name
2. With the help of __**init**__ constructor, create instances of class and pass arguments
3. Once arguments have been passed, then create **attributes** or **properties** of those instances. These instances represents the ***properties*** of Class Name in point 1.
4. Now the properties have been created, If you want you can create **Methods.**
5. **Methods** are created with ***def*** and name. This is basically the actions that a class will have for the instances. Methods are defined inside the class.
6. So, a class is a blueprint or a template used to create an object of that class. Class contains two portions; Attributes (point 2) and Methods (point 5)
7. Once the methods are defined and attributes are passed, we can then create Objects by passing the arguments of that in __init__.
8. We create objects based on the class by using the instances. Objects inherit attributes and methods defined inside the class.
9. In **OOP** it's a common practice to use __init__ constructor to initialize attributes of the class because attributes represent the data or properties associated with objects.
10. On the other hand, methods are used to define behavior that an object will have. Its an option for us whether an object to have method or not. Methods are always created inside class using def keyword.
11. Some classes may have multiple methods while some may not have any depending up on the code requirement
12. We can include any valid python code within methods for example:
    - Loops (For/While)
    - Logical Statements (if,elif,else)
    - Any other valid python statement or expression

CLASS VARIABLES

- Value of a variable that is same for every class is called class variable
- Class variable is shared by all instances (objects) of class
- They are defined within the class but outside of any instance methods.
- Class variables are associated with class itself rather than with specific instances of class.
- Class Variable is always built inside class
- For example in a mosque, everyone has a watch in which time may differ
  but Salah will be offered according to the time in Mosque's clock.
  So Mosque's clock is considered as a class variable which is same for everyone

```
class Myclass:
    class_variable = 0 # Class Variable
    def __init__(self,instance_variable):
        self.instance_variable = instance_variable   # Instance Variable
    def increment_class_variable(self):
        Myclass.class_variable += 1 # Accessing and Modifying the class variable
within method
# Create Instances of Myclass
Obj1 = Myclass(10)
Obj2 = Myclass(20)
# Accessing Class Variable
print(Obj1.class_variable)   # Output 0
print(Obj2.class_variable)   # Output 0
# Modifying the class variable through one instance
Obj1.increment_class_variable()
# Now, the class variable is updated for all instances
print(Obj1.class_variable) # Output 1
print(Obj2.class_variable) # Output 1
```

In the above example, *class_variable* is a class variable shared amongst all instances of **Myclass** class. You can access and modify class variables using class name or an instance of class. When you modify the class variable, the change is reflected across all instances of class.

Class Variables can be accessed and used outside the class. This can be done by accessing them using the class name itself.

```
class Myclass:
    class_variable = 0
# Accessing class variable outside the class
print(Myclass.class_variable) # Output 0
# Modifying class variable
Myclass.class_variable = 42
# Accessing the modified class variable
print(Myclass.class_variable) # output 42
```

Purpose of Class Variables

- They allow us to store data that can be shared amongst all instances (objects) of a class
- This can be useful when we want to maintain common information or configuration settings for all classes
- They are memory efficient than instance variables because they are shared amongst classes
- Class Variables are often used to store configuration settings for a class.

Inheritance

The English meaning of Inheritance is Inheritance refers to the *process of transmission of genes from parent to offspring*.

In **Python** Inheritance allows us to define a class that inherits all the methods and properties from another class.

- Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows you to create a new class based on an existing class
- The new class, called the "subclass" or "derived class," inherits attributes and methods from the existing class, known as the "**Super Class**" or "**parent/base class**."
- In Python, inheritance is used to promote code reuse and to establish a hierarchy of classes.

Super Class (Base Class or Parent Class):

- This is the existing class from which you want to inherit attributes and methods (class being inherited from).
- The base class defines the common characteristics and behaviors that can be shared among multiple classes.
- Any class can be a parent class, so the syntax is the same as creating any other class: Create a class named Person, with FirstName and LastName

```
class Person:
  def __init__(self, fname, lname):
    self.firstname = fname
    self.lastname = lname

  def printname(self):
    print(self.firstname, self.lastname)

#Use the Person class to create an object, and
then execute the printname method:

x = Person("John", "Doe")
x.printname()
```

Subclass (Derived Class or Child Class)

Subclass is the class that inherits from another class, also called derived class or Child Class.

- This is the new class that inherits attributes and methods from the base class.
- The subclass can also have its own attributes and methods in addition to what it inherits.

```
    class Student(Person):
        def __init__(self, fname, lname, year):
            super().__init__(fname, lname)
            self.graduationyear = year

        def welcome(self):
            print("Welcome", self.firstname, self.lastname, "to
the class of", self.graduationyear)
```

In this example, `Student` is a subclass of `Person`. It inherits attributes from `Person` and adds its own, like `graduationyear`.

To create a class that inherits the functionality from another class, send the parent class as a parameter when creating the child class

From Example 1 above, Create a class named Student, which will inherit the properties and methods from the Person class:

```
        x = Student("Mike", "Olsen")
        x.printname()
```

Adding New Attributes and Methods

- Subclasses can also have their own attributes and methods in addition to what they inherit from the base class.

```
        class Bird(Animal):
            def __init__(self, name, species):
                super().__init__(name)  # Call the constructor
of the base class
                self.species = species

            def speak(self):
                return f"{self.name} chirps!"

            def fly(self):
                return f"{self.name} is flying!"
```

In this example, the `Bird` class inherits from `Animal` but also has its own `species` attribute and `fly` method.

Note: When you add the **`__init__()`** function, the child class will no longer inherit the parent's **`__init__()`** function.

Key Tips and Tricks

1. Inheritance helps us to reuse code and build new things based on existing code. It allows to make new class that **inherits** attributes and methods from parent class.
2. In this way, we do not have to start the code from scratch and all over again.
3. We need to create a subclass or derived/child class when we want to use most of the attributes and methods from the parent class but also need to add some specific things.
4. No need to create a subclass if we want to make a completely new type of class with no connection to an existing one
5. Subclass is formed by passing name of the parent class as parameter (or inside parenthesis of subclass) inside the name of subclass.

```
class Parent:
    def __init__(self):
        self.value = value
classStudent(Person)
```

6. In the child class, we can add our own new attributes and methods. These additions WILL NOT change the parent class
7. Child class can have the same attributes as a parent class. This is useful when child class have to use some attributes and methods from parent class, and needs modifications.

Illustration:

Imagine we are modelling different vehicles, and we want to create classes for both cars and motorcycles. These vehicles share some common attributes but each have its own features uniquely.

```
# Parent Class (Superclass)
class Vehicle:
    def __init__(self,make,model,year):
        self.make = make
        self.model = model
        self.year = year
    def start_engine(self):
        print(f"The {self.make} {self.model}'s engine is running")

# Child Class 1
class Car(Vehicle):
    def __init__(self,make,model,year,num_doors):
        # calling parent class's constructor
        super().__init__(make,model,year)
        self.num_doors = num_doors
    def honk(self):
        print(f"The {self.make} {self.model} beeps its horn")
# Child Class 2
class Motorcycle(Vehicle):
    def __init__(self,make,model,year,has_kickstart):
    # Calling parent class's constructor
     super().__init__(make,model,year)
     self.has_kickstart = has_kickstart

    def wheelie(self):
        print(f"The {self.make} {self.model} does a Wheelie!")
# Creating Instances of child classes
my_car = Car("Honda","Civic",2022,4)
my_motorcycle = Motorcycle("BMW","X7",2023,True)

# Using inherited methods and attributes
my_car.start_engine()
my_car.honk()
my_motorcycle.start_engine()
my_motorcycle.wheelie()
```

In the above example:

- We have a vehicle class with common attributes like make, model and year. It also has method start_engine
- The Car class and Motorcycle class inherit from Vehicle class. They add specific attributes num_doors and has_kickstart. And methods honk and wheelie. While reusing the common attributes and start_engine method from Parent class.
- We create instances of both the Car and Motorcycle classes and use their methods to demonstrate how inheritance allows us to share code between classes while customizing them for specific purpose

This practical example above shows us how we can get help from inheritance by reusing the code and building specialized classes.

Method Overriding

- Subclasses can override methods inherited from the base class, providing their own implementations.
- This allows for **polymorphism**, where different objects can respond differently to the same method call. Here's an example with the speak method:

```
class Dog(Student):
    def speak(self):
        return f"{self.firstname} says Woof!"

class Cat(Student):
    def speak(self):
        return f"{self.firstname} says Meow!"
```

- Inheritance in Python helps in creating a more organized and efficient codebase by reusing and extending existing code. It also facilitates the modeling of real-world relationships between objects in a program.

However, To keep the inheritance of the parent's `__init__()` function, add a call to the parent's `__init__()` function:

```
class Student(Person):
  def __init__(self, fname, lname):
    Person.__init__(self, fname, lname)
```

Now we have successfully added the `__init__()` function, and kept the inheritance of the parent class, and we are ready to add functionality in the `__init__()` function.

Using the super() Function

- Python also has a super() function that will make the child class inherit all the methods and properties from its parent.
- It simplifies the process of calling the superclass's constructor, especially when there are multiple levels of inheritance.

```
class Student(Person):
    def __init__(self, fname, lname, year):
        super().__init__(fname, lname)
        self.graduationyear = year
```

By using the super() function, you do not have to use the name of the parent element, it will automatically inherit the methods and properties from its parent.

Add Properties

Add a property called graduationyear to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname):
    super().__init__(fname, lname)
    self.graduationyear = 2019
```

In the example below, the year 2019 should be a variable, and passed into the Student class when creating student objects. To do so, add another parameter in the __init__() function: Now, Add a year parameter, and pass the correct year when creating objects:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

x = Student("Mike", "Olsen", 2019)
```

Add Methods

Add a method called welcome to the Student class:

```
class Student(Person):
  def __init__(self, fname, lname, year):
    super().__init__(fname, lname)
    self.graduationyear = year

  def welcome(self):
    print("Welcome", self.firstname,
self.lastname, "to the class of", self.graduationyear)
```

If you add a method in the child class with the same name as a function in the parent class, the inheritance of the parent method will be overridden.

Pillars of Python

Pillars of Python Programming

There are 4 pillars in Python Programming.

1. Encapsulation
2. 2. Abstraction
3. 3. Inheritance (Already covered earlier.)
4. 4. Polymorphism

Encapsulation

- Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on **that** data into a single unit called a *class*.
    - It allows you to control access to the data and restrict external entities from directly modifying it.
    - In Python, this is achieved through classes and access modifiers like public, private, and protected.
- Encapsulation is a word formed from capsule. Anything inside a capsule is hidden, private and cannot be publicly used and cannot be publicly changed
- Encapsulation in python means to hide or protect methods or attributes of class.
- To access and update them we have 2 additional functions called **getter (to access)** and **setter (to update)**.
- Python offers **getter** and **setter** convenience methods to control access to the private instance variables for classes. The getter and setter methods are important because without them, the *private instance variables would not be accessible outside of the class*.
- **Getter** method allows to access the value of a private instance variable from outside a class, and the **setter** method allows to set the value of a private instance variable from outside a class.
- For example, if we have an account in any social media site such as Instagram. We have our password or other information protected that cannot be publicly accessed and changed. We have a function to change password By "forgot password".
- Second Example might be bank balances that are protected and hidden.

To Unhide the protected function there is a trick: **ObjName._ClassName__AttributeName.** Since python is not pure object oriented, encapsulation is not perfectly facilitated as it is in Java etc.

Protected Method are those that can only be accessed from inside the class and its subclasses. It is done by placing single underscore (_) before the name.

Private methods are those that can only be accessed from inside the class. It is done by placing double underscores (__) before the name.

Public methods are those that can be accessed from anywhere within or outside the class.

Illustration 1

```
class BankAccount:
    def __init__(self, account_number, balance):
        self._account_number = account_number  # Protected attribute
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        if amount > 0:
            self.__balance += amount
            print(f"Deposited ${amount}. New balance: ${self.__balance}")
        else:
            print("Invalid deposit amount.")

    def withdraw(self, amount):
        if amount > 0 and amount <= self.__balance:
            self.__balance -= amount
            print(f"Withdrew ${amount}. New balance: ${self.__balance}")
        else:
            print("Invalid withdrawal amount or insufficient balance.")

    def get_balance(self):
        return self.__balance
# Creating an instance of the BankAccount class
account = BankAccount("12345", 1000)
# Accessing protected attributes (conventionally considered non-public)
print("Account Number:", account._account_number)  # Accessing protected attribute

# Accessing private attributes (name mangling applied)
# This is not recommended; it's just to show the behavior
print("Balance:", account._BankAccount__balance)  # Accessing private attribute
# Using methods to deposit and withdraw
account.deposit(500)
account.withdraw(200)
account.withdraw(800)
# Accessing the balance through a method (recommended)
print("Current Balance:", account.get_balance())
```

In this example, we have a `BankAccount` class that encapsulates the account's data and methods. Here's how encapsulation is demonstrated:

1. **Attributes**: The class has two attributes: `_account_number` (protected) and `__balance` (private). Conventionally, *attributes starting with a single underscore are considered protected*, and *those starting with double underscores are considered private.*

    ii. **Methods**: The class defines methods for depositing, withdrawing, and retrieving the balance. These methods provide controlled access to the attributes, ensuring that the data is manipulated safely and following specific rules.

    iii. **Access Control**: While Python doesn't enforce strict access control, it uses name mangling to make it more difficult to directly access private attributes from outside the class. However, it's still possible to access them (as demonstrated in the example), but it's generally not recommended.

By encapsulating the data and providing controlled access through methods, encapsulation helps maintain the integrity of the data and provides a clear interface for interacting with the class, promoting good programming practices and making it easier to manage and maintain the code.

Illustration 2

```
class A():
    def __init__(self,name,age):
        self.__name = name
        self.__age = age
    def display(self): # Getter Function
        return "Name is: {} {} old.".format(self.__name,self.__age)

    def SetValue(self, name, age): # Setter Function
        self.__name = name
        self.__age = age

    def __hello(self): # Private Method
        return "Lol, Got Ya"

p1 = A("Talaal", 27)
print(p1)

p1.name # Will not be printed

# To print this we can use a hack as above:
 p1._A__name  # Talaal
```

Abstraction

- Abstraction is the process of simplifying complex systems by breaking them down into smaller, more manageable parts.
    - In Python, you can achieve abstraction through *classes* and *interfaces*
    - You create abstract classes to define common behavior and attributes, and then concrete classes inherit from these abstract classes to provide specific implementations.
    - It involves simplifying complex systems by breaking them down into smaller, more manageable parts.
- A class that is present in real world but not linked to any object is called Abstract class.
- For Example **Talaal** belongs to a **Living Things** Class but not directly associated to Living Thing Object. Due to ***the hierarchy of Living Thing***; In Living things there are plants, animals, insects, humans and etc but there is no direct object of Living Thing. Human class does not have a direct object, it will either be male class or female class. Living Things will be a parent class consisting many derived classes
- Another Example is Males and Females are humans and both share same features (methods) for example eating, walking, sleeping and speaking etc. and some methods and attributes are distinct. Similar methods and attributes will be in the Human class whereas the attributes and methods that are distinct from each other will have 2 derived classes; male class and female class. The General features will inherit from Human class. But in real world, there is no direct object of Human class. It will be either male class or female class.
- So any Parent Class to which we want that it does not have any Object but consists general features or methods and can be inherited is called Abstract Class.
- In object-oriented programming, an abstract class is a class that cannot be instantiated. However, you can create classes that inherit from an abstract class. Typically, you use an abstract class to create a blueprint for other classes.
- In Abstraction, object can be made from child class and not directly from parent class. We restrict the object from being inherit from Parent Class.
- Since python is not purely object oriented that is why we have to do most of the work through Packages. Most common package is ABC:

```
from abc import ABC, abstractmethod
```

Illustration 1:

```
from abc import ABC, abstractmethod
# Abstract class (Shape) representing a geometric shape
class Shape(ABC):
    def __init__(self, name):
        self.name = name

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

# Concrete class (Circle) inheriting from the abstract class Shape
class Circle(Shape):
    def __init__(self, name, radius):
        super().__init__(name)
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

    def perimeter(self):
        return 2 * 3.14 * self.radius

# Concrete class (Rectangle) inheriting from the abstract class Shape
class Rectangle(Shape):
    def __init__(self, name, length, width):
        super().__init__(name)
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * (self.length + self.width)

# Function that calculates and prints area and perimeter of a shape
def print_shape_info(shape):
    print(f"Shape: {shape.name}")
    print(f"Area: {shape.area()}")
    print(f"Perimeter: {shape.perimeter()}")

# Creating instances of Circle and Rectangle
circle = Circle("Circle", 5)
rectangle = Rectangle("Rectangle", 4, 6)

# Using the print_shape_info function to demonstrate abstraction
print_shape_info(circle)
print()
print_shape_info(rectangle)
```

In this example, we have used abstraction to create an abstract class Shape that defines the common interface for all geometric shapes. Here's how abstraction is demonstrated:

1. **Abstract Class**: `Shape` is an abstract class that defines two abstract methods, `area` and `perimeter`. An abstract class cannot be instantiated directly, but it provides a blueprint for concrete classes.
2. **Concrete Class**: `Circle` and `Rectangle` are concrete classes that inherit from the abstract class `Shape`. These classes provide specific implementations of the `area` and `perimeter` methods, which are required by the abstract class.
3. **Polymorphism**: The `print_shape_info` function demonstrates polymorphism. It can work with objects of any class that inherits from Shape, regardless of whether it's a `Circle` or a `Rectangle`. This showcases abstraction in action, as the function operates on the abstract concept of a "Shape" without knowing the specific shape type.

Abstraction allows you to create a clear separation between the interface (abstract class) and the implementation (concrete classes), making it easier to extend and maintain your code. It also promotes code reusability and helps manage complexity in larger software systems.

Inheritance has already been covered earlier. Please see earlier topics

Polymorphism

- The word Polymorphism refers to "many forms"
- In python, polymorphism means methods/functions/operators with same name that can be executed on many objects or classes.
- Polymorphism is a fundamental concept in object-oriented programming (OOP), including Python.
- In Python, polymorphism is significant because it enables flexibility, reusability, and abstraction in your code.

Significance of Polymorphism

1. **Code Reusability**: Polymorphism allows you to write code that can work with objects of different classes in a uniform way. This means you can reuse the same code for different objects that share a common interface or behavior, rather than having to write separate code for each specific class.
2. **Abstraction**: Polymorphism promotes a higher level of abstraction in your code. You can focus on what objects do (their behavior or methods) rather than what they are (their specific class or type). This simplifies the design and makes your code more maintainable.
3. **Flexibility**: Polymorphism makes your code more flexible and adaptable to changes. If you add a new class that conforms to the same interface (i.e., has the same methods), your existing code can work with objects of this new class without modification.
4. **Code Organization**: Polymorphism encourages better code organization through the use of inheritance and interfaces. You can define a common superclass or interface that specifies the methods all related classes should have, and then each subclass can provide its own implementation of those methods.
5. **Dynamic Behavior**: In Python, polymorphism is often achieved through dynamic method dispatch. This means that the appropriate method to execute is determined at runtime based on the actual object's class. This dynamic behavior is powerful because it allows for runtime flexibility and extensibility.

Illustration

```
class Shape:
    def area(self):
        pass

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius * self.radius

class Rectangle(Shape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

def calculate_area(shape):
    return shape.area()

circle = Circle(5)
rectangle = Rectangle(4, 6)

print(calculate_area(circle))     # Output: 78.5
print(calculate_area(rectangle))  # Output: 24
```

In this example, we have a Shape superclass with a common method `area()`.
Both `Circle` and `Rectangle` classes inherit from Shape and provide their own implementations of the `area()` method. The `calculate_area()` function can work with objects of any class that inherits from Shape, demonstrating polymorphism. This design makes it easy to add more shapes in the future without modifying the existing code.
In summary, polymorphism in Python is a powerful and essential concept in object-oriented programming that allows for code reuse, abstraction, flexibility, and dynamic behavior, ultimately making your code more maintainable and extensible.

Achieving Polymorphism

Polymorphism in Python is often achieved through **method overriding** and **duck typing**. Here's an overview of how it works:

1. **Method Overriding**: In Python, you can define a method in a subclass with the same name as a method in its superclass. This is called method overriding. Method overriding is a feature in object-oriented programming that allows a subclass to provide a different implementation of a method that is already defined in its superclass.

```
class Parent:
  def my_method(self):
    print("Parent method called")

class Child(Parent):
  def my_method(self):
    print("Child method called")

obj = Child()
obj.my_method() ### prints Child method called
```

2. **Duck Typing**: Python follows a dynamic typing system, and it's not necessary for objects to explicitly inherit from a common superclass to exhibit polymorphic behavior. Instead, Python relies on "duck typing," which means that an object's suitability is determined by its behavior (i.e., if it walks like a duck and quacks like a duck, it's treated as a duck). **See below for more details.**

```
class Bird:
    def speak(self):
        pass

class Duck(Bird):
    def speak(self):
        return "Quack!"

class Crow(Bird):
    def speak(self):
        return "Caw!"

def bird_sound(bird):
    return bird.speak()

duck = Duck()
crow = Crow()

print(bird_sound(duck))  # Output: "Quack!"
print(bird_sound(crow))  # Output: "Caw!"
```

There is another method used called **Method Overloading.** Method overloading is a feature in Python that allows a class to have multiple methods with the same name but with different parameters. This feature helps to provide flexibility and reusability to the code design. It is different from method overriding that allows a subclass to provide its implementation of a method defined in its superclass. Unlike other programming languages, Python DOES NOT support Method Overloading as python focuses on default arguments and variable length argument lists.

Function Polymorphism

- In programming polymorphism refers to methods/functions/operators with the same name that can be executed on many objects or classes.
- An example of a Python function that can be used on different objects is the `len()` function
- For *strings* `len()` returns the number of characters
- For *tuples* `len()` returns the number of items in the tuple
- For *dictionaries* `len()` returns the number of key/value pairs in the dictionary

Class Polymorphism

Polymorphism is often used in Class methods, where we can have multiple classes with the same method name.For example, say we have three classes: `Car`, `Boat`, and `Plane`, and they all have a method called `move()`:

Illustration

Different classes with the same method:

```
class Car:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Drive!")

class Boat:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Sail!")

class Plane:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang")       #Create a Car class
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat class
plane1 = Plane("Boeing", "747")     #Create a Plane class

for x in (car1, boat1, plane1):
  x.move()
```

Look at the *for* loop at the end. Because of polymorphism we can execute the same method for all three classes.

Inheritance Class Polymorphism

What about classes with child classes with the same name? Can we use polymorphism there?

Yes. If we use the example above and make a parent class called `Vehicle`, and make `Car`, `Boat`, `Plane` child classes of `Vehicle`, the child classes inherits the `Vehicle` methods, but can override them:

Illustration

Create a class called `Vehicle` and make `Car, Boat, Plane` child classes of `Vehicle`:

```
class Vehicle:
  def __init__(self, brand, model):
    self.brand = brand
    self.model = model

  def move(self):
    print("Move!")

class Car(Vehicle):
  pass

class Boat(Vehicle):
  def move(self):
    print("Sail!")

class Plane(Vehicle):
  def move(self):
    print("Fly!")

car1 = Car("Ford", "Mustang") #Create a Car object
boat1 = Boat("Ibiza", "Touring 20") #Create a Boat object
plane1 = Plane("Boeing", "747") #Create a Plane object

for x in (car1, boat1, plane1):
  print(x.brand)
  print(x.model)
  x.move()
```

- Child classes inherits the properties and methods from the parent class.
- In the example above you can see that the `Car` class is empty, but it inherits `brand`, `model`, and `move()` from `Vehicle`.
- The `Boat` and `Plane` classes also inherit `brand`, `model`, and `move()` from `Vehicle`, but they both override the `move()` method.
- Because of polymorphism we can execute the same method for all classes.

Decorators in Python:

- Decorators are the functions which helps to change and return other functions. It is simply a function that takes another function, updates it and then returns it.
- The *syntax* to add a decorator is to insert **"@"** symbol as a prefix before any class, method or function.
- Python decorators are powerful and versatile tools that allows us to modify the behavior of functions and methods. They are the way to extend functionality of a function or method without changing the source code.
- Decorator is function that takes another function as an argument and returns new function that modifies the behavior of the original function. The new function is referred to as decorated function
- Decorators are not limited to Abstraction only but can be used in various aspects of Python such as functions, classes or methods.
- They can be used for various purposes:
    1. Adding Authentication Checks to Functions
    2. Logging Function calls and their arguments
    3. Measuring execution time of functions
    4. Validating Function inputs and outputs
    5. Extending behavior of methods in classes

```
import time
def timing_decorator(func):
    def wrapper(*args, **kwargs):
        start_time = time.time()
        result = func(*args, **kwargs)
        end_time = time.time()
        print(f'{func.__name__} took {end_time - start_time:.2f} seconds to run')
        return result
    return wrapper

@timing_decorator
def slow_function():
    time.sleep(2)

slow_function()
```

In the above example, **timing_decorator** is applied to **slow_function**. It measures the time take by **slow_function** to execute and prints duration.

Python Type Checking

Python Type checking is performed on a real-time basis using dynamic Typing. For example, Python is dynamically typed.

This implies that a variable's type might vary over time. Likewise, Perl, Ruby, PHP, and JavaScript are dynamically typed.

Let's examine how a variable may change type in Python:

```
b ="hurry"
print(type(b))
b = 4
print(type(b))
<class 'str'>
<class 'int'>
```

The code above assigns the variable b to the string hurry and an integer value of 4. This ensures that Python correctly infers the variable's type.

Static Typing checks types during compilation. Some languages such as C, C++, and Java specify a variable's type, whereas others like Scala and Haskell enable type inference. Static typing avoids variable retyping.

Duck typing is a dynamic language system type. The method an object specifies is more important than its type

Static and dynamic typing in Python

A computer program's objects and elements are stored in memory and have variable names. When an object of a particular class is created, it will be allocated memory and invoked by its variable name.

You can consider this memory space as a box or container. This means that we have two components; an item and its packaging.

A box must be built to hold the thing it contains (i.e., a box made for keeping pens will not be optimal for carrying textiles). An object and box must be the same.

This is called static typing because both the object and the variable name (the box) must have the same type.

This explains why in statically typed languages like Java/C++, you must declare a variable's type. Even if the variable is empty, you may designate it as a box. Python does not support this functionality.

Python is a dynamically typed language. The variable name is more like a store price tag than a box. So the label is blank. If you ask its type, it will probably choose the tagged object.

A clothing tag can be reused on another garment. As a result, Python does not explicitly type variables.

Concept of Duck typing

We construct three classes: duck, goose, and hippo. They all swim. The duck and goose fly, but the hippo walks.

Next, we define the swim_fly() function, which accepts an animal and outputs its characteristics.

A creature missing the swim_fly() function will fail the Duck test due to a variable type issue.

Let's look at an example of Duck typing:

```
class Duck:
   def swim_fly(self):
         print("I am a duck, and I can swim and fly.")

class Goose:
     def swim_fly(self):
         print("I am a Goose, and I can swim and fly.")

class Hippo:
     def walk(self):
         print("I am a Hippo, and I can swim but can't fly.")

for obj in Duck(), Goose(), Hippo():
    obj.swim_fly()
I am a duck, and I can swim and fly.
I am a Goose, and I can swim and fly.
Traceback (most recent call last):
  File "main.py", line 21, in <module>
    obj.swim_fly()
AttributeError: 'Hippo' object has no attribute 'swim_fly'
```

In the three classes above, the duck and goose can swim and fly. However, the hippo class lacks the swim fly method. Therefore, its instance fails the test.

This gives us a basic notation for Duck typing. Custom types are more about implementing features than data types. Even though a goose isn't an actual duck, the swim_fly function turns it into one.

How to apply iteration in Duck typing

Iteration allows us to modify lists in Python. The for loops are a systematic approach to establishing an iteration.

Due to Duck typing, these objects are handled the same regardless of their application. Iteration requires **iter()** and **next()** methods.

In Python, we can design our iterator for outputting square numbers using methods like **iter()** and **next().**

```
class Number_squared:
  def __init__(identity, l=0, x=-1):
    identity.x = x
    identity.y = l
  def __iter__(identity):
    return identity
  def   next  (identity):
    if identity.y < identity.x:
      s = identity.y ** 2
      identity.y += 1
      return s
    else:
      raise StopIteration
for k in Number_squared(2, 6):
  print(k)
4
9
16
25
```

In the above code, we utilized the iter() and next() methods to create iterators of a custom class.The square numbers to be computed are between 2 and 6. We use a for loop to go through these numbers.

Implementation of custom len() function

Custom **len()** method uses the **sort()** method to sort a list. For example, consider sorting a list by name length. Here's how Duck typing works

```
def length(iterator):
   count = 0
   for item in iterator:

      count += 1
   return count
if   name   ==  "  main  ":
   iterator = input("Enter a string:- ")
   print(f"Length of {iterator} is {length(iterator)}")
```

The code above prompts the user to enter a string. It then calculates the length of the string and returns the result.

def length(iterator) is used to declare the function to retrieve the length of the iterator and sets the count to zero using count =0.

The for loop increments the count by adding one after every count and returns the value to the function.

iterator = input("Enter a string:- ") prompts the user to enter a string and returns the length of the string using print(f"Length of {iterator} is {length(iterator)}").

Conclusion

Duck typing is a perfect example of dynamic typing in Python. It prioritizes related functionality over specific data types.

Appendix - A: Special Methods/Magic Methods/Dunder Methods

Besides __ **init** __, __ **str** __ and __ **main** __ there are many more Special/Dunder/Magic methods: These methods enable you to define custom behavior for various operations on objects of your classes. Here are some commonly used special methods:

1. **`__init__(self, ...):`** The constructor method, called when an object is created from a class.
2. **`__repr__(self):`** Returns an "official" string representation of the object, typically used for debugging.
3. **`__eq__(self, other):`** Defines the behavior of the equality operator ==.
4. **`__lt__(self, other):`** Defines the behavior of the less-than operator <.
5. **`__le__(self, other):`** Defines the behavior of the less-than-or-equal-to operator <=.
6. **`__gt__(self, other):`** Defines the behavior of the greater-than operator >.
7. **`__ge__(self, other):`** Defines the behavior of the greater-than-or-equal-to operator >=.
8. **`__ne__(self, other):`** Defines the behavior of the not-equal-to operator !=.
9. **`__len__(self):`** Returns the length of the object when len(object) is called.
10. **`__getitem__(self, key):`** Enables indexing of objects, like `object[key]`.
11. **`__setitem__(self, key, value):`** Enables setting values using indexing, like `object[key] = value`.
12. **`__delitem__(self, key):`** Enables deleting items using indexing, like del `object[key]`.
13. **`__iter__(self):`** Returns an iterator object for the class, allowing iteration over the object.
14. **`__next__(self):`** Defines the behavior when using the next() function with an iterator.
15. **`__contains__(self, item):`** Defines the behavior of the in operator, like item in object.
16. **`__call__(self, ...):`** Allows you to call an object as if it were a function.
17. **`__enter__(self):`** Used for context management (with statement).
18. **`__exit__(self, exc_type, exc_value, traceback):`** Used for context management (with statement).
19. **`__add__(self, other):`** Defines the behavior of the addition operator +.
20. **`__sub__(self, other):`** Defines the behavior of the subtraction operator -.
21. **`__mul__(self, other):`** Defines the behavior of the multiplication operator *.
22. **`__truediv__(self, other):`** Defines the behavior of the division operator /.
23. **`__floordiv__(self, other):`** Defines the behavior of the floor division operator //.
24. **`__mod__(self, other):`** Defines the behavior of the modulo operator %.
25. **`__pow__(self, other, modulo=None):`** Defines the behavior of the exponentiation operator **.

Appendix B

LIST OF REGULAR EXPRESSIONS CHARACTERS

METACHARACTERS

Metacharacters are characters with a special meaning:

S.NO	Metacharachters	Description
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		

SETS

A set is a set of characters inside a pair of square brackets [] with a special meaning. Any Metacharacter inside the set will lose its speciality:

S.No	Sets	Description
1		
2		
3		
4		
5		
6		
7		
8		

RegEx Functions

Function	Description
findall	Returns a list containing all matches
search	Returns a Match object if there is a match anywhere in the string
split	Returns a list where the string has been split at each match
sub	Replaces one or many matches with a string

Special Sequences

A special sequence is a \ followed by one of the characters in the list above, and have a special meaning.

S.No	Special Sequence	Description
1		
2		
3		
4		
5		
6		
7		
8		
9		
10		