PYTHON

BASICS AND ALGORITHMS PROGRAMMING

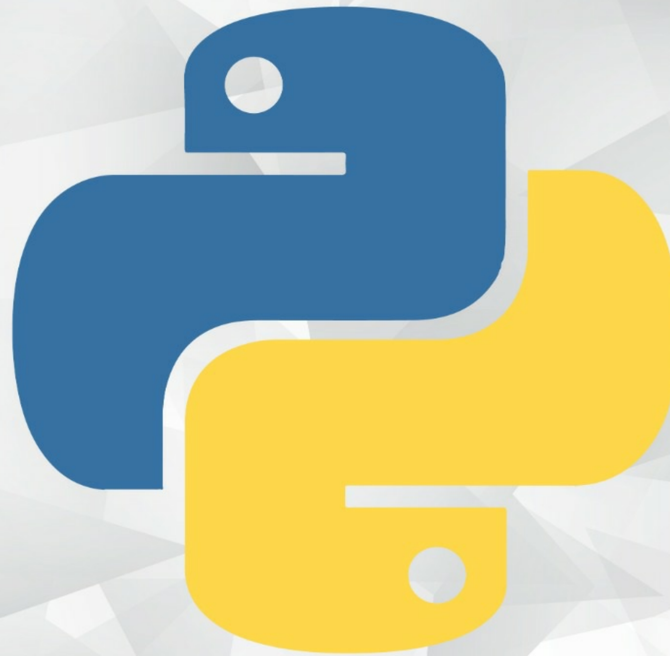
WRITTEN BY
SIDDHARTHA VADLAMUDI

PYTHON

BASICS AND

ALGORITHMS

PROGRAMMING



WRITTEN BY

SIDDHARTHA VADLAMUDI

About the author:

About the technical reviewer

Preface:

We find ourselves deep in the ocean of technological advancements found today, from self-driving cars to Nanotechnologies, social media applications such as Facebook, Snapchat or Instagram or even TikTok, which is commonly used to edit photos or videos according to your desires. If you want to make a website, just use some tools and it will automatically but what are those tools? How are they made? And How are they controlled? are some of the questions which come into our minds which has scaled the entire technological business world to the next level. Knowing these tools is essential if you want to start your own business fortunately there exists a language through which you can talk to computers according to your wishes and desires to create astounding applications such as video games, video editing features and cybersecurity to protect valuable data. To build these exciting applications you need to learn the fundamentals of python programming and the working various algorithms and their computational use parameters such as time and space complexity. Fortunately, this book will take you to a systematic walk-through of these algorithms from the very basics and it will be very instrumental for you and if you want to take some jobs in the field of programming then this book is perfect for your needs.

Contents

Chapter 1 Introduction to Python programming
- Why learn Python?
- Install Python on your system
- Install a coding text editor:

Chapter 2 Learn Variables and various Data Types in Python
- Variables:
- Some common errors with python scripts:
- Strings:
- Concatenating strings:
- Manipulating strings:
- Arithmetic operations:
- Difference between integer and float:
- Receiving inputs:
- Type conversions:
- Comments:
- Some rules of programming to keep in mind are:

Chapter 3 Lists in Python
- What are lists in Python?
- Modifying lists:
- Find the length of list:
- Slicing lists:
- Add and replace elements to list:
- Removing elements in a list:
- Organize lists:

Chapter 4 Working with for Loops and if-else statements
- What is a for Loop?
- Usage of for loops using examples:
- If-else statements in Python:
- and, or operation:

Chapter 5 Functions in python
Functions in python and how they are useful:
Coding examples of functions:
Define a function:
Passing variables to functions:

PART 2: INTRODUCING TO ALGORITHMS IN PYTHON

Chapter 6: Array algorithms in python
Two number sum:
Algorithm explanation:
Time and Space complexity:
Three number sum:
Time and Space complexity:
Smallest difference:
Time and Space complexity:
Validate Subsequence:
Time and Space complexity:
Product Sum:
Time and Space complexity:

Chapter 7: Sorting algorithms in python
Bubble sort:
Time and Space complexity:
Insertion sort:
Time and Space complexity:
Selection sort:
Time and Space complexity:

Chapter 8: Interesting algorithms in python
Depth first search:
Time and Space complexity:
Branch Sum:
Time and Space complexity:
Breadth first search:
Time and Space complexity:
Binary search:

Time and Space complexity:
Nth Fibonacci:
Time and Space complexity:
Palindrome Check:
Time and Space complexity:

Chapter 1

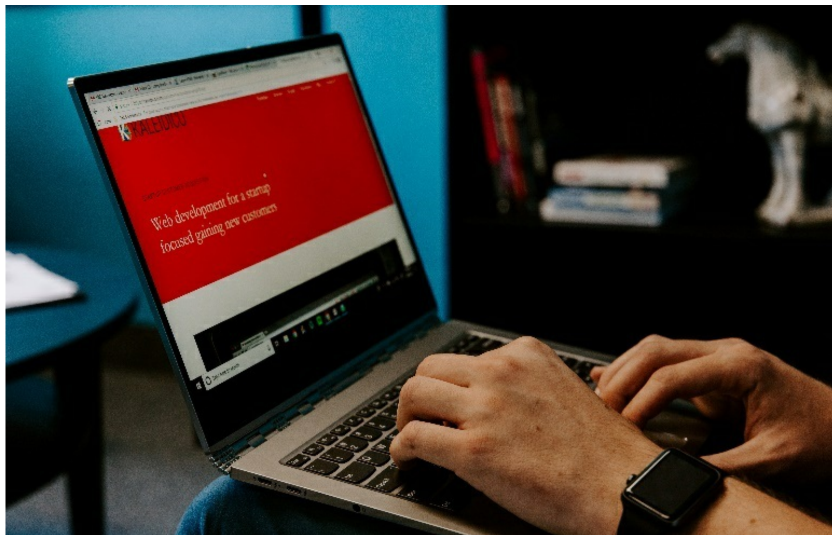
Introduction to Python programming

Every human being is born with a purpose and tries to create applications, which ultimately rewards the creator as well as benefits the society in order to earn fame, social status and of course, wealth, look at the world around you especially the internet world and you will find many applications that are in use in our daily life such as you watch videos on YouTube or you watch your favorite show on Netflix or if you play your favorite games or just want to communicate with your friends on Instagram and Facebook, a person like me would also want to create my very own website you might want that too and I also like editing my photos in a way to create new visual art fortunately everything that you have in mind is possible with the help of a programming language named as Python.

WHY LEARN PYTHON?

Python is the main programming language for many applications you use such as:

- **Website development** is one of the main applications of Python, many websites that you use today such as uploading videos on YouTube or live streaming your favourite show from the internet can be accomplished using Python and if you want to develop your own website with custom features would take only a few lines of code and you'll have a nice website setup with custom applications.

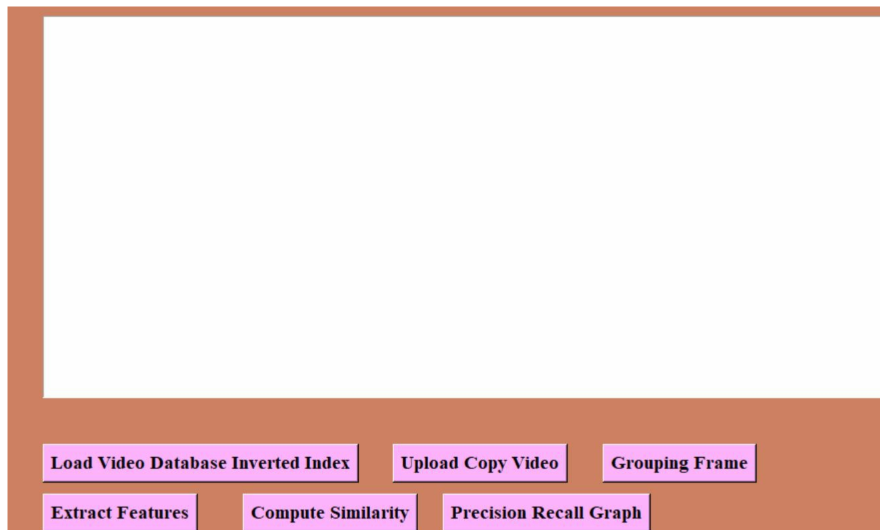


Web development using python

- Python is the backbone for popular **games** such as Battlefield 2, Frets on Fire, World of Tanks, Disney's Toontown Online, Vega Strike, and Civilization-IV. Any game that you have in mind can be made with the help of coding in python.

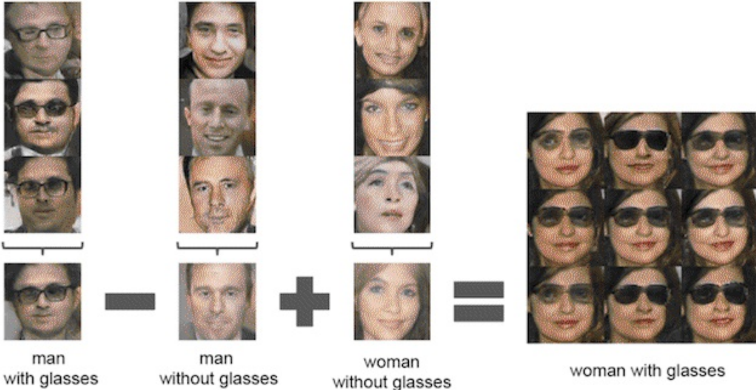
Counter strike game

- Suppose you want to install a new software for your computer such as DirectX then a **Graphical user interface (GUI)** you through the whole process, the GUI is commonly written in python.



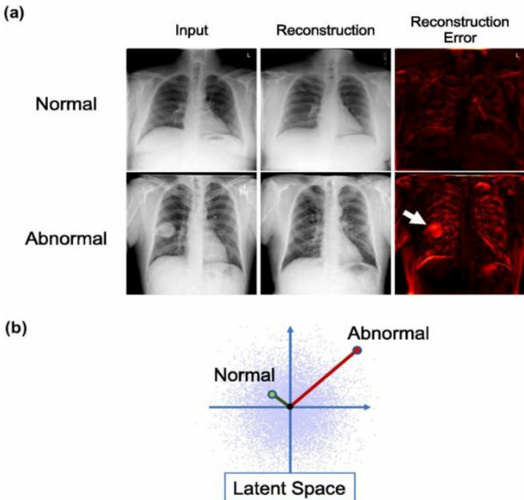
- Image processing and graphic design applications are common applications such as the filters used in Snapchat and Instagram are coded in python, Python's machine learning libraries provides ease of generating artwork used to edit your photos such as perfectly placing a hat on your head would involve the machine learning code in python to examine your facial features to perfectly place a hat on your head similarly python allows new

editable colours to be created with the help of image processing tools coded in python. The photo editing tools that are commonly used such as cropping, resizing and colour filtering are coded in this esteemed programming language.



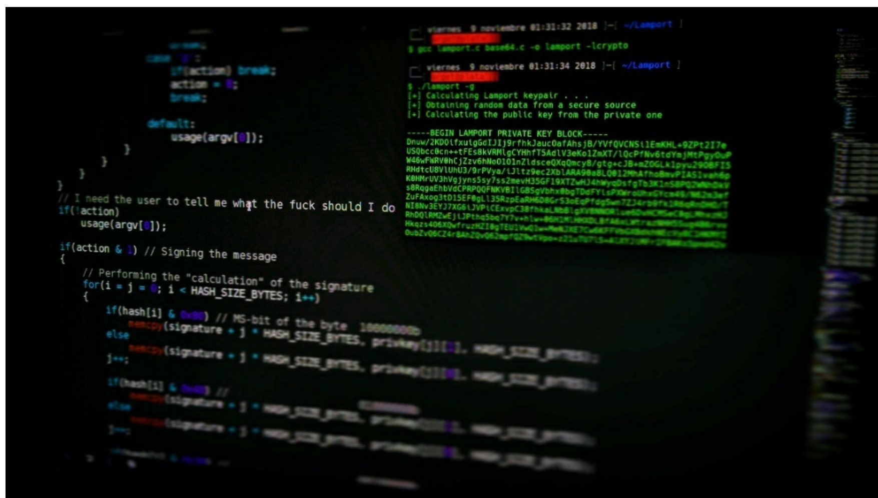
Examples of fake images produced by Python generative adversarial networks

- An interesting application of python is found in medical imaging analysis where python is used to find tumors in data. X-rays are commonly very minute information and sometimes it is even difficult for experts to analyze the things going on behind the hood fortunately python includes Artificial intelligence tools and techniques to find anomalies in data, precisely localize the data and generate the data for visualization.



Localization of tumors in X-rays

- Everything on the internet is not secure so another interesting field to make it secure is a field known as Cybersecurity which basically involves protecting the users data from being stolen so that it cannot be used in harmful ways an example of cyber-attack was on Facebook where more than 500 million Facebook users' details were published online on an underground website used by cyber criminals. Thus it is important for a Python programmer pursuing the field of cyber security to have a solid grasp on the language in order to protect valuable data like this to be exposed online.



Securing a server using python

INSTALL PYTHON ON YOUR SYSTEM

Python is very easy to install, follow the steps in order to install python on windows machine:

- Head to https://www.python.org/downloads/.
- The latest version at the time of writing is python 3.9.4 so download latest version of it in your local machine.

Looking for a specific release?

Python releases by version number:

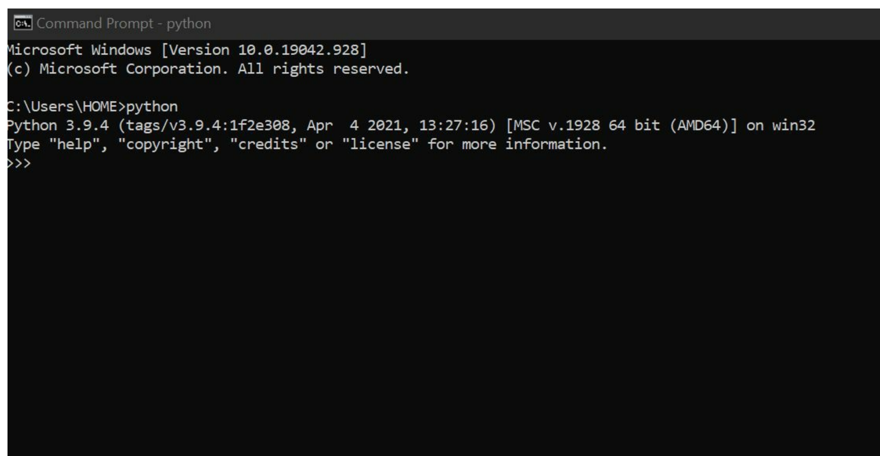
Release version	Release date		Click for more
Python 3.9.4	April 4, 2021	Download	Release Notes
Python 3.8.9	April 2, 2021	Download	Release Notes
Python 3.9.2	Feb. 19, 2021	Download	Release Notes
Python 3.8.8	Feb. 19, 2021	Download	Release Notes
Python 3.6.13	Feb. 15, 2021	Download	Release Notes
Python 3.7.10	Feb. 15, 2021	Download	Release Notes
Python 3.8.7	Dec. 21, 2020	Download	Release Notes

- You will be directed to a page once you open scroll down and hit Windows installer (64-bit) or (32-bit) according to your personal computer.

macOS 64-bit Intel installer	Mac OS X	for macOS 10.9 and later	2b974bfd787f941fb8f80b5b8084e569	29866341	SIG
macOS 64-bit universal2 installer	Mac OS X	for macOS 10.9 and later, including macOS 11 Big Sur on Apple Silicon (experimental)	9aa68872b9582c6c71151d5dd4f5ebca	37648771	SIG
Windows embeddable package (32-bit)	Windows		b4bd8ec0891891158000c6844222014d	7580762	SIG
Windows embeddable package (64-bit)	Windows		5c34eb7e79cfe8a92bf56b5168a459f4	8419530	SIG
Windows help file	Windows		aaacfe224768b5e4aa7583c12af68fb0	8859759	SIG
Windows installer (32-bit)	Windows		b790fdaff648f757bf0f233e4d05c053	27222976	SIG
Windows installer (64-bit)	Windows	Recommended	ebc65aaa142b1d6de450ce241c50e61c	28323440	SIG

- Click on the installed file and check mark according to this snapshot.

- Once the setup is complete, you can verify your installation by clicking on **cmd** in windows search bar and then type **python,** which will open the idle for you, below, is the screenshot of this.



- We can write a simple program just to have fun and get used to the command terminal please write **print (‘Hello World’)** and **1+1** according to screenshots below.

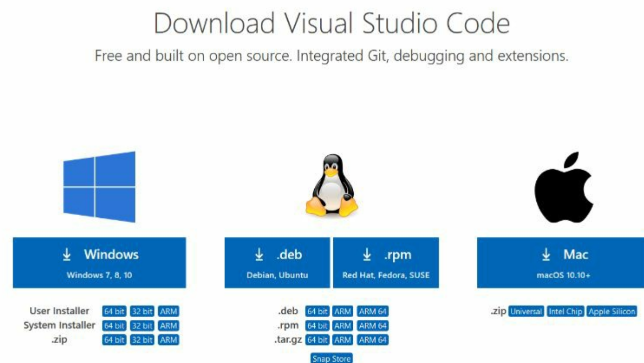
```
Command Prompt - python
Microsoft Windows [Version 10.0.19042.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\HOME>python
Python 3.9.4 (tags/v3.9.4:1f2e308, Apr  4 2021, 13:27:16) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print('Hello World')
Hello World
>>> 1+1
2
>>>
```

- print(‘’) command will print any text or result and we will get into the details of this later in further chapters.
- Next, we need to install a code text editor to make writing big and complicated codes easier for us.

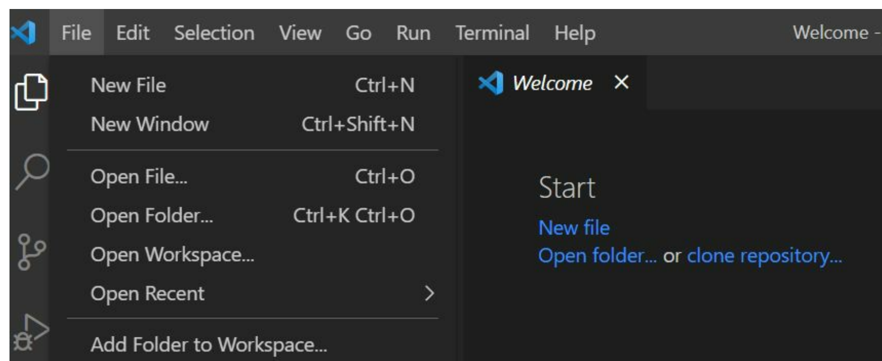
INSTALL A CODING TEXT EDITOR:

As we move on to further chapters we will be dealing with large and complicated codes it will not be easy to write everything on Idle so we need a code text editor in order to compile large codes altogether. I personally use **Visual studio Code** because it is very easy to use and supports writing in other programming languages as well. So let's download and install it.

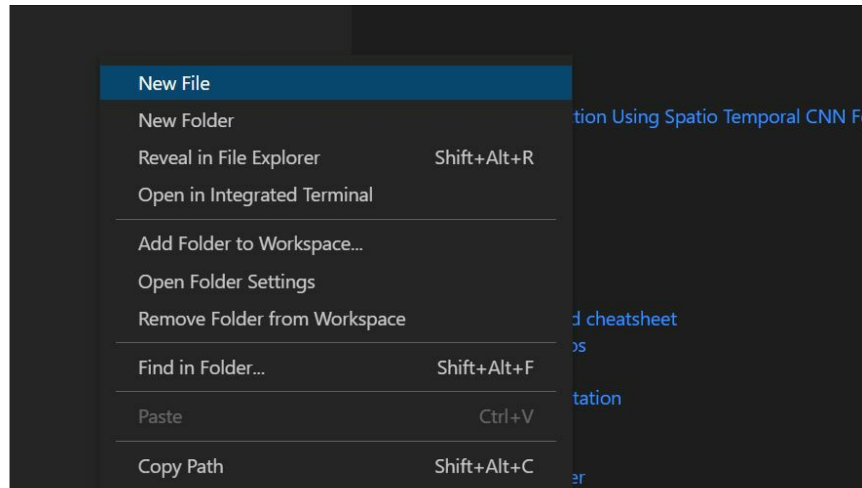
- Click on https://code.visualstudio.com/download. Choose windows and it will automatically start the download for you.



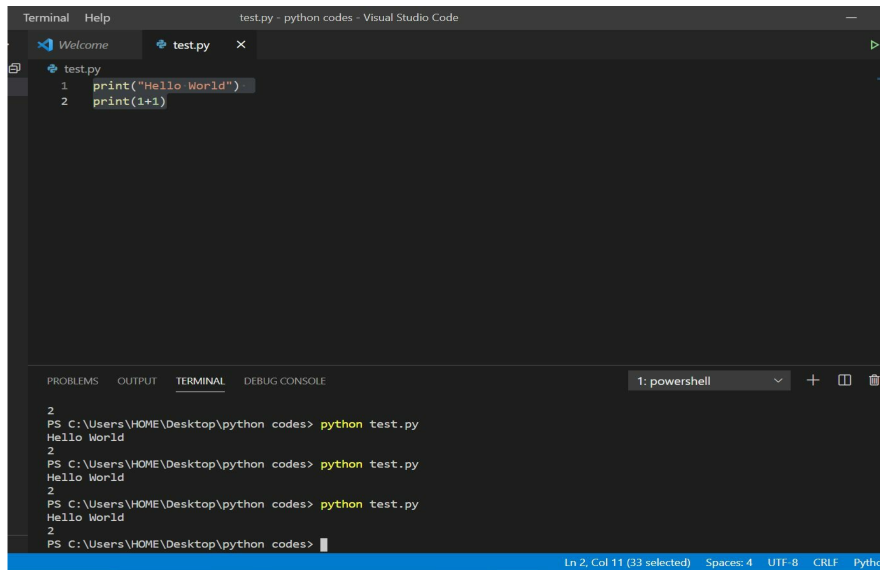
- Once downloaded open it. Also, make a new folder on your desktop named as **python codes.**



- Once opened select the folder that you have make and press **select folder,** which will automatically open the folder for you.
- To make a new file follow the screenshot below and name it as **test.py.**



- Now open the terminal on the top screen and select new terminal, which will open the terminal for you. Now write **print("Hello World")** and **print(1+1)** and to execute the code type **python test.py** in the terminal according to screenshots below.



Congratulations, you have just written your first python code, we will get to the details of the code in the next chapter but this is a huge milestone for you to get you started.

Chapter 2

Learn Variables and various Data Types in Python

Python can be used for many different applications but it is very important for you to have a solid grasp on the basics of python and we will slowly immerse in the capacious world of Python. This chapter aims at the following objectives:

- Variables
- Strings and various operations with it.
- Various data types.
- Concatenating strings and data types.
- Arithmetic operations.
- Type conversions.

Let us get started.

VARIABLES:

A variable by definition is **something that can be changed** and each variable has a certain value, make a new script named as **simple_variable.py** let us have a look at it.

```
Name = "John"
print(Name)
```

In this simple code, **Name** is the **variable** and **John is the value assigned** to this variable however, this can be overwritten very easily.

```
Name = "John"
print(Name)
Name = "Mike"
print(Name)
```

Now the **new variable is Mike** and the previous variable has been overwritten.

SOME COMMON ERRORS WITH PYTHON SCRIPTS:

Most people who start python programming often get errors in their script because they do not write the names properly in the print statement an example is:

```
Name = "John"
print(Nam)
```

File "C:\Users\HOME\Desktop\python codes\simple_variable.py", line 2, in <module>

print(Nam)

NameError: name 'Nam' is not defined

Every time you make a mistake python will tell you exactly what the mistake is and then you can correct that mistake.

STRINGS:

A string is simply a collection of characters, anything inside the quotation mark is considered as a string for example:

```
Name = "John"
print(Name)
```

This will tell you that the name is **John.**

CONCATENATING STRINGS:

You can also concatenate strings to make a longer string, such as:

```
Name1 = "John"
Name2 = "Alice"
print(Name1 +" " + Name2)
```

The output of this is:

John Alice

Let us break the code:

- First, we defined two names.
- We concatenated two names with the help of `" "` which simply defines a space between the two names if it is not used then there will be no space in between.

You can do whatever you like with the help of concatenated strings.

MANIPULATING STRINGS:

You can do other many things with string as well such as:

- To convert all characters to **lower case** use:

```
Name = "John"
print(Name.lower())
```

which outputs `john`

- To convert all characters to **upper case** use:

```
Name = "John"
print(Name.upper())
```

which outputs `JOHN`

- To format a name appropriately:

```
Name = "John the lion "
print(Name.title())
```

which outputs John The Lion

Replace characters in string:

To replace a character use the built in ***.replace(old, new)*** method which replaces old character with new character.

```
Name = "John"
print(Name.replace('h', 'o'))
```

which outputs `Joon`.

ARITHMETIC OPERATIONS:

Arithmetic operations are quite simple you can do addition; subtraction etc whatever you like consider the examples:

```
>>> 3+3
6
>>> 3-3
0
>>> 3 + (2*3)
9
>>> 3/3
1.0
>>> 1.2 + 1.1
2.3
```

DIFFERENCE BETWEEN INTEGER AND FLOAT:

An integer is a whole that can be positive, negative or zero but does not contain decimal places such the examples above `3 + 3 = 6` but `1.2 + 1.1 = 2.3` none of the numbers in it are integer but they are floats denoted by decimal places.

RECEIVING INPUTS:

To receive an input we use the ***input()*** method which basically prompts the user to give a certain input such as:

```
Name = input('Please input your name: ')
print("Hello " + Name)
```

The output will be:

Please input your name: Python

Hello Python

But if you give a number to it, it will throw an error like this:

```
Age = input('Please input your age: ')
age = 2021 - Age
print("Your age is " + age)
```

The output will be:

--

age = 2021 - Age

TypeError: unsupported operand type(s) for -: 'int' and 'str'

--

The error is dictating that the inputs are string but a string cannot be subtracted from integers ant this because the string input has not been converted to integers therefore we need to convert its datatype from string to integer.

TYPE CONVERSIONS:

To convert string age to integer we will use the built in ***int()*** command which will automatically convert the string to integer let us clarify this with an example.

```
birth_year = input('Please input your birth year: ')
age = 2021 - int(birth_year)
print( age)
```

The output will be:

Please input your birth year: 1999

22

In the later part of the book you will see that you will be dealing with a lot of conversions in the future and thus it is important to understand various datatypes as these are the foundations for future.

COMMENTS:

It is important to write comments on code so that the code is readable not just for you but for others as well and if you want to change some parameters of the code later on, you would know by comments the parameters and variables involved in a project. Let us check by an example:

```
#enter your birth year

birth_year = input('Please input your birth year: ')
#age calculation

age = 2021 - int(birth_year)

print( age)
```

SOME RULES OF PROGRAMMING TO KEEP IN MIND ARE:

- First of all you will be dealing with many errors on daily basis if you ever encounter an error stop right there and do something else such as take a quick walk or do some pushups or drink water and come back to it later on.
- Try to make your code as simple as possible and neat as well not just for your understanding but for others as well there should be a difference between you and others.
- When working at a project move one-step at a time, writing good code will earn you respect as well as a chance to collaborate on other projects and job opportunities as well.

Chapter 3

Lists in Python

We all go out on a monthly basis for getting groceries for home or buying some extra fancy items sometimes it is hard to keep track of the amount of money spent on a monthly basis, too keep track of our spending habits we all need list that depicts the cost break down of what we have spent so far. Fortunately, you can make lists in python to keep track of your spending habits, what is more interesting is that you can modify the elements of the list according to your needs. In this chapter, we will specifically look at:

- What are lists and how to make them.
- How to modify lists.
- How to organize lists.

WHAT ARE LISTS IN PYTHON?

A list by definition is simply a collection of values. All the elements in a list are ordered and can be modified according to your needs. Make a new file called **list.py** and to define a list in python the following notation is followed:

```
your_list = []
```

- The [] always indicate that the anything inside the brackets is a list.

A list can hold anything such as strings or numbers and to make a list of string items is shown as:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
print(your_list)
```

The output is:

--

['mangoes', 'oranges', 'bananas', 'grapes']

--

So you see you can make a list as large and versatile as you like.

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes', 0, 1, 2, 3]
print(your_list)
```

The output is:

['mangoes', 'oranges', 'bananas', 'grapes', 0, 1, 2, 3]

MODIFYING LISTS:

You can do many operations with a list such as:

Access elements in a list:

To access a certain element in a list you need to understand what indexing is. In python the first element in a list always start at index 0 and can be seen by the example where the first index is grabbed and printed:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
print(your_list[0])
```

The output is:

```
mangoes
```

The element that lies ahead is at index 1 and so onwards.

Access elements in a list in reverse order:

The reverse order of a list commences at index -1 where the last element of list is printed as there is no such thing as -0 similarly to access the second last element index -2 is used and demonstrated by the example:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
print(your_list[-1])
print(your_list[-2])
```

The output is:

----

grapes

bananas

FIND THE LENGTH OF LIST:

To find the length of list use len() which will return the total number of elements in a list.

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
print(len(your_list))
```

SLICING LISTS:

- To slice list we have to grab the element in a list by its first index up till the interested element +1 because if we grab it till the interested character, it will not be printed as shown by the example:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
print(your_list[0:2])
```

The output is:

['mangoes', 'oranges']

In this example, I wanted to have mangoes and oranges printed but I did not want the rest of the elements to be printed so I grabbed the index of mangoes [0] till the interested index [1] +1 which is at index [2] so the index [2] was not printed but the rest were printed.

- Similarly, if I want to omit the first character I can use the index method above or I can use the following method:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
print(your_list[1:])
```

The output is:

```
['oranges', 'bananas', 'grapes']
```

In this example I used the slicing operator to start my list from index [1] till the end of the list and omitting the index [0].

- If I want do not want to include the last element of the list but instead have the first three elements of the list only, I can use the

following procedure:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
print(your_list[:3])
```

The output is:

['mangoes', 'oranges', 'bananas']

This returns values from index (3-1) which is from index 0 until two.

ADD AND REPLACE ELEMENTS TO LIST:

- To add new elements to list we use the .**append**() method example can be seen below:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
your_list.append('kiwi')
print(your_list)
```

The output is:

['mangoes', 'oranges', 'bananas', 'grapes', 'kiwi']

- To replace and element with another element we grab its index and replace the element in that index with a new element.

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
your_list.append('kiwi')
your_list[0] ='guava'
print(your_list)
```

The output is:

['guava', 'oranges', 'bananas', 'grapes', 'kiwi']

It can be seen that the first element is replaced with a new one.

- You can also replace elements via slicing:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
your_list.append('kiwi')
your_list[0:1] ='guava' , 'chocolate'
print(your_list)
```

The output is:

['guava', 'chocolate', 'oranges', 'bananas', 'grapes', 'kiwi']

This replaces index 0 and 1 with the new elements.

REMOVING ELEMENTS IN A LIST:

- To remove an element in a list you can use the **del**() operator which will delete elements of the list.

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
del your_list[0]
print((your_list))
```

The output is:

['oranges', 'bananas', 'grapes']

- To remove elements using indexing we can specify the indexes of the element till and the last element which we want to delete + 1.

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
del your_list[0:2]
print((your_list))
```

The output is:

['bananas', 'grapes']

ORGANIZE LISTS:

- To sort a list use the **sorted()** method to organize the list in order for example:

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
sorted_list= sorted(your_list)
print((sorted_list))
```

The output is:

('sorted list is', ['bananas', 'grapes', 'mangoes', 'oranges'])

- You can even try various combinations such as printing the list in reverse order to reverse the list use .**reverse**() method.

```
your_list = ['mangoes', 'oranges', 'bananas', 'grapes']
sorted_list= sorted(your_list)
print(('sorted list is', sorted_list))
sorted_list.reverse()
print( 'reverse sorted list is', sorted_list )
```

The output is:

----

sorted list is ['bananas', 'grapes', 'mangoes', 'oranges']

reverse sorted list is ['oranges', 'mangoes', 'grapes', 'bananas']

Therefore, the list is sorted and reversed as well.

Chapter 4

Working with for Loops and if-else statements

In the previous chapters, we have seen that we have to deal with all kinds of data such as strings, numbers, floats and a lot more. In case of games, we want to loop forever so that the game can be played until the user decides to leave the game. In case of repetitive tasks, where we have to perform the same task every time, such as a long list of items we have to work with loops. In this chapter, we will specifically look at:

- What is for loop and how to use it in Python?
- Usage of for loops using examples.
- Working with for loops, functions and lists.
- If-else statements in Python

WHAT IS A FOR LOOP?

Well as the name suggests a loop is something that keeps on going until and unless a specific condition is met. In real life, you can say that you go to your work place on daily basis and you keep on traveling until you have arrived to your destination similar is the case with loops that you keep on repeating the code until a specific condition is met.

Example Counter strike Go game:

You might have played this game or would have played similar games, so a for loop in this case might be that if you press a certain key, the player keeps walking or shooting or doing both the tasks at the same time and if you leave any one of the keys, it stops doing that action so in this case the program keeps on looping and as soon as the desired condition is met, the loop stops.

USAGE OF FOR LOOPS USING EXAMPLES:

- Let us suppose we want out list of groceries to be displayed one we can do that using for loop.

```
fruits = ["apple", "banana", "cherry", "strawberry"]
for x in fruits:
  print(x)
```

The output is:

----

apple

banana

cherry

strawberry

In this case, we defined our list of fruits and printed all of them. What the for loop did is that it iterated over each item in the list and then printed it on screen. **Please note** that indentation here matters, after writing any code after semi colon python automatically brings it according to the required indentation. If you change it, it will throw an error.

- To print numbers we have to use the **range()** method which checks how many times it has to iterate in the loop, once the number is achieved it stops, this is a perfect example of not iterating infinitely, let us examine this:

```
for x in range(6):
  print(x)
```

The output is:

0

1

2

3

4

5

In this case, python starts at index 0 and iterates all the way to 5, 0 is counted as well.

- You can even use two for loops for checking more variables at once this is known as nested for loops:

```
attribute = ["sweet", "Loyal", "King"]
animals = ["Cats", "Dog", "Lion"]

for x in animals:
  for y in attribute:
    print(x, y)
```

The output is:

Cats sweet

Cats Loyal

Cats King

Dog sweet

Dog Loyal

Dog King

Lion sweet

Lion Loyal

Lion King

In this case for the first item cat and for each of its corresponding attribute the item is matched to every attribute, the for loop iterates over the animals and the second for loops attaches each attribute to that animal, once the second for loop finishes, the condition has been met, the second item is taken and mapped to every attribute, similarly for the third element.

IF-ELSE STATEMENTS IN PYTHON:

As the name suggests if and else are used if a certain condition is met, if it is not met then you want to break out of the loop and execute the other example. In real life, you can say that if you study smart and hard for an exam you will score well else you will not score good marks, let us clear this with some examples:

```
a = int(input('number a: '))
b =int(input('number b: '))
if b > a:
    print("b is greater than a")
elif a == b:
    print('b is equal to a')
else:
    print('a is greater than b')
```

The output is:

----

number a: 5

number b: 5

b is equal to a

In this case it is seen that the user is prompted for an input, if number a is greater than b then a certain condition is executed else if a is equal to b then something is executed else it is stated that a is greater than b. In this both the numbers are five so a is equal to b.

AND, OR OPERATION:

- and operation is executed if both of the test conditions are true so let us say if john has mangoes and milk then he can make a mango tart but if john has mangoes but he doesn't have milk then he cannot make mango tart.
- or operation works when either of the test condition is true.

Let us clarify with some examples:

```
a = int(input('number a: '))
b =int(input('number b: '))
c =10
if b > c and a>c:
    print("b and a are greater than c")
else:
    print("one or both conditions are not true")
```

The outputs are:

----

number a: 9

number b: 11

one or both conditions are not true

----

number a: 11

number b: 12

b and a are greater than c

In these cases, it is seen that when one of the condition is not true then whole loop exits and if both conditions are true only then it works.

- For the or operation:

```
a = int(input('number a: '))
b =int(input('number b: '))
```

```python
c =10
if b > c or a>c:
    print("one or both conditions are true")
else:
    print('both conditions are not true')
```

The outputs are:

----

number a: 2

number b: 11

one or both conditions are true

----

number a: 2

number b: 9

both conditions are not true

In these cases, it is seen that when one of the condition is not true, even then the whole condition is satisfied and if both conditions are not true only then it executes the else block.

Chapter 5

Functions in python

Human beings are quite complex, each part of our bodies are designed for a specific task such as your hand’s function is to pick and place items regarding to your needs, your legs function is to walk even the function of your brain is to thinks and convey messages to different parts of the body via neurons, all these functions of the body are called whenever they are required. In this chapter, we will learn how functions in programming follow the human analogy specifically we will look at:

- Functions in python and how they are useful.
- Coding examples of functions.

FUNCTIONS IN PYTHON AND HOW THEY ARE USEFUL:

So taking the analogy of the human body, a function is useful when you have certain part of your code to do a specific operation, take an example of a calculator whose function is to add, subtract and do numeric computations so if you want to use some part of a code for something and another part of the code to do something else you can wrap those independent tasks in functions and call them whenever needed.

CODING EXAMPLES OF FUNCTIONS:

Here are some coding examples we will start from how to define functions and doing simple operations with functions.

DEFINE A FUNCTION:

To define a function we will use a built in **def function_name()** which python will understand that a function is going to made with a certain name, the brackets can be left as empty, if there are certain variables that will be used then we need to define them:

```
def func():
    print ("function is called")

func()
```

The output is:

----

function is called

It is seen that function is defined and something is printed whenever the function is called.

PASSING VARIABLES TO FUNCTIONS:

If you want to make a calculator, you will definitely need some variables and some functions, which perform independent operations of addition, subtraction, multiplication, division etc. Some output would have to be returned as well so that it can be printed to screen.

```
def add(a,b):
    c = a+b
    return c
def sub(a,b):
    c = a-b
    return c
def mul(a,b):
    c = a*b
    return c
def div(a,b):
    c = a/b
    return c

print(add(2 ,2))
print(sub(2 ,2))
print(mul(2 ,2))
print(div(2 ,2))
```

The output is:

4

0

4

1.0

In this simplest calculator, it is seen that a function is made which takes two variables, each function performs its own operation and the output values are returned and printed.

PART 2

INTRODUCING TO ALGORITHMS IN PYTHON

Chapter 6:

Array algorithms in python

To ace some of the top coding interviews of Google there are various patterns of algorithms that are repeated, these algorithms enhance your current knowledge and skill to develop algorithm development skills to save time complexity and space complexity which is really essential when launching a business product or brand. The aims of this chapter are:

- Algorithms and their corresponding explanations in python.
- Understanding importance of time complexity.
- Understanding of space complexity.

TWO NUMBER SUM:

As the name suggests two number sum algorithm is designed to calculate the sum of two numbers given a target number from an array, let us clear the concept via looking at the algorithm.

Let us suppose you have an array such as:

```
[3, 5, -4, 8, 11, 1, -1, 6]
```

The sum of any two number from this array should be such that it equals the target number let us say that the target number is 10, the code would look as:

Code:

```
def twoNumberSum(array, targetSum):
    for n in array:
        #calculated subtraction
        residual = targetSum - n
        #check for unique numbers
        if residual in array and residual != n:
            return sorted([residual, n])
    return []
sum_ = twoNumberSum([3, 5, -4, 8, 11, 1, -1, 6], 10)
print(sum_)
```

ALGORITHM EXPLANATION:

- Since the target number is 10, we first calculate the residual; this is for storing any unique numbers in a variable.
- Then we check whether ***if residual in array*** or if any unique number is in the array, if the number is found, we store it.
- Then we check whether **residual is not equal to any number in n** if it is not we also store that unique number.
- Then we sort and return output.

Algorithm:

```
[3, 5, -4, 8, 11, 1, -1, 6]
```

- residual = 10-3 , 10-5 , 10-(-4), 10-11, 10-1, 10-(-1), 10-6
- if residual number is in array or if 7, 5, 14, -1, 9,11, 4
- And if residual is not equal to n
- Return the unique number of residual and n.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n). Here n is the length of the input array, the reason it is only O (n) times is that we are traversing the array only once, there are no double for loops. It runs in constant time only.
- Space complexity is also O (n).

THREE NUMBER SUM:

- Let us suppose that we have an array:

[12, 3, 1, 2, -6, 5, -8, 6]

- The aim is to find a three number sum, which equals to a target number 0.
- So first, we are going to sort the array according to ascending numbers:

[-8, -6, 1, 2, 3, 5, 6, 12]

- We will then apply the three number sum formula which is current sum = current number + left pointer + right pointer
- We set the current number at -8, the left pointer at -6 and the right pointer at 12 applying the formula we get -8 -6 + 12 = -2
- We update left pointer to be at 1 so we get -8 +1 + 12 = 5 so this results in a larger number and moving any further will give us larger numbers so we decrease right pointer from 12 to 6 and then -8-6+6 = -8 we keep on updating left pointer until left pointer reaches 2 and right pointer at 6 which gives a target sum of 0.
- We get another sum of zero when left pointer moves right and reaches three and right pointer reaches five so applying the formula we get -8+3+5 =0. After this, the pointers cannot move any further and the constant number -8 is updated to -6.
- Then the cycle starts again and this time left pointer starts at 1 and right pointer at 12, the sum is not equal to zero so right pointer decreases and at this point the sum = -6 +1 +6 =1 so we know that the right pointer has to be decreased again. This time the sum = -6 + 1+ 5 =0, at this stage this is our third set of numbers which equal zero. There are not any possible situations after this since moving left pointer upwards results in larger number and right number downwards results in a smaller number.

```
def three_number_sum(array, targetSum):
    array.sort()  # [-8, -6, 1, 2, 3, 5, 6, 12]
    triplates = []
    for i in range(len(array) - 2):
        # len(array)-2 because we dont want to include
```

```
        #last 2 because it will be for left and right index
        left_index = i + 1
        right_index = len(array) - 1
        while left_index < right_index:
            sum = array[i] + array[left_index] + array[right_index]
            if sum == targetSum:
                triplates.append([array[i], array[left_index], array[right_i
                left_index += 1
                right_index -= 1
            elif sum < targetSum:
                left_index += 1
            elif sum > targetSum:
                right_index -= 1
    return triplates
```

```
sample_input_array = [12, 3, 1, 2, -6, 5, -8, 6]
sample_sum = 0
result = three_number_sum(sample_input_array, sample_sum)
print(result)
```

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n^2). Here n is the length of the input array, the reason it is only O (n^2) times is that we are traversing the array with for and while loop and we are also performing a lot of operations such as calculating the sums and appending triplets to array.
- Space complexity is also O (n).

SMALLEST DIFFERENCE:

- Let us suppose that we have two arrays as shown below and we would like to calculate the smallest difference from both of these two arrays.

array_one = [-1, 5, 10, 20, 28, 3]

array_two = [26, 134, 135, 15, 17]

Algortihm Explanation:

- The first step would be to sort the two arrays such that they become:

array_one = [-1,3 ,5, 10, 20, 28]

array_two = [15, 17, 26, 134, 135]

- We have define two pointers, one for the left array and another for the right array.
- If left number < right number then update the left pointer but if left number > right number then update the right pointer.
- Initially -1 and 15 will be compared and their difference is calculated in this case -1 < 15 so left pointer is updated and 3 < 15, their difference is calculated.
- We keep on repeating this process until we reach 20 in this case left number > right number or 20 > 15 so we update right pointer as updating left pointer will give us negative values. Now 20 > 17 so we update right pointer again, now 20 < 26 so left pointer is updated and reaches 28. In this case 28 > 26, the difference is calculated and the process repeats for all the other numbers.

Code:

```
def smallestDifference(arrayOne, arrayTwo):
    array_one = sorted(arrayOne)
    array_two = sorted(arrayTwo)
    index_one = 0
    index_two = 0
```

```python
    smallestDiff = float("inf")  # Used for big initialisation
    result_pair = []
    while index_one < len(array_one) and index_two < len(array_two):
        first_num = array_one[index_one]
        second_num = array_two[index_two]
        currentDiff = abs(first_num - second_num)
        if currentDiff < smallestDiff:
            smallestDiff = currentDiff
            result_pair = [first_num, second_num]
        if first_num <= second_num:
            index_one += 1
        else:
            index_two += 1

    return result_pair

# My Solution O(n^2)
def small(arr1, arr2):
    arr1 = sorted(arr1)
    arr2 = sorted(arr2)
    diff = float("inf")
    for i in range(len(arr1)):
        for j in range(len(arr2)):
            if(abs(arr1[i]-arr2[j]) < diff):
                diff = abs(arr1[i]-arr2[j])
                m, n = i, j

    return [arr1[m], arr2[n]]

array_one = [-1, 5, 10, 20,24,31,54,45,6,43,44,245,67,88,43,56, 28, 3]
array_two = [26, 134, 135, 15, 17]
print(smallestDifference(array_one, array_two))
```

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (nlog(n) + mlog(m)). Here n is the length of the input array one and m is the length of the input array two, the reason it is only O (nlog(n) + mlog(m)) times is that we sorting both array.
- Space complexity is also O (1) because we are sorting the arrays in place and storing anything to consume additional memory.

VALIDATE SUBSEQUENCE:

- In this example we want to validate if a certain array is a subsequence of a main array let us clarify this with an example.
- Suppose that we have an array:

```
[2,3, 1 ,4 ,6,5 , 10, -1]
```

- We want to find whether

```
[1, 6, 10, -1]
```

Is a subsequence of our main array, so the algorithm would look as:

- Traverse through the array and check if the current item in the main array matches the item in the smaller array so 2 will be compared 1 if match is not found then the left pointer is updated and brought to 3, if another match is not found, the left pointer is again updated and brought to 3, it'll be compared with 1 and when match is not found left pointer will be updated.
- When 1 in the main array is compared with 1 from the subsequence array match is found, the right pointer is updated and left as well, then 4 is compared with 6 and when match is not found left pointer is updated again.

In other words:

```
if sequence[seqIdx] == val:
            seqIdx+=1
```

Code:

```
def validateSubsequence(arr, sequence):
    seqIdx=0
    for val in arr:
        if seqIdx == len(sequence):
            break
        if sequence[seqIdx] == val:
            seqIdx+=1
    return seqIdx == len(sequence)

x =validateSubsequence([2,3, 1 ,4 ,6,5 , 10, -1], [1, 6, 10, -1])
print(x)
```

The output is:

True

- As it is seen that every value in the main array is compared with the subsequence array, if a match is found then the index is updated and the next item is checked.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n). Here n is the length of the input array, the reason it is only O (n) times is that we are traversing the array only once, there are no double for loops. It runs in constant time only.
- Space complexity is also O (1).

PRODUCT SUM:

- Suppose we have an array that also has many sub arrays in it:

```
[5, 2, [7, -1], 3, [6, [-13,8],4 ]]
```

- As you can see that we have a main array and sub arrays inside such as [7,-1] and a subarray which also has another subarray such as [6, [-13, 8], 4].
- The program has to made such that we have the product sum of this array and trust me that the answer of this is going to be 12, let us see how the algorithm works.

Algorithm:

- Our main array is [5, 2, [7, -1], 3, [6, [-13, 8], 4]]
- Sum = 0 (at the start)
- Multiplier = mul = 1 (for the whole array)
- The multiplier mul will increase at each sub array (each sub array will have increase in multiplier, added and then multiplied with multiplier)
- At the start 5 + 2 = 7 with multiplier mul =1
- We reach sub array [7, -1], multiplier is increased mul = 1+1 =2
- So this sub array's sum = 7 +(-1) =6 x 2(mul) =12
- Add 12 with three and initial 7 = 15 +7 = 22
- We reach another subarray so its mul = 1+1 = 2
- We find another sub array in it so mul = 2+1 =3
- Solving subarray first -13 + 8 = -5 x3 = -15
- Then solving the bigger sub array = 6 + (-15) + 4 = -5
- Then multiplying this with the subarray's multiplier -5x2 = -10
- Adding all sums = 22 -10 =12 (answer)

The code will look as:

```
def productSum(array, multiplier = 1):
    sumProd = 0
    for element in array:
        if type(element) is list:
            sumProd += productSum(element, multiplier + 1)
```

```
        else:
            sumProd += element
    return sumProd * multiplier

product_sum = productSum([5, 2, [7, -1], 3, [
    6, [-13,8],4
]])
print(product_sum)
```

The output is:

```
12
```

Explanation:

- We define our sum equal to zero.
- For each element in array, we check if the element is a list, if it is then we increment our multiplier and add it to sum.
- Otherwise, we just add the number (element) to it.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n). Here n is the length of the input array, the reason it is only O (n) times is that we are traversing the array only once, there are no double for loops. It runs in constant time only.
- Space complexity is also O (1). We are not storing anything extra so it runs in constant time only.

Chapter 7:

Sorting algorithms in python

Sorting is a way to arrange the numbers in alphabetical order yet this is the most repeated questions in coding interviews in order to check your algorithmic knowledge and algorithm development skills.

BUBBLE SORT:

Bubble sort algorithm works by placing larger numbers to the right on every iteration, it is the simplest of the sorting algorithms, let us see the way it works:

We will using this array throughout the sorting series:

```
[8,5,2,9,5,6,3]
```

Algorithm:

- We start with and see whether 8>5 then we swap it so our array becomes [5,8,2,9,5,6,3].
- Then we check if 8>2 yes it is so swap it [5, 2, 8, 9, 5, 6, 3].
- Then we check if 8>9 no it is not so our greatest number is updated to 9 [5,2,8,9,5,6,3].
- Then we check if 9>5 yes it is so swap it [5,2,8,5,9,6,3].
- Then we check if 9>6 yes it is so swap it [5,2,8,5,6,9,3].
- Then we check if 9>3 yes it is so swap it [5,2,8,5,6,3,9].

Repeat this process again starting from the first element so we will have a sorted array in the end [2, 3, 5, 5, 6, 8, 9].

Code:

```
def bubbleSort(array):
    isSorted = False
    counter = 0
    while not isSorted:
        isSorted = True
        for i in range(len(array)- 1 - counter):
            if array[i] > array[i+1]:
                swap(i, i+1, array)
                isSorted = False
        counter+=1
    return array
```

```
def swap(m, n, arr):
    arr[m], arr[n] = arr[n], arr[m]

sort_ = bubbleSort([8,5,2,9,5,6,3])
print(sort_)
```

The output is:

```
[2, 3, 5, 5, 6, 8, 9]
```

Code explanation:

- We first check if the array is not sorted, if it is not then for all the elements in array we check if the current number > next number. If it is then swap the number.
- Keep on repeating until we get a sorted array and we can swap no more.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n^2). Here n is the length of the input array, the reason it is only O (n^2) times is that we are traversing the array multiple time with a while loop and we are looping through it until it is sorted. The best-case scenario is O (n) times when we were given a sorted array.
- Space complexity is also O (1). We are not storing anything extra so it runs in constant time only.

INSERTION SORT:

This is the simplest of the sorting algorithms and the skeleton of the algorithm is to put numbers in their correct position in ascending order, let us see the way that works:

Suppose you have an array:

```
[8,5,2,9,5,6,3]
```

We have to arrange the numbers in such a way such 2 is in the starting position and the rest of the elements are in ascending order:

Algorithm:

- Perform comparison such that if the current number < previous number or in other words, let us suppose the current number is 5 and previous number as seen in 8.
- Now 8 > 5 but it is arranged as 8, 5 so swap 5 and 8. The new array is [5,8,2,9,5,6,3]
- Now we move on to 2 which clearly is less than 8 8>2 or previous number > current number so swap its position [5, 2, 8, 9, 5, 6, 3], now 2 will be checked with 5 and it can be seen that 5>2 or previous number > current number so swap its position.

In code:

```
def insertionSort(array):
    for i in range(1,len(array)):
        j=i
        while j > 0 and array[j] < array[j-1]:
            #swap the array
            array[j],array[j-1] = array[j-1],array[j]
            j -= 1
    return array

sort_ = insertionSort([8,5,2,9,5,6,3])
print(sort_)
```

The output is:

```
[2, 3, 5, 5, 6, 8, 9]
```

Code explanation:

- Iterate through the array starting at index 0 because there is no previous number before it.
- If current number < previous number, then swap the array and decrease its index to fit it in its correct position.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n^2). Here n is the length of the input array, the reason it is only O (n^2) times is that we are traversing the array multiple time with a while loop and a for loop and we loop through it until it is sorted. The best-case scenario is O (n) times when we were given a sorted array.
- Space complexity is also O (1). We are not storing anything extra so it runs in constant time only.

SELECTION SORT:

This is another sorting algorithm but the major difference between it and insertion sort is that the selection sort can sort the numbers no matter where their index lies in the array, let us say that we have an array:

```
[8,5,2,9,5,6,3]
```

The numbers will be sorted to return the output in ascending order.

Algorithm:

- This algorithm focuses on getting the smallest number in every iteration.
- We start off with 8, then we check is 5<8, yes 5 is smaller than 8 so we move to the index of 5, then we check if 2<5, yes it is so we move to 9,5,6,3, all of these numbers are greater than 2 so we can swap 8 and 2 so that our array looks like: [2, 5, 8, 9, 5, 6, 3].
- Then we start at index 5 and perform comparisons where 5<8, 5<9, 5=5, 5<6 but when we get to three we see that 3 <5 so we swap its position and our array becomes: [2, 3, 8, 9, 5, 6, 5].
- Then we move on to 8 and perform comparisons so you can see 5 is the smallest number in this iteration so our array will become [2, 3, 5, 9, 8, 6, 5]. Similarly, we can sort the rest of the array like this.

Code:

```
def selectionSort(array):
    i = 0
    while i < len(array):
        min_element = i
        j = i+1
        while j < len(array):
            if array[min_element] > array[j]:
                min_element = j
            j += 1
        swap(array,i,min_element)
        i += 1
    return array

def swap(array,i,min_element):
    array[i],array[min_element] = array[min_element],array[i]
```

```
sort_ = selectionSort([8,5,2,9,5,6,3])
print(sort_)
```

The output is:

```
[2, 3, 5, 5, 6, 8, 9]
```

Explanation:

First, we start of at index 0 and declare that as min element.

Then we update our index and perform comparisons if min_element (current) > next element. If it is then we update min_element to be our next element, we keep on updating and checking for min_element and once we have found it, we can swap its position from the index we started.

We keep on repeating for the entire elements until we reach our output.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n^2). Here n is the length of the input array, the reason it is only O (n^2) times is that we are traversing the array multiple time with multiple while loops and we are looping through it until it is sorted. The best-case scenario is O (n) times when we were given a sorted array.
- Space complexity is also O (1). We are not storing anything extra so it runs in constant time only.

Chapter 8:

Interesting algorithms in python

There are many more algorithms in python that used in daily cases such as file handling, algorithmic document check and computations in a big network, each algorithm has different time complexity and space complexity, yet we will learn the most optimal solutions in terms of time and space.

DEPTH FIRST SEARCH:

Depth first search works by having children and parent mechanism where a parent can have multiple children; the left branch is solved first so in this case A, B and E will be appended first. Then we will move to the right so that the left most branch F and I will be appended in this list. Then moving to the right J will be appended, and then C will be appended. Therefore, in each parent node we will explore its children starting from the left and append that to the list.

Algorithm:

- Add a parent and add its children. Each children will in turn have their own children.
- Once added call the depth first search function that will first append the parent and then will append each child which in turn will append its own children first, once a particular branch is finished then only we can move to the next branch.
- So looking at this example A will be appended first and will move on to B which will first append its left most child E, once that is appended will move to the right child F and its sub children I and J will be appended then it will move to C and the process will continue just like this.

Code:

```
class Node:
    def __init__(self, name):
        self.children = []
        self.name = name

    def __repr__(self):
        return str(self.name)

    def addChild(self, name):
        self.children.append(Node(name))
        return self

    def depthFirstSearch(self, array):
        array.append(self.name)
        for n in self.children:
            n.depthFirstSearch(array)
        return array
```

```python
if __name__ == '__main__':
    root = Node('A')

    root.addChild('B')
    root.addChild('C')
    root.addChild('D')

    # append to B node E and F
    root.children[0].addChild('E')
    root.children[0].addChild('F')

    # append to D node G and H
    root.children[2].addChild('G')
    root.children[2].addChild('H')

    # append to F node I and J
    root.children[0].children[1].addChild('I')
    root.children[0].children[1].addChild('J')

    # append K to G node
    root.children[2].children[0].addChild('K')

    print(root.depthFirstSearch([]))
```

Explanation:

- In a class we add helper functions, the first function will initialize the parent and children.
- We make another function to add children nodes.
- Then we perform depth first search on parent, children nodes and any sub children nodes. For each child just append the name of the child to the list.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (v + e). Here v are the number of vertices and e are the number of edges, vertex is the node in the graph and lines are the edges or connectors.
- Space complexity is also O (1). We are not storing anything extra so it runs in constant time only.

BRANCH SUM:

In this, we simply add all the branches of the tree and calculate the total sum of the branch.

The idea is to recursively call the subtrees until we reach the end nodes with all the sums calculated.

Code:

```
class newNode:

    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def addBT(root):
    if (root == None):
        return 0
    return (root.key + addBT(root.left) +
                        addBT(root.right))

# Driver Code
if __name__ == '__main__':
    root = newNode(1)
    root.left = newNode(2)
    root.right = newNode(3)
    root.left.left = newNode(4)
    root.left.right = newNode(5)
    root.right.left = newNode(6)
    root.right.right = newNode(7)
    root.right.left.right = newNode(8)

    sum = addBT(root)

    print("Sum of all the nodes is:", sum)
```

Explanation:

In this code, you can simply see that all we are doing is adding the sums of the branches until we reach the leaf nodes and then add the total numbers found in the leaf nodes.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n). Here n is the length of the input array, the reason it is only O (n) times is that we are traversing the tree only once and calculating its total sum from each branch, there are no double for loops. It runs in constant time only.
- Space complexity is also O (n). N is the recursive call.

BREADTH FIRST SEARCH:

This algorithm is a bit different from the depth first search. It involves the children nodes to be added level by level so A will be added first, followed by B, C and D. Next E, F, G and H. Finally, I, J and K will be added, let us see the way the algorithm works.

Algorithm:

- We will use a Queue in the example that follows the first in first out (FIFO) mechanism. A current will be used to keep track of the parent node and its children will be added to the queue.
- Queue = [A]
- Current = A
- The children of the current will be added in the queue now:
- Queue = [A, B, C, D]
- Remove A from current and add it to an array remove A from queue.
- Queue = [B, C, D]
- Array = [A]
- Then make current = B and add its children to queue:
- Queue = [B, C, D, E, F]
- Remove B from current and add it to an array remove B from queue.
- Queue = [C, D, E, F]
- Array = [A, B]
- Then make current = C and add its children to queue but in this case, we do not have any children so leave it as it is:
- Queue = [C, D, E, F]
- Remove C from current and add it to an array remove C from queue.
- Queue = [D, E, F]
- Array = [A, B, C]

I hope you get the idea and this continues until we have added all the letters to the Array.

Code:

```
class Node:
    def __init__(self, name):
        self.children = []
        self.name = name

    def addChild(self, name):
        self.children.append(Node(name))
        return self

    def breadthFirstSearch(self, array):
        queue = [self]
        while len(queue) > 0:
            node = queue.pop(-1)
            array.append(node.name)
            for child in node.children:
                queue.insert(0, child)

        return array
```

```
if __name__ == '__main__':
    test1 = Node('A').addChild('B').addChild('C').addChild('D')
    test1.children[0].addChild('E').addChild('F')
    test1.children[2].addChild('G').addChild('H')
    test1.children[0].children[1].addChild('I').addChild('J')
    test1.children[2].children[0].addChild('K')

    print(test1.breadthFirstSearch([]))
```

The output is:

```
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K']
```

Explanation:

- First, we make a function to initialize all the names and the children.
- Next, we make a function to add the names of the children in the array of children.
- The code works by first initializing the queue and checking

whether the queue is empty or not empty, if it is not then we say, our node (current) is the first element in the queue and we append that to the array.

- We keep on inserting every child in the queue.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (v + e). Here v are the number of vertices and e are the number of edges, vertex is the node in the graph and lines are the edges or connectors.
- Space complexity is also O (v). We are storing v nodes only.

BINARY SEARCH:

This is a very interesting algorithm and is relatable to real life; let us suppose that your exam teacher is looking for your name in all the examination papers that are submitted. Suppose that all the examinations are in an alphabetical order, so the way to look at your paper is that the teacher will discard the pile of papers on the top of your name, the teacher will also eliminate the pile of papers beneath and this process will continue until your exam is found for review.

In binary search we perform the same steps by searching for the index of the element, which we are, interested in, let us see this in action.

Suppose that we have an array:

```
[0, 1, 21, 33, 45, 45, 61, 71, 72, 73]
```

We are interested to find the index of element 33 or in this case, it can be your exam paper, here is how the algorithm works.

Algorithm:

- To search for the index we are interested in we have to use the formula (last + first)//2, this will give us the absolute index of the array also we will use the left, middle and right pointers to keep track of left and right elements in the array, so in this case if we use the formula:

[0, 1, 21, 33, 45, 45, 61, 71, 72, 73]

- 0 in the array has index 0, our left pointer is currently there and 73 has index 9, our right pointer is currently there, so using this formula we get coordinate of middle pointer as well:

M = 0+9//2 = 4.5 = 4

- So all the elements in the array will be discarded from index 4, our left pointer will be at index 0, right pointer and middle pointer at 33, which means we will be left with:

[0, 1, 21, 33]

- Then we will use the formula again:

M = 0+3//2 = 1 so every element will be discarded from one and before:

[21, 33]

- Now we perform a comparison check to check whether 21 = 33, since it is not we can move our left pointer, which is at 33 to the right one so that they both overlap and discard 21.

Then we have out answer, let us see the way it works in code:

Code:

```
def binarySearch(array, target):
    # Write your code here.
    first = 0
    last = len(array) - 1
    while True:

        mid_idx = (last + first) // 2
        mid_element = array[mid_idx]

        if mid_element == target:
            return mid_idx
        elif first == last or first > last:
            return -1
        elif mid_element > target:
            last = mid_idx
        else:  # mid_element < target
            first = mid_idx + 1
```

```
if __name__ == '__main__':
    a = [0, 1, 21, 33, 45, 45, 61, 71, 72, 73]
    print(binarySearch(a, 33))
```

The output is:

3

Explanation:

- The left pointer is at index 0 and the right pointer is in the last element of the array.
- The mid index is calculated using the formula.
- If mid element reaches target return its index
- Else if first = last or first > last then just return the last index.
- Else if mid element > target then set then eliminate greater numbers.
- Else if mid element < target then eliminate the smaller numbers and increment the min index to point to the target element.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (log (n)). We are eliminating half of inputs every time.
- Space complexity is also O (1). We are not using additional memory to store, we are doing the operations in memory only.

NTH FIBONACCI:

This is yet an interesting algorithm and makes recursive calls to the previous numbers before it to calculate their sum, let us suppose we have some numbers:

[0, 1, 1, 2, 3, 5, 8, 13, 21]

The Fibonacci is a sum of the previous two numbers that results in an output number; let us see the way it works:

Algorithm:

Therefore, it works by calculating the sums of previous numbers such as:

2 = 1+1

3 =2+1

5= 3+2 and so on

- Therefore, it calculates the sums of the previous numbers.
- The output shows the terms that are computed in order to calculate the output.

Code:

```
def getNthFib(n):
    if n == 1:
        return 0
    elif n == 2:
        return 1
    else:
        return getNthFib(n-2) + getNthFib(n-1)

fib = getNthFib(6)
print(fib)
```

The output is:

5

- So five was the last number that was calculated in order to output

8 and it makes senses because 5+3 = 8 so the program outputted the number of computations it took to calculate 8 that are: [0, 1, 1, 2, 3, 5].

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (2^n) because we are calling previous two fibs every time or rather 2^n times.
- Space complexity is also O (n).

PALINDROME CHECK:

Suppose that you have some letters arranged in a particular way, such as:

'a, b, c, d, c, b, a'

We want to check whether the letters in the forward direction are equal to the letters in the reverse direction; here is the way the algorithm will work:

Algorithm:

Approach1:

- First, reverse the string using the reverse key word.
- Then append the reversed characters in a variable,
- Finally compare whether reversed_string == original string

Code:

```
def isPalindrome(string):
    reversedString = ""
    for i in reversed(range(len(string))):
        reversedString+= string[i] #Takes O(n) time
    return string == reversedString

palindrome = isPalindrome('a,b,c,d,c,b,a')
print(palindrome)
```

The output is:

```
True
```

Explanation:

- For each character in all reversed (characters), we append the alphabets in the reversed_String variable.
- Finally, we check whether our original string equals the reversed string.

Approach2:

Code:

```
def isPalindrome(string, i=0):
    j = len(string) - 1 - i
    if i>= j:
        return True
    else:
        return string[i] == string[j] and isPalindrome(string, i+1)

x= ['a','b','c','d','c','b','a']
print(len(x))
palindrome = isPalindrome(x)
print(palindrome)
```

Explanation:

- In this example, we have different cases, the first case is that we initialize a variable equal to the last element-i first.
- i is the iterator that will check the elements in forward and j is the element which will check in the reverse direction.
- If i>j then we have checked everything and the condition is complete.
- Otherwise, compare string[i] with string[j] to keep on checking whether they are equal and call the function recursively with our string in it and updating the iteration that will change j as well.

Algorithm:

- Here is how one iteration works,
- First i = 0
- j= len(array) -1 –i
- Now string[i] == string [j] in other words string [0] == string [6] in other words a==a
- Yes, it is so update i.
- Second iteration:
- i = 1
- j= len(array) -1 –i
- Now string[i] == string [j] in other words string [1] == string [5] in other words b==b

- Yes, it is so update i.

TIME AND SPACE COMPLEXITY:

- Time complexity of this algorithm is O (n^2) because each time we append to new string variables and compare them.
- Space complexity is also O (n). We use memory to store alphabets so it is not constant time.

Python is the most commonly used programming language and is used in almost all digital applications found today, we find ourselves immersed in various technological applications if you want to play a game or if you wish to edit your photos on Instagram or Snapchat or automate something such as sending automated messages to your customers on various platforms or even make up a website with your custom features then it is more than necessary for you to learn python as it is the backbone of all these digital applications. Some of the toughest coding interviews are also taken in python.

This book gives you a walkthrough of the fundamentals of the python programming language and it will allow you to prepare for algorithms and their parameters such as space and time complexity commonly asked in coding interviews.

This book will help you to learn:

- Uses of python in daily life
- Setting up your python environment
- String and variables
- Various operations in python
- Lists and arrays
- If-else statements
- Loops
- Functions
- Algorithms for coding interviews
- Time and space complexity of the algorithm