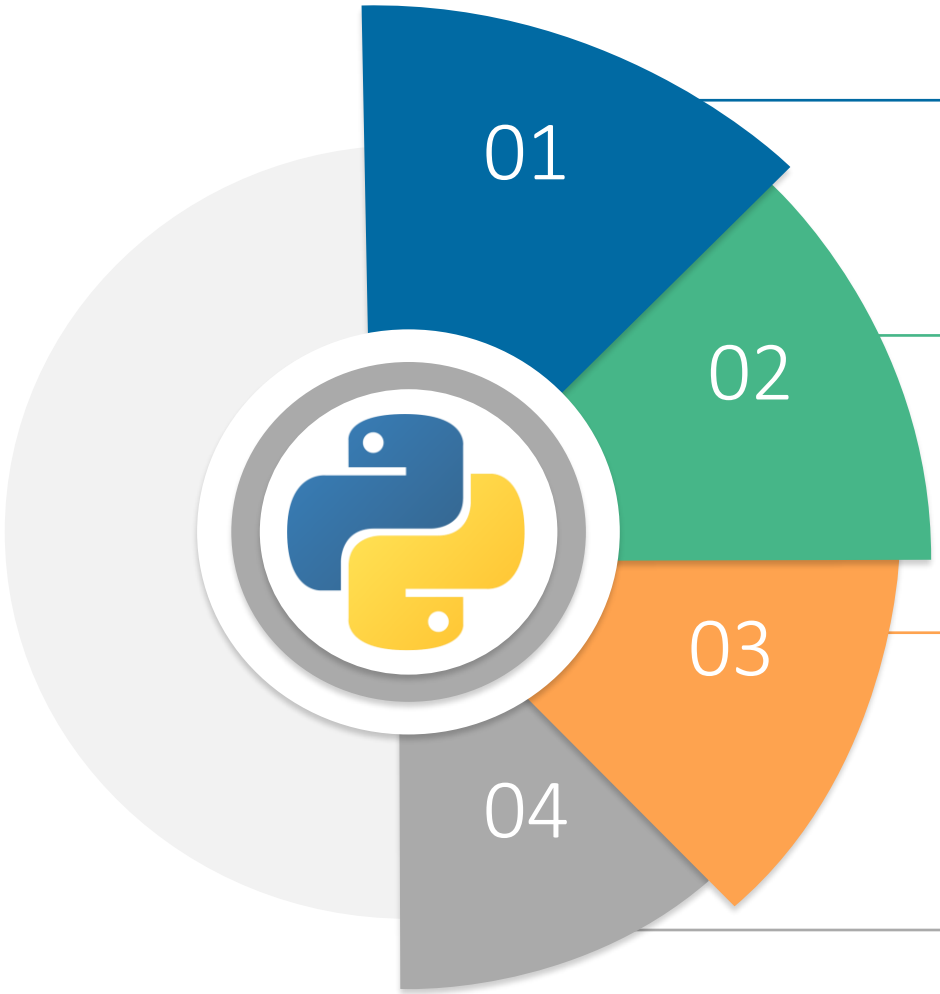
GKTCS INNOVATIONS

INTRODUCTION TO PYTHON

What is Python ?

GKTCS INNOVATIONS



Interactive

Interpreted

Object Oriented Scripting Language

High Level Programming Language

Difference between Programing language & Scripting language

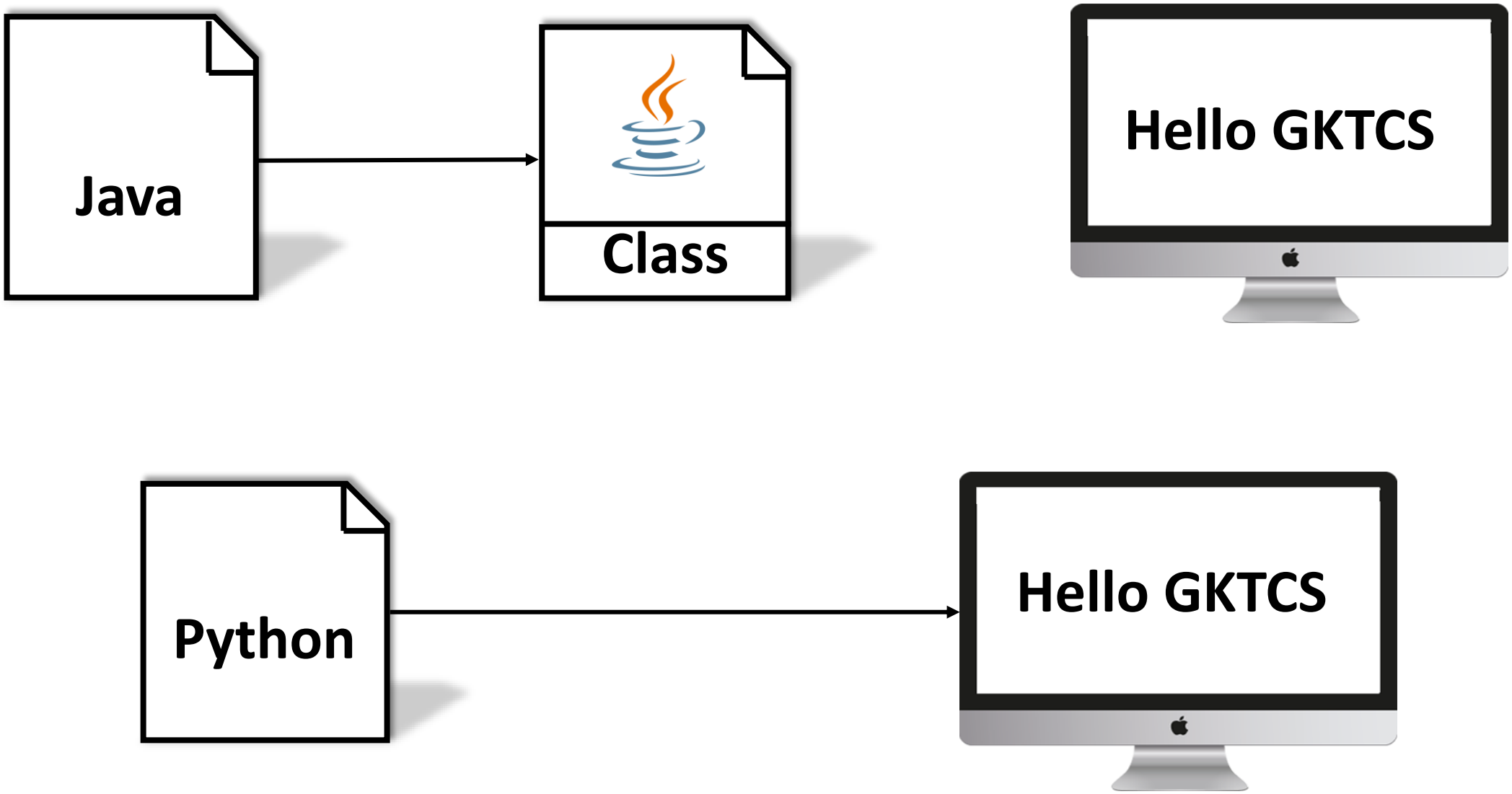
GKTCS INNOVATIONS

Java

```java
public class Main {
  public static void main(String[] args) {
      System.out.println("hello world");
  }
}
```

Python

```python
print("hello world");
```



GKTCS INNOVATIONS

PYTHON 2

VS

PYTHON 3

Legacy

Future

Library

Library

0100
0001

ASCII

Unicode

0000
0000
0100
0001

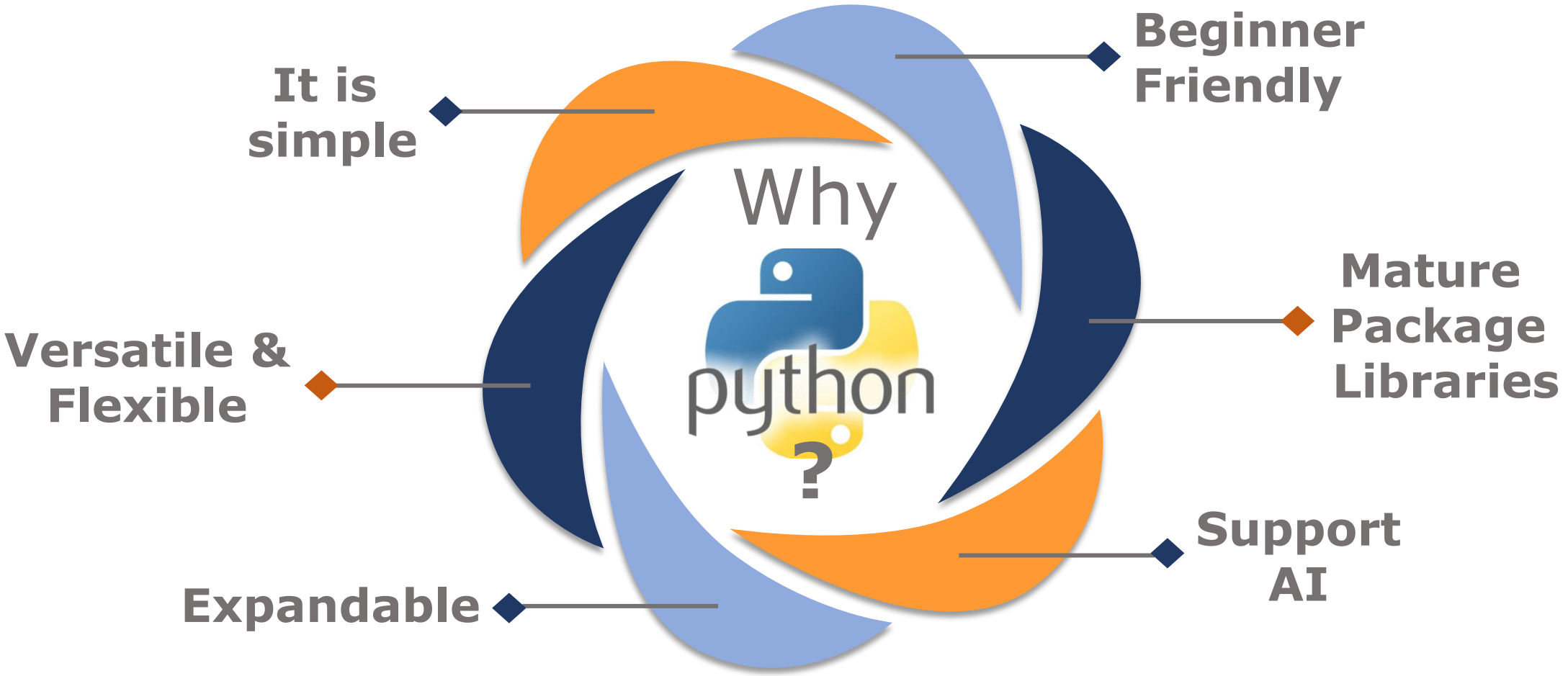
7/2=3

7/2=3.5

print “GKTCS”

print (“GKTCS”)

GKTCS INNOVATIONS

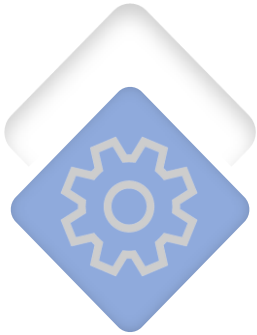


Advantages

GKTCS INNOVATIONS

Free & Open Source

Improved Productivity



Interpreted Language

Dynamically Typed

Vast Libraries Support

Object Oriented

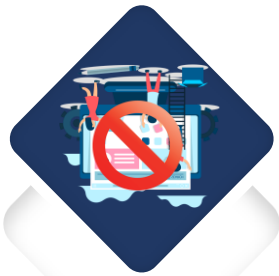
Disadvantages

GKTCS INNOVATIONS

Speed Limitations

Weak in Mobile Computing

Design Restrictions



Underdeveloped DB layers

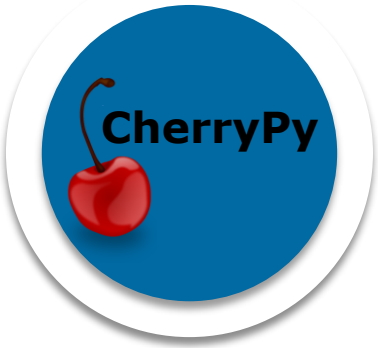
Web Frameworks

GKTCS INNOVATIONS

Flask

Web2Py

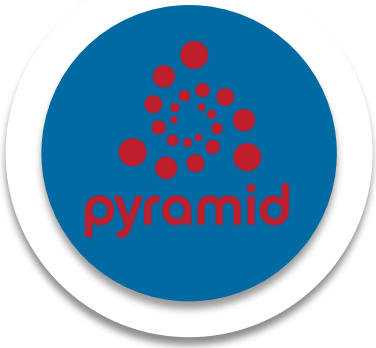
CherryPy



Django

Tornado

Pyramid



Bottle

Dash

CubicWeb

File Extensions in Python

GKTCS INNOVATIONS

01

.py

The normal extension for a Python source file

02

.pyc

The compiled bytecode

03

.pyd

A Windows DLL file

04

.pyo

A file created with optimizations

05

.pyw

A Python script for Windows

06

.pyz

A Python script archive

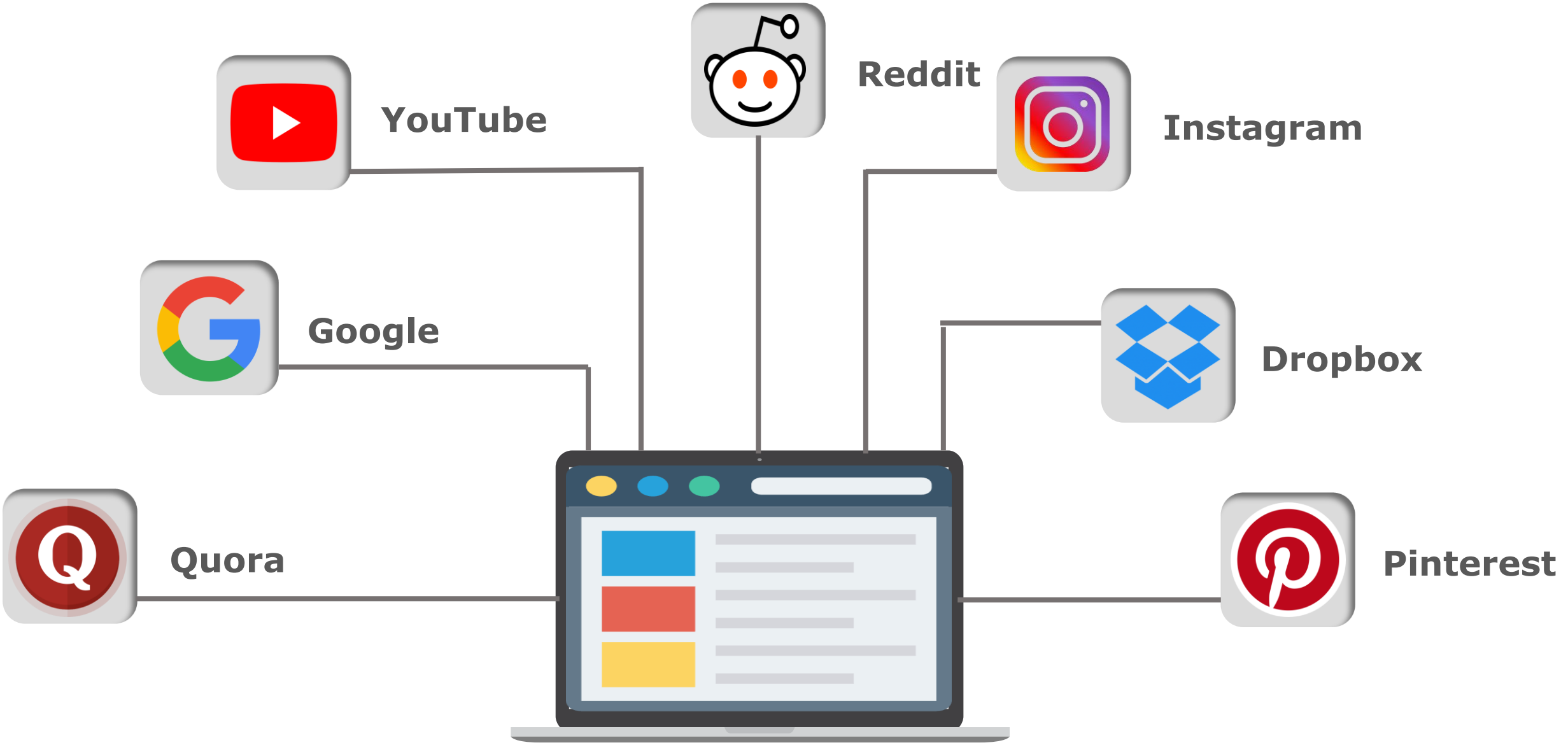
Applications Of Python

GKTCS INNOVATIONS

1. Web & Internet Development
2. Games and 3D Graphics
3. Software Development
4. Business Applications
5. Database Access
6. Network Programing

Popular website build with Python

GKTCS INNOVATIONS

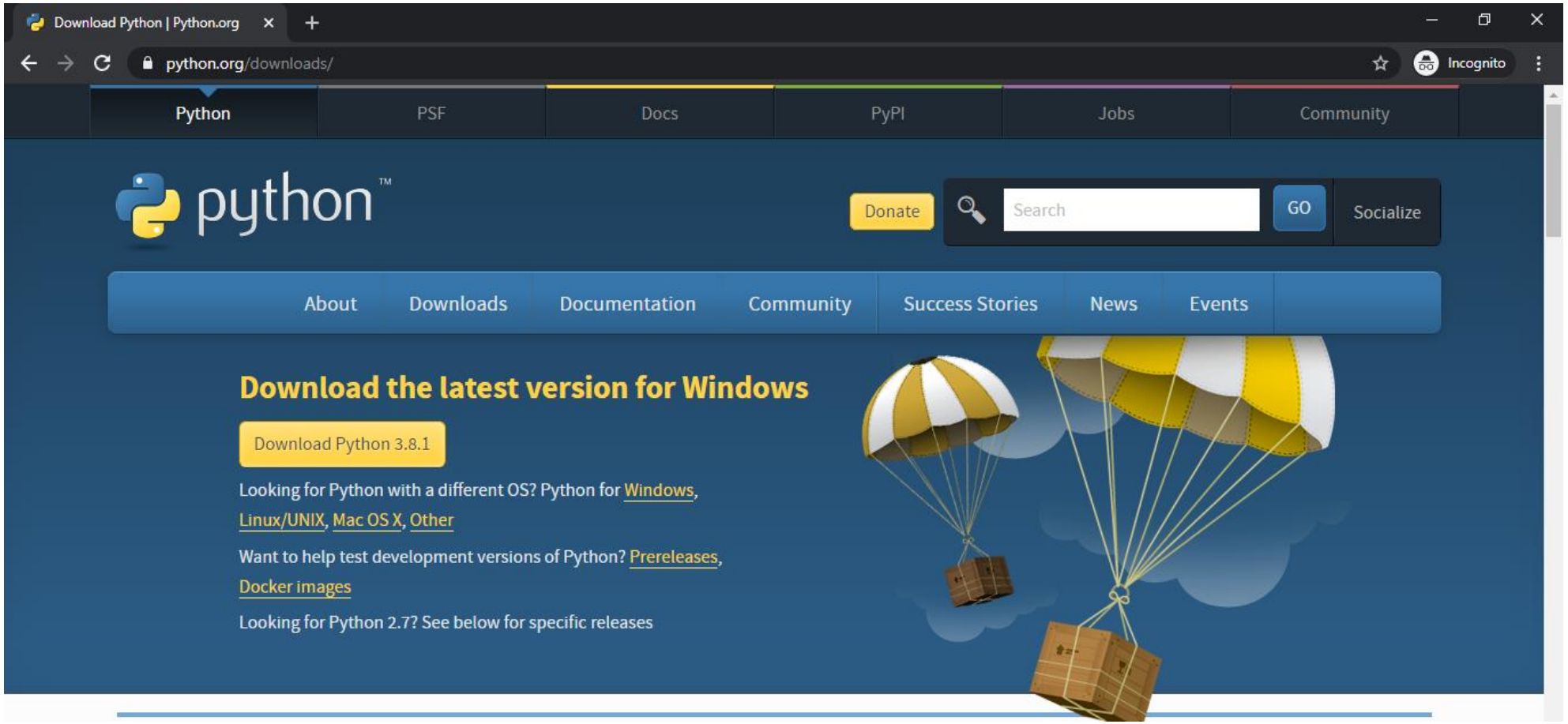


Installing Python on Windows

GKTCS INNOVATIONS

Step: 1

- To download and install Python, go to Python's official website http://www.python.org/downloads/



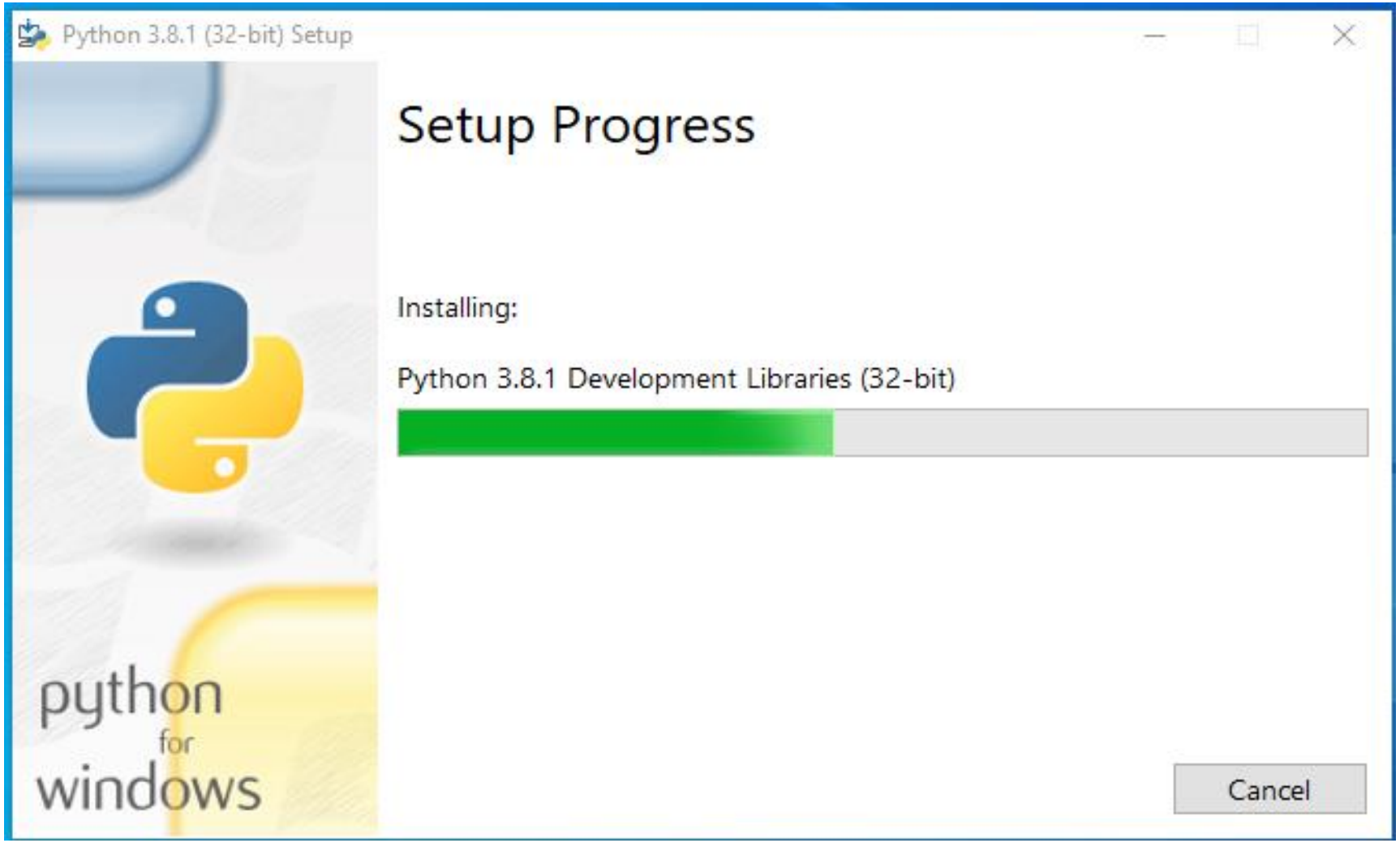
Step: 2

- When download is complete run .exe file to install Python.

Step: 3

GKTCS INNOVATIONS

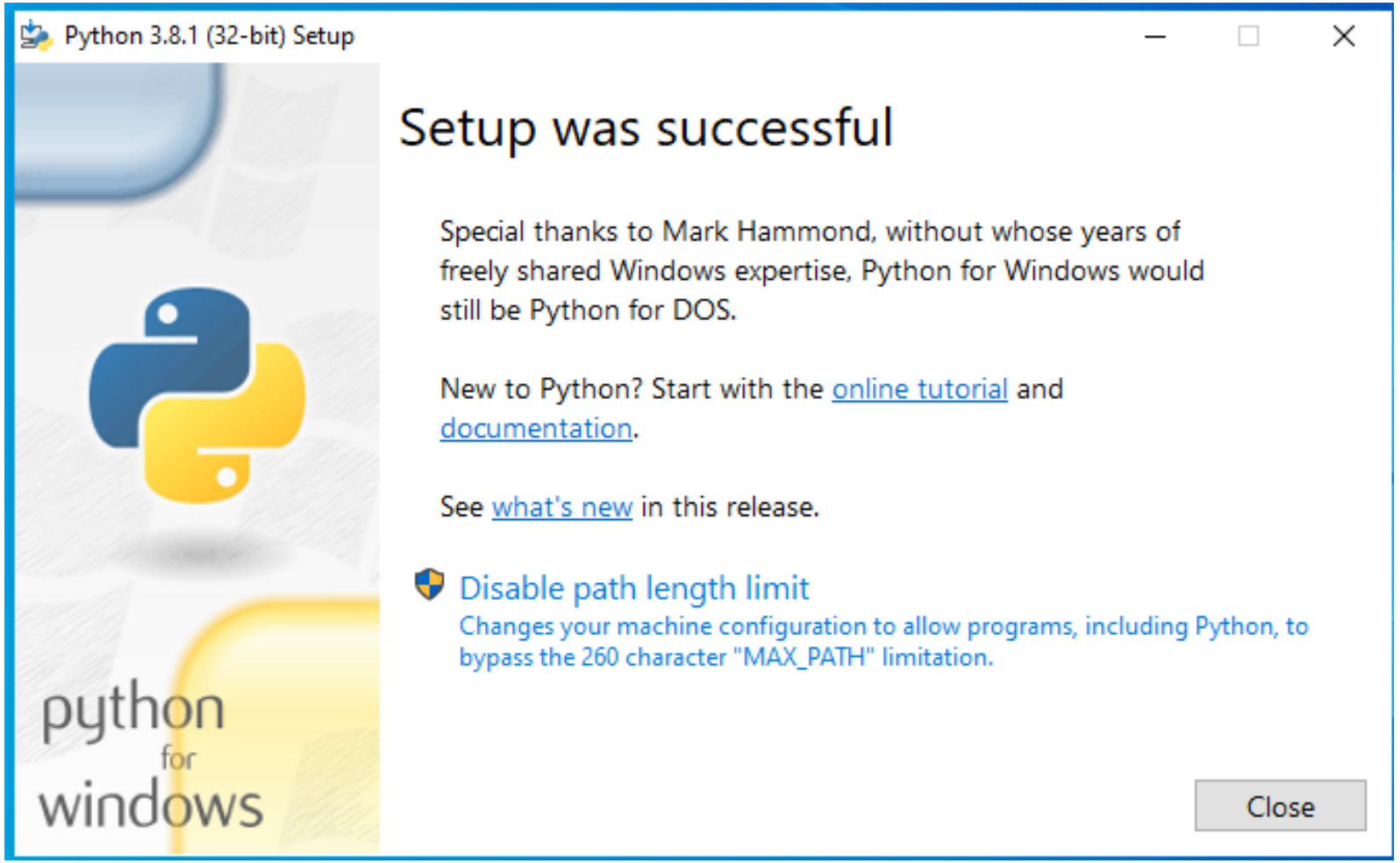
- You can see python installation.



Step: 4

GKTCS INNOVATIONS

- when installation was complete you can see message "setup was successful" on screen.



IDLE Development Environment

GKTCS INNOVATIONS

- **I**ntegrated **D**eve**L**opment **E**nvironment
- Text editor with smart indenting for creating python files.
- Menu commands for changing system settings and running files.

Python Interpreter

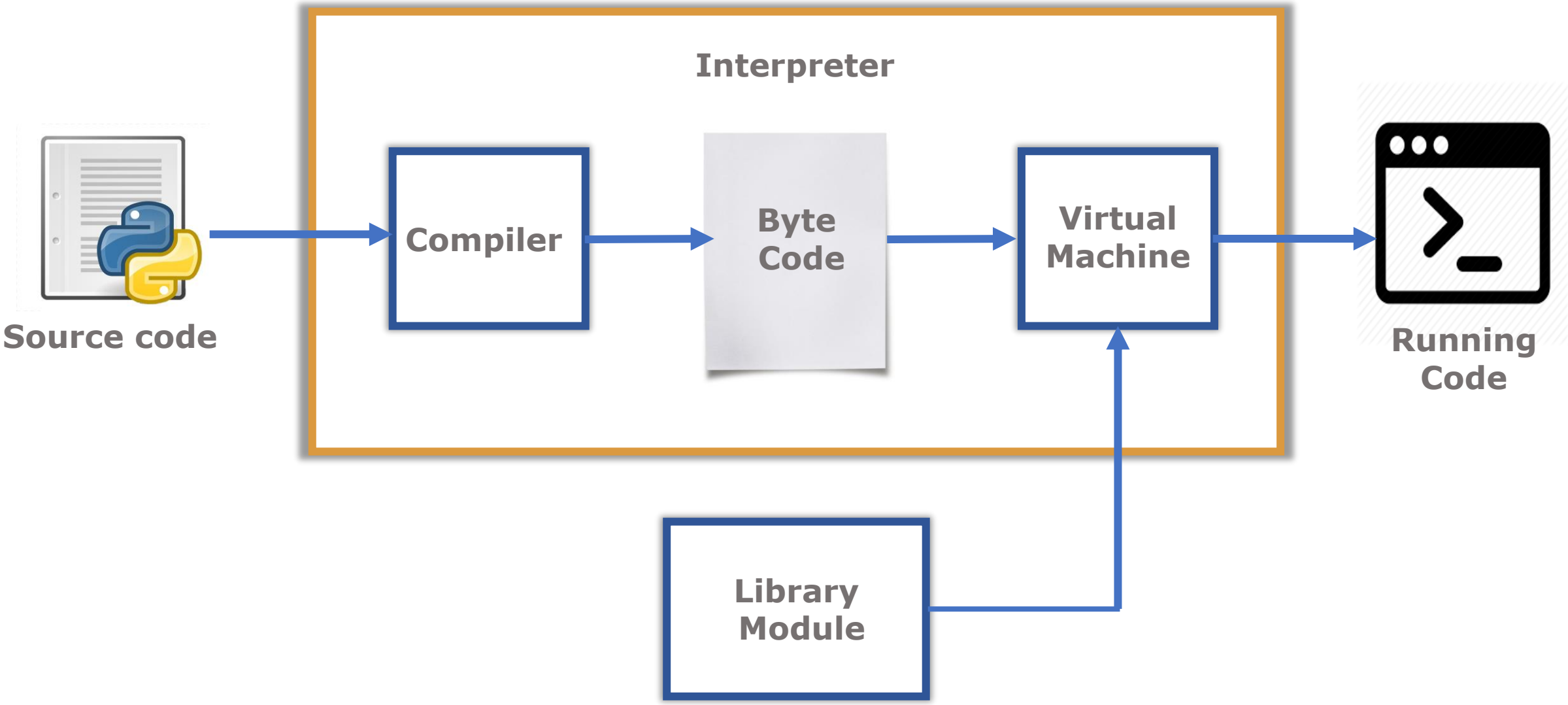
GKTCS INNOVATIONS

- Interactive Interface to python

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019,
22:39:24) [MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

How Python run's

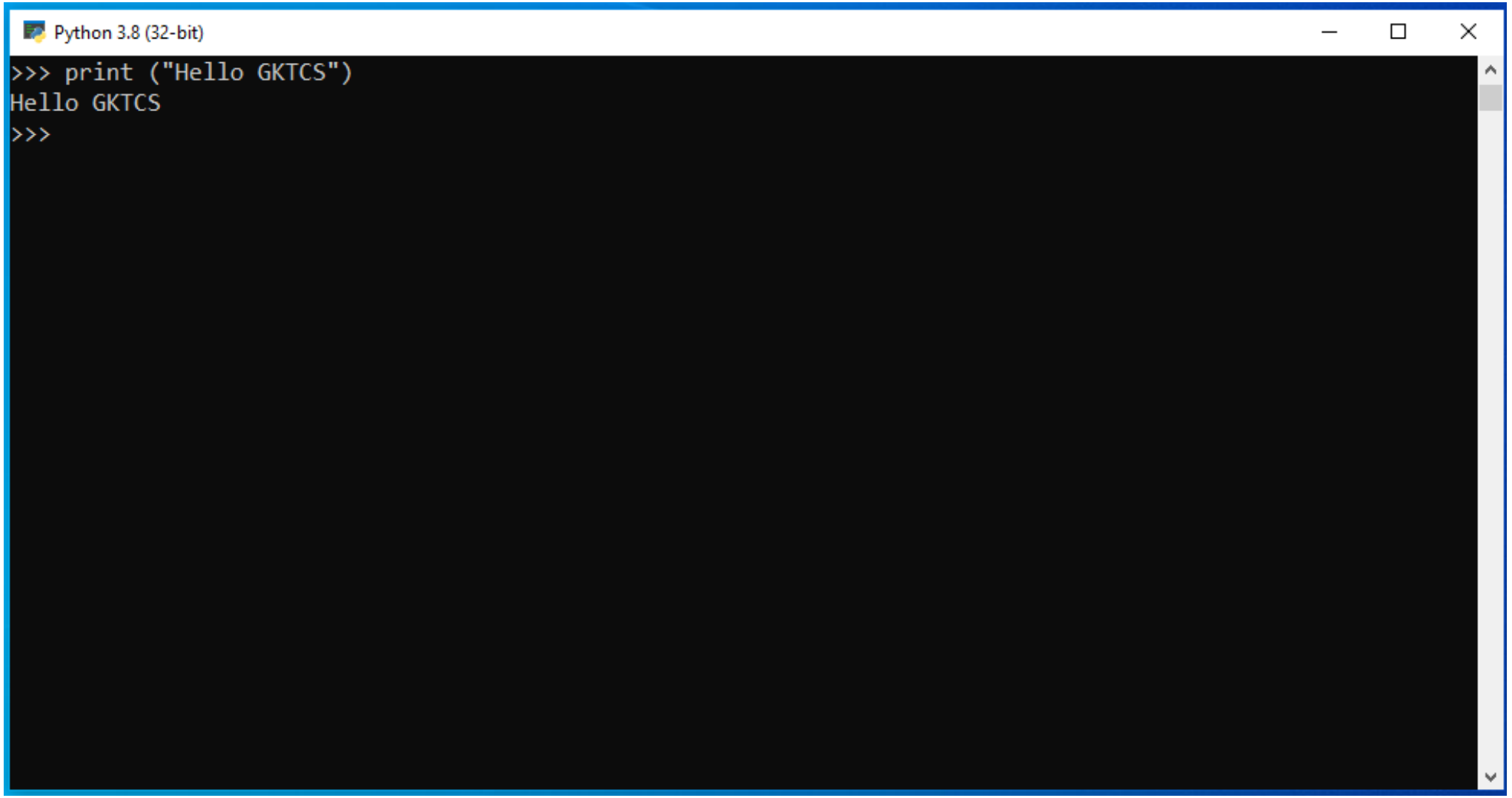
GKTCS INNOVATIONS



Running Python

GKTCS INNOVATIONS

- When you open the interpreter and type command



Datatypes

GKTCS INNOVATIONS

Text Type:	**str**
Numeric Types:	**int, float, complex**
Sequence Types:	**list, tuple, range**
Mapping Type:	**dict**
Set Types:	**set, frozenset**
Boolean Type:	**bool**
Binary Types:	**bytes, bytearray, memoryview**

Datatypes and Example

GKTCS INNOVATIONS

- When you assign a value to a variable data type is set :

int	float	str
a=10	a=2.5	a="GKTCS"

complex	list	tuple
a=2x	a = ["python", "Java", "Html"]	a = ("python", "Java", "Html")

GKTCS INNOVATIONS

dict

```
a = {
"name" : "Amit",
"age" : 25 }
```

set

```
a = { "python",
"Java", "Html" }
```

bool

```
a=True
```

complex

```
a=2x
```

bytes

```
a=b"GKTCS"
```

bytearray

```
a=bytearray(5)
```

Basic Datatypes

GKTCS INNOVATIONS

- **Integers(for numbers)**

  ```
  a=4+3    #answer is 7, integer addition
  ```

- **Floats**

  ```
  a=5/2    #answer is 2.5
  ```

- **Strings**

  Can use " " or ' ' to specify.

  "GKTCS" or 'GKTCS' are same.

String Methods

GKTCS INNOVATIONS

title()

Converts the first character of each word to upper case

upper()

Converts a string into upper case

lower()

Converts a string into lower case

isdigit()

Returns True if all characters in the string are digits

isupper()

Returns True if all characters in the string are in upper case.

swapcase()

Swaps cases, lower case becomes upper case and vice versa

Variables

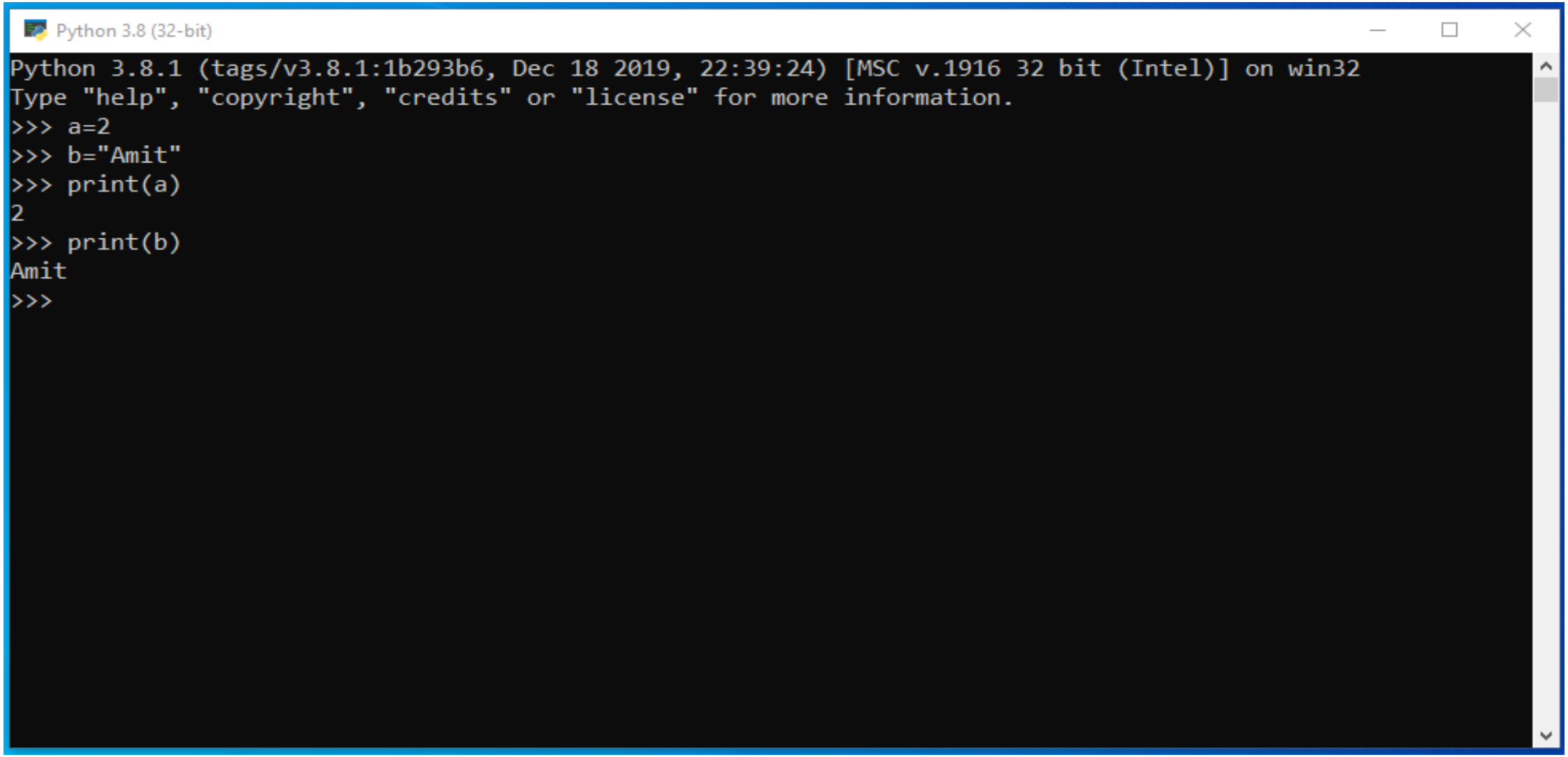
GKTCS INNOVATIONS

- Variables are use to store data values.
- A variable is created when you assign a value to it.

```
x = 2
y = "Amit"
print(x)
print(y)
```

Output

GKTCS INNOVATIONS



Rules for Python variables:

GKTCS INNOVATIONS

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Comments

GKTCS INNOVATIONS

Comments can be used to improve readability of the code.

1) Single-line comments

Simply create a line starting with the hash (#) character

```
#This would be a single line comment in Python
```

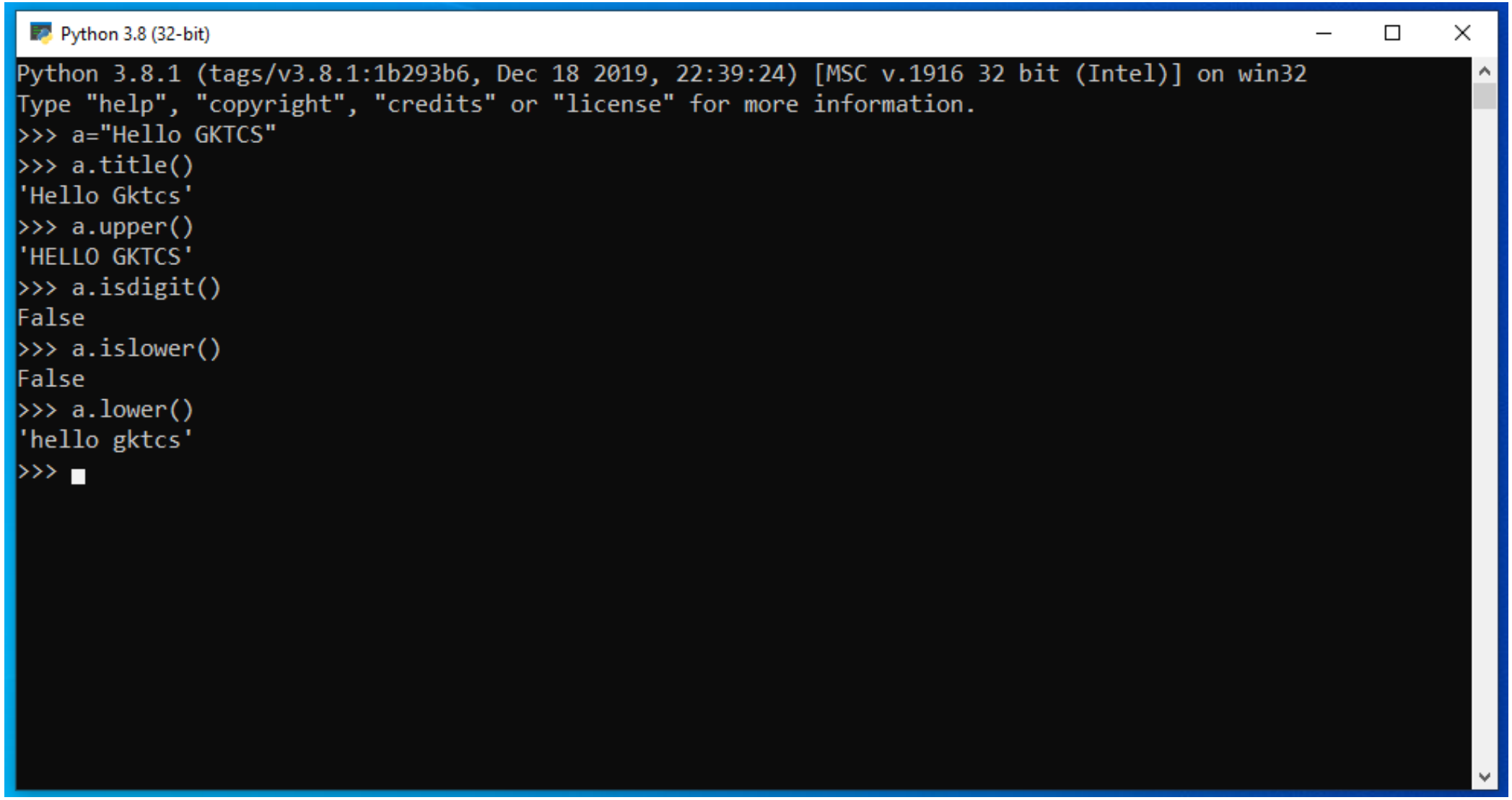
2) Multi-line comments

Created by adding a delimiter (""") on each end of the comment.

```
""" This would be a multiline comment in Python that
describes your code, your day, or anything you want it to """
```

Output

GKTCS INNOVATIONS



How to install Python on Windows?

Step 1: Select Version to Install Python

Visit the official page for Python https://www.python.org/downloads/ on the Windows operating system. Locate a reliable version of Python 3, preferably version 3.10.11, which was used in testing this tutorial. Choose the correct link for your device from the options provided: either **Windows installer (64-bit) or Windows installer (32-bit)** and proceed to download the executable file.

Python Releases for Windows

- Latest Python 3 Release - Python 3.11.3

Stable Releases

- Python 3.10.11 - April 5, 2023

  Note that Python 3.10.11 *cannot* be used on Windows 7 or earlier.

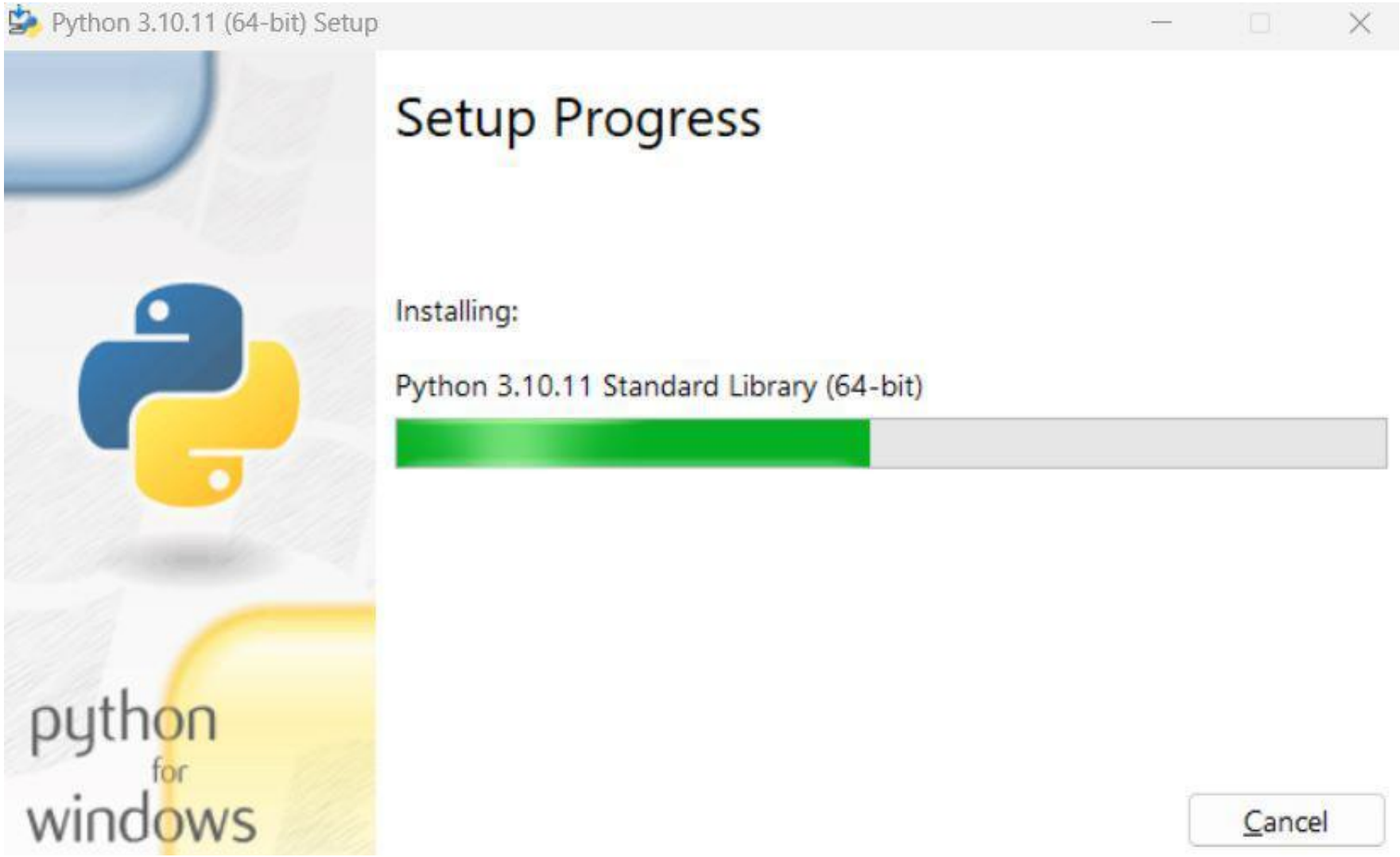
  - Download Windows embeddable package (32-bit)
  - Download Windows embeddable package (64-bit)
  - Download Windows help file
  - Download Windows installer (32 -bit)
  - Download Windows installer (64-bit)

Step 2: Downloading the Python Installer

Once you have downloaded the installer, open the .exe file, such as p**ython-3.10.11-amd64.exe**, by double-clicking it to launch the Python installer. Choose the option to Install the launcher for all users by checking the corresponding checkbox, so that all users of the computer can access the Python launcher application.Enable users to run Python from the command line by checking the Add python.exe to PATH checkbox.

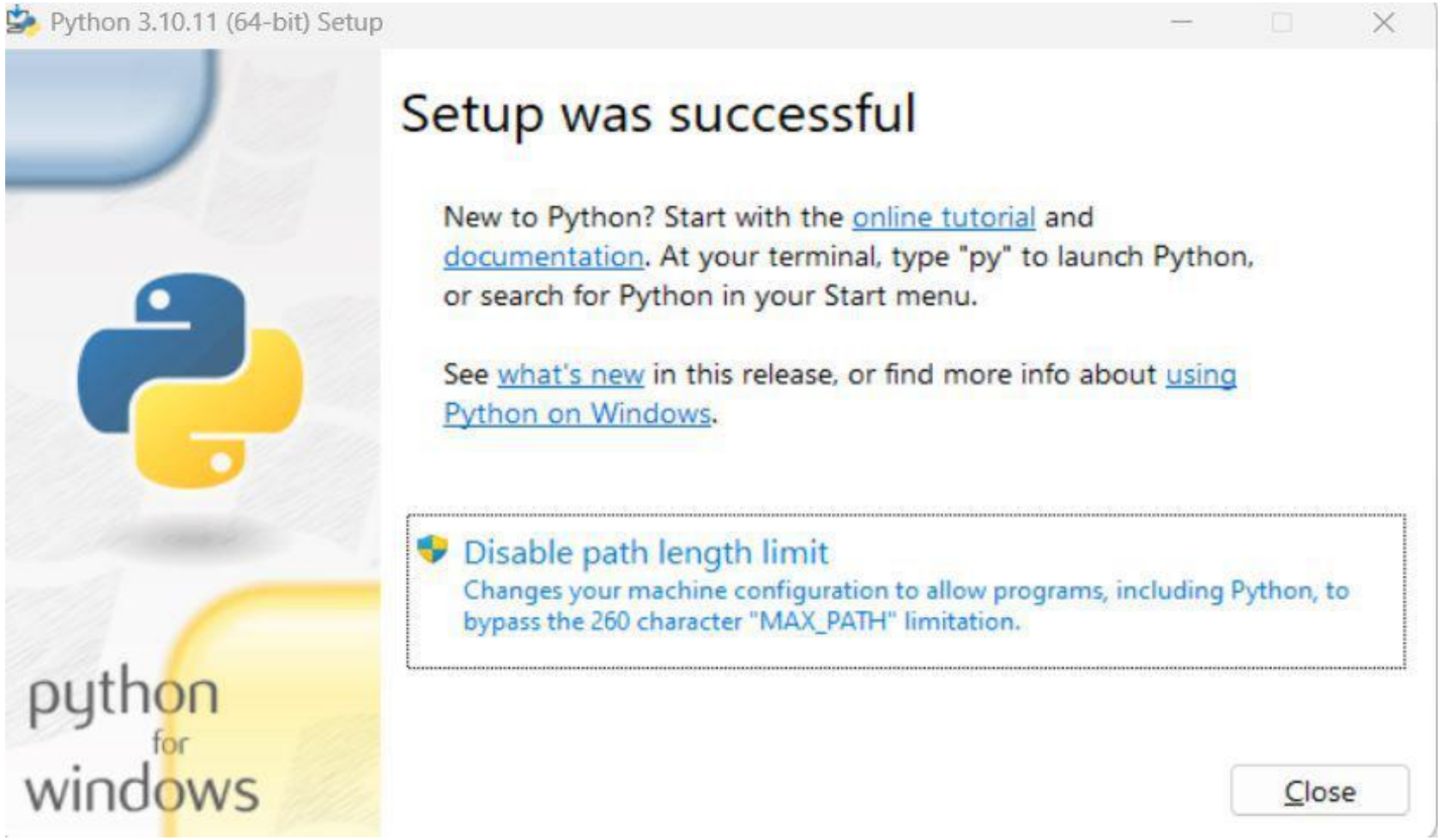

Python 3.10.11 (64-bit) Setup
Install Python 3.10.11 (64-bit)
Select Install Now to install Python with default settings, or choose
Customize to enable or disable features.
Install Now
C:\Users\ashub\AppData\Local\Programs\Python\Python310
Includes IDLE, pip and documentation
Creates shortcuts and file associations
Customize installation
Choose location and features
python
for
windows
Use admin privileges when installing py.exe
Add python.exe to PATH
Cancel

After Clicking the **Install Now Button** the setup will start installing Python on your Windows system. You will see a window like this.



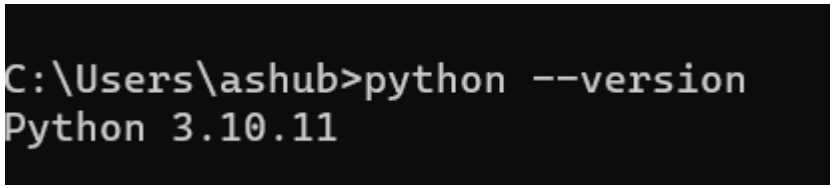
Step 3: Running the Executable Installer

After completing the setup. Python will be installed on your Windows system. You will see a successful message.



Step 4: Verify the Python Installation in Windows

Close the window after successful installation of Python. You can check if the installation of Python was successful by using either the command line or the I**ntegrated Development Environment (IDLE)**, which you may have installed. To access the command line, click on the Start menu and type “cmd” in the search bar. Then click on Command Prompt.



You can also check the version of Python by opening the IDLE application. Go to Start and enter IDLE in the search bar and then click the IDLE app, for example, **IDLE (Python 3.10.11 64-bit)**. If you can see the Python IDLE window then you are successfully able to download and installed

Python on Windows.


IDLE Shell 3.10.11
File Edit Shell Debug Options Window Help
Python 3.10.11 (tags/v3.10.11:7d4cc5a, Apr 5 2023, 00:38:17) [MSC v.1929 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>

How to Install Anaconda on Windows?

- Anaconda is an open-source software that contains Jupyter, spyder, etc that are used for large data processing, data analytics, heavy scientific computing. Anaconda works for R and python programming language. Spyder(sub-application of Anaconda) is used for python. Opencv for python will work in spyder. Package versions are managed by the package management system called conda. To begin working with Anaconda, one must get it installed first. Follow the below instructions to Download and install Anaconda on your system:
  Download and install Anaconda:
- Head over to anaconda.com and install the latest version of Anaconda. Make sure to download the “Python 3.7 Version” for the appropriate architecture.

Windows | macOS | Linux

Anaconda 2019.10 for Windows Installer

Python 3.7 version

Download

64-Bit Graphical Installer (462 MB)

32-Bit Graphical Installer (410 MB)

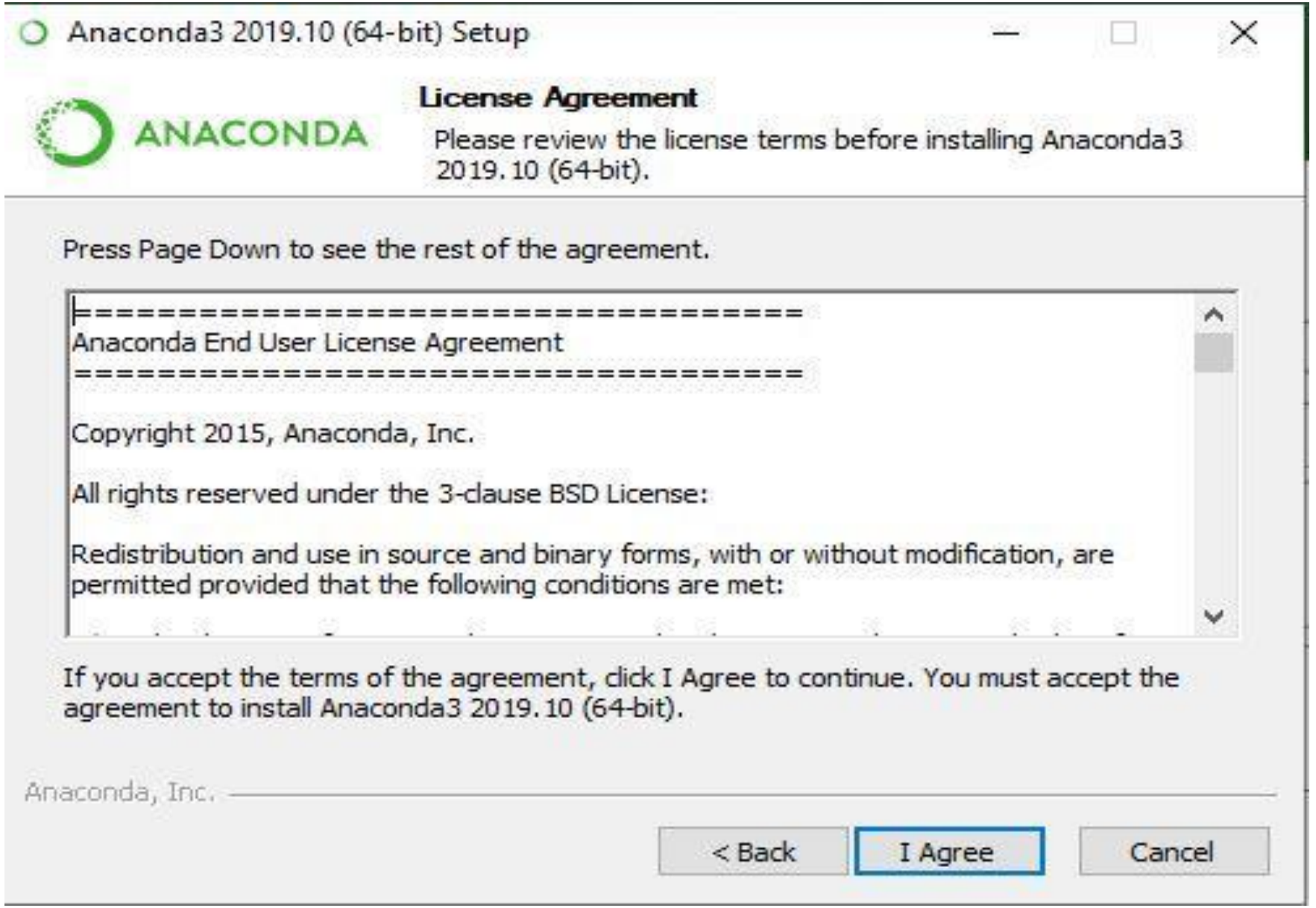
Python 2.7 version

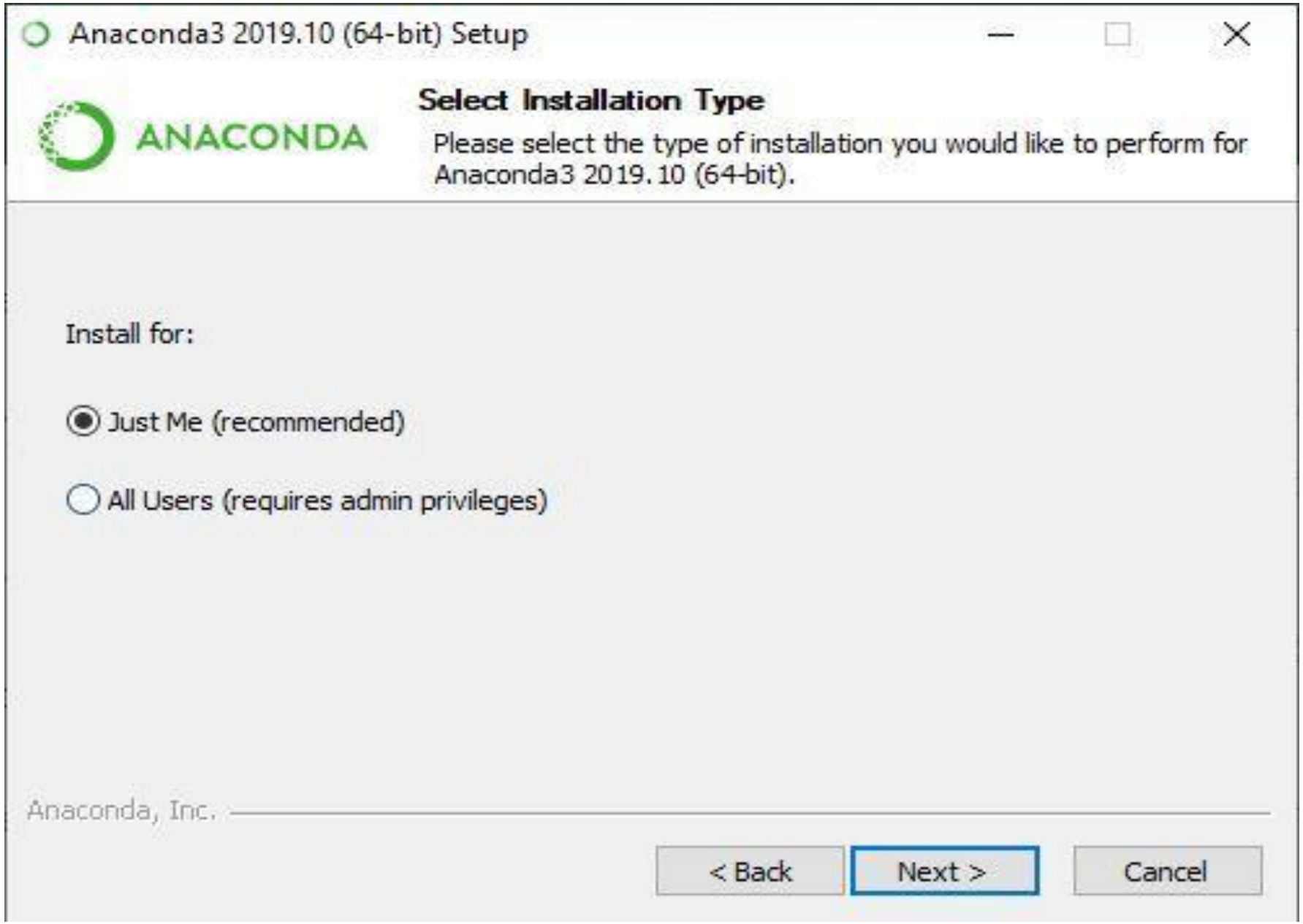
Download

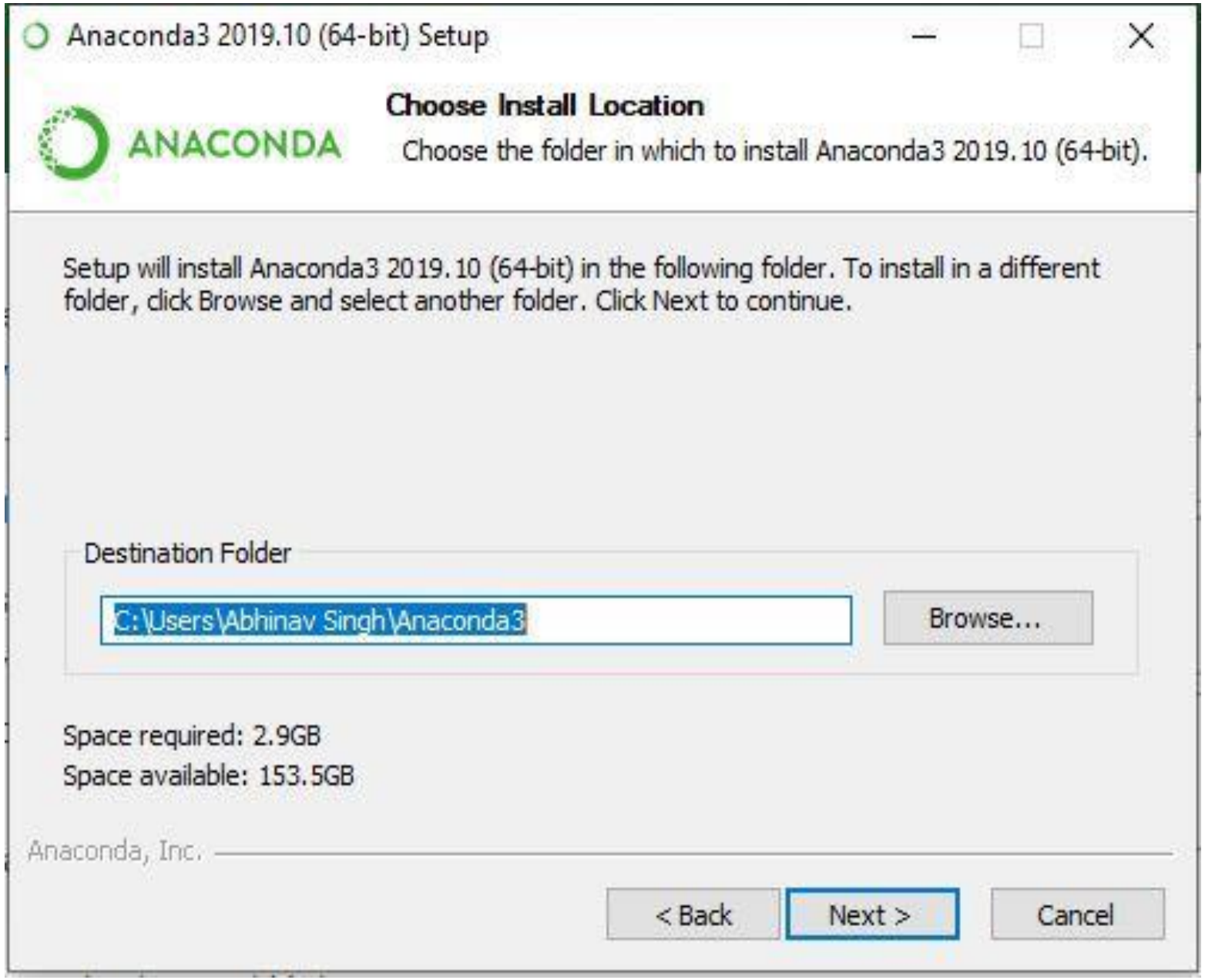
64-Bit Graphical Installer (413 MB)

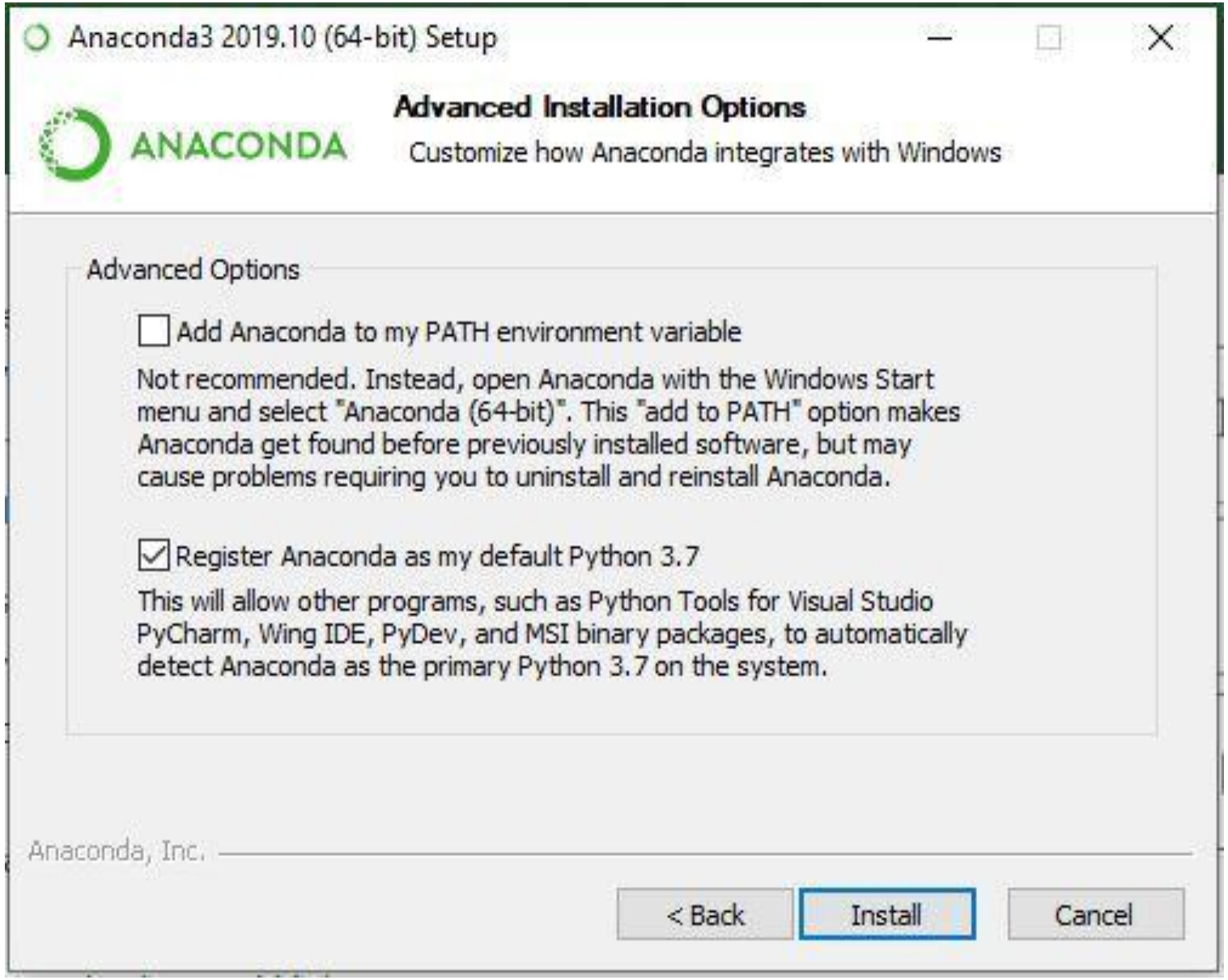
32-Bit Graphical Installer (356 MB)

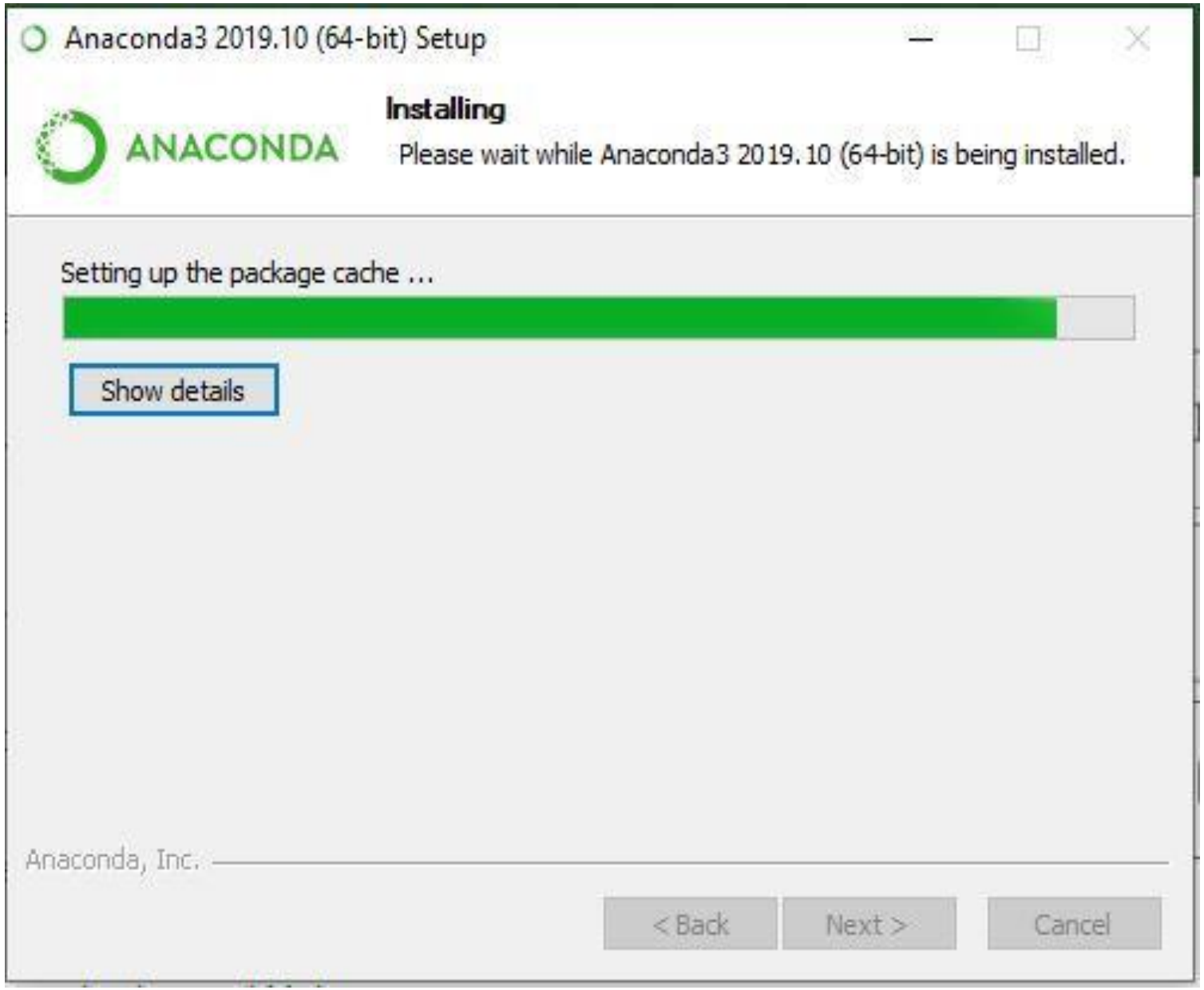

Anaconda3 2019.10 (64-bit) Setup
ANACONDA.
Welcome to Anaconda3 2019.10 (64-bit) Setup
Setup will guide you through the installation of Anaconda3 2019.10 (64-bit).
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Next >
Cancel


Anaconda3 2019.10 (64-bit) Setup
ANACONDA
License Agreement
Please review the license terms before installing Anaconda3 2019.10 (64-bit).
Press Page Down to see the rest of the agreement.
==========
Anaconda End User License Agreement
==========
Copyright 2015, Anaconda, Inc.
All rights reserved under the 3-clause BSD License:
Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 2019.10 (64-bit).
Anaconda, Inc.
< Back
I Agree
Cancel

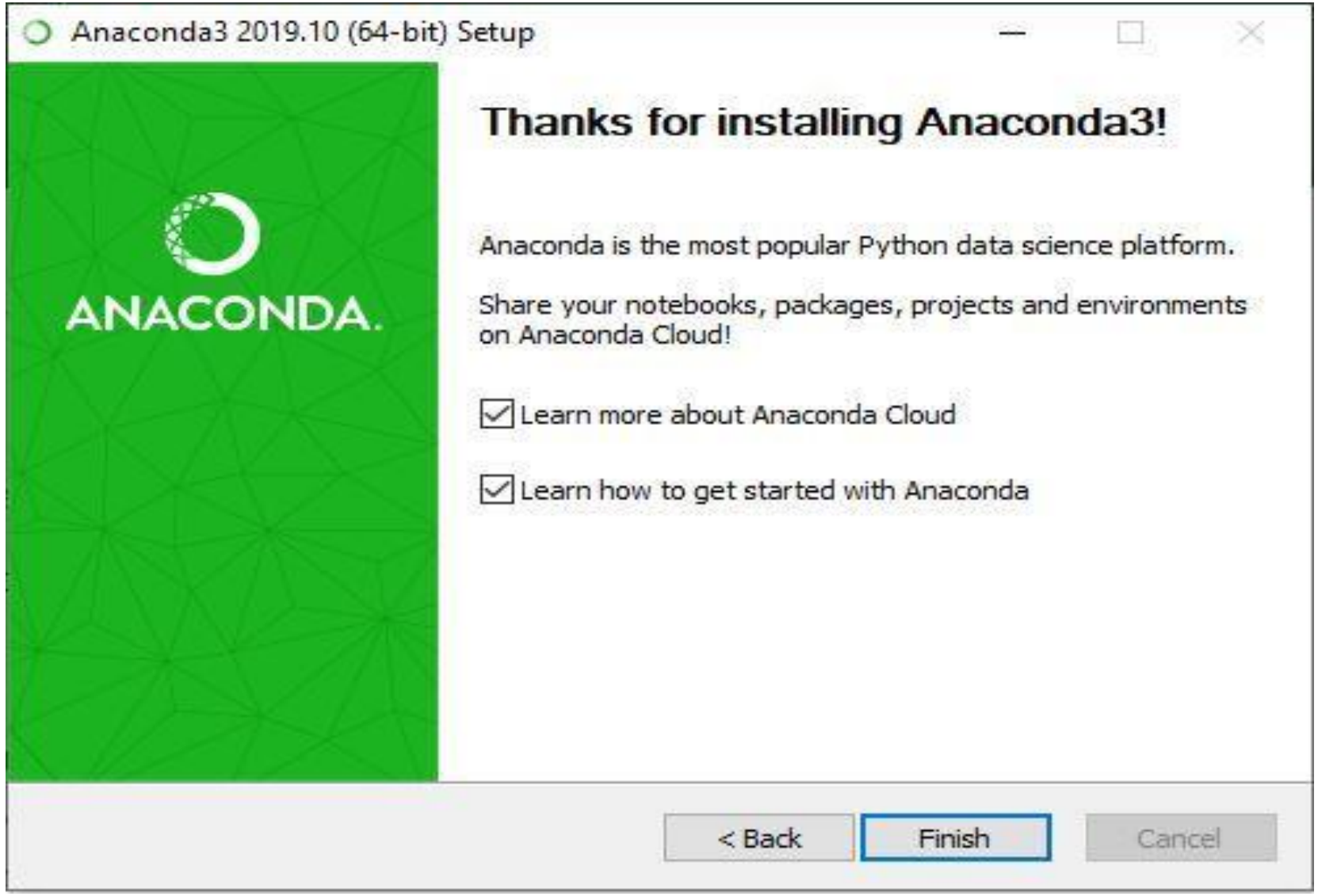

Anaconda3 2019.10 (64-bit) Setup
ANACONDA
Select Installation Type
Please select the type of installation you would like to perform for Anaconda3 2019.10 (64-bit).
Install for:
Just Me (recommended)
All Users (requires admin privileges)
Anaconda, Inc.
< Back
Next >
Cancel

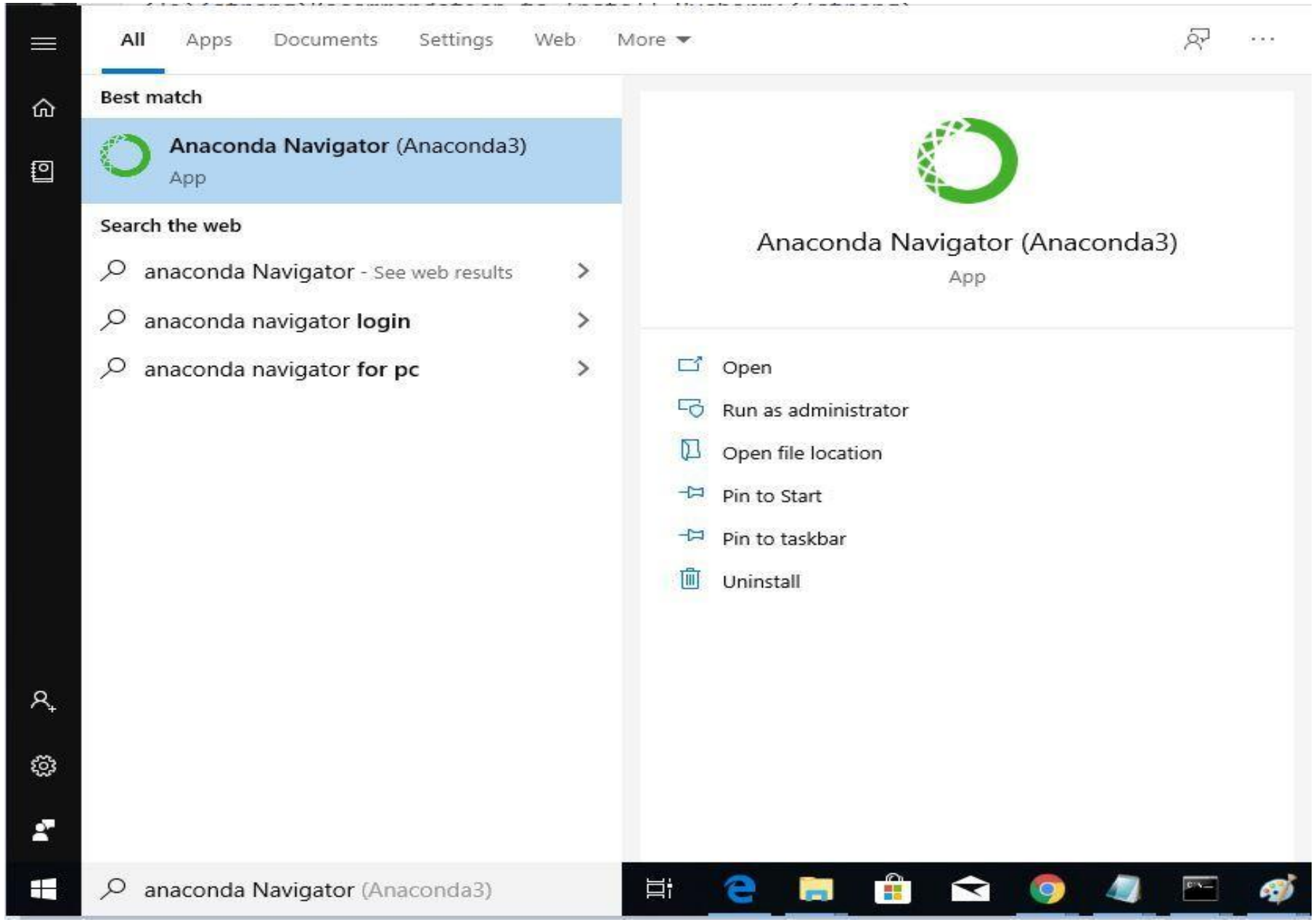

Anaconda3 2019.10 (64-bit) Setup
ANACONDA
Choose Install Location
Choose the folder in which to install Anaconda3 2019.10 (64-bit).
Setup will install Anaconda3 2019.10 (64-bit) in the following folder. To install in a different folder, click Browse and select another folder. Click Next to continue.
Destination Folder
C:\Users\Abhinav Singh\Anaconda3
Browse...
Space required: 2.9GB
Space available: 153.5GB
Anaconda, Inc.
< Back
Next >
Cancel

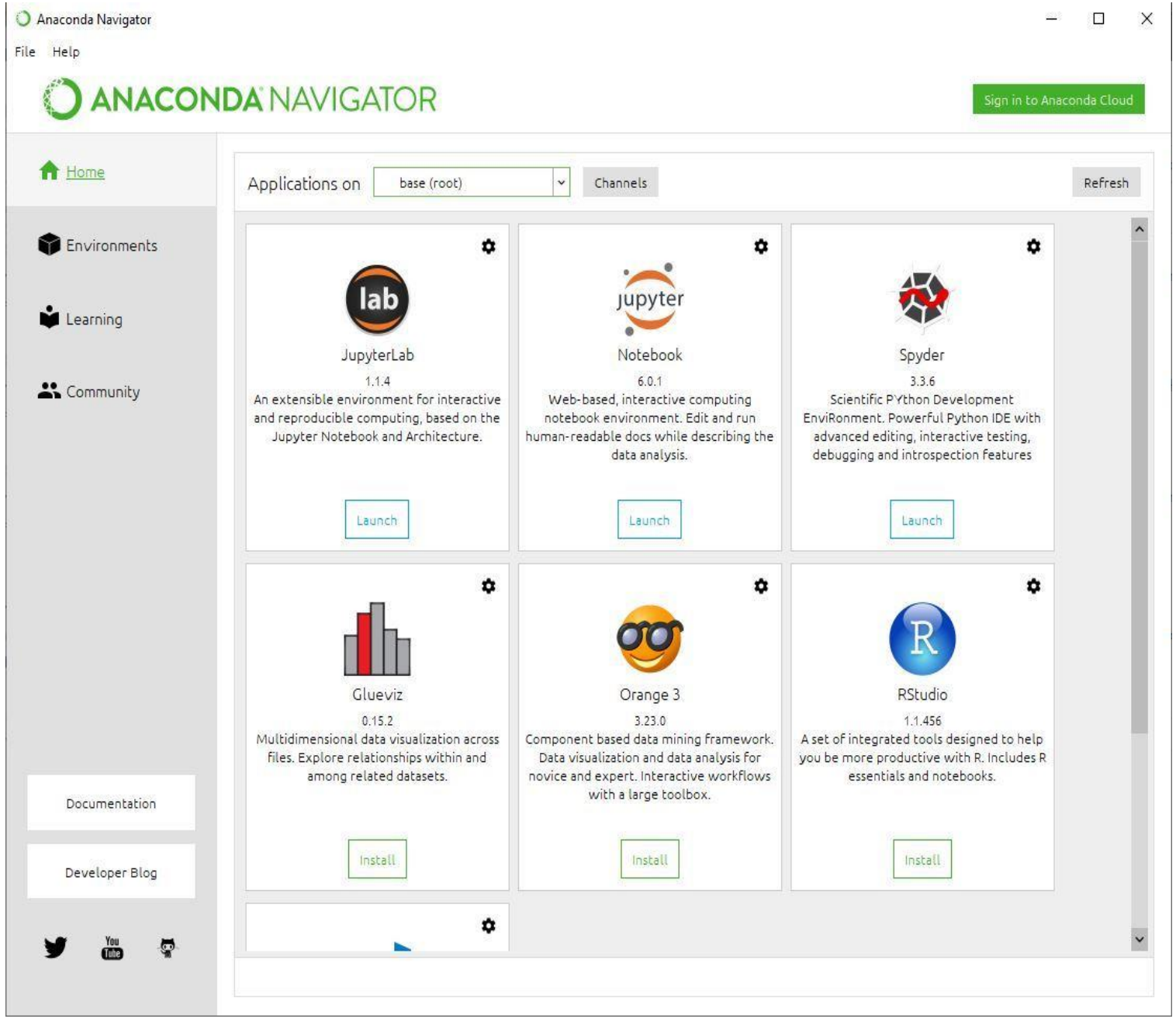

Anaconda3 2019.10 (64-bit) Setup
ANACONDA
Advanced Installation Options
Customize how Anaconda integrates with Windows
Advanced Options
Add Anaconda to my PATH environment variable
Not recommended. Instead, open Anaconda with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.
Register Anaconda as my default Python 3.7
This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.7 on the system.
Anaconda, Inc.
< Back
Install
Cancel


Anaconda3 2019.10 (64-bit) Setup
ANACONDA
Installing
Please wait while Anaconda3 2019.10 (64-bit) is being installed.
Setting up the package cache ...
Show details
Anaconda, Inc.
< Back
Next >
Cancel


Anaconda3 2019.10 (64-bit) Setup
ANACONDA
Anaconda3 2019.10 (64-bit)
Anaconda + JetBrains
Anaconda and JetBrains are working together to bring you Anaconda-powered environments tightly integrated in the PyCharm IDE.
PyCharm for Anaconda is available at:
https://www.anaconda.com/pycharm
ANACONDA.
PC
Anaconda, Inc.
< Back
Next >
Cancel


Anaconda3 2019.10 (64-bit) Setup
ANACONDA.
Thanks for installing Anaconda3!
Anaconda is the most popular Python data science platform.
Share your notebooks, packages, projects and environments on Anaconda Cloud!
Learn more about Anaconda Cloud
Learn how to get started with Anaconda
< Back
Finish
Cancel


All
Apps
Documents
Settings
Web
More
Best match
Anaconda Navigator (Anaconda3)
App
Search the web
anaconda Navigator - See web results
anaconda navigator login
anaconda navigator for pc
Anaconda Navigator (Anaconda3)
App
Open
Run as administrator
Open file location
Pin to Start
Pin to taskbar
Uninstall
anaconda Navigator (Anaconda3)


Anaconda Navigator
File Help
ANACONDA NAVIGATOR
Sign in to Anaconda Cloud
Home
Environments
Learning
Community
Documentation
Developer Blog
Applications on
base (root)
Channels
Refresh
JupyterLab
1.1.4
An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture.
Launch
Notebook
6.0.1
Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis.
Launch
Spyder
3.3.6
Scientific PYthon Development EnviRonment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features
Launch
Glueviz
0.15.2
Multidimensional data visualization across files. Explore relationships within and among related datasets.
Install
Orange 3
3.23.0
Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox.
Install
RStudio
1.1.456
A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.
Install

History of Python

- Python was first developed by ***Guido van Rossum*** in the late 80‟s and early 90‟s at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- ➢ It has been derived from many languages such as ABC, Modula-3, C, C++, Algol-68, SmallTalk, UNIX shell and other scripting languages.
- ➢ Since early 90‟s Python has been improved tremendously. Its version 1.0 was released in 1991, which introduced several new functional programming tools.

- While version 2.0included list comprehension was released in 2000 by the Be Open Python Labs team.
- ➢ Python 2.7 which is still used today will be supported till 2020.
- ➢ Currently Python 3.6.4 is already available. The newer versions have better features like flexible string representation e.t.c,
- ➢ Although Python is copyrighted, its source code is available under GNU General Public License (GPL) like that Perl.
- ➢ Python is currently maintained by a core development team at the institute which is directed by Guido Van Rossum.
- ➢ These days, from data to web development, Python has emerged as very powerful and popular language. It would be surprising to know that python is actually older than Java, R and JavaScrip

MODULES

Python Module is a file that contains built-in functions, classes,**its** and variables. There are many **Python modules**, each with its specific work.

A Python module is a file containing Python definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code.

Grouping related code into a module makes the code easier to understand and use. It also makes the code logically organized.

Create a Python Module

To create a Python module, write the desired code and save that in a file with **.py** extension. Let's understand it better with an example:

Example:

Let's create a simple calc.py in which we define two functions, one **add** and another **subtract**.

```
# A simple module, calc.py
def add(x, y):
        return (x+y)

def subtract(x, y):
        return (x-y)
```

Import module in Python

- We can import the functions, and classes defined in a module to another module using the **import statement** in some other Python source file.
- When the interpreter encounters an import statement, it imports the module if the module is present in the search path.

Syntax to Import Module in Python

```
import module


# importing module calc.py
import calc
print(calc.add(10, 2))
```

Output:

```
12
```

Python Import From Module

Python's from statement lets you import specific attributes from a module without importing the module as a whole.

```
# importing sqrt() and factorial from the
# module math
from math import sqrt, factorial
# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(sqrt(16))
print(factorial(6))
```

Output:

```
4.0
720
```

Directories List for Modules

Here, sys.path is a built-in variable within the sys module. It contains a list of directories that the interpreter will search for the required module.

```
# importing sys module
import sys
# importing sys.path
print(sys.path)
```

Output:

['/home/nikhil/Desktop/gfg', '/usr/lib/python38.zip', '/usr/lib/python3.8', '/usr/lib/python3.8/lib-dynload', '', '/home/nikhil/.local/lib/python3.8/site-packages', '/usr/local/lib/python3.8/dist-packages', '/usr/lib/python3/dist-packages', '/usr/local/lib/python3.8/dist-packages/IPython/extensions', '/home/nikhil/.ipython']

Renaming the Python Module

We can rename the module while importing it using the keyword.

Syntax: Import **Module_name** as **Alias_name**

```
# importing sqrt() and factorial from the
# module math
import math as mt

# if we simply do "import math", then
# math.sqrt(16) and math.factorial()
# are required.
print(mt.sqrt(16))
print(mt.factorial(6))
```

Output

```
4.0
720
```

Python Built-in modules

There are several built-in modules in Python, which you can import whenever you like.

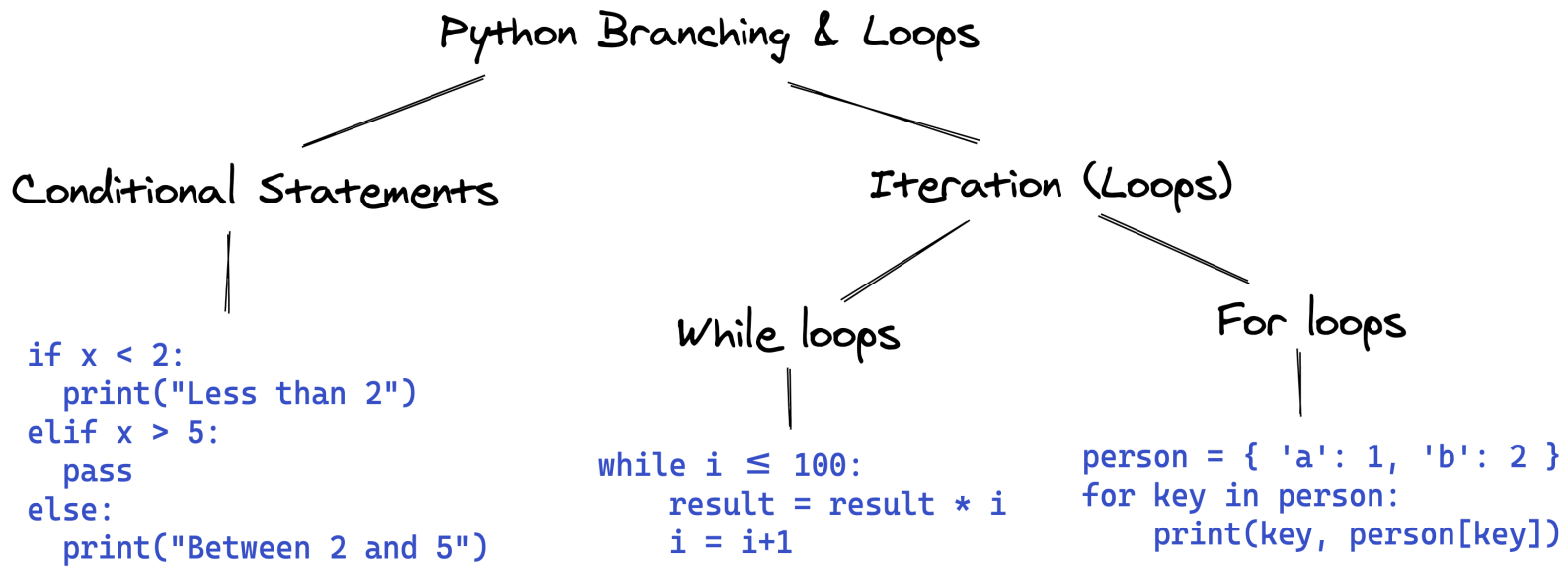
```
# importing built-in module math
import math

# using square root(sqrt) function contained
# in math module
print(math.sqrt(25))

# using pi function contained in math module
print(math.pi)

# 2 radians = 114.59 degrees
print(math.degrees(2))

# 60 degrees = 1.04 radians
print(math.radians(60))
```

```
print(math.factorial(4))

# importing built in module random
import random

# printing random integer between 0 and 5
print(random.randint(0, 5))

# print random floating point number between 0 and 1
print(random.random())

# random number between 0 and 100
print(random.random() * 100)

List = [1, 4, True, 800, "python", 27, "hello"]

# using choice function in random module for choosing
# a random element from a set such as a list
print(random.choice(List))
```

```
# Sine of 2 radians
print(math.sin(2))

# Cosine of 0.5 radians
print(math.cos(0.5))

# Tangent of 0.23 radians
print(math.tan(0.23))

# 1 * 2 * 3 * 4 = 24
```

```
importing built in module datetime
import datetime
from datetime import date
import time

# Returns the number of seconds since the
# Unix Epoch, January 1st 1970
print(time.time())

# Converts a number of seconds to a date object
print(date.fromtimestamp(454554))
```

Branching using Conditional Statements and Loops in Python



It covers the following topics:

- Branching with if, else and elif
- Nested conditions and if expressions
- Iteration with while loops
- Iterating over containers with for loops
- Nested loops, break and continue statements

Branching in Python programming

Branching in Python enables us to execute different statements based on different conditions. We can define multiple conditions using different methods, such as:

1. "**if-else**" statements, where a specific block of code is executed if a condition is met and another block of code is executed if the condition is not met.
2. Multiple "**if**" statements, where multiple conditions are checked, and the corresponding block of code is executed for the first true condition.
3. "**if-elif**" statements, where multiple conditions are checked in sequence, and the first true condition's corresponding block of code is executed.

If-else Statements

We can think of this type of conditioning as two opposite statements. For example, we collect the age of a passenger and based on that, we provide information about whether they are eligible for a window seat or not. If one statement is true, the other will be false, and vice versa.

```
age = 19

if (age > 18):
     print("You are eligible to take a window seat")
else:
     print("You are not eligible to take a window seat")
print("Enjoy your journey")

## Output
# You are eligible to take a window seat
# Enjoy your journey
```

Multiple “if” Statements

Here we treat all the conditions independent of each other. If we take the same example, we could have covered the exact execution sequence with multiple “if” statements. Let’s see,

```
age = 19

if (age > 18):
     print("You are eligible to take a window seat")
if (age <= 18):
     print("You are not eligible to take a window seat")

print("Enjoy your journey")
```

```
age = 19

if (age > 18):
     print("You are eligible to take a window seat")
if (age <= 18):
     print("You are not eligible to take a window seat")
if (age < 2):
     print("No seat allowed")

print("Enjoy your journey")
```

“if-elif” statements

An elif statement is a short form of “else if” which allows us to check additional conditions if the proceeding condition is false. If the proceeding condition is true, they will skip all the elif conditions. We can concatenate multiple additional conditions using multiple elif statements. Let’s see the code example snippet:

```
age = 15

if (age > 18):
     print("You are eligible to take a window seat")
elif (age < 2):
     print("No seat allowed")
else:
     print("You are not eligible to take a window seat")



## Output
#You are not eligible to take a window seat
```

Loops in Python – For, While and Nested Loops

- Python programming language provides two types of loops – **For loop** and **While loop** to handle looping requirements. Python provides three ways for executing the loops.

While Loop in Python

- In Python, a while loop is used to execute a block of statements repeatedly until a given condition is satisfied. When the condition becomes false, the line immediately after the loop in the program is executed.

While Loop Syntax:

```
while expression:
statement(s)
```

Example of Python While Loop

```
count = 0
while (count < 3):
        count = count + 1
        print("Hello Geek")
```

Output

```
Hello Geek
Hello Geek
Hello Geek
```

else statement with While Loop in Using Python

- The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.
- **Syntax of While Loop with else statement:**

```
while condition:
# execute these statements
else:
# execute these statements
```

Examples of While Loop with else statement:

```
count = 0
while (count < 3):
        count = count + 1
        print("Hello Geek")
else:
        print("In Else Block")
```

- **Output**

```
Hello Geek
Hello Geek
Hello Geek
In Else Block
```

For Loop in Python

For loops are used for sequential traversal. For example: traversing a list or string or array etc. In Python, there is “for in” loop which is similar to foreach loop in other languages. Let us learn how to use for loop in Python for sequential traversals with examples.

For Loop Syntax:

```
for iterator_var in sequence:
statements(s)
```

The code uses a Python for loop that iterates over the values from 0 to 3 (not including 4), as specified by the **range(0, n)** construct. It will print the values of ‘i' in each iteration of the loop

```
n = 4
for i in range(0, n):
        print(i)
```

Output

```
0
1
2
3
```

Exercise 1: Print First 10 natural numbers using while loop

```
# program 1: Print first 10 natural numbers
i = 1
while i <= 10:
    print(i)
    i += 1
```

Expected output:

```
1
2
3
4
5
6
7
8
9
10
```

Exercise 2: Print the following pattern

Write a program to print the following number pattern using a loop.

```
print("Number Pattern ")


# Decide the row count. (above pattern contains 5 rows)
row = 5
# start: 1
# stop: row+1 (range never include stop number in result)
# step: 1
# run loop 5 times
for i in range(1, row + 1, 1):
    # Run inner loop i+1 times
    for j in range(1, i + 1):
        print(j, end=' ')
    # empty line after each row
    print("")
```

Sample Output:

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

Exercise 3: Calculate the sum of all numbers from 1 to a given number

Write a program to accept a number from a user and calculate the sum of all numbers from 1 to a given number.

For example, if the user entered 10 the output should be 55 (1+2+3+4+5+6+7+8+9+10)

```
# s: store sum of all numbers
s = 0
n = int(input("Enter number "))
# run loop n times
# stop: n+1 (because range never include stop number in result)
for i in range(1, n + 1, 1):
    # add current number to sum variable
    s += i
print("\n")
print("Sum is: ", s)
```

Expected Output:

```
Enter number 10
Sum is:  55
```

Exercise 4: Write a program to print multiplication table of a given number

```
n = 2
# stop: 11 (because range never include stop number in result)
# run loop 10 times
for i in range(1, 11, 1):
    # 2 *i (current number)
    product = n * i
    print(product)
```

For example, num = 2 so the output should be

2
4
6
8
10
12
14
16
18
20

Exercise 5: Print list in reverse order using a loop

Using a reversed() function and for loop

```
list1 = [10, 20, 30, 40, 50]
# reverse list
new_list = reversed(list1)
# iterate reversed list
for item in new_list:
    print(item)
```

Expected output:

```
50
40
30
20
10
```

Exercise 6: Write a program to display all prime numbers within a range

```
start = 25
end = 50
print("Prime numbers between", start, "and", end, "are:")

for num in range(start, end + 1):
   # all prime numbers are greater than 1
   # if number is less than or equal to 1, it is not prime
   if num > 1:
      for i in range(2, num):
         # check for factors
         if (num % i) == 0:
            # not a prime number so break inner loop and
            # look for next number
            break
      else:
         print(num)
```

Given:

```
# range
start = 25
end = 50
```

Expected output:

```
Prime numbers between 25 and 50 are:
29
31
37
41
43
47
```

Exercise 7: Display Fibonacci series up to 10 terms

```
# first two numbers
num1, num2 = 0, 1

print("Fibonacci sequence:")
# run loop 10 times
for i in range(10):
    # print next number of a series
    print(num1, end="  ")
    # add last two numbers to get next number
    res = num1 + num2

    # update values
    num1 = num2
    num2 = res
```

For example, 0, 1, 1, 2, 3, 5, 8, 13, 21. The next number in this series above is 13+21 = 34.

Expected output:

Fibonacci sequence:

0 1 1 2 3 5 8 13 21 34

Exercise 8: Find the factorial of a given number

```
num = 5
factorial = 1
if num < 0:
    print("Factorial does not exist for negative numbers")
elif num == 0:
    print("The factorial of 0 is 1")
else:
    # run loop 5 times
    for i in range(1, num + 1):
        # multiply factorial by current number
        factorial = factorial * i
    print("The factorial of", num, "is", factorial)
```

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Expected output:

```
120
```

Exercise 9: Find the sum of the series upto n terms

Write a program to calculate the sum of series up to n term. For example, if n =5 the series will become 2 + 22 + 222 + 2222 + 22222 = 24690

```
n = 5
# first number of sequence
start = 2
sum_seq = 0
# run loop n times
for i in range(0, n):
    print(start, end="+")
    sum_seq += start
    # calculate the next term
    start = start * 10 + 2
print("\nSum of above series is:", sum_seq)
```

number of terms

n = 5

Expected output:

24690

Exercise 10: Print the following pattern

Write a program to print the following start pattern using the for loop

```
*
* *
* * *
* * * *
* * * * *
* * * *
* * *
* *
*
```

```
rows = 5
for i in range(0, rows):
    for j in range(0, i + 1):
        print("*", end=' ')
    print("\r")

for i in range(rows, 0, -1):
    for j in range(0, i - 1):
        print("*", end=' ')
    print("\r")
```

Python Lambda Functions

Python Lambda Functions are anonymous functions means that the function is without a name. As we already know the *def* keyword is used to define a normal function in Python. Similarly, the *lambda* keyword is used to define an anonymous function in Python.

Python Lambda Function Syntax

Syntax: lambda arguments : expression

- This function can have any number of arguments but only one expression, which is evaluated and returned.
- One is free to use lambda functions wherever function objects are required.
- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.
- It has various uses in particular fields of programming, besides other types of expressions in functions.

Python Lambda Function Example

In the example, we defined a lambda function(**upper**) to convert a string to its upper case using upper().

This code defines a lambda function named **`upper`** that takes a string as its argument and converts it to uppercase using the **`upper()`** method. It then applies this lambda function to the string 'GeeksforGeeks' and prints the result

```
str1 = 'GeeksforGeeks'

upper = lambda string: string.upper()
print(upper(str1))
```

Output:

```
GEEKSFORGEEKS
```

Use of Lambda Function in Python

Let's see some of the practical uses of the Python lambda function.

Condition Checking Using Python lambda function

Here, the **'format_numric'** calls the lambda function, and the num is passed as a parameter to perform operations.

```
format_numeric = lambda num: f"{num:e}" if isinstance(num, int) else
f"{num:,.2f}"

print("Int formatting:", format_numeric(1000000))
print("float formatting:", format_numeric(999999.789541235))
```

Output:

```
Int formatting: 1.000000e+06
float formatting: 999,999.79
```

Difference Between Lambda functions and def defined function

The code defines a cube function using both the **'def'** keyword and a lambda function. It calculates the cube of a given number (5 in this case) using both approaches and prints the results. The output is 125 for both the **'def'** and lambda functions, demonstrating that they achieve the same cube calculation.

```
def cube(y):
    return y*y*y

lambda_cube = lambda y: y*y*y
print("Using function defined with `def` keyword, cube:", cube(5))
print("Using lambda function, cube:", lambda_cube(5))
```

Output:

```
Using function defined with `def` keyword, cube: 125
Using lambda function, cube: 125
```

As we can see in the above example, both the **cube()** function and **lambda_cube()** function behave the same and as intended. Let's analyze the above example a bit more:

With lambda function	Without lambda function
Supports single-line sometimes statements that return some value.	Supports any number of lines inside a function block
Good for performing short operations/data manipulations.	Good for any cases that require multiple lines of code.
Using the lambda function can sometime reduce the readability of code.	We can use comments and function descriptions for easy readability.

Practical Uses of Python lambda function

Python Lambda Function with List Comprehension

On each iteration inside the list comprehension, we are creating a new lambda function with a default argument of x (where x is the current item in the iteration). Later, inside the for loop, we are calling the same function object having the default argument using **item()** and get the desired value. Thus, **is_even_list** stores the list of lambda function objects.

```
is_even_list = [lambda arg=x: arg * 10 for x in range(1, 5)]
```

```
for item in is_even_list:
    print(item())
```

Output:

```
10
20
30
40
```

Python Lambda Function with if-else

Here we are using the **Max** lambda function to find the maximum of two integers.

```
Max = lambda a, b : a if(a > b) else b
print(Max(1, 2))
```

Output:

```
2
```

Python Lambda with Multiple Statements

Lambda functions do not allow multiple statements, however, we can create two lambda functions and then call the other lambda function as a parameter to the first function. Let's try to find the second maximum element using lambda.

The code defines a list of sublists called **'List'**. It uses lambda functions to sort each sublist and find the second-largest element in each sublist. The result is a list of second-largest elements, which is then printed. The output displays the second-largest element from each sublist in the original list.

```
List = [[2,3,4],[1, 4, 16, 64],[3, 6, 9, 12]]

sortList = lambda x: (sorted(i) for i in x)
secondLargest = lambda x, f : [y[len(y)-2] for y in f(x)]
res = secondLargest(List, sortList)

print(res)
```

Output:

```
[3, 16, 9]
```

Lambda functions can be used along with built-in functions like filter(), map() and reduce().

Using lambda() Function with filter()

The filter() function in Python takes in a function and a list as arguments. This offers an elegant way to filter out all the elements of a sequence "sequence", for which the function returns True. Here is a small program that returns the odd numbers from an input list:

Filter out all odd numbers using filter() and lambda function

Here, lambda x: (x % 2 != 0) returns True or False if x is not even. Since filter() only keeps elements where it produces **True**, thus it removes all odd numbers that generated **False**.

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(filter(lambda x: (x % 2 != 0), li))
print(final_list)
```

Output:

```
[5, 7, 97, 77, 23, 73, 61]
```

Filter all people having age more than 18, using lambda and filter() function

The code filters a list of ages and extracts the ages of adults (ages greater than 18) using a lambda function and the **`filter`** function. It then prints the list of adult ages. The output displays the ages of individuals who are 18 years or older.

```
ages = [13, 90, 17, 59, 21, 60, 5]
adults = list(filter(lambda age: age > 18, ages))

print(adults)
```

Output:

```
[90, 59, 21, 60]
```

Using lambda() Function with map()

The map() function in Python takes in a function and a list as an argument. The function is called with a lambda function and a list and a new list is returned which contains all the lambda-modified items returned by that function for each item. Example:

Multiply all elements of a list by 2 using lambda and map() function

The code doubles each element in a list using a lambda function and the **`map`** function. It then prints the new list with the doubled elements. The output displays each element from the original list, multiplied by 2.

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]

final_list = list(map(lambda x: x*2, li))
print(final_list)
```

Output:

```
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

Transform all elements of a list to upper case using lambda and map() function

The code converts a list of animal names to uppercase using a lambda function and the **'map'** function. It then prints the list with the animal names in uppercase. The output displays the animal names in all uppercase letters.

```
animals = ['dog', 'cat', 'parrot', 'rabbit']
uppered_animals = list(map(lambda animal: animal.upper(), animals))

print(uppered_animals)
```

Output:

```
['DOG', 'CAT', 'PARROT', 'RABBIT']
```

Using lambda() Function with reduce()

The reduce() function in Python takes in a function and a list as an argument. The function is called with a lambda function and an iterable and a new reduced result is returned. This performs a repetitive operation over the pairs of the iterable. The reduce() function belongs to the ***functools*** module.

A sum of all elements in a list using lambda and reduce() function

The code calculates the sum of elements in a list using the **'reduce'** function from the **'functools'** module. It imports **'reduce'**, defines a list, applies a lambda function that adds two elements at a time, and prints the sum of all elements in the list. The output displays the computed sum.

```
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print(sum)
```

Output:

```
193
```

Here the results of the previous two elements are added to the next element and this goes on till the end of the list like (((((5+8)+10)+20)+50)+100).

Find the maximum element in a list using lambda and reduce() function

The code uses the **'functools'** module to find the maximum element in a list (**'lis'**) by employing the **'reduce'** function and a lambda function. It then prints the maximum element, which is 6 in this case.

```
import functools
lis = [1, 3, 5, 6, 2, ]
print("The maximum element of the list is : ", end="")
```

```
print(functools.reduce(lambda a, b: a if a > b else b, lis))
```

Output:

```
The maximum element of the list is : 6
```

Python Lists

Python Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). In simple language, a list is a collection of things, enclosed in [ ] and separated by commas.

The list is a sequence data type which is used to store the collection of data. Tuples and String are other types of sequence data types.

Example of list in Python

Here we are creating Python **List** using [].

```
Var = ["Geeks", "for", "Geeks"]
print(Var)
```

Output:

```
["Geeks", "for", "Geeks"]
```

Lists are the simplest containers that are an integral part of the Python language. Lists need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

Creating a List in Python

Lists in Python can be created by just placing the sequence inside the square brackets[]. Unlike Sets, a list doesn't need a built-in function for its creation of a list.

Note: Unlike Sets, the list may contain mutable elements.

Example 1: Creating a list in Python

```
# Python program to demonstrate
# Creation of List

# Creating a List
List = []
print("Blank List: ")
print(List)

# Creating a List of numbers
List = [10, 20, 14]
print("\nList of numbers: ")
print(List)

# Creating a List of strings and accessing
# using index
List = ["Geeks", "For", "Geeks"]
print("\nList Items: ")
```

```
print(List[0])
print(List[2])
```

Output

```
Blank List:
[]

List of numbers:
[10, 20, 14]

List Items:
Geeks
Geeks
```

Complexities for Creating Lists

Time Complexity: O(1)

Space Complexity: O(n)

Example 2: Creating a list with multiple distinct or duplicate elements

A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.

```
# Creating a List with
# the use of Numbers
# (Having duplicate values)
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
print("\nList with the use of Numbers: ")
print(List)

# Creating a List with
# mixed type of values
# (Having numbers and strings)
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
print("\nList with the use of Mixed Values: ")
print(List)
```

Output

```
List with the use of Numbers:
[1, 2, 4, 4, 3, 3, 3, 6, 5]

List with the use of Mixed Values:
[1, 2, 'Geeks', 4, 'For', 6, 'Geeks']
```

Accessing elements from the List

In order to access the list items refer to the index number. Use the index operator [ ] to access an item in a list. The index must be an integer. Nested lists are accessed using nested indexing.

Example 1: Accessing elements from list

```
# Python program to demonstrate
# accessing of element from list

# Creating a List with
# the use of multiple values
List = ["Geeks", "For", "Geeks"]

# accessing a element from the
# list using index number
print("Accessing a element from the list")
print(List[0])
print(List[2])
```

Output

```
Accessing a element from the list
Geeks
Geeks
```

Example 2: Accessing elements from a multi-dimensional list

```
# Creating a Multi-Dimensional List
# (By Nesting a list inside a List)
List = [['Geeks', 'For'], ['Geeks']]

# accessing an element from the
# Multi-Dimensional List using
# index number
print("Accessing a element from a Multi-Dimensional list")
print(List[0][1])
print(List[1][0])
```

Output

```
Accessing a element from a Multi-Dimensional list
For
Geeks
```

Negative indexing

In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in List[len(List)-3], it is enough to just write List[-3]. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```
List = [1, 2, 'Geeks', 4, 'For', 6, 'Geeks']

# accessing an element using
# negative indexing
print("Accessing element using negative indexing")

# print the last element of list
print(List[-1])

# print the third last element of list
print(List[-3])
```

Output

```
Accessing element using negative indexing
Geeks
For
```

Complexities for Accessing elements in a Lists:

Time Complexity: O(1)

Space Complexity: O(1)

Getting the size of Python list

Python len() is used to get the length of the list.

```
# Creating a List
List1 = []
print(len(List1))

# Creating a List of numbers
List2 = [10, 20, 14]
print(len(List2))
```

Output

```
0
3
```

Taking Input of a Python List

We can take the input of a list of elements as string, integer, float, etc. But the default one is a string.

Example 1:

```
# Python program to take space
# separated input as a string
# split and store it to a list
# and print the string list

# input the list as string
string = input("Enter elements (Space-Separated): ")

# split the strings and store it to a list
lst = string.split()
print('The list is:', lst)   # printing the list
```

Output:

```
Enter elements: GEEKS FOR GEEKS
The list is: ['GEEKS', 'FOR', 'GEEKS']
```

Example 2:

```
# input size of the list
n = int(input("Enter the size of list : "))
# store integers in a list using map,
# split and strip functions
lst = list(map(int, input("Enter the integer\
elements:").strip().split()))[:n]

# printing the list
print('The list is:', lst)
```

Output:

```
Enter the size of list : 4
Enter the integer elements: 6 3 9 10
The list is: [6, 3, 9, 10]
```

To know more see this.

Adding Elements to a Python List

Method 1: Using append() method

Elements can be added to the List by using the built-in **append()** function. Only one element at a time can be added to the list by using the append() method, for the addition of multiple elements with the append() method, loops are used. Tuples can also be added to the list with the use of the append method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of the append() method.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = []
print("Initial blank List: ")
print(List)

# Addition of Elements
# in the List
List.append(1)
List.append(2)
List.append(4)
print("\nList after Addition of Three elements: ")
print(List)

# Adding elements to the List
# using Iterator
for i in range(1, 4):
    List.append(i)
print("\nList after Addition of elements from 1-3: ")
print(List)

# Adding Tuples to the List
List.append((5, 6))
print("\nList after Addition of a Tuple: ")
print(List)
```

```
# Addition of List to a List
List2 = ['For', 'Geeks']
List.append(List2)
print("\nList after Addition of a List: ")
print(List)
```

Output

```
Initial blank List:
[]

List after Addition of Three elements:
[1, 2, 4]

List after Addition of elements from 1-3:
[1, 2, 4, 1, 2, 3]

List after Addition of a Tuple:
[1, 2, 4, 1, 2, 3, (5, 6)]

List after Addition of a List:
[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Geeks']]
```

Complexities for Adding elements in a Lists(append() method):

Time Complexity: O(1)

Space Complexity: O(1)

Method 2: Using insert() method

append() method only works for the addition of elements at the end of the List, for the addition of elements at the desired position, insert() method is used. Unlike append() which takes only one argument, the insert() method requires two arguments(position, value).

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1,2,3,4]
print("Initial List: ")
print(List)

# Addition of Element at
# specific Position
# (using Insert Method)
List.insert(3, 12)
List.insert(0, 'Geeks')
print("\nList after performing Insert Operation: ")
print(List)
```

Output

```
Initial List:
[1, 2, 3, 4]

List after performing Insert Operation:
```

```
['Geeks', 1, 2, 3, 12, 4]
```

Complexities for Adding elements in a Lists(insert() method):

Time Complexity: O(n)

Space Complexity: O(1)

Method 3: Using extend() method

Other than append() and insert() methods, there's one more method for the Addition of elements, **extend()**, this method is used to add multiple elements at the same time at the end of the list.

Note: append() and extend() methods can only add elements at the end.

```
# Python program to demonstrate
# Addition of elements in a List

# Creating a List
List = [1, 2, 3, 4]
print("Initial List: ")
print(List)

# Addition of multiple elements
# to the List at the end
# (using Extend Method)
List.extend([8, 'Geeks', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)
```

Output

```
Initial List:
[1, 2, 3, 4]

List after performing Extend Operation:
[1, 2, 3, 4, 8, 'Geeks', 'Always']
```

Complexities for Adding elements in a Lists(extend() method):

Time Complexity: O(n)

Space Complexity: O(1)

Reversing a List

Method 1: A list can be reversed by using the reverse() method in Python.

```
# Reversing a list
mylist = [1, 2, 3, 4, 5, 'Geek', 'Python']
mylist.reverse()
print(mylist)
```

Output

```
['Python', 'Geek', 5, 4, 3, 2, 1]
```

Method 2: Using the reversed() function:

The reversed() function returns a reverse iterator, which can be converted to a list using the list() function.

```
my_list = [1, 2, 3, 4, 5]
reversed_list = list(reversed(my_list))
print(reversed_list)
```

Output

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

Removing Elements from the List

Method 1: Using remove() method

Elements can be removed from the List by using the built-in **remove()** function but an Error arises if the element doesn't exist in the list. Remove() method only removes one element at a time, to remove a range of elements, the iterator is used. The remove() method removes the specified item.

Note: Remove method in List will only remove the first occurrence of the searched element.

Example 1:

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = [1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12]
print("Initial List: ")
print(List)

# Removing elements from List
# using Remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)
```

Output

```
Initial List:
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

List after Removal of two elements:
[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]
```

Example 2:

```
# Creating a List
List = [1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 11, 12]
# Removing elements from List
# using iterator method
for i in range(1, 5):
    List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)
```

Output

```
List after Removing a range of elements:
[5, 6, 7, 8, 9, 10, 11, 12]
```

Complexities for Deleting elements in a Lists(remove() method):

Time Complexity: O(n)

Space Complexity: O(1)

Method 2: Using pop() method

pop() function can also be used to remove and return an element from the list, but by default it removes only the last element of the list, to remove an element from a specific position of the List, the index of the element is passed as an argument to the pop() method.

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

# Removing element from the
# Set using the pop() method
List.pop()
print("\nList after popping an element: ")
print(List)

# Removing element at a
# specific location from the
# Set using the pop() method
List.pop(2)
print("\nList after popping a specific element: ")
print(List)
```

Output

```
List after popping an element:
[1, 2, 3, 4]

List after popping a specific element:
[1, 2, 4]
```

Complexities for Deleting elements in a Lists(pop() method):

Time Complexity: O(1)/O(n) (O(1) for removing the last element, O(n) for removing the first and middle elements)

Space Complexity: O(1)

Slicing of a List

We can get substrings and sublists using a slice. In Python List, there are multiple ways to print the whole list with all the elements, but to print a specific range of elements from the list, we use the Slice operation.

Slice operation is performed on Lists with the use of a colon(:).

To print elements from beginning to a range use:

[: Index]

To print elements from end-use:

[:-Index]

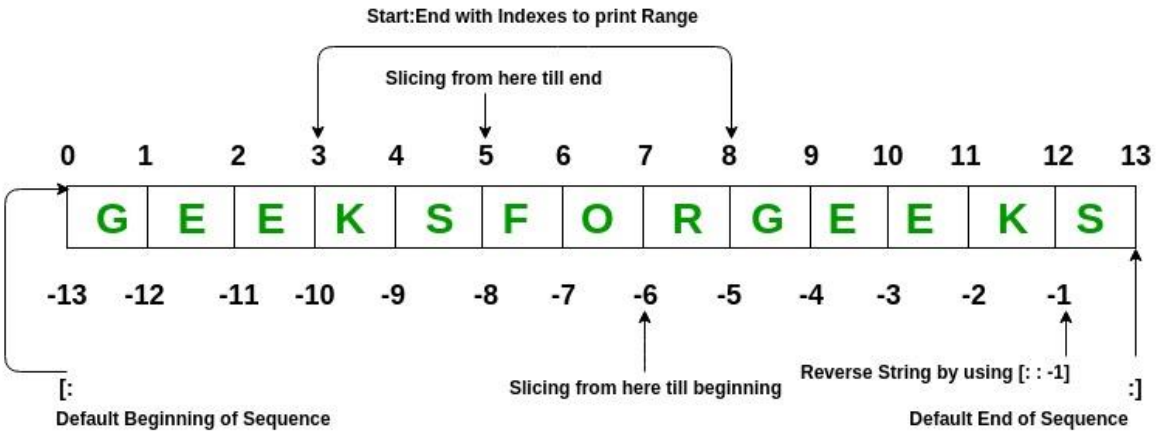
To print elements from a specific Index till the end use

[Index:]

To print the whole list in reverse order, use

[::-1]

Note – To print elements of List from rear-end, use Negative Indexes.



UNDERSTANDING SLICING OF LISTS:

- pr[0] accesses the first item, 2.
- pr[-4] accesses the fourth item from the end, 5.
- pr[2:] accesses [5, 7, 11, 13], a list of items from third to last.
- pr[:4] accesses [2, 3, 5, 7], a list of items from first to fourth.
- pr[2:4] accesses [5, 7], a list of items from third to fifth.
- pr[1::2] accesses [3, 7, 13], alternate items, starting from the second item.

```
# Python program to demonstrate
# Removal of elements in a List

# Creating a List
List = ['G', 'E', 'E', 'K', 'S', 'F',
        'O', 'R', 'G', 'E', 'E', 'K', 'S']
print("Initial List: ")
print(List)

# Print elements of a range
# using Slice operation
Sliced_List = List[3:8]
print("\nSlicing elements in a range 3-8: ")
print(Sliced_List)

# Print elements from a
# pre-defined point to end
Sliced_List = List[5:]
print("\nElements sliced from 5th "
      "element till the end: ")
print(Sliced_List)

# Printing elements from
# beginning till end
Sliced_List = List[:]
print("\nPrinting all elements using slice operation: ")
print(Sliced_List)
```

Output

```
Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Slicing elements in a range 3-8:
['K', 'S', 'F', 'O', 'R']

Elements sliced from 5th element till the end:
['F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Printing all elements using slice operation:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']
```

Negative index List slicing

```
# Creating a List
List = ['G', 'E', 'E', 'K', 'S', 'F',
        'O', 'R', 'G', 'E', 'E', 'K', 'S']
print("Initial List: ")
print(List)

# Print elements from beginning
# to a pre-defined point using Slice
Sliced_List = List[:-6]
print("\nElements sliced till 6th element from last: ")
print(Sliced_List)

# Print elements of a range
# using negative index List slicing
Sliced_List = List[-6:-1]
print("\nElements sliced from index -6 to -1")
```

```
print(Sliced_List)

# Printing elements in reverse
# using Slice operation
Sliced_List = List[::-1]
print("\nPrinting List in reverse: ")
print(Sliced_List)
```

Output

```
Initial List:
['G', 'E', 'E', 'K', 'S', 'F', 'O', 'R', 'G', 'E', 'E', 'K', 'S']

Elements sliced till 6th element from last:
['G', 'E', 'E', 'K', 'S', 'F', 'O']

Elements sliced from index -6 to -1
['R', 'G', 'E', 'E', 'K']

Printing List in reverse:
['S', 'K', 'E', 'E', 'G', 'R', 'O', 'F', 'S', 'K', 'E', 'E', 'G']
```

List Comprehension

Python List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc. A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

Syntax:

newList = [expression(element) for element in oldList if condition]

Example:

```
# Python program to demonstrate list
# comprehension in Python

# below list contains square of all
# odd numbers from range 1 to 10
odd_square = [x ** 2 for x in range(1, 11) if x % 2 == 1]
print(odd_square)
```

Output

```
[1, 9, 25, 49, 81]
```

For better understanding, the above code is similar to as follows:

```
# for understanding, above generation is same as,
odd_square = []

for x in range(1, 11):
    if x % 2 == 1:
        odd_square.append(x**2)

print(odd_square)
```

Output

```
[1, 9, 25, 49, 81]
```

Refer to the below articles to get detailed information about List Comprehension.

- Python List Comprehension and Slicing
- Nested List Comprehensions in Python
- List comprehension and ord() in Python

Basic Example on Python List

- Python program to interchange first and last elements in a list
- Python program to swap two elements in a list
- Python – Swap elements in String list
- Python | Ways to find length of list
- Maximum of two numbers in Python
- Minimum of two numbers in Python

To Practice the basic list operation, please read this article – Python List of program

List Methods

Function	Description
Append()	Add an element to the end of the list
Extend()	Add all elements of a list to another list
Insert()	Insert an item at the defined index
Remove()	Removes an item from the list
Clear()	Removes all items from the list
Index()	Returns the index of the first matched item
Count()	Returns the count of the number of items passed as an argument
Sort()	Sort items in a list in ascending order
Reverse()	Reverse the order of items in the list
copy()	Returns a copy of the list
pop()	Removes and returns the item at the specified index. If no index is provided, it removes and returns the last item.

To know more refer to this article – Python List methods

The operations mentioned above modify the list Itself.

Built-in functions with List

Function	Description
reduce()	apply a particular function passed in its argument to all of the list elements stores the intermediate result and only returns the final summation value

Function	Description
sum()	Sums up the numbers in the list
ord()	Returns an integer representing the Unicode code point of the given Unicode character
cmp()	This function returns 1 if the first list is “greater” than the second list
max()	return maximum element of a given list
min()	return minimum element of a given list
all()	Returns true if all element is true or if the list is empty
any()	return true if any element of the list is true. if the list is empty, return false
len()	Returns length of the list or size of the list
enumerate()	Returns enumerate object of the list
accumulate()	apply a particular function passed in its argument to all of the list elements returns a list containing the intermediate results
filter()	tests if each element of a list is true or not
map()	returns a list of the results after applying the given function to each item of a given iterable
lambda()	This function can have any number of arguments but only one expression, which is evaluated and returned

Mutable vs Immutable Objects in Python

In Python, Every variable in Python holds an instance of an object. There are two types of objects in Python i.e. **Mutable** and **Immutable objects**. Whenever an object is instantiated, it is assigned a unique object id. The type of the object is defined at the runtime and it can't be changed afterward. However, its state can be changed if it is a mutable object.

Mutable and Immutable Objects in Python

Let us see what are Python's Mutable vs Immutable Types in Python.

Mutable Objects in Python

Immutable Objects are of in-built datatypes like int, float, bool, string, Unicode, and tuple. In simple words, an immutable object can't be changed after it is created.

Example 1: In this example, we will take a tuple and try to modify its value at a particular index and print it. As a tuple is an immutable object, it will throw an error when we try to modify it.

```
# Python code to test that
# tuples are immutable

tuple1 = (0, 1, 2, 3)
tuple1[0] = 4
print(tuple1)
```

Error:

```
Traceback (most recent call last):
  File "e0eaddff843a8695575daec34506f126.py", line 3, in
    tuple1[0]=4
TypeError: 'tuple' object does not support item assignment
```

Example 2: In this example, we will take a Python string and try to modify its value. Similar to the tuple, strings are immutable and will throw an error.

```
# Python code to test that
# strings are immutable

message = "Welcome to GeeksforGeeks"
message[0] = 'p'
print(message)
```

Error:

```
Traceback (most recent call last):
  File "/home/ff856d3c5411909530c4d328eeca165b.py", line 3, in
    message[0] = 'p'
TypeError: 'str' object does not support item assignment
```

Mutable Objects in Python

Mutable Objects are of type Python list, Python dict, or Python set. Custom classes are generally mutable.

Are Lists Mutable in Python?

Yes, Lists are mutable in Python. We can add or remove elements from the list. In Python, mutability refers to the capability of an object to be changed or modified after its creation.

In Python, lists are a widely used data structure that allows the storage and manipulation of a collection of items. One of the key characteristics of lists is their mutability, which refers to the ability to modify the list after it has been created.

Example 1: Add and Remove items from a list in Python

In this example, we will take a Python List object and try to modify its value using the index. A list in Python is mutable, that is, it allows us to change its value once it is created. Lists have the ability to add and remove elements dynamically. Lists provide methods such as append(),insert(),extend(),remove() and pop().

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)

my_list.insert(1, 5)
print(my_list)

my_list.remove(2)
print(my_list)

popped_element = my_list.pop(0)
print(my_list)
print(popped_element)
```

Output

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

Example 2: Modify item from a dictionary in Python

Here is an example of dictionary that are mutable i.e., we can make changes in the Dictionary.

```
my_dict = {"name": "Ram", "age": 25}
new_dict = my_dict
new_dict["age"] = 37

print(my_dict)
```

```
print(new_dict)
```

Output

```
{'name': 'Ram', 'age': 37}
{'name': 'Ram', 'age': 37}
```

Example 3: Modify item from a Set in Python

Here is an example of Set that are mutable i.e., we can make changes in the set.

```
my_set = {1, 2, 3}
new_set = my_set
new_set.add(4)

print(my_set)
print(new_set)
```

Output

```
{1, 2, 3, 4}
{1, 2, 3, 4}
```

Python's Mutable vs Immutable

1. Mutable and immutable objects are handled differently in Python. Immutable objects are quicker to access and are expensive to **change** because it involves the creation of a copy. Whereas mutable objects are easy to change.

2. The use of mutable objects is recommended when there is a need to change the size or content of the object.

3. **Exception:** However, there is an exception in immutability as well. We know that a tuple in Python is immutable. But the tuple consists of a sequence of names with unchangeable bindings to objects. Consider a tuple

```
tup = ([3, 4, 5], 'myname')
```

The tuple consists of a string and a list. Strings are immutable so we can't change their value. But the contents of the list can change. **The tuple itself isn't mutable but contains items that are mutable.** As a rule of thumb, generally, Primitive-like types are probably immutable, and Customized Container-like types are mostly mutable.

Problem solving using list & function

Python list is the most widely used data structure, and a good understanding of it is necessary. list operations and manipulations

- list functions
- list slicing
- list comprehension

Exercise 1: Reverse a list in Python

Given:

```
list1 = [100, 200, 300, 400, 500]
```

Expected output:

```
[500, 400, 300, 200, 100]
```

Solution 1: list function `reverse()`

```
list1 = [100, 200, 300, 400, 500]
list1.reverse()
print(list1)
```

Solution 2: Using negative slicing

-1 indicates to start from the last item.

```
list1 = [100, 200, 300, 400, 500]
```

```
list1 = list1[::-1]
print(list1)
```

Exercise 2: Concatenate two lists index-wise

Write a program to add two lists index-wise. Create a new list that contains the 0th index item from both the list, then the 1st index item, and so on till the last element. any leftover items will get added at the end of the new list.

Given:

```
list1 = ["M", "na", "i", "Ke"]
list2 = ["y", "me", "s", "lly"]
```

Expected output:

```
['My', 'name', 'is', 'Kelly']
```

Use the `zip()` function. This function takes two or more iterables (like list, dict, string), aggregates them in a tuple, and returns it.

```
list1 = ["M", "na", "i", "Ke"]
list2 = ["y", "me", "s", "lly"]
list3 = [i + j for i, j in zip(list1, list2)]
print(list3)
```

Exercise 3: Turn every item of a list into its square

Given a list of numbers. write a program to turn every item of a list into its square.

Given:

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

Expected output:

```
[1, 4, 9, 16, 25, 36, 49]
```

Solution 1: Using loop and list method

- Create an empty result list
- Iterate a numbers list using a loop
- In each iteration, calculate the square of a current number and add it to the result list using the `append()` method.

```
numbers = [1, 2, 3, 4, 5, 6, 7]
# result list
res = []
for i in numbers:
    # calculate square and add to the result list
    res.append(i * i)
print(res)
```

Solution 2: Use list comprehension

```
numbers = [1, 2, 3, 4, 5, 6, 7]
res = [x * x for x in numbers]
print(res)
```

Exercise 4: Concatenate two lists in the following order

```
list1 = ["Hello ", "take "]
list2 = ["Dear", "Sir"]
```

Expected output:

```
['Hello Dear', 'Hello Sir', 'take Dear', 'take Sir']
```

```
list1 = ["Hello ", "take "]
list2 = ["Dear", "Sir"]

res = [x + y for x in list1 for y in list2]
print(res)
```

Exercise 5: Iterate both lists simultaneously

Given a two Python list. Write a program to iterate both lists simultaneously and display items from list1 in original order and items from list2 in reverse order.

Given

```
list1 = [10, 20, 30, 40]
list2 = [100, 200, 300, 400]
```

Expected output:

```
10 400

20 300
```

```
30 200

40 100
```

- The zip() function can take two or more lists, aggregate them in a tuple, and returns it.
- Pass the first argument as a list1 and seconds argument as a list2[::-1] (reverse list using list slicing)
- Iterate the result using a for loop

```
list1 = [10, 20, 30, 40]
list2 = [100, 200, 300, 400]

for x, y in zip(list1, list2[::-1]):
    print(x, y)
```

Exercise 6: Remove empty strings from the list of strings

```
list1 = ["Mike", "", "Emma", "Kelly", "", "Brad"]
```

Expected output:

```
["Mike", "Emma", "Kelly", "Brad"]
```

Use a filter() function to remove None type from the list

```
list1 = ["Mike", "", "Emma", "Kelly", "", "Brad"]

# remove None from list1 and convert result into list
res = list(filter(None, list1))
print(res)
```

Exercise 7: Add new item to list after a specified item

Write a program to add item 7000 after 6000 in the following Python List

Given:

```
list1 = [10, 20, [300, 400, [5000, 6000], 500], 30, 40]
```

Expected output:

```
[10, 20, [300, 400, [5000, 6000, 7000], 500], 30, 40]
```

Use the append() method

```
list1 = [10, 20, [300, 400, [5000, 6000], 500], 30, 40]

# understand indexing
# list1[0] = 10
# list1[1] = 20
# list1[2] = [300, 400, [5000, 6000], 500]
# list1[2][2] = [5000, 6000]
# list1[2][2][1] = 6000

# solution
```

```
list1[2][2].append(7000)
print(list1)
```

Exercise 8: Extend nested list by adding the sublist

You have given a nested list. Write a program to extend it by adding the sublist `["h", "i", "j"]` in such a way that it will look like the following list.

Given List:

```
list1 = ["a", "b", ["c", ["d", "e", ["f", "g"], "k"], "l"], "m", "n"]

# sub list to add
sub_list = ["h", "i", "j"]
```

Expected Output:

```
['a', 'b', ['c', ['d', 'e', ['f', 'g', 'h', 'i', 'j'], 'k'], 'l'], 'm', 'n']
```

```
list1 = ["a", "b", ["c", ["d", "e", ["f", "g"], "k"], "l"], "m", "n"]
sub_list = ["h", "i", "j"]

# understand indexing
# list1[2] = ['c', ['d', 'e', ['f', 'g'], 'k'], 'l']
# list1[2][1] = ['d', 'e', ['f', 'g'], 'k']
# list1[2][1][2] = ['f', 'g']

# solution
list1[2][1][2].extend(sub_list)
```

```
print(list1)
```

Exercise 9: Replace list's item with new value if found

You have given a Python list. Write a program to find value 20 in the list, and if it is present, replace it with 200. Only update the first occurrence of an item.

Given:

```
list1 = [5, 10, 15, 20, 25, 50, 20]
```

Expected output:

```
[5, 10, 15, 200, 25, 50, 20]
```

```
list1 = [5, 10, 15, 20, 25, 50, 20]

# get the first occurrence index
index = list1.index(20)

# update item present at location
list1[index] = 200
print(list1)
```

Exercise 10: Remove all occurrences of a specific item from a list.

Given a Python list, write a program to remove all occurrences of item 20.

Given:

```
list1 = [5, 20, 15, 20, 25, 50, 20]
```

Expected output:

```
[5, 15, 25, 50]
```

Solution 1: Use the list comprehension

```
list1 = [5, 20, 15, 20, 25, 50, 20]

# list comprehension
# remove specific items and return a new list
def remove_value(sample_list, val):
    return [i for i in sample_list if i != val]

res = remove_value(list1, 20)
print(res)
```

Solution 2: while loop (slow solution)

```
list1 = [5, 20, 15, 20, 25, 50, 20]

while 20 in list1:
    list1.remove(20)
print(list1)
```