Complete Python texted notes

What is Programming :-

- ***Programming is a way for us to tell computers what to do.***
- ***Computer is a very dumb machine and it only does what we tell it to do.***
- ***Hence we learn programming and tell computers to do what we are very slow at - computation.***
- ***If I ask you to calculate 5+6, you will immediately say 11.***
- ***How about 23453453 X 56456?***
- ***You will start searching for a calculator or jump to a new tab to calculate the same.***

What is Python?

- ***Python is a dynamically typed, general purpose programming language that supports an object-oriented programming approach as well as a functional programming approach.***
- ***Python is an interpreted and a high-level programming language.***
- ***It was created by Guido Van Rossum in 1989.***

Features of Python

- ***Python is simple and easy to understand.***

- ***It is Interpreted and platform-independent which makes debugging very easy.***
- ***Python is an open-source programming language.***
- ***Python provides very big library support. Some of the popular libraries include NumPy, Tensorflow, Selenium, OpenCV, etc.***
- ***It is possible to integrate other programming languages within python.***

What is Python used for

- ***Python is used in Data Visualization to create plots and graphical representations.***
- ***Python helps in Data Analytics to analyze and understand raw data for insights and trends.***
- ***It is used in AI and Machine Learning to simulate human behavior and to learn from past data without hard coding.***
- ***It is used to create web applications.***
- ***It can be used to handle databases.***
- ***It is used in business and accounting to perform complex mathematical operations along with quantitative and qualitative analysis.***

Why Replit?

- ***Replit is very easy to share tutorials and code.***

- ***You can easily fork this repl and continue learning in your own style. Video, code as well as text tutorial on the same page which makes things easy!***
- ***For fellow teachers out there, you create a .tutorial folder to create tutorials using replit.***

Day 2 - My Python Success Story:-

Why I love python (And you will too...)
Welcome to Day 2 of 100 days of code.
Let me start with a story! Back in my college, I used to learn C and C++ programming in depth, used to score good marks.
I created a bunch of printing, conditionals and loop program.
Now what?
I wanted to benefit from the same In my second year of college,
I started working (I mean actually working in the industry) with the python programming language.
I was not so good with it but I used to write code for a singaporean client and actually make good money without having to actually master Python.
Harry then got curious and started working on his Python skills even more.
I then got into web scraping and trust me I made some good easy money on Fiverr just by writing some python programs and charging on per webpage basis to my clients (I used to automate scraping)

I then learnt flask and got to work with Flask with a university professor abroad. Long story short, Python made a huge impact in my career.

What can Python do for you?

I want to show you some python programs I created which will surely inspire you to create your own versions of the same as we progress through this tutorial.
Do not try to recreate them just yet if you are a beginner and just started working on Python.
We will make progress gradually trust me

Day 3 - 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:

1. ***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.***
2. ***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***
- ***pip install pandas***

Using a module in Python (Usage):-

- ***We use the import syntax to import a module in Python. Here is an example code:***
- ***import panda***
- ***# Read and work with a file named 'words.csv'***

df = pandas.read_csv('words.csv')

print(df) # This will display first few rows from the words.csv file

Similarly we can install other modules and look into their documentations for usage instructions.
We will find ourselved doing this often in the later part of this course

Day 4 - Our First Program

Today we will write our first ever python program from scratch. It will consist of a bunch of print statements. print can be used to print something on the console in python

Quick Quiz

Write a program to print a poem in Python. Choose the poem of your choice and publish your repl

print("---Your poem here---")

Please make sure you attempt this. Might be easy for some of you but please finish each and every task

Day 5 - Comments, Escape sequence & Print in Python

Welcome to Day 5 of 100DaysOfCode. Today we will talk about Comments, Escape Sequences and little bit more about print statement in Python. We will also throw some light on Escape Sequences

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.

Example 1

#This is a 'Single-Line Comment'

print("This is a print statement.")

Output:

This is a print statement.

Example 2

```
print("Hello World !!!") #Printing Hello World
```

Output:

```
Hello World !!!
```

Example 3:

```
print("Python Program")
#print("Python Program")
```

Output:

```
Python Program
```

Multi-Line Comments:

To write multi-line comments you can use '#' at each line or you can use the multiline string.

Example 1: The use of '#'.

```
#It will execute a block of code if a specified condition is true.
#If the condition is false then it will execute another block of code.
p = 7
if (p > 5):
   print("p is greater than 5.")
else:
   print("p is not greater than 5.")
```

Output:

```
p is greater than 5.
```

Example 2: The use of multiline string.

```
"""This is an if-else statement.
It will execute a block of code if a specified condition is true.
If the condition is false then it will execute another block of code."""
```

```
p = 7

if (p > 5):

   print("p is greater than 5.")

else:

   print("p is not greater than 5.")
```

Output

p is greater than 5.

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:

```
print("This doesnt "execute")

print("This will \" execute")
```

More on Print statement

The syntax of a print statement looks something like this:

```
print(object(s), sep=separator, end=end, file=file, flush=flush)
```

Other Parameters of Print Statement

1. *object(s): Any object, and as many as you like. Will be converted to string before printed*
2. *sep='separator': Specify how to separate the objects, if there is more than one. Default is ' '*
3. *end='end': Specify what to print at the end. Default is '\n' (line feed)*
4. *file: An object with a write method. Default is sys.stdout*

Parameters 2 to 4 are optional

Day 6 - 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
```

These are four variables of different data types.

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))
```

By default, python provides the following built-in data types:

1. Numeric data: int, float, complex

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

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}
```

Operators

Python has different types of operators for different operations. To create a calculator we require arithmetic operators.

Arithmetic operators

Operator	*Operator Name*	*Example*
+	*Addition*	*15+7*
-	*Subtraction*	*15-7*
***	*Multiplication*	*5*7*
***	*Exponential*	*5**3*
/	*Division*	*5/3*
%	*Modulus*	*15%7*
//	*Floor Division*	*15//7*

Exercise

```
n = 15

m = 7

ans1 = n+m

print("Addition of",n,"and",m,"is", ans1)

ans2 = n-m

print("Subtraction of",n,"and",m,"is", ans2)

ans3 = n*m

print("Multiplication of",n,"and",m,"is", ans3)

ans4 = n/m

print("Division of",n,"and",m,"is", ans4)

ans5 = n%m

print("Modulus of",n,"and",m,"is", ans5)
```

ans6 = n//m

print("Floor Division of",n,"and",m,"is", ans6)

Explaination

Here 'n' and 'm' are two variables in which the integer value is being stored. Variables 'ans1' , 'ans2' ,'ans3', 'ans4','ans5' and 'ans6' contains the outputs corresponding to addition, subtraction,multiplication, division, modulus and floor division respectively.

Exercise 1 - Create a Calculator

Create a calculator capable of performing addition, subtraction, multiplication and division operations on two numbers. Your program should format the output in a readable manner!

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).*

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'>

Day 10 - Taking User Input in python

In python, we can take user input directly by using input() function.This input function gives a return value as string/character hence we have to pass that into a variable

Syntax:

variable=input()

But input function returns the value as string. Hence we have to typecast them whenever required to another datatype.

Example:

variable=int(input())

variable=float(input())

We can also display a text using input function. This will make input() function take user input and display a message as well

Example:

a=input("Enter the name: ")

print(a)

Output:

Enter the name: Harry

Harry

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
```

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".')
```

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)
```

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.

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.

Slicing Example:

pie = "ApplePie"

print(pie[:5]) #Slicing from Start

print(pie[5:]) #Slicing till End

print(pie[2:6]) #Slicing in between

print(pie[-8:]) #Slicing using negative index # first write the negative index as positive index and then do same slicling

Output:

Apple

Pie

pleP

ApplePie

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 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.

Example:

str1 = "AbcDEfghIJ"

print(str1.upper())

Output:

ABCDEFGHIJ

lower()

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

Example:

str1 = "AbcDEfghIJ"

print(str1.lower())

Output:

abcdefghij

strip() :

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

Example:

str2 = " Silver Spoon "

print(str2.strip)

Output:

Silver Spoon

rstrip() :

the rstrip() removes any trailing characters. Example:

str3 = "Hello !!!"

print(str3.rstrip("!"))

Output:

Hello

replace() :

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

str2 = "Silver Spoon"

print(str2.replace("Sp", "M"))

Output:

Silver Moon

split() :

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

Example:

str2 = "Silver Spoon"

print(str2.split(" ")) #Splits the string at the whitespace " ".

Output:

['Silver', 'Spoon']

There are various other string methods that we can use to modify our strings.

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.

Example:

str1 = "hello"

capStr1 = str1.capitalize()

print(capStr1)

str2 = "hello WorlD"

capStr2 = str2.capitalize()

print(capStr2)

Output:

Hello

Hello world

center() :

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

Example:

```
str1 = "Welcome to the Console!!!"

print(str1.center(50))
```

Output:

```
        Welcome to the Console!!!
```

We can also provide padding character. It will fill the rest of the fill characters provided by the user.

Example:

```
str1 = "Welcome to the Console!!!"

print(str1.center(50, "."))
```

Output:

```
.....Welcome to the Console!!!.....
```

count() :

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

Example:

```
str2 = "Abracadabra"

countStr = str2.count("a")

print(countStr)
```

Output:

4

endswith() :

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

Example :

str1 = "Welcome to the Console !!!"

print(str1.endswith("!!!"))

Output:

True

We can even also check for a value in-between the string by providing start and end index positions.

Example:

str1 = "Welcome to the Console !!!"

print(str1.endswith("to", 4, 10))

Output:

True

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.

Example:

str1 = "He's name is Dan. He is an honest man."

print(str1.find("is"))

Output:

10

As we can see, this method is somewhat similar to the index() method. The major difference being that index() raises an exception if value is absent whereas find() does not.

Example:

str1 = "He's name is Dan. He is an honest man."

print(str1.find("Daniel"))

Output:

-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.

Example:

str1 = "He's name is Dan. Dan is an honest man."

print(str1.index("Dan"))

Output:

13

As we can see, this method is somewhat similar to the find() method. The major difference being that index() raises an exception if value is absent whereas find() does not.

Example:

str1 = "He's name is Dan. Dan is an honest man."

print(str1.index("Daniel"))

Output:

ValueError: substring not found

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.

Example 1:

str1 = "WelcomeToTheConsole"

print(str1.isalnum())

Output:

True

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.

Example :

str1 = "Welcome"

print(str1.isalpha())

Output:

True

islower() :

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

Example:

str1 = "hello world"

print(str1.islower())

Output:

True

isprintable() :

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

Example :

str1 = "We wish you a Merry Christmas"

print(str1.isprintable())

Output:

True

isspace() :

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

Example:

```
str1 = "       "      #using Spacebar

print(str1.isspace())

str2 = "       "      #using Tab

print(str2.isspace())
```

Output:

True

True

istitle() :

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

Example:

```
str1 = "World Health Organization"

print(str1.istitle())
```

Output:

True

Example:

```
str2 = "To kill a Mocking bird"

print(str2.istitle())
```

Output:

False

isupper() :

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

Example :

```
str1 = "WORLD HEALTH ORGANIZATION"

print(str1.isupper())
```

Output:

True

startswith() :

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

Example :

```
str1 = "Python is a Interpreted Language"

print(str1.startswith("Python"))
```

Output:

True

swapcase() :

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

Example:

```
str1 = "Python is a Interpreted Language"

print(str1.swapcase())
```

Output:

pYTHON IS A iNTERPRETED lANGUAGE

title() :

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

Example:

```
str1 = "He's name is Dan. Dan is an honest man."

print(str1.title())
```

Output:

He'S Name Is Dan. Dan Is An Honest Man.

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.*

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.

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.

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.

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
```

Excersice 2: Good Morning Sir

Create a python program capable of greeting you with Good Morning, Good Afternoon and Good Evening. Your program should use time module to get the current hour. Here is a sample program and documentation link for you:

```
import time

timestamp = time.strftime('%H:%M:%S')

print(timestamp)

timestamp = time.strftime('%H')

print(timestamp)

timestamp = time.strftime('%M')

print(timestamp)
```

timestamp = time.strftime('%S')

print(timestamp)

 # https://docs.python.org/3/library/time.html#time.strftime

Match Case Statements

To implement switch-case like characteristics very similar to if-else functionality, we use a match case in python. If you are coming from a C, C++ or Java like language, you must have heard of switch-case statements. If this is your first language, dont worry as I will tell you everything you need to know about match case statements in this video!

A 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*

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.

Syntax:

match variable_name:

    case 'pattern1' : //statement1

    case 'pattern2' : //statement2

    ...

    case 'pattern n' : //statement n

Example:

x = 4

```
# x is the variable to match
match x:
    # if x is 0
    case 0:
        print("x is zero")
    # case with if-condition
    case 4 if x % 2 == 0:
        print("x % 2 == 0 and case is 4")
    # Empty case with if-condition
    case _ if x < 10:
        print("x is < 10")
    # default case(will only be matched if the above cases were not matched)
    # so it is basically just an else:
    case _:
        print(x)
```

Output:

```
x % 2 == 0 and case is 4
```

🎯 Pro Tip (Before You Learn Loops):

When you're not using loops, use a "guard clause" like this:

```
python
CopyEdit
if not (0 <= a <= 100):
    print("Invalid input")
else:
    # rest of the logic here
```

This prevents bad inputs from leaking into your main logic 💪

✅ 1. Use abs() function (BEST WAY)

python

CopyEdit

result = abs(a - b)

- *abs() means absolute value*
- *Example:*
  abs(5 - 8) → abs(-3) → 3

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:

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

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.

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

Continue Statement

The continue statement skips the rest of the loop statements and causes the next iteration to occur.

example

```
for i in [2,3,4,6,8,0]:
    if (i%2!=0):
        continue
    print(i)
```

output

2

4

6

8

0

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.

Example:

```
def name(fname, mname = "Jhon", lname = "Whatson"):
    print("Hello,", fname, mname, lname)
name("Amy")
```

Output:

Hello, Amy Jhon Whatson

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.

Example:

```
def name(fname, mname, lname):
    print("Hello,", fname, mname, lname)
name(mname = "Peter", lname = "Wesker", fname = "Jade")
```

Output:

Hello, Jade Peter Wesker

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.

Example 1: when number of arguments passed does not match to the actual function definition.

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

Output:

name("Peter", "Quill")

TypeError: name() missing 1 required positional argument: 'lname'

Example 2: when number of arguments passed matches to the actual function definition.

```
def name(fname, mname, lname):
    print("Hello,", fname, mname, lname)
name("Peter", "Ego", "Quill")
```

Output:

Hello, Peter Ego Quill

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.*

Example:

```
def name(*name):
   print("Hello,", name[0], name[1], name[2])
name("James", "Buchanan", "Barnes")
```

Output:

Hello, James Buchanan Barnes

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.*

Example:

```
def name(**name):
   print("Hello,", name["fname"], name["mname"], name["lname"])
name(mname = "Buchanan", lname = "Barnes", fname = "James")
```

Output:

Hello, James Buchanan Barnes

return Statement

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

Example:

```
def name(fname, mname, lname):
   return "Hello, " + fname + " " + mname + " " + lname
```

print(name("James", "Buchanan", "Barnes"))

Output:

Hello, James Buchanan Barnes

Python Lists

- *Lists are ordered collection of data items.*
- *They store multiple items in a single variable.*
- *List items are separated by commas and enclosed within square brackets [].*
- *Lists are changeable meaning we can alter them after creation.*

Example 1:

lst1 = [1,2,2,3,5,4,6]

lst2 = ["Red", "Green", "Blue"]

print(lst1)

print(lst2)

Output:

[1, 2, 2, 3, 5, 4, 6]

['Red', 'Green', 'Blue']

Example 2:

details = ["Abhijeet", 18, "FYBScIT", 9.8]

print(details)

Output:

['Abhijeet', 18, 'FYBScIT', 9.8]

As we can see, a single list can contain items of different data types.

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.

Example:

colors = ["Red", "Green", "Blue", "Yellow", "Green"]

 # [0] [1] [2] [3] [4]

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.

Example:

colors = ["Red", "Green", "Blue", "Yellow", "Green"]

 # [0] [1] [2] [3] [4]

print(colors[2])

print(colors[4])

print(colors[0])

Output:

Blue

Green

Red

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.

Example:

colors = ["Red", "Green", "Blue", "Yellow", "Green"]

 # [-5] [-4] [-3] [-2] [-1]

print(colors[-1])

print(colors[-3])

print(colors[-5])

Output:

Green

Blue

Red

Check whether an item in present in the list?

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

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]

if "Yellow" in colors:

  print("Yellow is present.")

else:

  print("Yellow is absent.")
```

Output:

Yellow is present.

```
colors = ["Red", "Green", "Blue", "Yellow", "Green"]

if "Orange" in colors:

  print("Orange is present.")

else:

  print("Orange is absent.")
```

Output:

Orange is absent.

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.

Syntax:

listName[start : end : jumpIndex]

Note: jump Index is optional. We will see this in later examples.

Example: printing elements within a particular range:

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow"]

print(animals[3:7]) #using positive indexes

print(animals[-7:-2])        #using negative indexes'
```

Output:

```
['mouse', 'pig', 'horse', 'donkey']

['bat', 'mouse', 'pig', 'horse', 'donkey']
```

Here, we provide index of the element from where we want to start and the index of the element till which we want to print the values.

Note: The element of the end index provided will not be included.

Example: printing all element from a given index till the end

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow"]

print(animals[4:])   #using positive indexes

print(animals[-4:])  #using negative indexes
```

Output:

```
['pig', 'horse', 'donkey', 'goat', 'cow']

['horse', 'donkey', 'goat', 'cow']
```

When no end index is provided, the interpreter prints all the values till the end.

Example: printing all elements from start to a given index

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow"]

print(animals[:6])   #using positive indexes

print(animals[:-3])  #using negative indexes
```

Output:

```
['cat', 'dog', 'bat', 'mouse', 'pig', 'horse']

['cat', 'dog', 'bat', 'mouse', 'pig', 'horse']
```

When no start index is provided, the interpreter prints all the values from start up to the end index provided.

Example: Printing alternate values

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow"]

print(animals[::2])          #using positive indexes

print(animals[-8:-1:2])      #using negative indexes
```

Output:

```
['cat', 'bat', 'pig', 'donkey', 'cow']

['dog', 'mouse', 'horse', 'goat']
```

Here, we have not provided start and index, which means all the values will be considered. But as we have provided a jump index of 2 only alternate values will be printed.

Example: printing every 3rd consecutive value withing a given range

```
animals = ["cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow"]

print(animals[1:8:3])
```

Output:

```
['dog', 'pig', 'goat
```

Here, jump index is 3. Hence it prints every 3rd element within given index.

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

Example 1:

colors = ["voilet", "indigo", "blue", "green"]

colors.sort()

print(colors)

num = [4,2,5,3,6,1,2,1,2,8,9,7]

num.sort()

print(num)

Output:

['blue', 'green', 'indigo', 'voilet']

[1, 1, 2, 2, 2, 3, 4, 5, 6, 7, 8, 9]

What if you want to print the list in descending order?
We must give reverse=True as a parameter in the sort method.

Example:

```
colors = ["voilet", "indigo", "blue", "green"]

colors.sort(reverse=True)

print(colors)

num = [4,2,5,3,6,1,2,1,2,8,9,7]

num.sort(reverse=True)

print(num)
```

Output:

```
['voilet', 'indigo', 'green', 'blue']

[9, 8, 7, 6, 5, 4, 3, 2, 2, 2, 1, 1]
```

The reverse parameter is set to False by default.

Note: Do not mistake the reverse parameter with the reverse method.

reverse()

This method reverses the order of the list.

Example:

```
colors = ["voilet", "indigo", "blue", "green"]

colors.reverse()

print(colors)

num = [4,2,5,3,6,1,2,1,2,8,9,7]

num.reverse()

print(num)
```

Output:

```
['green', 'blue', 'indigo', 'voilet']

[7, 9, 8, 2, 1, 2, 1, 6, 3, 5, 2, 4]
```

index()

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

Example:

```
colors = ["voilet", "green", "indigo", "blue", "green"]

print(colors.index("green"))

num = [4,2,5,3,6,1,2,1,3,2,8,9,7]

print(num.index(3))
```

Output:

```
1

3
```

count()

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

Example:

```
colors = ["voilet", "green", "indigo", "blue", "green"]

print(colors.count("green"))

num = [4,2,5,3,6,1,2,1,3,2,8,9,7]
```

Output:

```
2

3
```

copy()

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

Example:

```
colors = ["voilet", "green", "indigo", "blue"]

newlist = colors.copy()

print(colors)

print(newlist)
```

Output:

['voilet', 'green', 'indigo', 'blue']

['voilet', 'green', 'indigo', 'blue']

append():

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

Example:

```
colors = ["voilet", "indigo", "blue"]

colors.append("green")

print(colors)
```

Output:

['voilet', 'indigo', 'blue', 'green']

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.

Example:

```
colors = ["voilet", "indigo", "blue"]

#         [0]      [1]     [2]

colors.insert(1, "green")   #inserts item at index 1

# updated list: colors = ["voilet", "green", "indigo", "blue"]

#      indexs           [0]      [1]      [2]     [3]

print(colors)
```

Output:

['voilet', 'green', 'indigo', 'blue']

extend():

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

Example 1:

```
#add a list to a list
```

colors = ["voilet", "indigo", "blue"]

rainbow = ["green", "yellow", "orange", "red"]

colors.extend(rainbow)

print(colors)

Output:

['voilet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red']

Concatenating two lists:

You can simply concatenate two lists to join two lists.

Example:

colors = ["voilet", "indigo", "blue", "green"]

colors2 = ["yellow", "orange", "red"]

print(colors + colors2)

Output:

['voilet', 'indigo', 'blue', 'green', 'yellow', 'orange', 'red']

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.

Example 1:

tuple1 = (1,2,2,3,5,4,6)

tuple2 = ("Red", "Green", "Blue")

print(tuple1)

print(tuple2)

Output:

(1, 2, 2, 3, 5, 4, 6)

('Red', 'Green', 'Blue')

Example 2:

```
details = ("Abhijeet", 18, "FYBScIT", 9.8)

print(details)
```

Output:

```
('Abhijeet', 18, 'FYBScIT', 9.8)
```

Tuple Indexes

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

Example:

```
country = ("Spain", "Italy", "India",)

#          [0]     [1]     [2]
```

Accessing tuple items:

I. Positive Indexing:

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

Example:

```
country = ("Spain", "Italy", "India",)

#          [0]     [1]     [2]

print(country[0])

print(country[1])

print(country[2])
```

Output:

```
Spain

Italy

India
```

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.

Example:

```
country = ("Spain", "Italy", "India", "England", "Germany")

#          [0]     [1]     [2]      [3]       [4]

print(country[-1]) # Similar to print(country[len(country) - 1])

print(country[-3])

print(country[-4])
```

Output:

```
Germany

India

Italy
```

III. Check for item:

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

Example 1:

```
country = ("Spain", "Italy", "India", "England", "Germany")

if "Germany" in country:

   print("Germany is present.")

else:

   print("Germany is absent.")
```

Output:

```
Germany is present.
```

Example 2:

```
country = ("Spain", "Italy", "India", "England", "Germany")

if "Russia" in country:
```

```
    print("Russia is present.")
else:
    print("Russia is absent.")
```

Output:

Russia is absent.

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.

Syntax:

Tuple[start : end : jumpIndex]

Note: jump Index is optional. We will see this in given examples.

Example: Printing elements within a particular range:

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow")
print(animals[3:7])     #using positive indexes
print(animals[-7:-2])   #using negative indexes
```

Output:

('mouse', 'pig', 'horse', 'donkey')

('bat', 'mouse', 'pig', 'horse', 'donkey')

Here, we provide index of the element from where we want to start and the index of the element till which we want to print the values. Note: The element of the end index provided will not be included.

Example: Printing all element from a given index till the end

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow")
print(animals[4:])     #using positive indexes
print(animals[-4:])    #using negative indexes
```

Output:

('pig', 'horse', 'donkey', 'goat', 'cow')

('horse', 'donkey', 'goat', 'cow')

When no end index is provided, the interpreter prints all the values till the end.

Example: printing all elements from start to a given index

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow")

print(animals[:6])     #using positive indexes

print(animals[:-3])    #using negative indexes
```

Output:

('cat', 'dog', 'bat', 'mouse', 'pig', 'horse')

('cat', 'dog', 'bat', 'mouse', 'pig', 'horse')

When no start index is provided, the interpreter prints all the values from start up to the end index provided.

Example: Print alternate values

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow")

print(animals[::2])    #using positive indexes

print(animals[-8:-1:2]) #using negative indexes
```

Output:

('cat', 'bat', 'pig', 'donkey', 'cow')

('dog', 'mouse', 'horse', 'goat')

Here, we have not provided start and end index, which means all the values will be considered. But as we have provided a jump index of 2 only alternate values will be printed.

Example: printing every 3rd consecutive withing given range

```
animals = ("cat", "dog", "bat", "mouse", "pig", "horse", "donkey", "goat", "cow")

print(animals[1:8:3])
```

Output:

('dog', 'pig', 'goat')

Here, jump index is 3. Hence it prints every 3rd element within given index.

Manipulating Tuples

Tuples are immutable, hence if you want to add, remove or change tuple items, then first you must convert the tuple to a list. Then perform operation on that list and convert it back to tuple.

Example:

```
countries = ("Spain", "Italy", "India", "England", "Germany")

temp = list(countries)

temp.append("Russia")      #add item

temp.pop(3)              #remove item

temp[2] = "Finland"       #change item

countries = tuple(temp)

print(countries)
```

Output:

```
('Spain', 'Italy', 'Finland', 'Germany', 'Russia')
```

Thus, we convert the tuple to a list, manipulate items of the list using list methods, then convert list back to a tuple.

However, we can directly concatenate two tuples without converting them to list.

Example:

```
countries = ("Pakistan", "Afghanistan", "Bangladesh", "ShriLanka")

countries2 = ("Vietnam", "India", "China")

southEastAsia = countries + countries2

print(southEastAsia)
```

Output:

```
('Pakistan', 'Afghanistan', 'Bangladesh', 'ShriLanka', 'Vietnam', 'India', 'China')
```

Tuple methods

As tuple is immutable type of collection of elements it have limited built in methods.They are explained below

count() Method

The count() method of Tuple returns the number of times the given element appears in the tuple.

Syntax:

tuple.count(element)

Example

Tuple1 = (0, 1, 2, 3, 2, 3, 1, 3, 2)

res = Tuple1.count(3)

print('Count of 3 in Tuple1 is:', res)

Output

3

index() method

The Index() method returns the first occurrence of the given element from the tuple.

Syntax:

tuple.index(element, start, end)

Note: This method raises a ValueError if the element is not found in the tuple.

Example

Tuple = (0, 1, 2, 3, 2, 3, 1, 3, 2)

res = Tuple.index(3)

print('First occurrence of 3 is', res)

Output

3

Excersice 2: Good Morning Sir

Create a python program capable of greeting you with Good Morning, Good Afternoon and Good Evening. Your program should use time module to get the current hour. Here is a sample program and documentation link for you:

```
import time

timestamp = time.strftime('%H:%M:%S')

print(timestamp)

timestamp = time.strftime('%H')

print(timestamp)

timestamp = time.strftime('%M')

print(timestamp)

timestamp = time.strftime('%S')

print(timestamp)

# https://docs.python.org/3/library/time.html#time.strftime
```

Create a program capable of displaying questions to the user like KBC. Use List data type to store the questions and their correct answers. Display the final amount the person is taking home after playing the game.

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.")
```

Output:

Hello, My name is Tushar and I'm 23 years old.

In the above code, we have used the f-string to format the string. It evaluates at runtime; we can put all valid Python expressions in them.

We can use it in a single statement as well.

Example

```
print(f"{2 * 30})"
```

Output:

60

Docstrings in python

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

Example

```
def square(n):

   '''Takes in a number n, returns the square of n'''

   print(n**2)

square(5)
```

Here,

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

Output:

25

Here is another example:

def add(num1, num2):

"""

Add up two integer numbers.

This function simply wraps the ``+`` operator, and does not

do anything interesting, except for illustrating what

the docstring of a very simple function looks like.

Parameters

----

num1 : int

First number to add.

num2 : int

Second number to add.

Returns

----

int

The sum of ``num1`` and ``num2``.

See Also

----

subtract : Subtract one integer from another.

Examples

```
>>> add(2, 2)
4
>>> add(25, 0)
25
>>> add(10, -10)
0
"""
return num1 + num2
```

Python Comments vs Docstrings

Python Comments

Comments are descriptions that help programmers better understand the intent and functionality of the program. They are completely ignored by the Python interpreter.

Python docstrings

As mentioned above, Python docstrings are strings used right after the definition of a function, method, class, or module (like in Example 1). They are used to document our code.

We can access these docstrings using the doc attribute.

Python doc attribute

Whenever string literals are present just after the definition of a function, module, class or method, they are associated with the object as their doc attribute. We can later use this attribute to retrieve this docstring.

Example

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

Output:

Takes in a number n, returns the square of n

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.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

*Although never is often better than *right* now.*

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

Easter egg

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.

Example:

```
info = {"Carla", 19, False, 5.9, 19}

print(info)
```

Output:

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

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.

Quick Quiz: Try to create an empty set. Check using the type() function whether the type of your variable is a set

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.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}

cities3 = cities.union(cities2)

print(cities3)
```

Output:

```
{'Tokyo', 'Madrid', 'Kabul', 'Seoul', 'Berlin', 'Delhi'}
```

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}

cities.update(cities2)

print(cities)
```

Output:

```
{'Berlin', 'Madrid', 'Tokyo', 'Delhi', 'Kabul', 'Seoul'}
```

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.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}
```

cities3 = cities.intersection(cities2)

print(cities3)

Output:

{'Madrid', 'Tokyo'}

Example :

cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}

cities.intersection_update(cities2)

print(cities)

Output:

{'Tokyo', 'Madrid'}

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.

Example:

cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}

cities3 = cities.symmetric_difference(cities2)

print(cities3)

Output:

{'Seoul', 'Kabul', 'Berlin', 'Delhi'}

Example:

cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}

cities.symmetric_difference_update(cities2)

print(cities)

Output:

{'Kabul', 'Delhi', 'Berlin', 'Seoul'}

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.

Example:

cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Seoul", "Kabul", "Delhi"}

cities3 = cities.difference(cities2)

print(cities3)

Output:

{'Tokyo', 'Madrid', 'Berlin'}

Example:

cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Seoul", "Kabul", "Delhi"}

print(cities.difference(cities2))

Output:

{'Tokyo', 'Berlin', 'Madrid'}

Set Methods

There are several in-built methods used for the manipulation of set.They are explained below

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.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Tokyo", "Seoul", "Kabul", "Madrid"}

print(cities.isdisjoint(cities2))
```

Output:

False

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.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Seoul", "Kabul"}

print(cities.issuperset(cities2))

cities3 = {"Seoul", "Madrid","Kabul"}

print(cities.issuperset(cities3))
```

Output:

False
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.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Delhi", "Madrid"}

print(cities2.issubset(cities))
```

Output:

True

add()

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

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities.add("Helsinki")

print(cities)
```

Output:

```
{'Tokyo', 'Helsinki', 'Madrid', 'Berlin', 'Delhi'}
```

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.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities2 = {"Helsinki", "Warsaw", "Seoul"}

cities.update(cities2)

print(cities)
```

Output:

```
{'Seoul', 'Berlin', 'Delhi', 'Tokyo', 'Warsaw', 'Helsinki', 'Madrid'}
```

remove()/discard()

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

Example :

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities.remove("Tokyo")

print(cities)
```

Output:

```
{'Delhi', 'Berlin', 'Madrid'}
```

The main difference between remove and discard is that, if we try to delete an item which is not present in set, then remove() raises an error, whereas discard() does not raise any error.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities.remove("Seoul")

print(cities)
```

Output:

```
KeyError: 'Seoul'
```

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.

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

item = cities.pop()

print(cities)

print(item)
```

Output:

```
{'Tokyo', 'Delhi', 'Berlin'} Madrid
```

del

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

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

del cities

print(cities)
```

Output:

NameError: name 'cities' is not defined We get an error because our entire set has been deleted and there is no variable called cities which contains a set.

What if we don't want to delete the entire set, we just want to delete all items within that set?

clear():

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

Example:

```
cities = {"Tokyo", "Madrid", "Berlin", "Delhi"}

cities.clear()

print(cities)
```

Output:

```
set()
```

Check if item exists

You can also check if an item exists in the set or not.

Example

```
info = {"Carla", 19, False, 5.9}

if "Carla" in info:
   print("Carla is present.")
else:
   print("Carla is absent.")
```

Output:

```
Carla is present.
```

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}

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

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])

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'])

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)])

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.

Example:

info = {'name':'Karan', 'age':19, 'eligible':True}

print(info)

info.update({'age':20})

info.update({'DOB':2001})

print(info)

Output:

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

{'name': 'Karan', 'age': 20, 'eligible': True, 'DOB': 2001}

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.

Example:

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

Output:

{}

pop():

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

Example:

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

Output:

{'name': 'Karan', 'age': 19}

popitem():

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

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True, 'DOB':2003}
info.popitem()
print(info)
```

Output:

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

del:

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

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True, 'DOB':2003}

del info['age']

print(info)
```

Output:

{'name': 'Karan', 'eligible': True, 'DOB': 2003}

If key is not provided, then the del keyword will delete the dictionary entirely.

Example:

```
info = {'name':'Karan', 'age':19, 'eligible':True, 'DOB':2003}

del info

print(info)
```

Output:

NameError: name 'info' is not defined

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.

Syntax

```
for counter in sequence:
    #Statements inside for loop block
```

```
else:
   #Statements inside else block
```

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.

Exceptions in Python

Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).

When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled. If not handled, the program will crash.

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.

Syntax:

```
try:
    #statements which could generate
    #exception
except:
    #Soloution of generated exception
```

Example:

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

Output:

Enter an integer: 6.022

Number entered is not an integer.

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.

Syntax:

```
try:
  #statements which could generate
```

```
  #exception
except:
  #solution of generated exception
finally:
   #block of code which is going to
   #execute in any situation
```

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.

Raising Custom errors

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

```
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 built-in exceptions in Python and why it is important to handle exceptions. However, sometimes we may need to create our own custom exceptions that serve our purpose.

Defining Custom Exceptions

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

Here's the syntax to define custom exceptions:

```
class CustomError(Exception):

 # code ...

 pass

try:

 # code ...

except CustomError:

 # code...
```

This is useful because sometimes we might want to do something when a particular exception is raised. For example, sending an error report to the admin, calling an api, etc.

Create a program capable of displaying questions to the user like KBC. Use List data type to store the questions and their correct answers. Display the final amount the person is taking home after playing the game.

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:

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

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.

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))

This will output a list of all the names defined in the math module, including functions like sqrt and pi, as well as other variables and constants.

*In summary, the import statement in Python allows you to access the functions and variables defined in a module from within your current script. You can import the entire module, specific functions or variables, or use the * wildcard to import everything. You can also use the as keyword to rename a module, and the dir function to view the contents of a module.*

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:

```
def main():
    # Code to be run when the script is run directly
    print("Running script directly")
if __name__ == "__main__":
    main()
```

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.

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.

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)

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")

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.

Write a python program to translate a message into secret code language. Use the rules below to translate normal English into secret code language

Coding:

if the word contains atleast 3 characters, remove the first letter and append it at the end now append three random characters at the starting and the end else: simply reverse the string

Decoding:

if the word contains less than 3 characters, reverse it else: remove 3 random characters from start and end. Now remove the last letter and append it to the beginning

Your program should ask whether you want to code or decode

st = input("Enter message")

words = st.split(" ")

coding = input("1 for Coding or 0 for Decoding")

coding = True if (coding=="1") else False

print(coding)

```
if(coding):
  nwords = []
  for word in words:
    if(len(word)>=3):
      r1 = "dsf"
      r2 = "jkr"
      stnew = r1+ word[1:] + word[0] + r2
      nwords.append(stnew)
    else:
      nwords.append(word[::-1])
  print(" ".join(nwords))
else:
  nwords = []
  for word in words:
    if(len(word)>=3):
      stnew = word[3:-3]
      stnew = stnew[-1] + stnew[:-1]
      nwords.append(stnew)
    else:
      nwords.append(word[::-1])
  print(" ".join(nwords))
```

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.

I hope this tutorial has helped clarify the differences between local and global variables and how to use the global keyword in Python. Thank you for watching!

Python provides several ways to manipulate files. Today, we will discuss how to handle files in Python.

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).*

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)
```

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!')
```

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
```

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.

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.

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.

Lambda functions are often used in conjunction with higher-order functions, such as map, filter, and reduce which we will look into later.

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))
```

*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.

I hope this helps clarify the difference between is and == in Python!

Snake, Water and Gun is a variation of the children's game "rock-paper-scissors" where players use hand gestures to represent a snake, water, or a gun. The gun beats the snake, the water beats the gun, and the snake beats the water. Write a python program to create a Snake Water Gun game in Python using if-else statements. Do not create any fancy GUI. Use proper functions to check for win.

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

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:

```
obj1 = Details()
```

Now we can print values:

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.

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***

We will see the explaination and example of each type of inheritance in the later tutorials

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

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.

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.

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

Snake, Water and Gun is a variation of the children's game "rock-paper-scissors" where players use hand gestures to represent a snake, water, or a gun. The gun beats the snake, the water beats the gun, and the snake beats the water. Write a python program to create a Snake Water Gun game in Python using if-else statements. Do not create any fancy GUI. Use proper functions to check for win.

Write a Library class with no_of_books and books as two instance variables. Write a program to create a library from this Library class and show how you can print all books, add a book and get the number of books using different methods. Show that your program doesnt persist the books after the program is stopped!

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.

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.

```
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
```

In the example above, 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.

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.

Packager files

Write a program to clear the clutter inside a folder on your computer. You should use os module to rename all the png images from 1.png all the way till n.png where n is the number of png files in that folder. Do the same for other file formats. For example:

- *sfdsf.png --> 1.png*
- *vfsf.png --> 2.png*
- *this.png --> 3.png*
- *design.png --> 4.png*
- *name.png --> 5.png*

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.

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.

Write a program to clear the clutter inside a folder on your computer. You should use os module to rename all the png images from 1.png all the way till n.png where n is the number of png files in that folder. Do the same for other file formats. For example:

- *sfdsf.png --> 1.png*
- *vfsf.png --> 2.png*
- *this.png --> 3.png*
- *design.png --> 4.png*
- *name.png --> 5.png*

Write a program to manipulate pdf files using pyPDF. Your programs should be able to merge multiple pdf files into a single pdf. You are welcome to add more functionalities

pypdf is a free and open-source pure-python PDF library capable of splitting, merging, cropping, and transforming the pages of PDF files. It can also add custom data, viewing options, and passwords to PDF files. pypdf can retrieve text and metadata from PDFs as well.

Operator Overloading in Python: An Introduction

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.*

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:

```
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__

For example, if you want to overload the + operator to add two instances of a custom class, you would define the add method:

```
class Point:
    def __init__(self, x, y):
        self.x = x
```

```
        self.y = y

    def __add__(self, other):

        return Point(self.x + other.x, self.y + other.y)
```

It's important to note that operator overloading is not limited to the built-in operators, you can overload any user-defined operator as well.

Conclusion

Operator overloading is a powerful feature in Python that allows you to create more readable and intuitive code. By redefining the behavior of mathematical and comparison operators for custom data types, you can write code that is both concise and expressive. However, it's important to use operator overloading wisely, as overloading the wrong operator or using it inappropriately can lead to confusing or unexpected behavior.

Single Inheritance in Python

Single inheritance is a type of inheritance where a class inherits properties and behaviors from a single parent class. This is the simplest and most common form of inheritance.

Syntax

The syntax for single inheritance in Python is straightforward and easy to understand. To create a new class that inherits from a parent class, simply specify the parent class in the class definition, inside the parentheses, like this:

```
class ChildClass(ParentClass):

    # class body
```

Example

Let's consider a simple example of single inheritance in Python. Consider a class named "Animal" that contains the attributes and behaviors that are common to all animals.

```
class Animal:

    def __init__(self, name, species):

        self.name = name
```

```
        self.species = species


    def make_sound(self):
        print("Sound made by the animal")
```

If we want to create a new class for a specific type of animal, such as a dog, we can create a new class named "Dog" that inherits from the Animal class.

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


    def make_sound(self):
        print("Bark!")
```

The Dog class inherits all the attributes and behaviors of the Animal class, including the __init__ method and the make_sound method. Additionally, the Dog class has its own __init__ method that adds a new attribute for the breed of the dog, and it also overrides the make_sound method to specify the sound that a dog makes.

Single inheritance is a powerful tool in Python that allows you to create new classes based on existing classes. It allows you to reuse code, extend it to fit your needs, and make it easier to manage complex systems. Understanding single inheritance is an important step in becoming proficient in object-oriented programming in Python.

Multiple Inheritance in Python

Multiple inheritance is a powerful feature in object-oriented programming that allows a class to inherit attributes and methods from multiple parent classes. This can be useful in situations where a class needs to inherit functionality from multiple sources.

Syntax

In Python, multiple inheritance is implemented by specifying multiple parent classes in the class definition, separated by commas.

```
class ChildClass(ParentClass1, ParentClass2, ParentClass3):
    # class body
```

In this example, the ChildClass inherits attributes and methods from all three parent classes: ParentClass1, ParentClass2, and ParentClass3.

It's important to note that, in case of multiple inheritance, Python follows a method resolution order (MRO) to resolve conflicts between methods or attributes from different parent classes. The MRO determines the order in which parent classes are searched for attributes and methods.

Example

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

    def make_sound(self):
        print("Sound made by the animal")

class Mammal:
    def __init__(self, name, fur_color):
        self.name = name
        self.fur_color = fur_color

class Dog(Animal, Mammal):
    def __init__(self, name, breed, fur_color):
        Animal.__init__(self, name, species="Dog")
```

```
        Mammal.__init__(self, name, fur_color)
        self.breed = breed

    def make_sound(self):
        print("Bark!")
```

In this example, the Dog class inherits from both the Animal and Mammal classes, so it can use attributes and methods from both parent classes.

Multilevel Inheritance in Python

Multilevel inheritance is a type of inheritance in object-oriented programming where a derived class inherits from another derived class. This type of inheritance allows you to build a hierarchy of classes where one class builds upon another, leading to a more specialized class.

In Python, multilevel inheritance is achieved by using the class hierarchy. The syntax for multilevel inheritance is quite simple and follows the same syntax as single inheritance.

Syntax

```
class BaseClass:
    # Base class code

class DerivedClass1(BaseClass):
    # Derived class 1 code

class DerivedClass2(DerivedClass1):
    # Derived class 2 code
```

In the above example, we have three classes: BaseClass, DerivedClass1, and DerivedClass2. The DerivedClass1 class inherits from the BaseClass, and the DerivedClass2 class inherits from the DerivedClass1 class. This creates a

hierarchy where DerivedClass2 has access to all the attributes and methods of both DerivedClass1 and BaseClass.

Example

Let's take a look at an example to understand how multilevel inheritance works in Python. Consider the following classes:

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

    def show_details(self):
        print(f"Name: {self.name}")
        print(f"Species: {self.species}")

class Dog(Animal):
    def __init__(self, name, breed):
        Animal.__init__(self, name, species="Dog")
        self.breed = breed

    def show_details(self):
        Animal.show_details(self)
        print(f"Breed: {self.breed}")

class GoldenRetriever(Dog):
    def __init__(self, name, color):
        Dog.__init__(self, name, breed="Golden Retriever")
        self.color = color
```

```
    def show_details(self):
        Dog.show_details(self)
        print(f"Color: {self.color}")
```

In this example, we have three classes: Animal, Dog, and GoldenRetriever. The Dog class inherits from the Animal class, and the GoldenRetriever class inherits from the Dog class.

Now, when we create an object of the GoldenRetriever class, it has access to all the attributes and methods of the Animal class and the Dog class. We can also see that the GoldenRetriever class has its own attributes and methods that are specific to the class.

```
dog = GoldenRetriever("Max", "Golden")
dog.show_details()
```

Output:

```
Name: Max
Species: Dog
Breed: Golden Retriever
Color: Golden
```

As we can see from the output, the GoldenRetriever object has access to all the attributes and methods of the Animal and Dog classes, and, it has also added its own unique attributes and methods. This is a powerful feature of multilevel inheritance, as it allows you to create more complex and intricate classes by building upon existing ones.

Another important aspect of multilevel inheritance is that it allows you to reuse code and avoid repeating the same logic multiple times. This can lead to better maintainability and readability of your code, as you can abstract away complex logic into base classes and build upon them.

In conclusion, multilevel inheritance is a powerful feature in object-oriented programming that allows you to create complex and intricate classes by building upon existing ones. It provides the benefits of code reuse, maintainability, and readability, while also requiring careful consideration to avoid potential problems.

Hybrid Inheritance in Python

Hybrid inheritance is a combination of multiple inheritance and single inheritance in object-oriented programming. It is a type of inheritance in which multiple inheritance is used to inherit the properties of multiple base classes into a single derived class, and single inheritance is used to inherit the properties of the derived class into a sub-derived class.

In Python, hybrid inheritance can be implemented by creating a class hierarchy, in which a base class is inherited by multiple derived classes, and one of the derived classes is further inherited by a sub-derived class.

Syntax

The syntax for implementing Hybrid Inheritance in Python is the same as for implementing Single Inheritance, Multiple Inheritance, or Hierarchical Inheritance.

Here's the syntax for defining a hybrid inheritance class hierarchy:

```
class BaseClass1:
  # attributes and methods
class BaseClass2:
  # attributes and methods
class DerivedClass(BaseClass1, BaseClass2):
  # attributes and methods
```

Example

Consider the example of a Student class that inherits from the Person class, which in turn inherits from the Human class. The Student class also has a Program class that it is associated with.

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

```
  def show_details(self):
    print("Name:", self.name)
    print("Age:", self.age)

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

  def show_details(self):
    Human.show_details(self)
    print("Address:", self.address)

class Program:
  def __init__(self, program_name, duration):
    self.program_name = program_name
    self.duration = duration

  def show_details(self):
    print("Program Name:", self.program_name)
    print("Duration:", self.duration)

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

```
def show_details(self):
    Person.show_details(self)
    self.program.show_details()
```

In this example, the Student class inherits from the Person class, which in turn inherits from the Human class. The Student class also has an association with the Program class. This is an example of Hybrid Inheritance in action, as it uses both Single Inheritance and Association to achieve the desired inheritance structure.

To create a Student object, we can do the following:

```
program = Program("Computer Science", 4)
student = Student("John Doe", 25, "123 Main St.", program)
student.show_details()
```

Output

```
Name: John Doe
Age: 25
Address: 123 Main St.
Program Name: Computer Science
Duration: 4
```

As we can see from the output, the Student object has access to all the attributes and methods of the Person and Human classes, as well as the Program class through association.

In this way, hybrid inheritance allows for a flexible and powerful way to inherit attributes and behaviors from multiple classes in a hierarchy or chain.

Hierarchical Inheritance

Hierarchical Inheritance is a type of inheritance in Object-Oriented Programming where multiple subclasses inherit from a single base class. In other words, a

single base class acts as a parent class for multiple subclasses. This is a way of establishing relationships between classes in a hierarchical manner.

Here's an example to illustrate the concept of hierarchical inheritance in Python:

```
class Animal:
    def __init__(self, name):
        self.name = name
    def show_details(self):
        print("Name:", self.name)
class Dog(Animal):
    def __init__(self, name, breed):
        Animal.__init__(self, name)
        self.breed = breed
    def show_details(self):
        Animal.show_details(self)
        print("Species: Dog")
        print("Breed:", self.breed)
class Cat(Animal):
    def __init__(self, name, color):
        Animal.__init__(self, name)
        self.color = color
    def show_details(self):
        Animal.show_details(self)
        print("Species: Cat")
        print("Color:", self.color)
```

In the above code, the Animal class acts as the base class for two subclasses, Dog and Cat. The Dog class and the Cat class inherit the attributes

and methods of the Animal class. However, they can also add their own unique attributes and methods.

Here's an example of creating objects of the Dog and Cat classes and accessing their attributes and methods:

dog = Dog("Max", "Golden Retriever")

dog.show_details()

cat = Cat("Luna", "Black")

cat.show_details()

Output:

Name: Max

Species: Dog

Breed: Golden Retriever

Name: Luna

Species: Cat

Color: Black

As we can see from the outputs, the Dog and Cat classes have inherited the attributes and methods of the Animal class, and have also added their own unique attributes and methods.

In conclusion, hierarchical inheritance is a way of establishing relationships between classes in a hierarchical manner. It allows multiple subclasses to inherit from a single base class, which helps in code reuse and organization of code in a more structured manner.

Packager files

Write a program to manipulate pdf files using pyPDF. Your programs should be able to merge multiple pdf files into a single pdf. You are welcome to add more functionalities

pypdf is a free and open-source pure-python PDF library capable of splitting, merging, cropping, and transforming the pages of PDF files. It can also add custom data, viewing options, and passwords to PDF files. pypdf can retrieve text and metadata from PDFs as well.

Write a program to pronounce list of names using win32 API. If you are given a list l as follows:

l = ["Rahul", "Nishant", "Harry"]

Your program should pronouce:

Shoutout to Rahul

Shoutout to Nishant

Shoutout to Harry

Note: If you are not using windows, try to figure out how to do the same thing using some other package

The time Module in Python

The time module in Python provides a set of functions to work with time-related operations, such as timekeeping, formatting, and time conversions. This module is part of the Python Standard Library and is available in all Python installations, making it a convenient and essential tool for a wide range of applications. In this day 84 tutorial, we'll explore the time module in Python and see how it can be used in different scenarios.

time.time()

The time.time() function returns the current time as a floating-point number, representing the number of seconds since the epoch (the point in time when the time module was initialized). The returned value is based on the computer's system clock and is affected by time adjustments made by the operating system, such as daylight saving time. Here's an example:

```
import time

print(time.time())

# Output: 1602299933.233374
```

As you can see, the function returns the current time as a floating-point number, which can be used for various purposes, such as measuring the duration of an operation or the elapsed time since a certain point in time.

time.sleep()

The time.sleep() function suspends the execution of the current thread for a specified number of seconds. This function can be used to pause the program for a certain period of time, allowing other parts of the program to run, or to synchronize the execution of multiple threads. Here's an example:

```
import time

print("Start:", time.time())

time.sleep(2)

print("End:", time.time())

# Output:

# Start: 1602299933.233374

# End: 1602299935.233376
```

As you can see, the function time.sleep() suspends the execution of the program for 2 seconds, allowing other parts of the program to run during that time.

time.strftime()

The time.strftime() function formats a time value as a string, based on a specified format. This function is particularly useful for formatting dates and times in a human-readable format, such as for display in a GUI, a log file, or a report. Here's an example:

```
import time

t = time.localtime()

formatted_time = time.strftime("%Y-%m-%d %H:%M:%S", t)

print(formatted_time)

# Output: 2022-11-08 08:45:33
```

As you can see, the function time.strftime() formats the current time (obtained using time.localtime()) as a string, using a specified format. The format string contains codes that represent different parts of the time value, such as the year, the month, the day, the hour, the minute, and the second.

Conclusion

The time module in Python provides a set of functions to work with time-related operations, such as timekeeping, formatting, and time conversions. Whether you are writing a script, a library, or an application, the time module is a powerful tool that can help you perform time-related tasks with ease and efficiency. So, if you haven't already, be sure to check out the time module in Python and see how it can help you write better, more efficient code.

Creating Command Line Utilities in Python

Command line utilities are programs that can be run from the terminal or command line interface, and they are an essential part of many development workflows. In Python, you can create your own command line utilities using the built-in argparse module.

Syntax

Here is the basic syntax for creating a command line utility using argparse in Python:

```
import argparse

parser = argparse.ArgumentParser()

# Add command line arguments

parser.add_argument("arg1", help="description of argument 1")

parser.add_argument("arg2", help="description of argument 2")

# Parse the arguments

args = parser.parse_args()

# Use the arguments in your code

print(args.arg1)

print(args.arg2)
```

Examples

Here are a few examples to help you get started with creating command line utilities in Python:

Adding optional arguments

The following example shows how to add an optional argument to your command line utility:

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument("-o", "--optional", help="description of optional argument", default="default_value")

args = parser.parse_args()

print(args.optional)
```

Adding positional arguments

The following example shows how to add a positional argument to your command line utility:

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument("positional", help="description of positional argument")

args = parser.parse_args()

print(args.positional)
```

Adding arguments with type

The following example shows how to add an argument with a specified type:

```
import argparse

parser = argparse.ArgumentParser()

parser.add_argument("-n", type=int, help="description of integer argument")

args = parser.parse_args()

print(args.n)
```

Conclusion

Creating command line utilities in Python is a straightforward and flexible process thanks to the argparse module. With a few lines of code, you can create powerful and customizable command line tools that can make your development workflow

easier and more efficient. Whether you're working on small scripts or large applications, the argparse module is a must-have tool for any Python developer.

The Walrus Operator in Python

The Walrus Operator is a new addition to Python 3.8 and allows you to assign a value to a variable within an expression. This can be useful when you need to use a value multiple times in a loop, but don't want to repeat the calculation.

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

Here's an example of how you can use the Walrus Operator in a while loop:

```
numbers = [1, 2, 3, 4, 5]

while (n := len(numbers)) > 0:

    print(numbers.pop())
```

In this example, the length of the numbers list is assigned to the variable n using the Walrus Operator. The value of n is then used in the condition of the while loop, so that the loop will continue to execute until the numbers list is empty.

Another example of using the Walrus Operator in an if statement:

```
names = ["John", "Jane", "Jim"]

if (name := input("Enter a name: ")) in names:

    print(f"Hello, {name}!")

else:

    print("Name not found.")
```

Here is another example

```
# 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)
```

In this example, the user input is assigned to the variable name using the Walrus Operator. The value of name is then used in the if statement to determine whether it is in the names list. If it is, the corresponding message is printed, otherwise, a different message is printed.

It is important to note that the Walrus Operator should be used sparingly as it can make code less readable if overused.

In conclusion, the Walrus Operator is a useful tool for Python developers to have in their toolkit. It can help streamline code and reduce duplication, but it should be used with care to ensure code readability and maintainability.

Shutil Module in Python

Shutil is a Python module that provides a higher level interface for working with file and directories. The name "shutil" is short for shell utility. It provides a convenient and efficient way to automate tasks that are commonly performed on files and directories. In this repl, we'll take a closer look at the shutil module and its various functions and how they can be used in Python.

Importing shutil

The syntax for importing the shutil module is as follows:

```
import shutil
```

Functions

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

- *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.*

Examples

Here are some examples of how you can use the shutil module in your Python code:

```
import shutil

# Copying a file

shutil.copy("src.txt", "dst.txt")

# Copying a directory

shutil.copytree("src_dir", "dst_dir")

# Moving a file

shutil.move("src.txt", "dst.txt")

# Deleting a directory

shutil.rmtree("dir")
```

As you can see, the shutil module provides a simple and efficient way to perform common file and directory-related tasks in Python. Whether you need to copy, move, delete, or preserve metadata about files and directories, the shutil module has you covered.

In conclusion, the shutil module is a powerful tool for automating file and directory-related tasks in Python. Whether you are a beginner or an experienced Python developer, the shutil module is an essential tool to have in your toolbox.

Write a program to pronounce list of names using win32 API. If you are given a list l as follows:

l = ["Rahul", "Nishant", "Harry"]

Your program should pronouce:

Shoutout to Rahul

Shoutout to Nishant

Shoutout to Harry

Note: If you are not using windows, tr

Requests module in python

The Python Requests module is an HTTP library that enables developers to send HTTP requests in Python. This module enables you to send HTTP requests using Python code and makes it possible to interact with APIs and web services.

Installation

pip install requests

Get Request

Once you have installed the Requests module, you can start using it to send HTTP requests. Here is a simple example that sends a GET request to the Google homepage:

```
import requests

response = requests.get("https://www.google.com")

print(response.text)
```

Post Request

Here is another example that sends a POST request to a web service and includes a custom header:

```
import requests

url = "https://api.example.com/login"

headers = {

   "User-Agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/58.0.3029.110 Safari/537.36",

   "Content-Type": "application/json"

}

data = {

   "username": "myusername",

   "password": "mypassword"

}

response = requests.post(url, headers=headers, json=data)

print(response.text)
```

In this example, we send a POST request to a web service to authenticate a user. We include a custom User-Agent header and a JSON payload with the user's credentials.

bs4 Module

There is another module called BeautifulSoup which is used for web scraping in Python. I have personally used bs4 module to finish a lot of freelancing task.

Exercise 10

Use the NewsAPI and the requests module to fetch the daily news related to different topics. Go to: https://newsapi.org/ and explore the various options to build you application

Generators in Python

Generators in Python are special type of functions that allow you to create an iterable sequence of values. A generator function returns a generator object, which can be used to generate the values one-by-one as you iterate over it.

Generators are a powerful tool for working with large or complex data sets, as they allow you to generate the values on-the-fly, rather than having to create and store the entire sequence in memory.

Creating a Generator

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:

```
def my_generator():
    for i in range(5):
        yield i
gen = my_generator()
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
print(next(gen))
# Output:
# 0
# 1
# 2
# 3
# 4
```

As you can see, the generator function my_generator() returns a generator object, which can be used to generate the values in the range 0 to 4. The next() function is used to request the next value from the generator, and the generator resumes its execution until it encounters another yield statement or until it reaches the end of the function.

Using a Generator

Once you have created a generator, you can use it in a variety of ways, such as in a for loop, a list comprehension, or a generator expression. Here's an example:

```
gen = my_generator()

for i in gen:
    print(i)

# Output:
# 0
# 1
# 2
# 3
# 4
```

As you can see, the generator can be used in a for loop, just like any other iterable sequence. The generator is used to generate the values one-by-one as the loop iterates over it.

Benefits of Generators

Generators offer several benefits over other types of sequences, such as lists, tuples, and sets. One of the main benefits of generators is that they allow you to generate the values on-the-fly, rather than having to create and store the entire sequence in memory. This makes generators a powerful tool for working with large or complex data sets, as you can generate the values as you need them, rather than having to store them all in memory at once.

Another benefit of generators is that they are lazy, which means that the values are generated only when they are requested. This allows you to generate the values in a more efficient and memory-friendly manner, as you don't have to generate all the values up front.

Conclusion

Generators in Python are a powerful tool for working with large or complex data sets, allowing you to generate the values on-the-fly and store only what you need in memory. Whether you are working with a large dataset, performing complex calculations, or generating a sequence of values, generators are a must-have tool in your programming toolkit. So, if you haven't already, be sure to check out

generators in Python and see how they can help you write better, more efficient code.

Function Caching in Python

Function caching is a technique for improving the performance of a program by storing the results of a function call so that you can reuse the results instead of recomputing them every time the function is called. This can be particularly useful when a function is computationally expensive, or when the inputs to the function are unlikely to change frequently.

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
```

As you can see, the functools.lru_cache decorator is used to cache the results of the fib function. The maxsize parameter is used to specify the maximum number of results to cache. If maxsize is set to None, the cache will have an unlimited size.

Benefits of Function Caching

Function caching can have a significant impact on the performance of a program, particularly for computationally expensive functions. By caching the results of a function, you can avoid having to recompute the results every time the function is called, which can save a significant amount of time and computational resources.

Another benefit of function caching is that it can simplify the code of a program by removing the need to manually cache the results of a function. With

the functools.lru_cache decorator, the caching is handled automatically, so you can focus on writing the core logic of your program.

Conclusion

Function caching is a technique for improving the performance of a program by storing the results of a function so that you can reuse the results instead of recomputing them every time the function is called. In Python 3, function caching can be achieved using the functools.lru_cache decorator, which provides an easy and efficient way to cache the results of a function. Whether you're writing a computationally expensive program, or just want to simplify your code, function caching is a great technique to have in your toolbox.

#Exercise 10 Use the NewsAPI and the requests module to fetch the daily news related to different topics. Go to: https://newsapi.org/ and explore the various options to build you application

Exercise 11 - Drink Water Reminder

Write a python program which reminds you of drinking water every hour or two. Your program can either beep or send desktop notifications for a specific operating system

Regular Expressions in Python

Regular expressions, or "regex" for short, are a powerful tool for working with strings and text data in Python. They allow you to match and manipulate strings based on patterns, making it easy to perform complex string operations with just a few lines of code.

Metacharacters in regular expressions

[] Represent a character class

^ Matches the beginning

$ Matches the end

. Matches any character except newline

? Matches zero or one occurrence.

| Means OR (Matches with any of the characters

separated by it.

** Any number of occurrences (including 0 occurrences)*

+ One or more occurrences

{} Indicate number of occurrences of a preceding RE

to match.

() Enclose a group of REs

Find list of more meta characters here - https://www.ibm.com/docs/en/rational-clearquest/9.0.1?topic=tags-meta-characters-in-regular-expressions

Importing re Package

In Python, regular expressions are supported by the re module. The basic syntax for working with regular expressions in Python is as follows:

```
import re
```

Searching for a pattern in re using re.search() Method

re.search() method either returns None (if the pattern doesn't match), or a re.MatchObject that contains information about the matching part of the string. This method stops after the first match, so this is best suited for testing a regular expression more than extracting data. We can use re.search method like this to search for a pattern in regular expression:

```
# Define a regular expression pattern
pattern = r"expression"
# Match the pattern against a string
text = "Hello, world!"
match = re.search(pattern, text)
if match:
    print("Match found!")
else:
    print("Match not found.")
```

Searching for a pattern in re using re.findall() Method

You can also use the re.findall function to find all occurrences of the pattern in a string:

```
import re

pattern = r"expression"

text = "The cat is in the hat."

matches = re.findall(pattern, text)

print(matches)

# Output: ['cat', 'hat']
```

Replacing a pattern

The following example shows how to replace a pattern in a string:

```
import re

pattern = r"[a-z]+at"

text = "The cat is in the hat."

matches = re.findall(pattern, text)

print(matches)

# Output: ['cat', 'hat']

new_text = re.sub(pattern, "dog", text)

print(new_text)

# Output: "The dog is in the dog."
```

Extracting information from a string

The following example shows how to extract information from a string using regular expressions:

```
import re

text = "The email address is example@example.com."

pattern = r"\w+@\w+\.\w+"

match = re.search(pattern, text)

if match:
```

```
    email = match.group()
    print(email)
# Output: example@example.com
```

Conclusion

Regular expressions are a powerful tool for working with strings and text data in Python. Whether you're matching patterns, replacing text, or extracting information, regular expressions make it easy to perform complex string operations with just a few lines of code. With a little bit of practice, you'll be able to use regular expressions to solve all sorts of string-related problems in Python.

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.

Syntax

Here is the basic syntax for creating an asynchronous function in Python:

```
import asyncio
async def my_async_function():
    # asynchronous code here
    await asyncio.sleep(1)
    return "Hello, Async World!"
async def main():
    result = await my_async_function()
    print(result)
asyncio.run(main())
```

Another way to schedule tasks concurrently is as follows:

```
L = await asyncio.gather(
```

```
    my_async_function(),
    my_async_function(),
    my_async_function(),
  )
print(L)
```

Async IO is a powerful programming pattern that allows for high-performance and concurrent I/O operations in Python. With the asyncio module and asynchronous functions, you can write efficient and scalable code that can handle large amounts of data and I/O operations without blocking the main thread. Whether you're working on web applications, network services, or data processing pipelines, async IO is an essential tool for any Python developer.

Multithreading in Python

Multithreading is a technique in programming that allows multiple threads of execution to run concurrently within a single process. In Python, we can use the threading module to implement multithreading. In this tutorial, we will take a closer look at the threading module and its various functions and how they can be used in Python.

Importing Threading

We can use threading by importing the threading module.

```
import threading
```

Creating a thread

To create a thread, we need to create a Thread object and then call its start() method. The start() method runs the thread and then to stop the execution, we use the join() method. Here's how we can create a simple thread.

```
import threading
def my_func():
  print("Hello from thread", threading.current_thread().name)
  thread = threading.Thread(target=my_func)
  thread.start()
```

thread.join()

Functions

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

- ***threading.Thread(target, args): This function creates a new thread that runs the target function with the specified arguments.***
- ***threading.Lock(): This function creates a lock that can be used to synchronize access to shared resources between threads.***

Creating multiple threads

Creating multiple threads is a common approach to using multithreading in Python. The idea is to create a pool of worker threads and then assign tasks to them as needed. This allows you to take advantage of multiple CPU cores and process tasks in parallel.

```
import threading

def thread_task(task):
    # Do some work here
    print("Task processed:", task)

if __name__ == '__main__':
    tasks = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    threads = []
    for task in tasks:
        thread = threading.Thread(target=thread_task, args=(task,))
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
```

Using a lock to synchronize access to shared resources

When working with multithreading in python, locks can be used to synchronize access to shared resources among multiple threads. A lock is an object that acts as a semaphore, allowing only one thread at a time to execute a critical section of code. The lock is released when the thread finishes executing the critical section.

```
import threading

def increment(counter, lock):
    for i in range(10000):
        lock.acquire()
        counter += 1
        lock.release()
if __name__ == '__main__':
    counter = 0
    lock = threading.Lock()
    threads = []
    for i in range(2):
        thread = threading.Thread(target=increment, args=(counter, lock))
        threads.append(thread)
        thread.start()
    for thread in threads:
        thread.join()
    print("Counter value:", counter)
```

Conclusion

As you can see, the threading module provides a simple and efficient way to implement multithreading in Python. Whether you need to create a new thread, run a function across multiple input values, or synchronize access to shared resources, the threading module has you covered.

In conclusion, the threading module is a powerful tool for parallelizing code in Python. Whether you are a beginner or an experienced Python developer, the

threading module is an essential tool to have in your toolbox. With multithreading, you can take advantage of multiple CPU cores and significantly improve the performance of your code.

Multiprocessing in Python

Multiprocessing is a Python module that provides a simple way to run multiple processes in parallel. It allows you to take advantage of multiple cores or processors on your system and can significantly improve the performance of your code. In this repl, we'll take a closer look at the multiprocessing module and its various functions and how they can be used in Python.

Importing Multiprocessing

We can use multiprocessing by importing the multiprocessing module.

```
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.***

Creating a pool of worker processes

Creating a pool of worker processes is a common approach to using multiprocessing in Python. The idea is to create a pool of worker processes and then assign tasks to them as needed. This allows you to take advantage of multiple CPU cores and process tasks in parallel.

```
from multiprocessing import Pool

def process_task(task):
    # Do some work here
    print("Task processed:", task)

if __name__ == '__main__':
    tasks = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    with Pool(processes=4) as pool:
        results = pool.map(process_task, tasks)
```

Using a queue to communicate between processes

When working with multiple processes, it is often necessary to pass data between them. One way to do this is by using a queue. A queue is a data structure that allows data to be inserted at one end and removed from the other end. In the context of multiprocessing, a queue can be used to pass data between processes.

```
def producer(queue):
    for i in range(10):
        queue.put(i)
def consumer(queue):
    while True:
        item = queue.get()
```

```
        print(item)
queue = multiprocessing.Queue()
p1 = multiprocessing.Process(target=producer, args=(queue,))
p2 = multiprocessing.Process(target=consumer, args=(queue,))
p1.start()
p2.start()
```

Using a lock to synchronize access to shared resources

When working with multiprocessing in python, locks can be used to synchronize access to shared resources among multiple processes. A lock is an object that acts as a semaphore, allowing only one process at a time to execute a critical section of code. The lock is released when the process finishes executing the critical section.

```
def increment(counter, lock):
    for i in range(10000):
        lock.acquire()
        counter.value += 1
        lock.release()
if __name__ == '__main__':
    counter = multiprocessing.Value('i', 0)
    lock = multiprocessing.Lock()
    p1 = multiprocessing.Process(target=increment, args=(counter, lock))
    p2 = multiprocessing.Process(target=increment, args=(counter, lock))
    p1.start()
    p2.start()
    p1.join()
    p2.join()
    print("Counter value:", counter.value)
```

Conclusion

As you can see, the multiprocessing module provides a simple and efficient way to run multiple processes in parallel. Whether you need to create a new process, run a function across multiple input values, communicate data between processes, or synchronize access to shared resources, the multiprocessing module has you covered.

In conclusion, the multiprocessing module is a powerful tool for parallelizing code in Python. Whether you are a beginner or an experienced Python developer, the multiprocessing module is an essential tool to have in your toolbox.

Exercise 11 - Drink Water Reminder (Solution)

Write a python program which reminds you of drinking water every hour or two. Your program can either beep or send desktop notifications for a specific operating system

Conclusion

Congratulations on completing the 100 days of Python code challenge! You have likely gained a solid foundation in the language and developed a range of skills, from basic syntax to more advanced concepts such as object-oriented programming. However, this is just the beginning of your journey with Python. There are many more topics to explore, including machine learning, web development, game development, and more.

Where to go from here:

To continue your learning journey, consider exploring the following resources:

- *Python books: There are many excellent books on Python that can help you deepen your knowledge and skills. Some popular options include "Python Crash Course" by Eric Matthes, "Automate the Boring Stuff with Python" by Al Sweigart, and "Fluent Python" by Luciano Ramalho. I would also like to recommend "Hands on Machine Learning book by Aurélien Géron"*
- *YouTube Projects: There are many YouTube projects available which can be watched after you have some basic understanding of python*
- *Python communities: There are many online communities where you can connect with other Python learners and experts, ask questions, and share your knowledge. Some popular options include the Python subreddit, the Python Discord server, and the Python community on Stack Overflow.*

- ***GitHub repositories: GitHub is a great resource for finding Python projects, libraries, and code snippets. Some useful repositories to check out include "awesome-python" (a curated list of Python resources), "scikit-learn" (a machine learning library), and "django" (a web development framework).***

Link to some resources

- ***Tkinter - You can learn Tkinter which is used to create GUIs from here :***
- ***Machine Learning - I loved this playlist from Google Developers***
- ***Django - For Django, try the tutorial from the official documentation. Trust me its really good***

Overall, the key to mastering Python (or any programming language) is to keep practicing and experimenting. Set yourself challenges, work on personal projects, and stay curious. Good luck!