PYTHON (codewith harry)

BASICS

Python is a popular programming language created by Guido Van Rossum in 1989. It is simple, easy to understand, and supports both object-oriented and functional programming. As an interpreted and platform-independent language, Python makes debugging convenient. It has a vast library support, including NumPy, Tensorflow, and OpenCV. Python is utilized in various fields such as data visualization, analytics, AI, machine learning, web development, and database management. It is also employed in business and accounting for complex mathematical operations and analysis.

Modules and pip in Python

Module is like a code library which can be used to borrow code written by somebody else in our python program. There are two types of modules in python:

Built in Modules - These modules are ready to import and use and ships with the python interpreter. there is no need to install such modules explicitly.

External Modules - These modules are imported from a third party file or can be installed using a package manager like pip or conda. Since this code is written by someone else, we can install different versions of a same module with time.

The pip command

It can be used as a package manager pip to install a python module. Lets install a module called pandas using the following command

Comments, Escape sequence & Print in Python

Python Comments

A comment is a part of the coding file that the programmer does not want to execute, rather the programmer uses it to either explain a block of code or to avoid the execution of a specific part of code while testing.

Single-Line Comments:

To write a comment just add a '#' at the start of the line.

Escape Sequence Characters

To insert characters that cannot be directly used in a string, we use an escape sequence character.

An escape sequence character is a backslash \ followed by the character you want to insert.

An example of a character that cannot be directly used in a string is a double quote inside a string that is surrounded by double quotes:

Variables and Data Types

What is a variable?

Variable is like a container that holds data. Very similar to how our containers in kitchen holds sugar, salt etc Creating a variable is like creating a placeholder in memory and assigning it some value. In Python its as easy as writing:

```
a = 1
b = True
c = "Harry"
d = None
```

What is a Data Type?

Data type specifies the type of value a variable holds. This is required in programming to do various operations without causing an error.
In python, we can print the type of any operator using type function:

```
a = 1
print(type(a))
b = "1"
print(type(b))
```

1. Numeric data: int, float, complex

- int: 3, -8, 0
- float: 7.349, -9.0, 0.0000001
- complex: 6 + 2j

2. Text data: str

str: "Hello World!!!", "Python Programming"

3. Boolean data:

Boolean data consists of values True or False

4. Sequenced data: list, tuple

list: A list is an ordered collection of data with elements separated by a comma and enclosed within square brackets. Lists are mutable and can be modified after creation.

Example:

```
list1 = [8, 2.3, [-4, 5], ["apple", "banana"]]
print(list1)
```

Output:

```
[8, 2.3, [-4, 5], ['apple', 'banana']]
```

Tuple: A tuple is an ordered collection of data with elements separated by a comma and enclosed within parentheses. Tuples are immutable and can not be modified after creation.

Example:

```
tuple1 = (("parrot", "sparrow"), ("Lion", "Tiger"))
print(tuple1)
```

Output:

```
(('parrot', 'sparrow'), ('Lion', 'Tiger'))
```

5. Mapped data: dict

dict: A dictionary is an unordered collection of data containing a key:value pair. The key:value pairs are enclosed within curly brackets.

Example:

```
dict1 = {"name":"Sakshi", "age":20, "canVote":True}
print(dict1)
```

Output:

```
{'name': 'Sakshi', 'age': 20, 'canVote': True}
```

Typecasting in python

The conversion of one data type into the other data type is known as type casting in python or type conversion in python.

Python supports a wide variety of functions or methods like: int(), float(), str(), ord(), hex(), oct(), tuple(), set(), list(), dict(), etc. for the type casting in python.

Two Types of Typecasting:

1. Explicit Conversion (Explicit type casting in python)

2. Implicit Conversion (Implicit type casting in python).

IMMUTABLE , MUTABLE DIFFERENCE

In Python, "mutable" and "immutable" refer to whether an object's state can be changed after it is created.

- **Mutable Objects:** These are objects whose state (content) can be modified after creation. Lists, dictionaries, and sets are examples of mutab objects in Python. For instance, you can add or remove elements from a list or dictionary after it's been created.
- **Immutable Objects:** These are objects whose state cannot be modified after creation. Tuples, strings, and integers are examples of immutab objects in Python. Once created, you cannot change the value of a specific element within a tuple or a character within a string.

Mutable Example (List):

```
my_list = [1, 2, 3]
my_list[0] = 4  # Modifying the first element
print(my_list)  # Output: [4, 2, 3]
```

Immutable Example (Tuple):

```
my_tuple = (1, 2, 3)
my_tuple[0] = 4  # This will raise an error because tuples are immutable
```

Explicit typecasting:

The conversion of one data type into another data type, done via developer or programmer's intervention or manually as per the requirement, is known as explicit type conversion.
It can be achieved with the help of Python’s built-in type conversion functions such as int(), float(), hex(), oct(), str(), etc .

Example of explicit typecasting:

```
string = "15"
number = 7
string_number = int(string) #throws an error if the string is not a valid
integer
sum= number + string_number
print("The Sum of both the numbers is: ", sum)
```

Output:

```
The Sum of both the numbers is 22
```

Implicit type casting:

Data types in Python do not have the same level i.e. ordering of data types is not the same in Python. Some of the data types have higher-order, and some have lower order. While performing any operations on variables with different data types in Python, one of the variable's data types will be changed to the higher data type. According to the level, one data type is converted into other by the Python interpreter itself (automatically). This is called, implicit typecasting in python. Python converts a smaller data type to a higher data type to prevent data loss.

Example of implicit type casting:

```
# Python automatically converts
# a to int
a = 7
print(type(a))
# Python automatically converts b to float
b = 3.0
print(type(b))
# Python automatically converts c to float as it is a float addition
c = a + b
print(c)
print(type(c))
```

Ouput:

```
<class 'int'>
<class 'float'>
10.0
<class 'float'>
```

What are strings?

In python, anything that you enclose between single or double quotation marks is considered a string. A string is essentially a sequence or array of textual data. Strings are used when working with Unicode characters.

Example

```
name = "Harry"
print("Hello, " + name)
```

Output

Hello, Harry

⌗Multiline Strings⌗

If our string has multiple lines, we can create them like this:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

Note: It does not matter whether you enclose your strings in single or double quotes, the output remains the same.

Sometimes, the user might need to put quotation marks in between the strings. Example, consider the sentence: He said, “I want to eat an apple”.

How will you print this statement in python?: `He said, "I want to eat an apple".` We will definitely use single quotes for our convenience

```
print('He said, "I want to eat an apple".')
```

Accessing Characters of a String

In Python, string is like an array of characters. We can access parts of string by using its index which starts from 0.
Square brackets can be used to access elements of the string.

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

Looping through the string

We can loop through strings using a for loop like this:

```
for character in name:
    print(character)
```

Above code prints all the characters in the string name one by one!

String Slicing & Operations on String

Length of a String

We can find the length of a string using len() function.

Example:

```
fruit = "Mango"
len1 = len(fruit)
print("Mango is a", len1, "letter word.")
```

Output:

Mango is a 5 letter word.

Loop through a String:

Strings are arrays and arrays are iterable. Thus we can loop through strings.

Example:

```
alphabets = "ABCDE"
for i in alphabets:
    print(i)
```

Output:

```
A
B
C
D
E
```

String as an array

A string is essentially a sequence of characters also called an array. Thus we can access the elements of this array.

Example:

```
pie = "ApplePie"
print(pie[:5])
print(pie[6])  #returns character at specified index
```

Output:

```
Apple
i
```

Note: This method of specifying the start and end index to specify a part of a string is called slicing.

String methods

Python provides a set of built-in methods that we can use to alter and modify the strings

upper() :

The upper() method converts a string to upper case.

lower():

The lower() method converts a string to lower case.

split() :

The split() method splits the given string at the specified instance and returns the separated strings as list items.

center() :

The center() method aligns the string to the center as per the parameters given by the user.

strip() :

The strip() method removes any white spaces before and after the string.

rstrip() :

the rstrip() removes any trailing characters.

count() :

The count() method returns the number of times the given value has occurred within the given string.

replace() :

The replace() method replaces all occurences of a string with another string

capitalize() :

The capitalize() method turns only the first character of the string to uppercase and the rest other characters of the string are turned to lowercase. The string has no effect if the first character is already uppercase.

endswith() :

The endswith() method checks if the string ends with a given value. If yes then return True, else return False.

find() :

The find() method searches for the first occurrence of the given value and returns the index where it is present. If given value is absent from the string then return -1.

index() :

The index() method searches for the first occurrence of the given value and returns the index where it is present. If given value is absent from the string then raise an exception.

isalnum() :

The isalnum() method returns True only if the entire string only consists of A-Z, a-z, 0-9. If any other characters or punctuations are present, then it returns False.

isalpha() :

The isalnum() method returns True only if the entire string only consists of A-Z, a-z. If any other characters or punctuations or numbers(0-9) are present, then it returns False.

islower() :

The islower() method returns True if all the characters in the string are lower case, else it returns False.

isprintable() :

The isprintable() method returns True if all the values within the given string are printable, if not, then return False.

isspace() :

The isspace() method returns True only and only if the string contains white spaces, else returns False.

title() :

The title() method capitalizes each letter of the word within the string.

istitle() :

The istitile() returns True only if the first letter of each word of the string is capitalized, else it returns False.

isupper() :

The isupper() method returns True if all the characters in the string are upper case, else it returns False.

startswith() :

The endswith() method checks if the string starts with a given value. If yes then return True, else return False.

swapcase() :

The swapcase() method changes the character casing of the string. Upper case are converted to lower case and lower case to upper case.

if-else Statements

Sometimes the programmer needs to check the evaluation of certain expression(s), whether the expression(s) evaluate to True or False. If the expression evaluates to False, then the program execution follows a different path than it would have if the expression had evaluated to True.

Based on this, the conditional statements are further classified into following types:

- if
- if-else
- if-else-elif
- nested if-else-elif.

Example:

```
applePrice = 210
budget = 200
if (applePrice <= budget):
    print("Alexa, add 1 kg Apples to the cart.")
else:
    print("Alexa, do not add Apples to the cart.")
```

Output:

```
Alexa, do not add Apples to the cart.
```

An if.....else statement evaluates like this:

if the expression evaluates True:

Execute the block of code inside if statement. After execution return to the code out of the if.....else block.\

if the expression evaluates False:

Execute the block of code inside else statement. After execution return to the code out of the if.....else block.

Nested if statements

We can use if, if-else, elif statements inside other if statements as well.
Example:

```
num = 18
if (num < 0):
    print("Number is negative.")
elif (num > 0):
    if (num <= 10):
        print("Number is between 1-10")
    elif (num > 10 and num <= 20):
        print("Number is between 11-20")
    else:
        print("Number is greater than 20")
else:
    print("Number is zero")
```

Output:

```
Number is between 11-20
```

elif Statements

Sometimes, the programmer may want to evaluate more than one condition, this can be done using an elif statement.

Working of an elif statement

Execute the block of code inside if statement if the initial expression evaluates to True. After execution return to the code out of the if block.

Execute the block of code inside the first elif statement if the expression inside it evaluates True. After execution return to the code out of the if block.

Execute the block of code inside the second elif statement if the expression inside it evaluates True. After execution return to the code out of the if block.

.

.

.

Execute the block of code inside the nth elif statement if the expression inside it evaluates True. After execution return to the code out of the if block.

Execute the block of code inside else statement if none of the expression evaluates to True. After execution return to the code out of the if block.

Example:

```
num = 0
if (num < 0):
   print("Number is negative.")
elif (num == 0):
   print("Number is Zero.")
else:
   print("Number is positive.")
```

Output:

```
Number is Zero.
```

Match Case Statements

Match statement will compare a given variable's value to different shapes, also referred to as the pattern. The main idea is to keep on comparing the variable with all the present patterns until it fits into one.

The match case consists of three main entities :

1. **The match keyword**
2. **One or more case clauses**
3. **Expression for each case**
4.

The case clause consists of a pattern to be matched to the variable, a condition to be evaluated if the pattern matches, and a set of statements to be executed if the pattern matches.

```
# example:-
x = int(input("enter your  age:-"))

match x:

    case _ if x > 18 :
        print("you are eligible for voting")

    case _ if x <= 18:
        print("you are not eligible for voting")
```

output:-

```
enter your  age:-5
you are not eligible for voting
```

Introduction to Loops

Sometimes a programmer wants to execute a group of statements a certain number of times. This can be done using loops. Based on this loops are further classified into following main types;

- for loop
- while loop

The for Loop

for loops can iterate over a sequence of iterable objects in python. Iterating over a sequence is nothing but iterating over strings, lists, tuples, sets and dictionaries.

Example: iterating over a string:

```
name = 'Abhishek'
for i in name:
    print(i, end=", ")
```

Output:

A, b, h, i, s, h, e, k,

Example: iterating over a list:

```
colors = ["Red", "Green", "Blue", "Yellow"]
for x in colors:
    print(x)
```

Output:

Red
Green
Blue
Yellow

Similarly, we can use loops for lists, sets and dictionaries.

range():

What if we do not want to iterate over a sequence? What if we want to use for loop for a specific number of times?

Here, we can use the range() function.

Example:

```
for k in range(5):
    print(k)
```

Output:

0
1
2
3
4

Here, we can see that the loop starts from 0 by default and increments at each iteration.

But we can also loop over a specific range.

Example:

```
for k in range(4,9):
    print(k)
```

Output:

4
5
6
7
8

Quick Quiz

Explore about third parameter of range (ie range(x, y, z))

Python while Loop

As the name suggests, while loops execute statements while the condition is True. As soon as the condition becomes False, the interpreter comes out of the while loop.

Example:

```
count = 5
while (count > 0):
 print(count)
 count = count - 1
```

Output:

```
5
4
3
2
1
```

Here, the count variable is set to 5 which decrements after each iteration. Depending upon the while loop condition, we need to either increment or decrement the counter variable (the variable count, in our case) or the loop will continue forever.

Else with While Loop

We can even use the else statement with the while loop. Essentially what the else statement does is that as soon as the while loop condition becomes False, the interpreter comes out of the while loop and the else statement is executed

Example:

```
x = 5
while (x > 0):
   print(x)
   x = x - 1
else:
   print('counter is 0')
```

Output:

```
5
4
3
2
1
counter is 0
```

Do-While loop in python

Do..while is a loop in which a set of instructions will execute at least once (irrespective of the condition) and then the repetition of loop's body will depend on the condition passed at the end of the while loop. It is also known as an exit-controlled loop.

How to emulate do while loop in python?

To create a do while loop in Python, you need to modify the while loop a bit in order to get similar behavior to a do while loop. The most common technique to emulate a do-while loop in Python is to use an infinite while loop with a break statement wrapped in an if statement that checks a given condition and breaks the iteration if that condition becomes true:

Explanation

This loop uses True as its formal condition. This trick turns the loop into an infinite loop. Before the conditional statement, the loop runs all the required processing and updates the breaking condition. If this condition evaluates to true, then the break statement breaks out of the loop, and the program execution continues its normal path.

Example

```
while True:
  number = int(input("Enter a positive number: "))
  print(number)
  if not number > 0:
    break
```

Output

```
Enter a positive number: 1
1
Enter a positive number: 4
4
Enter a positive number: -1
-1
```

break statement

The break statement enables a program to skip over a part of the code. A break statement terminates the very loop it lies within.

example

```
for i in range(1,101,1):
   print(i ,end=" ")
   if(i==50):
       break
   else:
       print("Mississippi")
print("Thank you")
```

output

```
1 Mississippi
2 Mississippi
3 Mississippi
4 Mississippi
5 Mississippi
.
.
50 Mississippi
```

Python Functions

A function is a block of code that performs a specific task whenever it is called. In bigger programs, where we have large amounts of code, it is advisable to create or use existing functions that make the program flow organized and neat.
There are two types of functions:

1. Built-in functions
2. User-defined functions

Built-in functions:

These functions are defined and pre-coded in python. Some examples of built-in functions are as follows:
min(), max(), len(), sum(), type(), range(), dict(), list(), tuple(), set(), print(), etc.

User-defined functions:

We can create functions to perform specific tasks as per our needs. Such functions are called user-defined functions.

Syntax:

```
def function_name(parameters):
  pass
  # Code and Statements
```

- Create a function using the def keyword, followed by a function name, followed by a paranthesis (()) and a colon(:).
- Any parameters and arguments should be placed within the parentheses.
- Rules to naming function are similar to that of naming variables.
- Any statements and other code within the function should be indented.

Calling a function:

We call a function by giving the function name, followed by parameters (if any) in the parenthesis.

Example:

```
def name(fname, lname):
    print("Hello,", fname, lname)

name("Sam", "Wilson")
```

Output:

```
Hello, Sam Wilson
```

Function Arguments and return statement

There are four types of arguments that we can provide in a function:

- Default Arguments
- Keyword Arguments
- Variable length Arguments
- Required Arguments

Default arguments:

We can provide a default value while creating a function. This way the function assumes a default value even if a value is not provided in the function call for that argument.

Keyword arguments:

We can provide arguments with key = value, this way the interpreter recognizes the arguments by the parameter name. Hence, the the order in which the arguments are passed does not matter.

Required arguments:

In case we don't pass the arguments with a key = value syntax, then it is necessary to pass the arguments in the correct positional order and the number of arguments passed should match with actual function definition.

Variable-length arguments:

Sometimes we may need to pass more arguments than those defined in the actual function. This can be done using variable-length arguments.
There are two ways to achieve this:

Arbitrary Arguments:

While creating a function, pass a * before the parameter name while defining the function. The function accesses the arguments by processing them in the form of tuple.

Keyword Arbitrary Arguments:

While creating a function, pass a * before the parameter name while defining the function. The function accesses the arguments by processing them in the form of dictionary.

return Statement

The return statement is used to return the value of the expression back to the calling function.

List Index

Each item/element in a list has its own unique index. This index can be used to access any particular item from the list. The first item has index [0], second item has index [1], third item has index [2] and so on.

Accessing list items

We can access list items by using its index with the square bracket syntax []. For example colors[0] will give "Red", colors[1] will give "Green" and so on...

Positive Indexing:

As we have seen that list items have index, as such we can access items using these indexes.

Negative Indexing:

Similar to positive indexing, negative indexing is also used to access items, but from the end of the list. The last item has index [-1], second last item has index [-2], third last item has index [-3] and so on

Range of Index:

You can print a range of list items by specifying where you want to start, where do you want to end and if you want to skip elements in between the range.

List Comprehension

List comprehensions are used for creating new lists from other iterables like lists, tuples, dictionaries, sets, and even in arrays and strings.

Syntax:

List = [Expression(item) for item in iterable if Condition]

Expression: It is the item which is being iterated.

Iterable: It can be list, tuples, dictionaries, sets, and even in arrays and strings.

Condition: Condition checks if the item should be added to the new list or not.

Example 1: Accepts items with the small letter “o” in the new list

```
names = ["Milo", "Sarah", "Bruno", "Anastasia", "Rosa"]
namesWith_O = [item for item in names if "o" in item]
print(namesWith_O)
```

Output:

```
['Milo', 'Bruno', 'Rosa']
```

Example 2: Accepts items which have more than 4 letters

```
names = ["Milo", "Sarah", "Bruno", "Anastasia", "Rosa"]
namesWith_O = [item for item in names if (len(item) > 4)]
print(namesWith_O)
```

Output:

```
['Sarah', 'Bruno', 'Anastasia']
```

List Methods

list.sort()

This method sorts the list in ascending order. The original list is updated

reverse()

This method reverses the order of the list.

insert():

This method inserts an item at the given index. User has to specify index and the item to be inserted within the insert() method.

index()

This method returns the index of the first occurrence of the list item.

count()

Returns the count of the number of items with the given value

extend():

This method adds an entire list or any other collection datatype (set, tuple, dictionary) to the existing list.

copy()

Returns copy of the list. This can be done to perform operations on the list without modifying the original list.

append():

This method appends items to the end of the existing list.

Concatenating two lists:

You can simply concatenate two lists to join two lists.

Python Tuples

Tuples are ordered collection of data items. They store multiple items in a single variable. Tuple items are separated by commas and enclosed within round brackets (). Tuples are unchangeable meaning we can not alter them after creation

Tuple Indexes

I. Positive Indexing:

As we have seen that tuple items have index, as such we can access items using these indexes.

II. Negative Indexing:

Similar to positive indexing, negative indexing is also used to access items, but from the end of the tuple. The last item has index [-1], second last item has index [-2], third last item has index [-3] and so on.

III. Check for item:

We can check if a given item is present in the tuple. This is done using the `in` keyword

IV. Range of Index:

You can print a range of tuple items by specifying where do you want to start, where do you want to end and if you want to skip elements in between the range.

String formatting in python

String formatting can be done in python using the format method.

```
txt = "For only {price:.2f} dollars!"
print(txt.format(price = 49))
```

f-strings in python

It is a new string formatting mechanism introduced by the PEP 498. It is also known as Literal String Interpolation or more commonly as F-strings (f character preceding the string literal). The primary focus of this mechanism is to make the interpolation easier.
When we prefix the string with the letter 'f', the string becomes the f-string itself. The f-string can be formatted in much same as the str.format() method. The f-string offers a convenient way to embed Python expression inside string literals for formatting.

Example

```
val = 'Geeks'
print(f"{val}for{val} is a portal for {val}.")
name = 'Tushar'
age = 23
print(f"Hello, My name is {name} and I'm {age} years old.")
```

Docstrings in python

Python docstrings are the string literals that appear right after the definition of a function, method, class, or module.

Here,
'''Takes in a number n, returns the square of n''' is a docstring which will not appear in output

Example

```
def square(n):
    '''Takes in a number n, returns the square of n'''
    print(n**2)
square(5)
```

Output :25

PEP 8

PEP 8 is a document that provides guidelines and best practices on how to write Python code. It was written in 2001 by Guido van Rossum, Barry Warsaw, and Nick Coghlan. The primary focus of PEP 8 is to improve the readability and consistency of Python code.
PEP stands for Python Enhancement Proposal, and there are several of them. A PEP is a document that describes new features proposed for Python and documents aspects of Python, like design and style, for the community.

The Zen of Python

Long time Pythoneer Tim Peters succinctly channels the BDFL's guiding principles for Python's design into 20 aphorisms, only 19 of which have been written down. IT IS ALSO KNOWN AN AS A EASTER EGG IN PYTHON

It can be used by ;- import this

Recursion in python

Recursion is the process of defining something in terms of itself.

Python Recursive Function

In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

Example:

```
def factorial(num):
    if (num == 1 or num == 0):
        return 1
    else:
        return (num * factorial(num - 1))

# Driver Code
num = 7;
print("Number: ",num)
print("Factorial: ",factorial(num))
```

Output:

```
number:  7
Factorial:  5040
```

Python Sets

Sets are unordered collection of data items. They store multiple items in a single variable. Set items are separated by commas and enclosed within curly brackets {}. Sets are unchangeable, meaning you cannot change items of the set once created. Sets do not contain duplicate items.

Here we see that the items of set occur in random order and hence they cannot be accessed using index numbers. Also sets do not allow duplicate values.

Example:

```
info = {"Carla", 19, False, 5.9, 19}
print(info)
```

Output:

```
{False, 19, 5.9, 'Carla'}
```

Accessing set items:

Using a For loop

You can access items of set using a for loop.

Example:

```
info = {"Carla", 19, False, 5.9}
for item in info:
   print(item)
```

Output:

```
False
Carla
19
5.9
```

Joining Sets

Sets in python more or less work in the same way as sets in mathematics. We can perform operations like union and intersection on the sets just like in mathematics.

I. union() and update():

The union() and update() methods prints all items that are present in the two sets. The union() method returns a new set whereas update() method adds item into the existing set from another set.

II. intersection and intersection_update():

The intersection() and intersection_update() methods prints only items that are similar to both the sets. The intersection() method returns a new set whereas intersection_update() method updates into the existing set from another set.

III. symmetric_difference and symmetric_difference_update():

The symmetric_difference() and symmetric_difference_update() methods prints only items that are not similar to both the sets. The symmetric_difference() method returns a new set whereas symmetric_difference_update() method updates into the existing set from another set.

IV. difference() and difference_update():

The difference() and difference_update() methods prints only items that are only present in the original set and not in both the sets. The difference() method returns a new set whereas difference_update() method updates into the existing set from another set.

Set Methods

clear():

This method clears all items in the set and prints an empty set.

isdisjoint():

The isdisjoint() method checks if items of given set are present in another set. This method returns False if items are present, else it returns True.

issuperset():

The issuperset() method checks if all the items of a particular set are present in the original set. It returns True if all the items are present, else it returns False.

issubset():

The issubset() method checks if all the items of the original set are present in the particular set. It returns True if all the items are present, else it returns False.

del

del is not a method, rather it is a keyword which deletes the set entirely.

pop()

This method removes the last item of the set but the catch is that we don't know which item gets popped as sets are unordered. However, you can access the popped item if you assign the pop() method to a variable.

remove()/discard()

We can use remove() and discard() methods to remove items form list.

add()

If you want to add a single item to the set use the add() method.

update()

If you want to add more than one item, simply create another set or any other iterable object(list, tuple, dictionary), and use the update() method to add it into the existing set.

Python Dictionaries

Dictionaries are ordered collection of data items. They store multiple items in a single variable. Dictionary items are key-value pairs that are separated by commas and enclosed within curly brackets {}.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info)
```

Output:

```
{'name': 'Karan', 'age': 19, 'eligible': True}
```

II. Accessing multiple values:

We can print all the values in the dictionary using values() method.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info.values())
```

Output:

```
dict_values(['Karan', 19, True])
```

IV. Accessing key-value pairs:

We can print all the key-value pairs in the dictionary using items() method.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info.items())
```

Output:

```
dict_items([('name', 'Karan'), ('age', 19), ('eligible', True)])
```

Accessing Dictionary items:

I. Accessing single values:

Values in a dictionary can be accessed using keys. We can access dictionary values by mentioning keys either in square brackets or by using get method.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info['name'])
print(info.get('eligible'))
```

Output:

```
Karan
True
```

III. Accessing keys:

We can print all the keys in the dictionary using keys() method.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True}
print(info.keys())
```

Output:

```
dict_keys(['name', 'age', 'eligible'])
```

Dictionary Methods

Dictionary uses several built-in methods for manipulation.They are listed below

update()

The update() method updates the value of the key provided to it if the item already exists in the dictionary, else it creates a new key-value pair.

Removing items from dictionary:

There are a few methods that we can use to remove items from dictionary.

clear():

The clear() method removes all the items from the list.

pop():

The pop() method removes the key-value pair whose key is passed as a parameter.

popitem():

The popitem() method removes the last key-value pair from the dictionary.

del:

we can also use the del keyword to remove a dictionary item.

Python - else in Loop

As you have learned before, the else clause is used along with the if statement.
Python allows the else keyword to be used with the for and while loops too. The else block appears after the body of the loop. The statements in the else block will be executed after all iterations are completed. The program exits the loop only after the else block is executed.

Example:

```
for x in range(5):
   print ("iteration no {} in for loop".format(x+1))
else:
   print ("else block in loop")
print ("Out of loop")
```

Output:

```
iteration no 1 in for loop
iteration no 2 in for loop
iteration no 3 in for loop
iteration no 4 in for loop
iteration no 5 in for loop
else block in loop
Out of loop
```

Exception Handling

Exception handling is the process of responding to unwanted or unexpected events when a computer program runs. Exception handling deals with these events to avoid the program or system crashing, and without this process, exceptions would disrupt the normal operation of a program.

Python try...except

try….. except blocks are used in python to handle errors and exceptions. The code in try block runs when there is no error. If the try block catches the error, then the except block is executed.

Example:

```
try:
    num = int(input("Enter an integer: "))
except ValueError:
    print("Number entered is not an integer.")
```

Raising Custom errors

In python, we can raise custom errors by using the `raise` keyword.

Example:-
```
salary = int(input("Enter salary amount: "))
if not 2000 < salary < 5000:
    raise ValueError("Not a valid salary")
```
In the previous tutorial, we learned about different bui

Defining Custom Exceptions

In Python, we can define custom exceptions by creating a new class that is derived from the built-in Exception class.

Finally Clause

The finally code block is also a part of exception handling. When we handle exception using the try and except block, we can include a finally block at the end. The finally block is always executed, so it is generally used for doing the concluding tasks like closing file resources or closing database connection or may be ending the program execution with a delightful message.

The finally block is executed irrespective of the outcome of try……except…..else blocks
One of the important use cases of finally block is in a function which returns a value.

Example:

```
try:
    num = int(input("Enter an integer: "))
except ValueError:
    print("Number entered is not an integer.")
else:
    print("Integer Accepted.")
finally:
    print("This block is always executed.")
```

Output 1:

```
Enter an integer: 19
Integer Accepted.
This block is always executed.
```

Output 2:

```
Enter an integer: 3.142
Number entered is not an integer.
This block is always executed.
```

If ... Else in One Line:-

There is also a shorthand syntax for the if-else statement that can be used when the condition being tested is simple and the code blocks to be executed are short. Here's an example:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

You can also have multiple else statements on the same line:

Conclusion

The shorthand syntax can be a convenient way to write simple if-else statements, especially when you want to assign a value to a variable based on a condition. However, it's not suitable for more complex situations where you need to execute multiple statements or perform more complex logic. In those cases, it's best to use the full if-else syntax.

Example

One line if else statement, with 3 conditions:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

Another Example

```
result = value_if_true if condition else value_if_false
```

This syntax is equivalent to the following if-else statement:

```
if condition:
    result = value_if_true
else:
    result = value_if_false
```

Enumerate function in python

The enumerate function is a built-in function in Python that allows you to loop over a sequence (such as a list, tuple, or string) and get the index and value of each element in the sequence at the same time. Here's a basic example of how it works:

```
# Loop over a list and print the index and value of each element
fruits = ['apple', 'banana', 'mango']
for index, fruit in enumerate(fruits):
    print(index, fruit)
```

The output of this code will be:

```
0 apple
1 banana
2 mango
```

As you can see, the enumerate function returns a tuple containing the index and value of each element in the sequence. You can use the for loop to unpack these tuples and assign them to variables, as shown in the example above.

Changing the start index:-

By default, the enumerate function starts the index at 0, but you can specify a different starting index by passing it as an argument to the enumerate function:

```
# Loop over a list and print the index (starting at 1) and value of each element
fruits = ['apple', 'banana', 'mango']
for index, fruit in enumerate(fruits, start=1):
   print(index, fruit)
```

This will output:

```
1 apple
2 banana
3 mango
```

The enumerate function is often used when you need to loop over a sequence and perform some action with both the index and value of each element. For example, you might use it to loop over a list of strings and print the index and value of each string in a formatted way:

```
fruits = ['apple', 'banana', 'mango']
for index, fruit in enumerate(fruits):
   print(f'{index+1}: {fruit}')
```

This will output:

```
1: apple
2: banana
3: mango
```

In addition to lists, you can use the enumerate function with any other sequence type in Python, such as tuples and strings. Here's an example with a tuple:

```
# Loop over a tuple and print the index and value of each element
colors = ('red', 'green', 'blue')
for index, color in enumerate(colors):
    print(index, color)
```

And here's an example with a string:

```
# Loop over a string and print the index and value of each character
s = 'hello'
for index, c in enumerate(s):
    print(index, c)
```

Virtual Environment

A virtual environment is a tool used to isolate specific Python environments on a single machine, allowing you to work on multiple projects with different dependencies and packages without conflicts. This can be especially useful when working on projects that have conflicting package versions or packages that are not compatible with each other.

To create a virtual environment in Python, you can use the venv module that comes with Python. Here's an example of how to create a virtual environment and activate it:

```
# Create a virtual environment
python -m venv myenv

# Activate the virtual environment (Linux/macOS)
source myenv/bin/activate

# Activate the virtual environment (Windows)
myenv\Scripts\activate.bat
```

Once the virtual environment is activated, any packages that you install using pip will be installed in the virtual environment, rather than in the global Python environment. This allows you to have a separate set of packages for each project, without affecting the packages installed in the global environment.

To deactivate the virtual environment, you can use the deactivate command:

```
# Deactivate the virtual environment
deactivate
```

The "requirements.txt" file:-

In addition to creating and activating a virtual environment, it can be useful to create a requirements.txt file that lists the packages and their versions that your project depends on. This file can be used to easily install all the required packages in a new environment.
To create a requirements.txt file, you can use the pip freeze command, which outputs a list of installed packages and their versions.
For example:

```
# Output the list of installed packages and their versions to a file
pip freeze > requirements.txt
```

To install the packages listed in the requirements.txt file, you can use the pip install command with the -r flag:

```
# Install the packages listed in the requirements.txt file
pip install -r requirements.txt
```

Using a virtual environment and a requirements.txt file can help you manage the dependencies for your Python projects and ensure that your projects are portable and can be easily set up on a new machine.

How importing in python works

Importing in Python is the process of loading code from a Python module into the current script. This allows you to use the functions and variables defined in the module in your current script, as well as any additional modules that the imported module may depend on.

To import a module in Python, you use the import statement followed by the name of the module. For example, to import the math module, which contains a variety of mathematical functions, you would use the following statement:

```
import math
```

Once a module is imported, you can use any of the functions and variables defined in the module by using the dot notation. For example, to use the sqrt function from the math module, you would write:

```
import math

result = math.sqrt(9)
print(result)  # Output: 3.0
```

from keyword

You can also import specific functions or variables from a module using the from keyword. For example, to import only the sqrt function from the math module, you would write:

```
from math import sqrt

result = sqrt(9)
print(result)  # Output: 3.0
```

You can also import multiple functions or variables at once by separating them with a comma:

```
from math import sqrt, pi

result = sqrt(9)
print(result)  # Output: 3.0

print(pi)  # Output: 3.141592653589793
```

importing everything

It's also possible to import all functions and variables from a module using the * wildcard. However, this is generally not recommended as it can lead to confusion and make it harder to understand where specific functions and variables are coming from.

```
from math import *

result = sqrt(9)
print(result)  # Output: 3.0

print(pi)  # Output: 3.141592653589793
```

Python also allows you to rename imported modules using the as keyword. This can be useful if you want to use a shorter or more descriptive name for a module, or if you want to avoid naming conflicts with other modules or variables in your code.

The "as" keyword

```
import math as m

result = m.sqrt(9)
print(result)  # Output: 3.0

print(m.pi)  # Output: 3.141592653589793
```

The dir function

Finally, Python has a built-in function called dir that you can use to view the names of all the functions and variables defined in a module. This can be helpful for exploring and understanding the contents of a new module.

```
import math

print(dir(math))
```

`if "__name__ == "__main__"` *in Python:-*

The `if __name__ == "__main__"` idiom is a common pattern used in Python scripts to determine whether the script is being run directly or being imported as a module into another script.

In Python, the `__name__` variable is a built-in variable that is automatically set to the name of the current module. When a Python script is run directly, the `__name__` variable is set to the string `__main__` When the script is imported as a module into another script, the `__name__` variable is set to the name of the module.
Here's an example of how the if `__name__` == `__main__` idiom can be used:

```python
def main():
    # Code to be run when the script is run directly
    print("Running script directly")

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

Is it a necessity?

It's important to note that the `if __name__ == "__main__"` idiom is not required to run a Python script. You can still run a script without it by simply calling the functions or running the code you want to execute directly. However, the `if __name__ == "__main__"` idiom can be a useful tool for organizing and separating code that should be run directly from code that should be imported and used as a module.
In summary, the `if __name__ == "__main__"` idiom is a common pattern used in Python scripts to determine whether the script is being run directly or being imported as a module into another script. It allows you to reuse code from a script by importing it as a module into another script, without running the code in the original script.

In this example, the main function contains the code that should be run when the script is run directly. The if statement at the bottom checks whether the `__name__` variable is equal to `__main__`. If it is, the main function is called.

Why is it useful?

This idiom is useful because it allows you to reuse code from a script by importing it as a module into another script, without running the code in the original script. For example, consider the following script:

```
def main():
    print("Running script directly")

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

If you run this script directly, it will output "Running script directly". However, if you import it as a module into another script and call the main function from the imported module, it will not output anything:

```
import script

script.main()  # Output: "Running script directly"
```

This can be useful if you have code that you want to reuse in multiple scripts, but you only want it to run when the script is run directly and not when it's imported as a module.

os Module in Python

The os module in Python is a built-in library that provides functions for interacting with the operating system. It allows you to perform a wide variety of tasks such as reading and writing files, interacting with the file system, and running system commands.
Here are some common tasks you can perform with the os module:
Reading and writing files The os module provides functions for opening, reading, and writing files. For example, to open a file for reading, you can use the open function:

```
import os

# Open the file in read-only mode
f = os.open("myfile.txt", os.O_RDONLY)

# Read the contents of the file
contents = os.read(f, 1024)

# Close the file
os.close(f)
```

To open a file for writing, you can use the os.O_WRONLY flag:

```
import os

# Open the file in write-only mode
f = os.open("myfile.txt", os.O_WRONLY)

# Write to the file
os.write(f, b"Hello, world!")

# Close the file
os.close(f)
```

Running system commands

Finally, the os module provides functions for running system commands. For example, you can use the os.system function to run a command and get the output:

```
import os

# Run the "ls" command and print the output
output = os.system("ls")
print(output)  # Output: ['myfile.txt', 'otherfile.txt']
```

You can also use the os.popen function to run a command and get the output as a file-like object:

```
import os

# Run the "ls" command and get the output as a file-like object
f = os.popen("ls")

# Read the contents of the output
output = f.read()
print(output)  # Output: ['myfile.txt', 'otherfile.txt']

# Close the file-like object
f.close()
```

In summary, the os module in Python is a built-in library that provides a wide variety of functions for interacting with the operating system. It allows you to perform tasks such as reading and writing files, interacting with the file system, and running system commands.

Interacting with the file system

The os module also provides functions for interacting with the file system. For example, you can use the os.listdir function to get a list of the files in a directory:

```
import os

# Get a list of the files in the current directory
files = os.listdir(".")
print(files)  # Output: ['myfile.txt', 'otherfile.txt']
```

You can also use the os.mkdir function to create a new directory:

```
import os

# Create a new directory
os.mkdir("newdir")
```

local and global variables

Before we dive into the differences between local and global variables, let's first recall what a variable is in Python.
A variable is a named location in memory that stores a value. In Python, we can assign values to variables using the assignment operator =. For example:

```
x = 5
y = "Hello, World!"
```

Now, let's talk about local and global variables.
A local variable is a variable that is defined within a function and is only accessible within that function. It is created when the function is called and is destroyed when the function returns.
On the other hand, a global variable is a variable that is defined outside of a function and is accessible from within any function in your code.
Here's an example to help clarify the difference:

```
x = 10 # global variable

def my_function():
 y = 5 # local variable
 print(y)

my_function()
print(x)
print(y) # this will cause an error because y is a local variable and is not accessible outside of the function
```

In this example, we have a global variable x and a local variable y. We can access the value of the global variable x from within the function, but we cannot access the value of the local variable y outside of the function

The global keyword

Now, what if we want to modify a global variable from within a function? This is where the global keyword comes in.
The global keyword is used to declare that a variable is a global variable and should be accessed from the global scope. Here's an example:

```
x = 10 # global variable

def my_function():
 global x
 x = 5 # this will change the value of the global variable x
 y = 5 # local variable

my_function()
print(x) # prints 5
print(y) # this will cause an error because y is a local variable and is not accessible outside of the function
```

In this example, we used the global keyword to declare that we want to modify the global variable x from within the function. As a result, the value of x is changed to 5.
It's important to note that it's generally considered good practice to avoid modifying global variables from within functions, as it can lead to unexpected behavior and make your code harder to debug.

Opening a File

Before we can perform any operations on a file, we must first open it. Python provides the open() function to open a file. It takes two arguments: the name of the file and the mode in which the file should be opened. The mode can be 'r' for reading, 'w' for writing, or 'a' for appending.
Here's an example of how to open a file for reading:

```
f = open('myfile.txt', 'r')
```

By default, the open() function returns a file object that can be used to read from or write to the file, depending on the mode.

Modes in file

There are various modes in which we can open files.

1. read (r): This mode opens the file for reading only and gives an error if the file does not exist. This is the default mode if no mode is passed as a parameter.
2. write (w): This mode opens the file for writing only and creates a new file if the file does not exist.
3. append (a): This mode opens the file for appending only and creates a new file if the file does not exist.
4. create (x): This mode creates a file and gives an error if the file already exists.
5. text (t): Apart from these modes we also need to specify how the file must be handled. t mode is used to handle text files. t refers to the text mode. There is no difference between r and rt or w and wt since text mode is the default. The default mode is 'r' (open for reading text, synonym of 'rt').
6. binary (b): used to handle binary files (images, pdfs, etc).

Writing to a File

To write to a file, we first need to open it in write mode.

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

We can then use the write() method to write to the file.

```
f = open('myfile.txt', 'w')
f.write('Hello, world!')
```

Keep in mind that writing to a file will overwrite its contents. If you want to append to a file instead of overwriting it, you can open it in append mode.

```
f = open('myfile.txt', 'a')
f.write('Hello, world!')
```

Reading from a File

Once we have a file object, we can use various methods to read from the file. The read() method reads the entire contents of the file and returns it as a string.

```
f = open('myfile.txt', 'r')
contents = f.read()
print(contents)
```

Closing a File

It is important to close a file after you are done with it. This releases the resources used by the file and allows other programs to access it.

To close a file, you can use the close() method.

```
f = open('myfile.txt', 'r')
# ... do something with the file
f.close()
```

The 'with' statement

Alternatively, you can use the with statement to automatically close the file after you are done with it.

```
with open('myfile.txt', 'r') as f:
    # ... do something with the file
```

writelines() method

The writelines() method in Python writes a sequence of strings to a file. The sequence can be any iterable object, such as a list or a tuple.

Here's an example of how to use the writelines() method:

```
f = open('myfile.txt', 'w')
lines = ['line 1\n', 'line 2\n', 'line 3\n']
f.writelines(lines)
f.close()
```

This will write the strings in the lines list to the file myfile.txt. The \n characters are used to add newline characters to the end of each string.

Keep in mind that the writelines() method does not add newline characters between the strings in the sequence. If you want to add newlines between the strings, you can use a loop to write each string separately:

```
f = open('myfile.txt', 'w')
lines = ['line 1', 'line 2', 'line 3']
for line in lines:
    f.write(line + '\n')
f.close()
```

It is also a good practice to close the file after you are done with it.

readlines() method

The readline() method reads a single line from the file. If we want to read multiple lines, we can use a loop.

```
f = open('myfile.txt', 'r')
while True:
    line = f.readline()
    if not line:
        break
    print(line)
```

The readlines() method reads all the lines of the file and returns them as a list of strings.

seek() and tell() functions

In Python, the seek() and tell() functions are used to work with file objects and their positions within a file. These functions are part of the built-in io module, which provides a consistent interface for reading and writing to various file-like objects, such as files, pipes, and in-memory buffers.

seek() function

The seek() function allows you to move the current position within a file to a specific point. The position is specified in bytes, and you can move either forward or backward from the current position. For example:

```
with open('file.txt', 'r') as f:
 # Move to the 10th byte in the file
 f.seek(10)

 # Read the next 5 bytes
 data = f.read(5)
```

tell() function

The tell() function returns the current position within the file, in bytes.This can be useful for keeping track of your location within the file or for seeking to a specific position relative to the current position. For example:

```
with open('file.txt', 'r') as f:
 # Read the first 10 bytes
 data = f.read(10)

 # Save the current position
 current_position = f.tell()

 # Seek to the saved position
 f.seek(current_position)
```

truncate() function

When you open a file in Python using the open function, you can specify the mode in which you want to open the file. If you specify the mode as 'w' or 'a', the file is opened in write mode and you can write to the file. However, if you want to truncate the file to a specific size, you can use the truncate function.
Here is an example of how to use the truncate function:

```
with open('sample.txt', 'w') as f:
        f.write('Hello World!')
          f.truncate(5)

with open('sample.txt', 'r') as f:
    print(f.read())
```

Lambda Functions in Python

In Python, a lambda function is a small anonymous function without a name. It is defined using the lambda keyword and has the following syntax:

```
lambda arguments: expression
```

Lambda functions are often used in situations where a small function is required for a short period of time. They are commonly used as arguments to higher-order functions, such as map, filter, and reduce.

Here is an example of how to use a lambda function:

```
# Function to double the input
def double(x):
 return x * 2

# Lambda function to double the input
lambda x: x * 2
```

The above lambda function has the same functionality as the double function defined earlier. However, the lambda function is anonymous, as it does not have a name.

Lambda functions can have multiple arguments, just like regular functions. Here is an example of a lambda function with multiple arguments:

```
# Function to calculate the product of two numbers
def multiply(x, y):
   return x * y

# Lambda function to calculate the product of two numbers
lambda x, y: x * y
```

Lambda functions can also include multiple statements, but they are limited to a single expression. For example:

```
# Lambda function to calculate the product of two numbers,
# with additional print statement
lambda x, y: print(f'{x} * {y} = {x * y}')
```

In the above example, the lambda function includes a print statement, but it is still limited to a single expression.

Map, Filter and Reduce

In Python, the map, filter, and reduce functions are built-in functions that allow you to apply a function to a sequence of elements and return a new sequence. These functions are known as higher-order functions, as they take other functions as arguments.

map

The map function applies a function to each element in a sequence and returns a new sequence containing the transformed elements. The map function has the following syntax:

```
map(function, iterable)
```

The function argument is a function that is applied to each element in the iterable argument. The iterable argument can be a list, tuple, or any other iterable object.

Here is an example of how to use the map function:

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# Double each number using the map function
doubled = map(lambda x: x * 2, numbers)

# Print the doubled numbers
print(list(doubled))
```

map applies a given function to each item in an iterable (like a list) and returns a map object (which can be converted to a list).

In the above example, the lambda function lambda x: x * 2 is used to double each element in the numbers list. The map function applies the lambda function to each element in the list and returns a new list containing the doubled numbers.

filter

The filter function filters a sequence of elements based on a given predicate (a function that returns a boolean value) and returns a new sequence containing only the elements that meet the predicate. The filter function has the following syntax:

`filter(predicate, iterable)`

The predicate argument is a function that returns a boolean value and is applied to each element in the iterable argument. The iterable argument can be a list, tuple, or any other iterable object.

Here is an example of how to use the filter function:

```
# List of numbers
numbers = [1, 2, 3, 4, 5]

# Get only the even numbers using the filter function
evens = filter(lambda x: x % 2 == 0, numbers)

# Print the even numbers
print(list(evens))
```

In the above example, the lambda function lambda x: x % 2 == 0 is used to filter the numbers list and return only the even numbers. The filter function applies the lambda function to each element in the list and returns a new list containing only the even numbers.

reduce

The reduce function is a higher-order function that applies a function to a sequence and returns a single value. It is a part of the functools module in Python and has the following syntax:

```
reduce(function, iterable)
```

The function argument is a function that takes in two arguments and returns a single value. The iterable argument is a sequence of elements, such as a list or tuple.

The reduce function applies the function to the first two elements in the iterable and then applies the function to the result and the next element, and so on. The reduce function returns the final result.

Here is an example of how to use the reduce function:

```
from functools import reduce

# List of numbers
numbers = [1, 2, 3, 4, 5]

# Calculate the sum of the numbers using the reduce function
sum = reduce(lambda x, y: x + y, numbers)

# Print the sum
print(sum)
```

In the above example, the reduce function applies the lambda function lambda x, y: x + y to the elements in the numbers list. The lambda function adds the two arguments x and y and returns the result. The reduce function applies the lambda function to the first two elements in the list (1 and 2), then applies the function to the result (3) and the next element (3), and so on. The final result is the sum of all the elements in the list, which is 15.

It is important to note that the reduce function requires the functools module to be imported in order to use it.

'is' vs '==' in Python

In Python, is and == are both comparison operators that can be used to check if two values are equal. However, there are some important differences between the two that you should be aware of.

The ‘is’ operator compares the identity of two objects, while the ‘==’ operator compares the values of the objects. This means that is will only return True if the objects being compared are the exact same object in memory, while == will return True if the objects have the same value.

For example:

```
a = [1, 2, 3]
b = [1, 2, 3]

print(a == b)  # True
print(a is b)  # False
```

In this case, a and b are two separate lists that have the same values, so == returns True. However, a and b are not the same object in memory, so is returns False.

One important thing to note is that, in Python, strings and integers are immutable, which means that once they are created, their value cannot be changed. This means that, for strings and integers, is and == will always return the same result:

```
a = "hello"
b = "hello"

print(a == b)  # True
print(a is b)  # True

a = 5
b = 5

print(a == b)  # True
print(a is b)  # True
```

In these cases, a and b are both pointing to the same object in memory, so is and == both return True.

For mutable objects such as lists and dictionaries, is and == can behave differently. In general, you should use == when you want to compare the values of two objects, and use is when you want to check if two objects are the same object in memory.

Introduction to Object-oriented programming

Introduction to Object-Oriented Programming in Python: In programming languages, mainly there are two approaches that are used to write program or code.

- 1). Procedural Programming
- 2). Object-Oriented Programming

The procedure we are following till now is the “Procedural Programming” approach. So, in this session, we will learn about Object Oriented Programming (OOP).

The basic idea of object-oriented programming (OOP) in Python is to use classes and objects to represent real-world concepts and entities.

A class is a blueprint or template for creating objects. It defines the properties and methods that an object of that class will have. Properties are the data or state of an object, and methods are the actions or behaviors that an object can perform.

An object is an instance of a class, and it contains its own data and methods. For example, you could create a class called "Person" that has properties such as name and age, and methods such as speak() and walk(). Each instance of the Person class would be a unique object with its own name and age, but they would all have the same methods to speak and walk.

One of the key features of OOP in Python is encapsulation, which means that the internal state of an object is hidden and can only be accessed or modified through the object's methods. This helps to protect the object's data and prevent it from being modified in unexpected ways.

Another key feature of OOP in Python is inheritance, which allows new classes to be created that inherit the properties and methods of an existing class. This allows for code reuse and makes it easy to create new classes that have similar functionality to existing classes.

Polymorphism is also supported in Python, which means that objects of different classes can be treated as if they were objects of a common class. This allows for greater flexibility in code and makes it easier to write code that can work with multiple types of objects.

In summary, OOP in Python allows developers to model real-world concepts and entities using classes and objects, encapsulate data, reuse code through inheritance, and write more flexible code through polymorphism.

Python Class and Objects

A class is a blueprint or a template for creating objects, providing initial values for state (member variables or attributes), and implementations of behavior (member functions or methods). The user-defined objects are created using the class keyword.

Creating a Class:

Let us now create a class using the class keyword.

```
class Details:
    name = "Rohan"
    age = 20
```

Example:

```
obj1 = Details()
```

Now we can print values:

Creating an Object:

Object is the instance of the class used to access the properties of the class Now lets create an object of the class.

Example:

```
class Details:
    name = "Rohan"
    age = 20

obj1 = Details()
print(obj1.name)
print(obj1.age)
```

Output:

```
Rohan
20
```

self parameter

The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
It must be provided as the extra parameter inside the method definition.
VO OBJECT JISPE METHOD CALL HO RAHA HAI

Example:

```
class Details:
    name = "Rohan"
    age = 20

    def desc(self):
        print("My name is", self.name, "and I'm", self.age, "years old.")

obj1 = Details()
obj1.desc()
```

Output:

```
My name is Rohan and I'm 20 years old.
```

Constructors

A constructor is a special method in a class used to create and initialize an object of a class. There are different types of constructors. Constructor is invoked automatically when an object of a class is created. A constructor is a unique function that gets called automatically when an object is created of a class. The main purpose of a constructor is to initialize or assign values to the data members of that class. It cannot return any value other than None.

Syntax of Python Constructor

```
def __init__(self):
    # initializations
```

init is one of the reserved functions in Python. In Object Oriented Programming, it is known as a constructor.

Types of Constructors in Python

1. Parameterized Constructor
2. Default Constructor

Parameterized Constructor in Python

When the constructor accepts arguments along with self, it is known as parameterized constructor.
These arguments can be used inside the class to assign the values to the data members.

Example:

```
class Details:
    def __init__(self, animal, group):
        self.animal = animal
        self.group = group

obj1 = Details("Crab", "Crustaceans")
print(obj1.animal, "belongs to the",
obj1.group, "group.")
```

Output:

```
Crab belongs to the Crustaceans group.
```

Default Constructor in Python

When the constructor doesn't accept any arguments from the object and has only one argument, self, in the constructor, it is known as a Default constructor.

Example:

```
class Details:
 def __init__(self):
    print("animal Crab belongs to
Crustaceans group")
obj1=Details()
```

Output:

```
animal Crab belongs to Crustaceans
group
```

Python Decorators

Python decorators are a powerful and versatile tool that allow you to modify the behavior of functions and methods. They are a way to extend the functionality of a function or method without modifying its source code.
A decorator is a function that takes another function as an argument and returns a new function that modifies the behavior of the original function. The new function is often referred to as a "decorated" function. The basic syntax for using a decorator is the following:

```
@decorator_function
def my_function():
    pass
```

The @decorator_function notation is just a shorthand for the following code:

```
def my_function():
    pass
my_function = decorator_function(my_function)
```

Decorators are often used to add functionality to functions and methods, such as logging, memoization, and access control.

Practical use case

One common use of decorators is to add logging to a function. For example, you could use a decorator to log the arguments and return value of a function each time it is called:

```
import logging

def log_function_call(func):
    def decorated(*args, **kwargs):
        logging.info(f"Calling {func.__name__}
with args={args}, kwargs={kwargs}")
        result = func(*args, **kwargs)
        logging.info(f"{func.__name__} returned
{result}")
        return result
    return decorated

@log_function_call
def my_function(a, b):
    return a + b
```

In this example, the log_function_call decorator takes a function as an argument and returns a new function that logs the function call before and after the original function is called.

Conclusion

Decorators are a powerful and flexible feature in Python that can be used to add functionality to functions and methods without modifying their source code. They are a great tool for separating concerns, reducing code duplication, and making your code more readable and maintainable.

In conclusion, python decorators are a way to extend the functionality of functions and methods, by modifying its behavior without modifying the source code. They are used for a variety of purposes, such as logging, memoization, access control, and more. They are a powerful tool that can be used to make your code more readable, maintainable, and extendable.

Getters

Getters in Python are methods that are used to access the values of an object's properties. They are used to return the value of a specific property, and are typically defined using the @property decorator. Here is an example of a simple class with a getter method:

```
class MyClass:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        return self._value
```

In this example, the MyClass class has a single property, _value, which is initialized in the **init** method. The value method is defined as a getter using the @property decorator, and is used to return the value of the _value property.

To use the getter, we can create an instance of the MyClass class, and then access the value property as if it were an attribute:

```
>>> obj = MyClass(10)
>>> obj.value
10
```

Setters

It is important to note that the getters do not take any parameters and we cannot set the value through getter method.For that we need setter method which can be added by decorating method with @property_name.setter

Here is an example of a class with both getter and setter:

```
class MyClass:
    def __init__(self, value):
        self._value = value

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, new_value):
        self._value = new_value
```

We can use setter method like this:

```
>>> obj = MyClass(10)
>>> obj.value = 20
>>> obj.value
20
```

In conclusion, getters are a convenient way to access the values of an object's properties, while keeping the internal representation of the property hidden. This can be useful for encapsulation and data validation.

Inheritance in python

When a class derives from another class. The child class will inherit all the public and protected properties and methods from the parent class. In addition, it can have its own properties and methods,this is called as inheritance.

Python Inheritance Syntax

```
class BaseClass:
 Body of base class
class DerivedClass(BaseClass):
 Body of derived class
```

Derived class inherits features from the base class where new features can be added to it. This results in re-usability of code.

Types of inheritance:

1. Single inheritance
2. Multiple inheritance
3. Multilevel inheritance
4. Hierarchical Inheritance
5. Hybrid Inheritance

Single Inheritance:

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

Example:

```
class Parent:
    def func1(self):
        print("This function is in parent class.")
class Child(Parent):
    def func2(self):
        print("This function is in child class.")
object = Child()
object.func1()
object.func2()
```

Output:

```
This function is in parent class.
This function is in child class.
```

Multiple Inheritance:

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.

Example:

```
class Mother:
    mothername = ""
    def mother(self):
        print(self.mothername)
class Father:
    fathername = ""
    def father(self):
        print(self.fathername)
class Son(Mother, Father):
    def parents(self):
        print("Father name is :", self.fathername)
        print("Mother :", self.mothername)
s1 = Son()
s1.fathername = "Mommy"
s1.mothername = "Daddy"
s1.parents()
```

Output:

```
Father name is : Mommy
Mother name is : Daddy
```

Multilevel Inheritance :

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class. This is similar to a relationship representing a child and a grandfather.

Example:

```
class Grandfather:
    def __init__(self, grandfathername):
        self.grandfathername = grandfathername
class Father(Grandfather):
    def __init__(self, fathername, grandfathername):
        self.fathername = fathername
        Grandfather.__init__(self, grandfathername)
class Son(Father):
    def __init__(self, sonname, fathername, grandfathername):
        self.sonname = sonname
        Father.__init__(self, fathername, grandfathername)
    def print_name(self):
        print('Grandfather name :', self.grandfathername)
        print("Father name :", self.fathername)
        print("Son name :", self.sonname)
s1 = Son('Prince', 'Rampal', 'Lal mani')
print(s1.grandfathername)
s1.print_name()
```

Output:

```
George
Grandfather name : George
Father name : Philip
Son name : Charles
```

Hybrid Inheritance:

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

Example

```
class School:
    def func1(self):
        print("This function is in school.")
class Student1(School):
    def func2(self):
        print("This function is in student 1. ")
class Student2(School):
    def func3(self):
        print("This function is in student 2.")
class Student3(Student1, School):
    def func4(self):
        print("This function is in student 3.")
object = Student3()
object.func1()
object.func2()
```

Output:

```
This function is in school.
This function is in student 1.
```

Hierarchical Inheritance:

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.

Example:

```
class Parent:
    def func1(self):
        print("This function is in parent class.")

class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")

class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Output:

```
This function is in parent class.
This function is in child 1.
This function is in parent class.
This function is in child 2.
```

Access Specifiers/Modifiers

Access specifiers or access modifiers in python programming are used to limit the access of class variables and class methods outside of class while implementing the concepts of inheritance.
Let us see the each one of access specifiers in detail:

Types of access specifiers

1. Public access modifier
2. Private access modifier
3. Protected access modifier

Public Access Specifier in Python

All the variables and methods (member functions) in python are by default public. Any instance variable in a class followed by the 'self' keyword ie. self.var_name are public accessed.

Example:

```
class Student:
   # constructor is defined
   def __init__(self, age, name):
       self.age = age                # public variable
       self.name = name              # public variable

obj = Student(21,"Harry")
print(obj.age)
print(obj.name)
```

Output:

```
21
Harry
```

Private Access Modifier

By definition, Private members of a class (variables or methods) are those members which are only accessible inside the class. We cannot use private members outside of class.
In Python, there is no strict concept of "private" access modifiers like in some other programming languages. However, a convention has been established to indicate that a variable or method should be considered private by prefixing its name with a double underscore (__). This is known as a "weak internal use indicator" and it is a convention only, not a strict rule. Code outside the class can still access these "private" variables and methods, but it is generally understood that they should not be accessed or modified.

Example:

```
class Student:
    def __init__(self, age, name):
        self.__age = age       # An indication of private variable

        def __funName(self):  # An indication of private function
            self.y = 34
            print(self.y)

class Subject(Student):
    pass

obj = Student(21,"Harry")
obj1 = Subject
```

```
# calling by object of class Student
print(obj.__age)
print(obj.__funName())

# calling by object  of class Subject
print(obj1.__age)
print(obj1.__funName())
```

Output:

```
AttributeError: 'student' object has no attribute '__age'
AttributeError: 'student' object has no method '__funName()'
AttributeError: 'subject' object has no attribute '__age'
AttributeError: 'student' object has no method '__funName()'
```

Private members of a class cannot be accessed or inherited outside of class. If we try to access or to inherit the properties of private members to child class (derived class). Then it will show the error.

Name mangling

Name mangling in Python is a technique used to protect class-private and superclass-private attributes from being accidentally overwritten by subclasses. Names of class-private and superclass-private attributes are transformed by the addition of a single leading underscore and a double leading underscore respectively.

```
class MyClass:
    def __init__(self):
        self._nonmangled_attribute = "I am a nonmangled attribute"
        self.__mangled_attribute = "I am a mangled attribute"

my_object = MyClass()

print(my_object._nonmangled_attribute) # Output: I am a nonmangled attribute
print(my_object.__mangled_attribute) # Throws an AttributeError
print(my_object._MyClass__mangled_attribute) # Output: I am a mangled attribute
```

In the example above, the attribute _nonmangled_attribute is marked as nonmangled by convention, but can still be accessed from outside the class. The attribute __mangled_attribute is private and its name is "mangled" to _MyClass__mangled_attribute, so it can't be accessed directly from outside the class, but you can access it by calling _MyClass__mangled_attribute

Protected Access Modifier

In object-oriented programming (OOP), the term "protected" is used to describe a member (i.e., a method or attribute) of a class that is intended to be accessed only by the class itself and its subclasses. In Python, the convention for indicating that a member is protected is to prefix its name with a single underscore (_). For example, if a class has a method called _my_method, it is indicating that the method should only be accessed by the class itself and its subclasses.
It's important to note that the single underscore is just a naming convention, and does not actually provide any protection or restrict access to the member. The syntax we follow to make any variable protected is to write variable name followed by a single underscore (_) ie. _varName.

Example:

```
class Student:
    def __init__(self):
        self._name = "Harry"

    def _funName(self):       # protected method
        return "CodeWithHarry"

class Subject(Student):       #inherited class
    pass

obj = Student()
obj1 = Subject()

# calling by object of Student class
print(obj._name)
print(obj._funName())
# calling by object of Subject class
print(obj1._name)
print(obj1._funName())
```

Output:

```
Harry
CodeWithHarry

Harry
CodeWithHarry
```

Static methods :-

In Python are methods that belong to a class rather than an instance of the class. They are defined using the @staticmethod decorator and do not have access to the instance of the class (i.e. self). They are called on the class itself, not on an instance of the class. Static methods are often used to create utility functions that don't need access to instance data.

```
class Math:
    @staticmethod
    def add(a, b):
        return a + b
result = Math.add(1, 2)
print (result) # Output: 3
```

In this example, the add method is a static method of the Math class. It takes two parameters a and b and returns their sum. The method can be called on the class itself, without the need to create an instance of the class.

```
class Math:
 def __init__(self, num):
  self.num = num

 def addtonum(self, n):
  self.num = self.num +n

 @staticmethod
 def add(a, b):
     return a + b

# result = Math.add(1, 2)
# print(result) # Output: 3
a = Math(5)
print(a.num)
a.addtonum(6)
print(a.num)

print(Math.add(7, 2))
```

Instance vs class variables

In Python, variables can be defined at the class level or at the instance level. Understanding the difference between these types of variables is crucial for writing efficient and maintainable code.

Class Variables

Class variables are defined at the class level and are shared among all instances of the class. They are defined outside of any method and are usually used to store information that is common to all instances of the class. For example, a class variable can be used to store the number of instances of a class that have been created.

```
class MyClass:
    class_variable = 0

    def __init__(self):
        MyClass.class_variable += 1

    def print_class_variable(self):
        print(MyClass.class_variable)


obj1 = MyClass()
obj2 = MyClass()

obj1.print_class_variable() # Output: 2
obj2.print_class_variable() # Output: 2
```

In the example above, the class_variable is shared among all instances of the class MyClass. When we create new instances of MyClass, the value of class_variable is incremented. When we call the print_class_variable method on obj1 and obj2, we get the same value of class_variable.

Instance Variables

Instance variables are defined at the instance level and are unique to each instance of the class. They are defined inside the **init** method and are usually used to store information that is specific to each instance of the class. For example, an instance variable can be used to store the name of an employee in a class that represents an employee.

In the example , each instance of the class MyClass has its own value for the name variable. When we call the print_name method on obj1 and obj2, we get different values for name.

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

    def print_name(self):
        print(self.name)

obj1 = MyClass("John")
obj2 = MyClass("Jane")

obj1.print_name() # Output: John
obj2.print_name() # Output: Jane
```

Summary

In summary, class variables are shared among all instances of a class and are used to store information that is common to all instances. Instance variables are unique to each instance of a class and are used to store information that is specific to each instance. Understanding the difference between class variables and instance variables is crucial for writing efficient and maintainable code in Python.
It's also worth noting that, in python, class variables are defined outside of any methods and don't need to be explicitly declared as class variable. They are defined in the class level and can be accessed via classname.varibale_name or self.class.variable_name. But instance variables are defined inside the methods and need to be explicitly declared as instance variable by using self.variable_name.

Python Class Methods

Python Class Methods: An Introduction

In Python, classes are a way to define custom data types that can store data and define functions that can manipulate that data. One type of function that can be defined within a class is called a "method." In this blog post, we will explore what Python class methods are, why they are useful, and how to use them.

What are Python Class Methods?

A class method is a type of method that is bound to the class and not the instance of the class. In other words, it operates on the class as a whole, rather than on a specific instance of the class. Class methods are defined using the "@classmethod" decorator, followed by a function definition. The first argument of the function is always "cls," which represents the class itself.

Why Use Python Class Methods?

Class methods are useful in several situations. For example, you might want to create a factory method that creates instances of your class in a specific way. You could define a class method that creates the instance and returns it to the caller. Another common use case is to provide alternative constructors for your class. This can be useful if you want to create instances of your class in multiple ways, but still have a consistent interface for doing so.

How to Use Python Class Methods

To define a class method, you simply use the "@classmethod" decorator before the method definition. The first argument of the method should always be "cls," which represents the class itself. Here is an example of how to define a class method:

```
class ExampleClass:
    @classmethod
    def factory_method(cls, argument1, argument2):
        return cls(argument1, argument2)
```

In this example, the "factory_method" is a class method that takes two arguments, "argument1" and "argument2." It creates a new instance of the class "ExampleClass" using the "cls" keyword, and returns the new instance to the caller.

It's important to note that class methods cannot modify the class in any way. If you need to modify the class, you should use a class level variable instead.

Conclusion

Python class methods are a powerful tool for defining functions that operate on the class as a whole, rather than on a specific instance of the class. They are useful for creating factory methods, alternative constructors, and other types of methods that operate at the class level. With the knowledge of how to define and use class methods, you can start writing more complex and organized code in Python.

Class Methods as Alternative Constructors

In object-oriented programming, the term "constructor" refers to a special type of method that is automatically executed when an object is created from a class. The purpose of a constructor is to initialize the object's attributes, allowing the object to be fully functional and ready to use.

However, there are times when you may want to create an object in a different way, or with different initial values, than what is provided by the default constructor. This is where class methods can be used as alternative constructors.

A class method belongs to the class rather than to an instance of the class. One common use case for class methods as alternative constructors is when you want to create an object from data that is stored in a different format, such as a string or a dictionary. For example, consider a class named "Person" that has two attributes: "name" and "age". The default constructor for the class might look like this:

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

But what if you want to create a Person object from a string that contains the person's name and age, separated by a comma? You can define a class method named "from_string" to do this:

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

    @classmethod
    def from_string(cls, string):
        name, age = string.split(',')
        return cls(name, int(age))
```

Now you can create a Person object from a string like this:

```
person = Person.from_string("John Doe, 30")
```

Another common use case for class methods as alternative constructors is when you want to create an object with a different set of default values than what is provided by the default constructor. For example, consider a class named "Rectangle" that has two attributes: "width" and "height". The default constructor for the class might look like this:

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

But what if you want to create a Rectangle object with a default width of 10 and a default height of 5? You can define a class method named "square" to do this:

```
class Rectangle:
 def __init__(self, width, height):
    self.width = width
    self.height = height

 @classmethod
 def square(cls, size):
    return cls(size, size)
```

Now you can create a square rectangle like this:

```
rectangle = Rectangle.square(10)
```

⭕dir(), __dict__ and help() methods in python

We must look into dir(), __dict__() and help() attribute/methods in python. They make it easy for us to understand how classes resolve various functions and executes code. In Python, there are three built-in functions that are commonly used to get information about objects: dir(), **dict**, and help(). Let's take a look at each of them:

♦The dir() method

dir(): The dir() function returns a list of all the attributes and methods (including dunder methods) available for an object. It is a useful tool for discovering what you can do with an object. Example:

```
>>> x = [1, 2, 3]
>>> dir(x)
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
'__gt__', '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
'__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__', '__setattr__',
'__setitem__', '__sizeof__', '__str__', '__subclasshook__', 'append', 'clear', 'copy',
'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

The __dict__ attribute

__dict__: The __dict__ attribute returns a dictionary representation of an object's attributes. It is a useful tool for introspection. Example:

```
>>> class Person:
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> p = Person("John", 30)
>>> p.__dict__
```

Output

```
{'name': 'John', 'age': 30}
```

The help() mehthod

help(): The help() function is used to get help documentation for an object, including a description of its attributes and methods. Example:

```
>>> help(str)
Help on class str in module builtins:

class str(object)
 |  str(object='') -> str
 |  str(bytes_or_buffer[, encoding[, errors]]) -> str
 |
 |  Create a new string object from the given object. If encoding or
 |  errors is specified, then the object must expose a data buffer
 |  that will be decoded using the given encoding and error handler.
 |  Otherwise, returns the result of object.__str__() (if defined)
 |  or repr(object).
 |  encoding defaults to sys.getdefaultencoding().
 |  errors defaults to 'strict'.
```

In conclusion, dir(), **dict**, and help() are useful built-in functions in Python that can be used to get information about objects. They are valuable tools for introspection and discovery.

Inheritance vs. `super` Keyword in Python

Inheritance

Definition: Inheritance is a fundamental concept in object-oriented programming where a new class (child or subclass) is created based on an existing class (parent or superclass). The child class inherits attributes and methods from the parent class, allowing for code reuse and the creation of hierarchical class structures.

Key Points:

- **Code Reuse:** Inheritance allows the child class to reuse code from the parent class, reducing redundancy.
- **Hierarchy:** Creates a hierarchical relationship between classes.
- **Overriding:** Child classes can override methods from the parent class to provide specific behavior.

When to Use:

- When you have common functionality that should be shared across multiple classes.
- When you want to create a logical hierarchy between classes.

`super` Keyword

Definition: The `super` keyword is used to call a method from the parent class within a child class. It is typically used within an overridden method to extend or modify the behavior of the parent class method.

Key Points:

- **Access Parent Methods:** Allows access to methods and properties of the parent class.
- **Extend Functionality:** Used to extend or modify the behavior of a parent class method in the child class.
- **Initialization:** Commonly used in constructors to initialize the parent class.

When to Use:

- When you want to extend the behavior of a method from the parent class rather than completely overriding it.
- When you need to initialize a parent class within a child's constructor.

python

```python
class Animal:
    def speak(self):
        return "Animal speaks"

class Dog(Animal):
    def bark(self):
        return "Dog barks"

# Usage
dog = Dog()
print(dog.speak())  # Inherited from Animal
print(dog.bark())   # Defined in Dog
```

python

```python
class Animal:
    def speak(self):
        return "Animal speaks"

class Dog(Animal):
    def speak(self):
        parent_speak = super().speak()  # Call the speak method from Animal class
        return f"{parent_speak} and Dog barks"

# Usage
dog = Dog()
print(dog.speak())  # Output: "Animal speaks and Dog barks"
```

Differences

- **Purpose:**
  - **Inheritance:** Defines a new class based on an existing class to reuse code and create a hierarchical structure.
  - **super:** Used within a child class to call methods from the parent class, often to extend or modify the inherited methods.
- **Usage:**
  - **Inheritance:** Declared by specifying the parent class in parentheses after the child class name (`` `class Child(Parent):` ``).
  - **super:** Used within methods of the child class (`` `super().method_name()` ``).
- **Functionality:**
  - **Inheritance:** Inherits all attributes and methods from the parent class, allowing for overriding and extension.
  - **super:** Specifically calls the parent class's methods or properties from the child class to maintain or extend behavior.

Summary

- **Inheritance:** Establishes a relationship between classes, enabling a child class to inherit and override methods and properties from a parent class.
- **super:** Used within the context of inheritance to call and extend methods from the parent class within the child class.

Super keyword in Python

The super() keyword in Python is used to refer to the parent class. It is especially useful when a class inherits from multiple parent classes and you want to call a method from one of the parent classes.
When a class inherits from a parent class, it can override or extend the methods defined in the parent class. However, sometimes you might want to use the parent class method in the child class. This is where the super() keyword comes in handy.
Here's an example of how to use the super() keyword in a simple inheritance scenario:

```
class ParentClass:
    def parent_method(self):
        print("This is the parent method.")

class ChildClass(ParentClass):
    def child_method(self):
        print("This is the child method.")
        super().parent_method()

child_object = ChildClass()
child_object.child_method()
```

Output:

```
This is the child method.
This is the parent method.
```

In this example, we have a ParentClass with a parent_method and a ChildClass that inherits from ParentClass and overrides the child_method. When the child_method is called, it first prints "This is the child method." and then calls the parent_method using the super() keyword.

The super() keyword is also useful when a class inherits from multiple parent classes. In this case, you can specify the parent class from which you want to call the method.

Here's an example:

```
class ParentClass1:
    def parent_method(self):
        print("This is the parent method of ParentClass1.")

class ParentClass2:
    def parent_method(self):
        print("This is the parent method of ParentClass2.")

class ChildClass(ParentClass1, ParentClass2):
    def child_method(self):
        print("This is the child method.")
        super().parent_method()

child_object = ChildClass()
child_object.child_method()
```

Output:

```
This is the child method.
This is the parent method of ParentClass1.
```

In this example, the ChildClass inherits from both ParentClass1 and ParentClass2. The child_method calls the parent_method of the first parent class using the super() keyword.

In conclusion, the super() keyword is a useful tool in Python when you want to call a parent class method in a child class. It can be used in inheritance scenarios with a single parent class or multiple parent classes.

Magic/Dunder Methods in Python

These are special methods that you can define in your classes, and when invoked, they give you a powerful way to manipulate objects and their behaviour.
Magic methods, also known as “dunders” from the double underscores surrounding their names, are powerful tools that allow you to customize the behaviour of your classes. They are used to implement special methods such as the addition, subtraction and comparison operators, as well as some more advanced techniques like descriptors and properties.
Let’s take a look at some of the most commonly used magic methods in Python.

__init__ method

The **init** method is a special method that is automatically invoked when you create a new instance of a class. This method is responsible for setting up the object’s initial state, and it is where you would typically define any instance variables that you need. Also called "constructor", we have discussed this method already

__str__ and __repr__ methods

The **str** and **repr** methods are both used to convert an object to a string representation. The **str** method is used when you want to print out an object, while the **repr** method is used when you want to get a string representation of an object that can be used to recreate the object.

__len__ method

The **len** method is used to get the length of an object. This is useful when you want to be able to find the size of a data structure, such as a list or dictionary.

__call__ method

The **call** method is used to make an object callable, meaning that you can pass it as a parameter to a function and it will be executed when the function is called. This is an incredibly powerful tool that allows you to create objects that behave like functions.
These are just a few of the many magic methods available in Python. They are incredibly powerful tools that allow you to customize the behaviour of your objects, and can make your code much cleaner and easier to understand. So if you’re looking for a way to take your Python code to the next level, take some time to learn about these magic methods.

Method Overriding in Python

Method overriding is a powerful feature in object-oriented programming that allows you to redefine a method in a derived class. The method in the derived class is said to override the method in the base class. When you create an instance of the derived class and call the overridden method, the version of the method in the derived class is executed, rather than the version in the base class.
In Python, method overriding is a way to customize the behavior of a class based on its specific needs. For example, consider the following base class:

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

In this base class, the area method is defined, but does not have any implementation. If you want to create a derived class that represents a circle, you can override the area method and provide an implementation that calculates the area of a circle:

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

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

In this example, the Circle class inherits from the Shape class, and overrides the area method. The new implementation of the area method calculates the area of a circle, based on its radius.

It's important to note that when you override a method, the new implementation must have the same method signature as the original method. This means that the number and type of arguments, as well as the return type, must be the same.

Another way to customize the behavior of a class is to call the base class method from the derived class method. To do this, you can use the super function. The super function allows you to call the base class method from the derived class method, and can be useful when you want to extend the behavior of the base class method, rather than replace it.

For example, consider the following base class:

```
class Shape:
    def area(self):
        print("Calculating area...")
```

In this base class, the area method prints a message indicating that the area is being calculated. If you want to create a derived class that represents a circle, and you also want to print a message indicating the type of shape, you can use the super function to call the base class method, and add your own message:

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

    def area(self):
        print("Calculating area of a circle...")
        super().area()
        return 3.14 * self.radius * self.radius
```

In this example, the Circle class overrides the area method, and calls the base class method using the super function. This allows you to extend the behavior of the base class method, while still maintaining its original behavior.

In conclusion, method overriding is a powerful feature in Python that allows you to customize the behavior of a class based on its specific needs. By using method overriding, you can create more robust and reliable code, and ensure that your classes behave in the way that you need them to. Additionally, by using the super function, you can extend the behavior of a base class method, rather than replace it, giving you even greater flexibility and control over the behavior of your classes.

Operator Overloading in Python:

Operator Overloading is a feature in Python that allows developers to redefine the behavior of mathematical and comparison operators for custom data types.

This means that you can use the standard mathematical operators (+, -, *, /, etc.) and comparison operators (>, <, ==, etc.) in your own classes, just as you would for built-in data types like int, float, and str.

```python
class Vector:
  def __init__(self, i, j, k):
    self.i = i
    self.j = j
    self.k = k

  def __str__(self):
    return f"{self.i}i + {self.j}j + {self.k}k"

  def __add__(self, x):
    return Vector(self.i + x.i,  self.j+x.j, self.k+x.k)
v1 = Vector(3, 5, 6)
print(v1)

v2 = Vector(1, 2, 9)
print(v2)

print(v1 + v2)
print(type(v1 + v2))
```

```
3i + 5j + 6k
1i + 2j + 9k
4i + 7j + 15k
<class '__main__.Vector'>
```

Why do we need operator overloading?

Operator overloading allows you to create more readable and intuitive code. For instance, consider a custom class that represents a point in 2D space. You could define a method called 'add' to add two points together, but using the + operator makes the code more concise and readable:

```python
p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2
print(p3.x, p3.y) # prints 4, 6
```

How to overload an operator in Python?

You can overload an operator in Python by defining special methods in your class. These methods are identified by their names, which start and end with double underscores (__). Here are some of the most commonly overloaded operators and their corresponding special methods:

```
+ : __add__
- : __sub__
* : __mul__
/ : __truediv__
< : __lt__
> : __gt__
== : __eq__
```

Walrus operator:-

The main purpose of the walrus operator is to assign values to variables as part of an expression. In simpler terms, it allows you to both assign a value to a variable and use that variable within the same expression.

The Walrus Operator is represented by the `:=` syntax and can be used in a variety of contexts including while loops and if statements.

Why Use the Walrus Operator?

Using the walrus operator can make your code more concise and readable, especially in situations where you need to both assign a value and use it within the same expression. It can help avoid repetitive code and make complex expressions easier to understand.

When Should You Use It?

You can use the walrus operator in various scenarios, such as:

- Checking the length of a string or list and performing an action based on that length.
- Assigning the result of a function call to a variable and using it immediately.
- Simplifying code that involves complex expressions or conditions.

```
# walrus operator :=

# new to Python 3.8
# assignment expression aka walrus operator
# assigns values to variables as part of a larger expression

# happy = True
# print(happy)

# print(happy := True)

# foods = list()
# while True:
#   food = input("What food do you like?: ")
#       if food == "quit":
#           break
#   foods.append(food)

foods = list()
while (food := input("What food do you like?: ")) != "quit":
    foods.append(food)
```

Shutil:-

The `shutil` (short for "shell utilities") module in Python provides a higher-level interface for file operations that are more convenient than the low-level operations provided by the `os` module.

It allows you to perform various file operations such as copying, moving, renaming, and deleting files and directories.

:- Import shutil

```
# Create a zip archive
shutil.make_archive('archive', 'zip', 'directory_to_archive')

# Extract a zip archive
shutil.unpack_archive('archive.zip', 'extracted_directory')
```

- `shutil.copy(src, dst)` : This function copies the file located at src to a new location specified by dst. If the destination location already exists, the original file will be overwritten.
- `shutil.copy2(src, dst)` : This function is similar to shutil.copy, but it also preserves more metadata about the original file, such as the timestamp.
- `shutil.copytree(src, dst)` : This function recursively copies the directory located at src to a new location specified by dst. If the destination location already exists, the original directory will be merged with it.
- `shutil.move(src, dst)` : This function moves the file located at src to a new location specified by dst. This function is equivalent to renaming a file in most cases.
- `shutil.rmtree(path)` : This function recursively deletes the directory located at path, along with all of its contents. This function is similar to using the rm -rf command in a shell.

REQUESTS

The `requests` module in Python allows you to send HTTP requests to web servers and retr
from them. It simplifies the process of making web requests and handling the responses.

GET: Retrieves data from the server without modifying anything on the server.
Example:

```
response = requests.get('https://api.example.com/data')
```

POST: Submits data to be processed by the server. Often used for creating new resources.
Example:

```
response = requests.post('https://api.example.com/create', data={'key': 'value'})
```

PUT: Updates an existing resource on the server with the provided data.
Example:

```
response = requests.put('https://api.example.com/update/123', data={'key': 'new_value'})
```

DELETE: Removes a resource from the server.
Example:

```
response = requests.delete('https://api.example.com/delete/123')
```

```
import json

import requests

response = requests.get('https://jsonplaceholder.typicode.com/posts')
print(response.status_code)  # 200
print(json.dumps(response.json(), indent=5))  # List of posts in JSON format
```

```
import json
import requests
response = requests.get('https://jsonplaceholder.typicode.com/posts')
print(response.status_code)  # 200
posts = response.json()
print(json.dumps(posts, indent=4))  # List of posts in pretty JSON format

# Save the JSON response to a file
with open('posts.json', 'w') as json_file:
    json.dump(posts, json_file, indent=4)
```

GENERATORS:-

Generators in Python are a way to create iterators, which are objects that can be iterated over, like lists or tuples.

However, unlike lists, generators do not store all the values in memory at once; instead, they generate the values on the fly, as they are needed.

This makes them memory efficient, especially when dealing with large datasets.

In Python, you can create a generator by using the `yield` statement in a function. The `yield` statement returns a value from the generator and suspends the execution of the function until the next value is requested. Here's an example:

Comparing these three approaches, here's a summary:

- **Lists** store all elements in memory at once, making them suitable for a finite collection of data.
- **Tuples** behave similarly to lists but are immutable, meaning their values cannot be changed after creation.
- **Generators** produce values on-the-fly using the `yield` keyword, making them memory-efficient, especially for large datasets or infinite sequences. They're suitable for situations where you don't need to store all values in memory simultaneously.

API vs SDK

API (Application Programming Interface):

***What it does:** An API specifies how software components should interact. It defines the methods and data formats that applications can use to request and exchange information with each other.

***Example:** Think of an API like a menu at a restaurant. The menu lists the dishes available, along with descriptions and prices. Customers (applications) can choose items from the menu (make requests), and the kitchen (server or service) prepares and serves the requested dishes.

SDK (Software Development Kit):

- **What it does:** An SDK is a package of tools, libraries, and documentation that developers use to build applications for a specific platform or service. It provides everything needed to develop software, including pre-built components and resources.

- **Example:** Imagine you're building a model car. The SDK is like a complete kit that includes all the necessary parts, tools, and instructions. It provides the chassis, wheels, engine, and other components, along with detailed assembly instructions. You can use these resources to build your car without having to create every part from scratch.

Key Difference:

- **API:** Specifies how software components communicate and interact. It's like a menu that defines what services are available and how to access them.
- **SDK:** Provides developers with tools and resources to build applications for a specific platform or service. It's like a complete kit that includes everything needed to develop software, from pre-built components to documentation.

In essence, an API tells you what you can do, while an SDK gives you the tools to do it.

APIs are like interfaces to interact with existing software or services, while SDKs provide the tools and resources needed to create new software applications or integrate with existing ones.

So, APIs allow you to access functionality that's already built,

while SDKs help you build something new.

FUNCTION CACHING

Function caching in Python is a technique used to store the results of expensive function calls and reuse them when the same inputs occur again.

This can significantly speed up your programs, especially when the function is called multiple times with the same arguments.

In Python, function caching can be achieved using the functools.lru_cache decorator. The functools.lru_cache decorator is used to cache the results of a function so that you can reuse the results instead of recomputing them every time the function is called.
Here's an example:

```
import functools

@functools.lru_cache(maxsize=None)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

print(fib(20))
# Output: 6765
```

```
from functools import lru_cache
import time
@lru_cache(maxsize=100)
 # The maxsize parameter specifies the number of results to cache.

def expensive_function(n):
    time.sleep(2)  # Simulating a time-consuming calculation
    return n * n

# Without caching
start = time.time()
print(expensive_function(4))  # First call, takes time
print("Time taken:", time.time() - start)

start = time.time()
print(expensive_function(4))  # Second call, should be instant
print("Time taken:", time.time() - start)
```

REGULAR EXPRESSION


Regular expressions (regex or `re`) are patterns used to match character combinations in strings.

In Python, the `re` module provides functions to work with regular expressions. Regular expressions are extremely useful for parsing, searching, and manipulating strings.

`re.match()`

- **Description:** Determines if the regex matches at the start of the string.

`re.search()`

- **Description:** Searches the string for the first location where the regex pattern produces a match.

`re.findall()`

- **Description:** Returns a list of all non-overlapping matches of the pattern in the string.

`re.sub()`

- **Description:** Replaces the matches in the string with a replacement string.

`re.finditer()`

- **Description:** Returns an iterator yielding match objects for all non-overlapping matches of the pattern in the string.

```python
import re

data = "Contact us at support@example.com or call 123-456-7890. Visit
https://example.com for more info."

# Find all email addresses
emails = re.findall(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b',
data)
print("Emails:", emails)

# Find all phone numbers
phones = re.findall(r'\d{3}-\d{3}-\d{4}', data)
print("Phone Numbers:", phones)

# Replace URLs with a placeholder
cleaned data = re.sub(r'https?://\S+', '[URL]', data)
print("Cleaned Data:", cleaned_data)
```

```
Emails: ['support@example.com']
Phone Numbers: ['123-456-7890']
Cleaned Data: Contact us at support@example.com or call 123-456-7890. Visit [URL] for more info.

Process finished with exit code 0
```

Async IO in Python

Asynchronous I/O, or async for short, is a programming pattern that allows for high-performance I/O operations in a concurrent and non-blocking manner. In Python, async programming is achieved through the use of the `asyncio` module and asynchronous functions.

python Copy code

```python
async def function_name(arguments):
    # Asynchronous operations
    result = await async_operation()
    return result
```

Here's what happens:

- `await async_operation()`: This line pauses the execution of the `function_name` function.
- It waits until the `async_operation()` coroutine (which is an asynchronous function) completes its task.
- Once `async_operation()` finishes, `function_name()` resumes execution and assigns the result to `result`.

So, to answer your question directly: `await` will pause the execution of `function_name()` until `async_operation()` completes its task.

```
import asyncio
async def fetch_data(url):
print(f"Fetching data from {url}...")
response = await  some_async_function(url)

return response

async def main():
 tasks = [
fetch_data("https://example.com/resource1"),

fetch_data("https://example.com/resource2"),

fetch_data("https://example.com/resource3"),
 ]

results = await asyncio.gather(*tasks)
print("All tasks completed:", results)
# Run the async main function
asyncio.run(main())
```

Async IO in Python:

Uses:

- **Concurrency:** Handles multiple tasks concurrently within a single thread.
- **Improved Performance:** Enhances efficiency for I/O-bound operations.
- **Scalability:** Scales well for applications with many simultaneous connections.
- **Simplicity:** Uses a single-threaded event loop model for managing concurrency.

Advantages:

- **Non-Blocking I/O:** Allows applications to handle other tasks while waiting for I/O operations.
- **Improved Responsiveness:** Ensures applications remain responsive to user interactions.
- **Resource Efficiency:** Manages many connections/tasks with low resource overhead.

Disadvantages:

- **Complexity:** Async code can be more complex to write and debug.
- **Debugging Challenges:** Requires careful handling of concurrency issues.
- **Limited Use Cases:** Best suited for I/O-bound tasks, not CPU-bound operations.

In summary, async IO in Python offers efficient concurrency management for I/O-bound applications, improving performance and scalability while requiring careful handling of complexity and debugging challenges.

The `await` statement in Python is used within `async` functions to pause the execution of the function until the awaited coroutine.

It allows asynchronous functions to wait for results from concurrent tasks without blocking the entire program, enabling efficient use of resources and improved responsiveness in I/O-bound applications.

MULTI THREADING

1. **What is Multithreading?**

   - Multithreading lets your computer do multiple things at once, like playing with different toys at the same time.
   - In Python, threads are like little workers in your computer that can handle different tasks, but they all share the same space (like your playroom) and sometimes need to take turns.

2. **Creating Threads:**

   - You can create threads using a special module called `threading`.
   - Each thread can do a different task, like counting numbers or drawing pictures, all happening together.

3. **When is Multithreading Useful?**

   - It's great for tasks where you're waiting for something to happen, like downloading files from the internet or reading from a file.
   - Threads can keep doing other things while they wait, making your program seem faster.

```
import threading
import time

def count_numbers():
    for i in range(1, 6):
        print(f"Number: {i}")
        time.sleep(1)  # Wait for 1 second between each number

def draw_letters():
    for char in 'ABCDE':
        print(f"Letter: {char}")
        time.sleep(1)  # Wait for 1 second between each letter

if __name__ == "__main__":
    thread1 = threading.Thread(target=count_numbers)
    thread2 = threading.Thread(target=draw_letters)

    thread1.start()  # Start counting numbers
    thread2.start()  # Start drawing letters

    thread1.join()  # Wait for counting to finish
    thread2.join()  # Wait for drawing to finish

    print("All done!")
```

Technical Explanation:

Definition:

Multithreading is the concurrent execution of multiple threads within a single process. Each thread runs independently, performing different tasks or parts of the same task simultaneously.

How it Works:

- **Thread Creation:** You create threads using the `threading.Thread` class.
- **Concurrency:** Threads run concurrently, sharing the process's resources, such as memory and file handles.
- **Global Interpreter Lock (GIL):** In CPython, the Global Interpreter Lock (GIL) prevents multiple native threads from executing Python bytecodes simultaneously. This means that while threads can be useful for I/O-bound tasks, they don't provide a performance boost for CPU-bound tasks in CPython.

Advantages:

1. **Improved I/O Operations:** Multithreading is beneficial for I/O-bound tasks, such as reading/writing files or network operations.
2. **Resource Sharing:** Threads share the same memory space, making data sharing between them efficient.
3. **Responsive Applications:** In GUI applications, multithreading can keep the interface responsive while performing background tasks.

Disadvantages:

1. **GIL Limitation:** The GIL in CPython limits the performance improvement for CPU-bound tasks.
2. **Complexity:** Managing multiple threads can be complex, leading to potential issues like race conditions and deadlocks.
3. **Context Switching Overhead:** Switching between threads can introduce overhead, impacting performance.

MULTIPROCESSING

Multiprocessing in Easy Language

Definition: Multiprocessing in Python refers to the ability to run multiple processes simultaneously, where each process runs independently and can execute its own tasks.

How It Works:

1. **Processes vs. Threads**:
   - **Threads** (as discussed earlier) are like workers sharing the same resources (like memory) and executing tasks concurrently within the same program.
   - **Processes**, on the other hand, are more like separate programs running independently on your computer. They have their own memory space and run completely separate from each other.
2. **Creating Processes**:
   - In Python, you can create processes using the `multiprocessing` module.
   - Each process can perform tasks concurrently, utilizing different CPU cores if available, which can lead to better performance for certain types of tasks.
3. **Use Cases**:
   - **CPU-bound Tasks**: Tasks that require a lot of processing power, such as mathematical calculations, benefit from multiprocessing. Each process can utilize a separate CPU core, speeding up overall computation.
   - **Independent Tasks**: Processes are useful when tasks don't need to share a lot of data or interact closely with each other. They operate independently and communicate via more structured mechanisms like queues or shared memory.

```
import multiprocessing
```

Now, to use multiprocessing we need to create a `process object` which calls a `start()` method. The start() method runs the process and then to stop the execution, we use the `join()` method. Here's how we can create a simple process.

Creating a process

```
import multiprocessing
def my_func():
  print("Hello from process", multiprocessing.current_process().name)
  process = multiprocessing.Process(target=my_func)
  process.start()
  process.join()
```

Functions

The following are some of the most commonly used functions in the multiprocessing module:

- `multiprocessing.Process(target, args)` : This function creates a new process that runs the target function with the specified arguments.
- `multiprocessing.Pool(processes)` : This function creates a pool of worker processes that can be used to parallelize the execution of a function across multiple input values.
- `multiprocessing.Queue()` : This function creates a queue that can be used to communicate data between processes.
- `multiprocessing.Lock()` : This function creates a lock that can be used to synchronize access to shared resources between processes.