This is a python handbook to help anyone to learn and use python
As accurate and fast and to the point as possible.

Content table

Chapter 0 .

1- What is programming.

in general there are things around us and those things can be created or achieved by a specific steps that makes a formula to achieve it.
in our case we will give the computer instructions in a specific order to achieve a specific algorithm that'll achieve us a specific need.
more in depth look about why this type of thing possible and true.

(the big picture)

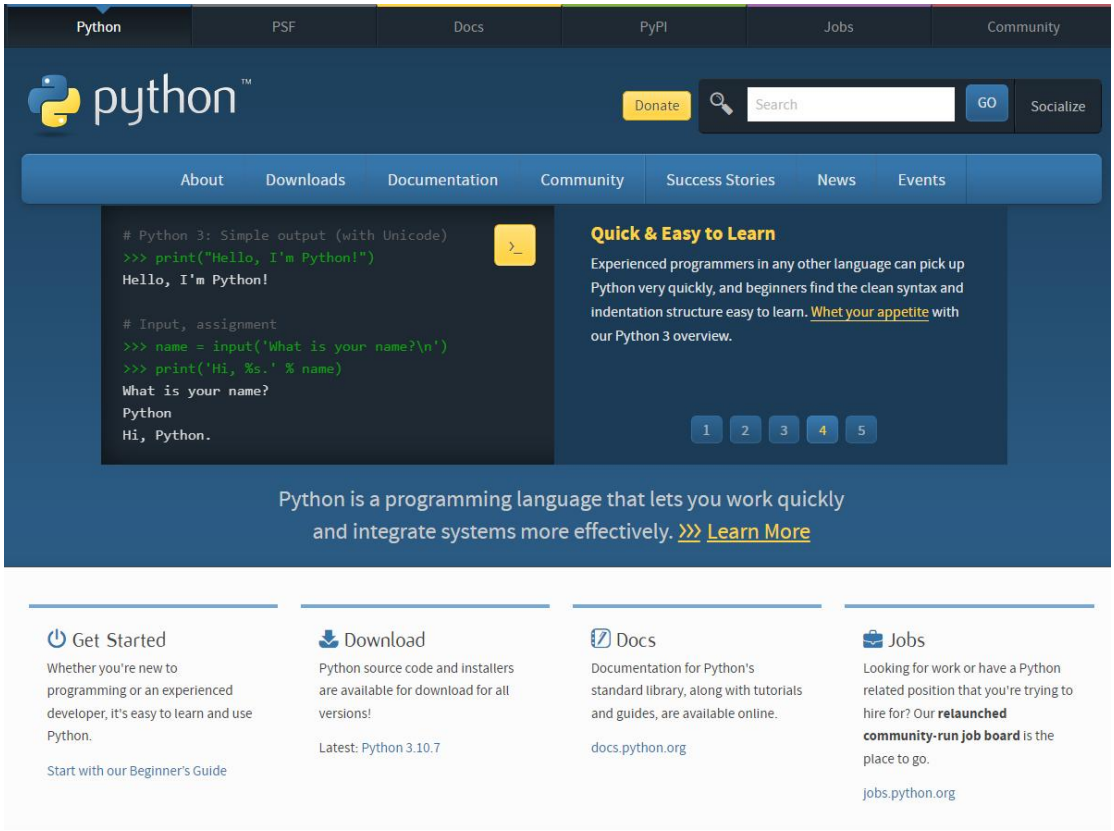
it's simply returns back to the start of the universe .
what ever caused the universe it had some properties described by the physics.
those properties are controlling everything that happens in the universe .
The things that are happening in the universe are actions that are controlled by the universal laws that determines the outcome of that action .
simply described as a function.
functions are simplifying this idea by saying: there is some thing that is happening on an input that will change it to be the output.
and all the possible changes that could happen are Determined by the laws of the function.
so every output is having a certain input that is being manipulated somehow and changed based on certain laws that controls that change what's the results to it.
programming is basically finding the ways that we can change some inputs to resolve it to a specific output we want**.**

2- General programming concepts.

after we understood the concept of programming let's now understand how programming with computers work.
from the CPU hardware design level it is already configured to do some basic functions like addition and subtraction etc.
addressing each operation Will trigger it.
and of course if we want to do some operations we will need some place to Store our operands that we will do the operation on. and that's the job of the memory
And we will need to control the flow between both and this is no easy task for us so here it comes the operating system that will control all of that hardware management resources and give us an interface to interact with it
from the programming language
these interfaces are called APIs
So if we want to access or use or etc. to a specific hardware device we will need to talk to the API that defines the way we can talk to it from the operating system.
so if we want to talk to the memory there is an API to talk to the memory and if we want to talk to the GPU there is an API to talk to the GPU . if we wanted to talk to the network interface there is a specific API for the network interface. all of those APIs are controlled and defined by the operating system .

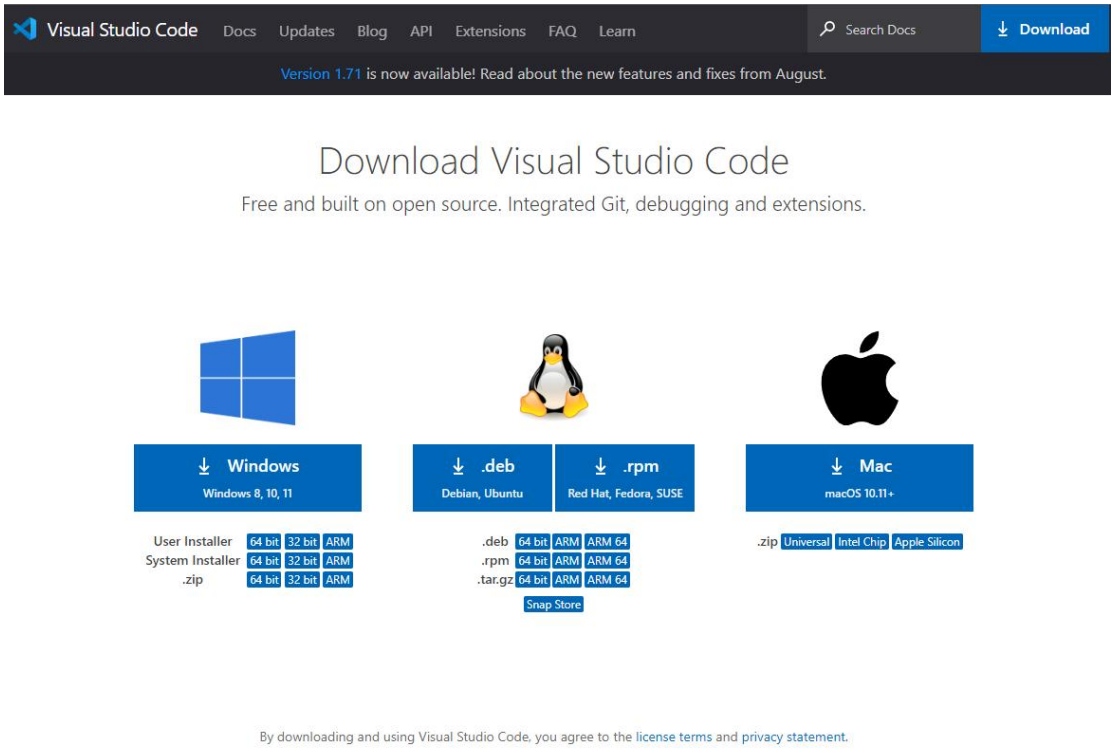
3- Setting the programming environment for python.

a) Downloading the python interpreter from the official python https://www.python.org/

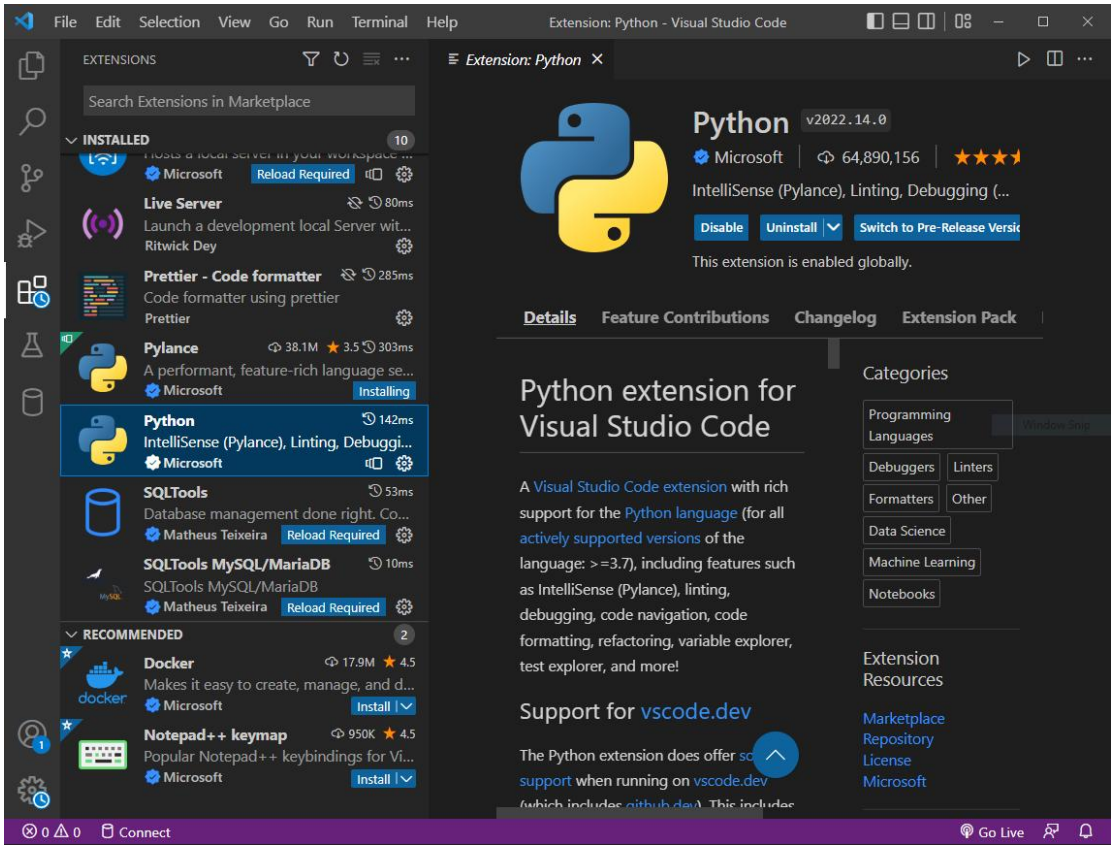


And make sure to add it to path.

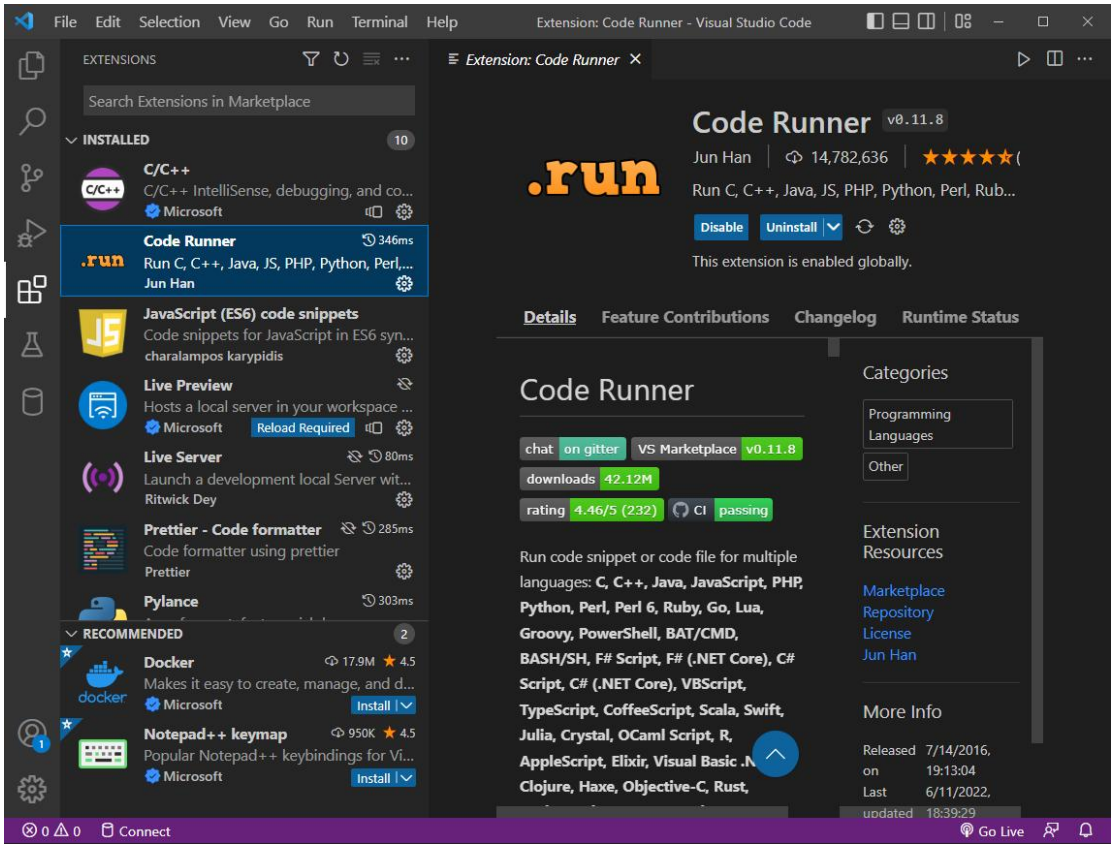
B) Download any text editor like VS-code from
https://code.visualstudio.com/download .



Download the python module extension to the Vs-code text editor.

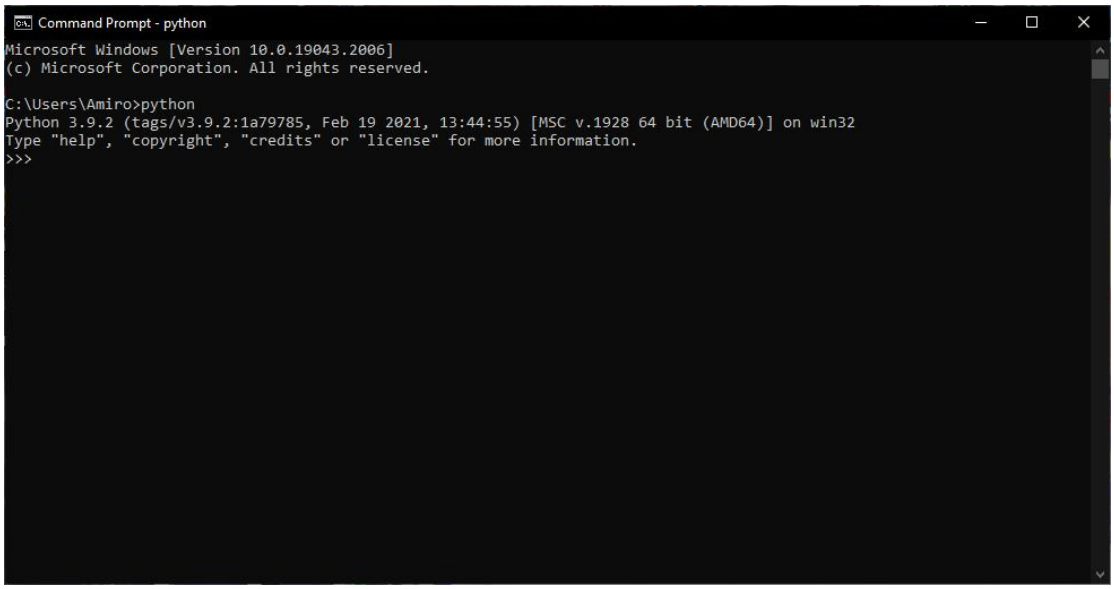


Download the code runner extension for running the code easily.



C) After we got everything ready now we need to understand the possible ways that we can set python and run it

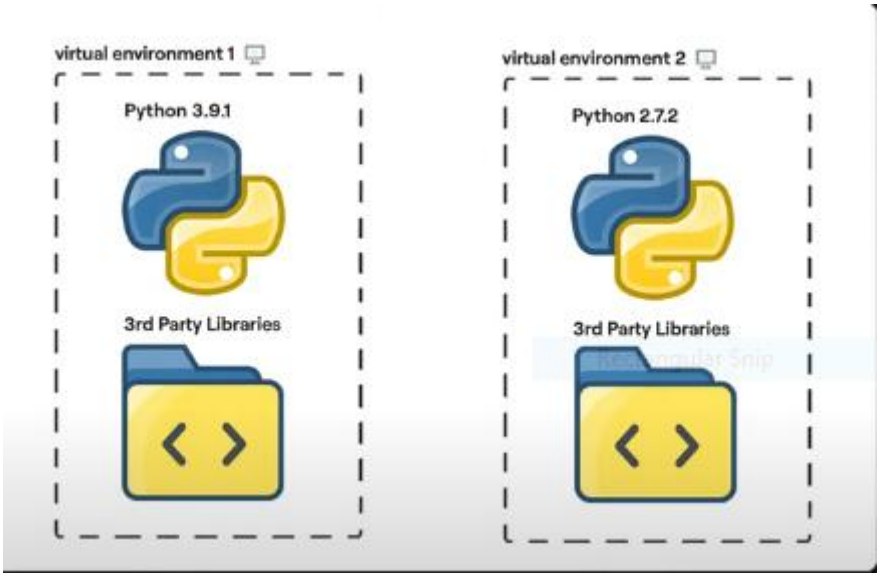
1- We can run it from the command line (CMD) directly by typing <python>
And run it directly .



2- We can make a python file by < name.py > and write python code with are text editor and run it later by a going to CMD and then going to the file location then to run it We would write < python name.py > and if u are using linux we would say

< python3 name.py >.

D) every programming environment is just the place and the configuration of the place that we are writing our code in like the libraries that we are using and the files and the metadata related to it and Versions.
The virtual environment is an environment that is separated from the standard environment and it is quarantined and anything we do inside of it is just going to stay inside of it and want to make any changes to the standard python
so we can say that the virtual environment is just like a dream. whatever we do inside of it will never be in the real life.
In other words**(in python when we build any project there are some dependencies , libraries and modules that we are using .**
there are different versions of those and if we want to make a project with specific version numbers for each module or library that we are using we need to install it without affecting the default module in our python environment.
so we will make a virtual environment and install what other version we want from those libraries and modules without affecting the default environment and is completely separated from it)
And each virtual environment can have any:
1- version of the python interpreter.
2- Any version of any library/dependence we want.



To make a virtual environment in windows:
1- First make a folder to place your virtual environment in.
2- Open powershell and then go to the folder that you want to place your virtual environment by using the commands (pwd=present working directory, ls=list, cd=change directory).
3- Now to create the virtual environment use the python command for creating it (python -m venv name_of_virtual_environment)
4- To start using the environment open the mother directory that contains the environment with your text editor like vscode and make a python file in the mother directory.

E) Pip:

Pip is the package manager of python that manages the downloads/uninstalls/updates of the python packages and dependencies

U can install pip or other package managers if you like

But it comes with default python installation to use pip and get around with it use

The manual :

```
Commands:
  install                 Install packages.
  download                Download packages.
  uninstall               Uninstall packages.
  freeze                  Output installed packages in requirements format.
  inspect                 Inspect the python environment.
  list                    List installed packages.
  show                    Show information about installed packages.
  check                   Verify installed packages have compatible dependencies.
  config                  Manage local and global configuration.
  search                  Search PyPI for packages.
  cache                   Inspect and manage pip's wheel cache.
  index                   Inspect information available from package indexes.
  wheel                   Build wheels from your requirements.
  hash                    Compute hashes of package archives.
  completion              A helper command used for command completion.
  debug                   Show information useful for debugging.
  help                    Show help for commands.

General Options:
  -h, --help              Show help.
  --debug                 Let unhandled exceptions propagate outside the
                          main subroutine, instead of logging them to
                          stderr.
  --isolated              Run pip in an isolated mode, ignoring
                          environment variables and user configuration.
  --require-virtualenv    Allow pip to only run in a virtual
                          environment; exit with an error otherwise.
  -v, --verbose           Give more output. Option is additive, and can
                          be used up to 3 times.
  -V, --version           Show version and exit.
  -q, --quiet             Give less output. Option is additive, and can
                          be used up to 3 times (corresponding to
                          WARNING, ERROR, and CRITICAL logging levels).
  --log <path>            Path to a verbose appending log.
  --no-input              Disable prompting for input.
  --proxy <proxy>         Specify a proxy in the form
                          scheme://[user:passwd@]proxy.server:port.
  --retries <retries>     Maximum number of retries each connection
                          should attempt (default 5 times).
  --timeout <sec>         Set the socket timeout (default 15 seconds).
  --exists-action <action>    Default action when a path already exists:
                          (s)witch, (i)gnore, (w)ipe, (b)ackup, (a)bort.
  --trusted-host <hostname>   Mark this host or host:port pair as trusted,
                          even though it does not have valid or any
                          HTTPS.
```

```
--cert <path>              Path to PEM-encoded CA certificate bundle. If
                           provided, overrides the default. See 'SSL
                           Certificate Verification' in pip documentation
                           for more information.
--client-cert <path>       Path to SSL client certificate, a single file
                           containing the private key and the certificate
                           in PEM format.
--cache-dir <dir>          Store the cache data in <dir>.
--no-cache-dir             Disable the cache.
--disable-pip-version-check
                           Don't periodically check PyPI to determine
                           whether a new version of pip is available for
                           download. Implied with --no-index.
--no-color                 Suppress colored output.
--no-python-version-warning
                           Silence deprecation warnings for upcoming
                           unsupported Pythons.
--use-feature <feature>    Enable new functionality, that may be backward
                           incompatible.
--use-deprecated <feature>  Enable deprecated functionality, that will be
                           removed in the future.
```

1- Pip -help.
2- To install use < pip install name_something >
3- To uninstall use < pip uninstall name_something >
4- To cheek version use < pip package_name -V >
5- To update pip use < python -m pip install --upgrade pip >

Chapter 1 .

The python programming language.

Importing libraries:

If we need to use some libraries/dependencies we will use the (import).
if we wanted specific thing from a library we will use it name and (from) the main library name.
If we wanted to give it an allays we would use (as).
Ex\

```
Import tkinter
Import button from tkinter
Import button from tkinter as btn
```

Data types:

As we said 'programming is just taking data and then doing something with it and outputting the data after that' so how can we obtain and define the data in python ?.
To obtain and the data we first must understand what are the data that we are taking. And those types are . . .

Native: the built in python basic data
1- Integers "numbers like (1,2,3..)".
2- Floats "float numbers (1.32, 3.567..)".
3- Double "float numbers with more values (4.567876.. , 5.788766 ..)".
4- Chars "single symbols (4, t, h ..)"
5- Strings " test or words (hello, python_is_cool …)".
6- Boolean " boolean statements like (true or false)

Non-native: the user can make his own data if needed
1-Classes (blue print that defines something and what can it do).

Storing data:

After we understood the types data that we will use when we are programming
Its really important to understand how we can store the data so we can use it. and there are some Ways that we can store the data so we can use it like
1- Variables (a place to store data that can be called by a name or symbol like X,Y,V ..etc).
2- Arrays (a place that we can store multiple data elements with different types in a contentious manner in memory).
3- Tuples (a place to store multiple data with different types in a none-contentious manner in memory).
4- Others (data structures like trees linked lists).

Taking/making data:

variables
To define a variable that stores a data type in python

1- make a name for the variable like x or my_var.
2- To store something in it u need to use the ‘ = ‘ symbol so you can assign something to it.
3- Now after the ‘ = ’ put the data you want to store but

a) If you want to store a number just put it after the ‘ = ‘.
b) If you want to store a test you must put it between double cots “ “ .
c) If you want to store a char you must put it between single cots ‘ ‘ .
d) If you want to store a boolean value juts write it after the ‘ = ‘

Ex\

```
my_var1 = 123
my_var2 = “hello world”
my_var3 = ‘a’
my_var4 = 4.567
My_var5 = TRUE
```

arrays

Now if we wanted to make an array we need to

1- Name it like my_array.
2- Put a = .
3- Put a square brackets [] after the = .
4- Put some data between the square brackets separated with (,).

a) For text elements we must put it between double cots (“ “).
b) For numbers we put them without the double cots.

Ex\

```
My_array = [“hello”, 34, ‘g’, True]
```

Tuples

Now if we wanted to make a tuple we need to

1- Name it like my_tuple.
2- Put a = .
3- Put () after the = .
4- Put some data between the () separated with (,).

c) For text elements we must put it between double cots (“ “).
d) For numbers we put them without the double cots.

Ex\

```
My_tuple = (‘v’, 34, “hello” )
```

The key difference between the tuples and lists is that **while the tuples are immutable objects the lists are mutable**. This means that tuples cannot be changed while the lists can be modified. Tuples are more memory efficient than the lists

Inputting data:

We use that way to make data and store it
But if we wanted to take date as an input we would use the (input()) instruction and we can prompt the use and say something to him when he is inputting by putting what we want to say in the () of the input function between cots.
but we will need to store that input in a variable so we can use it. And we will do that by putting it after the ' = '.
Ex\

```
My_input = input("input something")
```

Printing data to output:

To output something we will use the (print()) command.
By putting data in between the brackets () of the print command it will output it.
But it have some rolls
1- If we want to print text we must put it between (' ' , " ").
2- If we want to print something but not a text we will just pass is in the brackets without it between the cots.
Ex\

```
print("hello world")
print(123)
x = 23
print(x)          # x is not like 'x' in a text but x represents a variable that
```

contains something like a number or a text and the print will print it content but not it as a text.

Converting data types:

Sometimes we might need to change the data type of some data so we can use it in something else to do that we would use the (name of the data type that we want and give it a () and in it we would put the data that we want to change it type) as follows ..
1- If we want to convert to integer use (int(something)).
2- If we want to convert to string use (str(something)).
3- If we want to convert to float use (float(something)).
4- If we want to convert to double use (double(something)).
5- If we want to convert to char use (char(something)).
Ex\

```
Int("45")
Str(435)
```

Control flow statements:

Control flow is the way that we can control the flow and stream of the data and Gide it and also control it in the system for processing
And we use them when we want to make a decision based on a condition or loop to something . and we have tools to help us do that like
1- The - if , elif , else - statements.
2- Loops like - for , while - loops.

if statements

We can use the if statements when we have a decision to make based on a condition by Saying (if(something ‘ >, <, == ,<=, >=,’ something else): then do something).
Ex\

```
If(23 > 20):
  print(“its bigger”)
```

else

And now if we wanted to Handel the (if not part) of the condition by saying else.
Ex\

```
If(23 > 30):
  print(“its bigger”)
else:
 Print(“its not”)
```

elif

And now if we wanted to do something else with other condition.
We would use the elif statemen
Ex\

```
If (20 < 30):
  print(“its bigger”)
elif(20 > 10):
  print(“20 is bigger than 10”)
```

if statements and logical operators

When we specify a condition we can say (if something and other thing or something but something not other thing).
The and , or , but , not are the logical operators that we can use to make more specify a conditions. To make specify a condition in python we use the
1- And represented by (&, and).
2- Or represented by (| , or).
3- Xor represented (^ , xor).
4- Not represented (~ , not).

Ex\

```
If ( 20 > 10 and 20 < 30 ):
  Print(“ it is ”)
```

Ex\

```
If ( 12 <= 10 or 12 > 23):
  Print(“ it is ”)
```

Keep in mind that the == is saying (the same as) something.
and = is being used for (assignment).

Ex\

```
x = 20
If ( x == 20 ):
  print(“true”)
```

Working with loops

Loops are super useful because they help us when we have repetition needs
Like looping for a set of elements in a list or a range of numbers and do something for each element in each rotation. And we have mostly 2 types of loops
The (for, while) loop.
And constructed by

```
For element in x:
  Do something
```

```
while something:
  do something
```

1- For.

The for loop is a type of loop that we can use when we know how much we want to loop on something like a range or a set of elements in an array
Its used for known number of a iterations
Ex\

```
x = [1,2,3,4,5,6]
for element in x:
  Print(element)
```

Ex\

```
x = 10
For element in range(0, x):
  Print(element)
```

2- While.

We use the while loop when we don’t know how many times we want to iterate of something
Or when a condition is true we want to do something

Ex\

```
x = True
while x:
 print(“ hello”)
```

Working with files

Working with files is an important thing . and we need it so we can read, write and store information for further processing and history tracking for databases etc..

Python’s “with open(...) as ...”

Reading and writing data to files using Python is pretty straightforward. To do this, you must first open files in the appropriate mode. Here’s an example of how to use Python’s “with open(...) as ...” pattern to open a text file and read its contents:
The “with” opens and closes automatically if the user didn’t close the file

```
with open('data.txt', 'r') as f:
    data = f.read()
```

Thus the normal case

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

To read all lines use

```
f = open('story.txt')for line in f:
    print(line)     # Or do whatever you wish to line
myfile.close()     # Good habit: close a file when you are done with it.
```

Or for single spaces

```
f = open('story.txt')
for line in f:
    line = line.strip('\n')
    print(line)
```

```
Or use the read method
filename = "story.txt"
myfile = open(filename)
s = myfile.read()  # Read the whole file into string s.print(s)myfile.c
lose()
```

Make sure your files path is in the same directory as the python file
Or you would need to pacify its full path.
In addition if we had more than one element in the file
We must print it using loops as in ..

open() takes a filename and a mode as its arguments. r opens the file in read only mode. To write data to a file, pass in w as an argument instead:

```
with open('data.txt', 'w') as f:
    data = 'some data to be written to the file'
    f.write(data)
```

Working with classes and objects

Classes and objects, are one of the most simple ideas that any human can understand.
You should ask yourself one question. that is ..
if I know that computers only understand numbers that means if I wanted to make the computer understand things around me, I must represent them with numbers ! Like integers or float maybe even double.
But unfortunately,
sometimes the things that we want to express to the computer are not always easy to be described as ONE number they might be complex and maybe contain a lot of things like a car you can't describe a car by a number alone!
But you can describe the parts that creates the car like the car frame, engine type , number of tires, etc.
Also, you can describe the house with it basic components like the area of the house number of floors, number of rooms, etc.
So the big idea here is ..
that if I found a complex thing and I cannot describe it easily,
I will use the idea of divide and conquer .
I will basically say that this big problem is a collection of many small problems that I can handle easily and solve so I will divided it to each small problem that can be described as a number easily like the cars tires for example, or the floors in the house and then if I described a good amount of small problems that defined this one big problem, then I will be able to represent this complex object and this is what we call in programming and "class" and whatever the thing that is representing is called an object.
now after we learned how we can describe a complex object like a car or a house, etc.
We can further increase the description to another level that we can now describe how this object behaves in the car example the car can move and drive that's a thing a car can do, or as we call it a function that a car can execute.
so we can represent a way that describe this function in the car class
What are the type of things that a car can do that we can describe as a function like it can honk the horn for example.
So classes can describe a complex object from the simple properties that represent this object, and also can describe the functions that this object can do .

In python a class can be created by using the class operator and the name of the class AKA the object that we would like to represent

```
class car:
```

Now you can describe the elements that represents a car
Like engine or model NO. Tiers , etc ..

```
class car:
    #properties of car
    tiers='4'
    color='red'
```

Now are car blue_print is ready
Now we can create cars
But all of are cars would have red color and 4 tiers

To create a car AKA a car object wed just invoke the car class to a car name like car1
Then to look at our cars properties
We can access them using the '.' dot operator
We can access any propriety like name or color or even functions

```
car1 = car
print(car1.color)
```

But now if we wanted to create a car and change it color into other color
We'd need to create a thing that takes are wanted properties and creates the car for us
This is called the constructor
And it's a function that when we create an object it'll be invoked automatically
That would take some parameters that we'd pass in that'll represent are wanted car properties and assign them to the class data
And we create a constructor by using the def __init__ (self):
Special constructor function .
Thatll be invoked when we create a car object
And take the color we want and the number of tiers as as input
So that it'll assign it to the color / tiers or the car properties
And the self parameter represents the object name car 1

```
class car:
    def __init__(self,color_wanted,tiers_wanted):
        self.color=color_wanted
        self.tiers=tiers_wanted
car1= car('red',4)
print(car1.color)
print(car1.tiers)
```

Working with sockets and networking

pythons sock programming in other words known as bison. Communication programming, sockets are. Nothing but a virtual end point. That we would want to connect to receive connection into.
Bumping up from the famous communication scheme of sender and receiver to suck, it's gonna be initiated one as a sander and other as a receiver, AK server, and a client in this type of socket can be configured to be any type of socket for example, like a TCP socket or UDP socket also can be configured to be a Bluetooth socket and other types of sockets

Background

Sockets have a long history. Their use originated with ARPANET in 1971 and later became an API in the Berkeley Software Distribution (BSD) operating system released in 1983 called Berkeley sockets.

When the Internet took off in the 1990s with the World Wide Web, so did network programming. Web servers and browsers weren't the only applications taking advantage of newly connected networks and using sockets. Client-server applications of all types and sizes came into widespread use.

Today, although the underlying protocols used by the socket API have evolved over the years, and new ones have developed, the low-level API has remained the same.

The most common type of socket applications are client-server applications, where one side acts as the server and waits for connections from clients. This is the type of application that you'll be creating in this tutorial. More specifically, you'll focus on the socket API for Internet sockets, sometimes called Berkeley or BSD sockets. There are also Unix domain sockets, which can only be used to communicate between processes on the same host.

Python's socket module provides an interface to the Berkeley sockets API. This is the module that you'll use in this tutorial.

The primary socket API functions and methods in this module are:

- socket()
- .bind()
- .listen()
- .accept()
- .connect()
- .connect_ex()
- .send()
- .recv()
- .close()

Python provides a convenient and consistent API that maps directly to system calls, their C counterparts. In the next section, you'll learn how these are used together.

As part of its standard library, Python also has classes that make using these low-level socket functions easier. Although it's not covered in this tutorial, you can check out the socketserver module, a framework for network servers. There are also many modules available that implement higher-level Internet protocols like HTTP and SMTP. For an overview, see Internet Protocols and Support.

TCP Sockets

You're going to create a socket object using socket.socket(), specifying the socket type as socket.SOCK_STREAM. When you do that, the default protocol that's used is the Transmission Control Protocol (TCP). This is a good default and probably what you want.

Why should you use TCP? The Transmission Control Protocol (TCP):

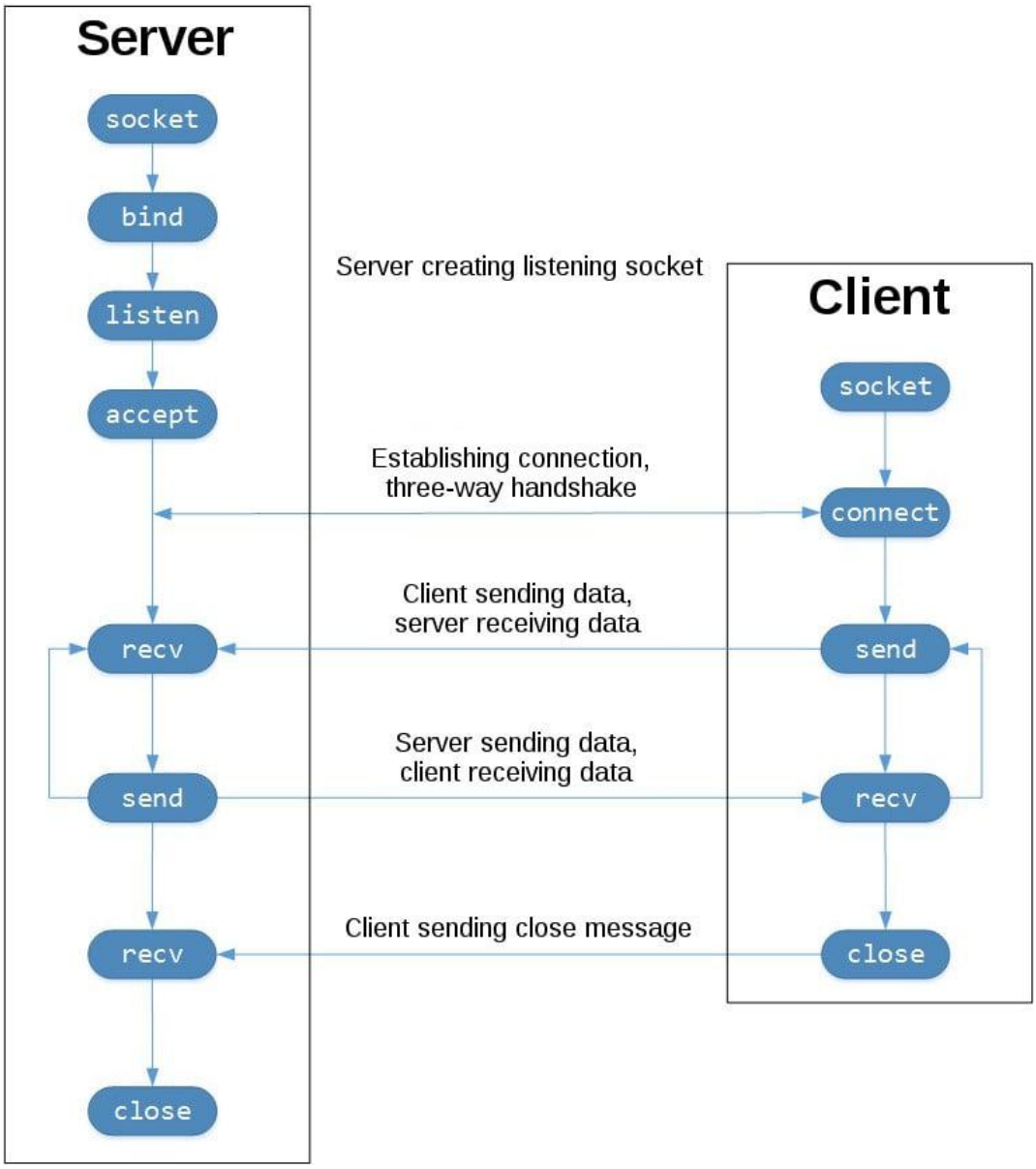
- **Is reliable:** Packets dropped in the network are detected and retransmitted by the sender.
- **Has in-order data delivery:** Data is read by your application in the order it was written by the sender.

In contrast, User Datagram Protocol (UDP) sockets created with socket.SOCK_DGRAM aren't reliable, and data read by the receiver can be out-of-order from the sender's writes.

Why is this important? Networks are a best-effort delivery system. There's no guarantee that your data will reach its destination or that you'll receive what's been sent to you.

Network devices, such as routers and switches, have finite bandwidth available and come with their own inherent system limitations. They have CPUs, memory, buses, and interface packet buffers, just like your clients and servers. TCP relieves you from having to worry about packet loss, out-of-order data arrival, and other pitfalls that invariably happen when you're communicating across a network.

To better understand this, check out the sequence of socket API calls and data flow for TCP:



A listening socket does just what its name suggests. It listens for connections from clients. When a client connects, the server calls .accept() to accept, or complete, the connection.

The client calls .connect() to establish a connection to the server and initiate the three-way handshake. The handshake step is important because it ensures that each side of the connection is reachable in the network, in other words that the client can reach the server and vice-versa. It may be that only one host, client, or server can reach the other.

In the middle is the round-trip section, where data is exchanged between the client and server using calls to .send() and .recv().

At the bottom, the client and server close their respective sockets.

Echo Client and Server

Now that you've gotten an overview of the socket API and how the client and server communicate, you're ready to create your first client and server. You'll begin with a simple implementation. The server will simply echo whatever it receives back to the client.

Echo Server

Here's the server:

```
# echo-server.py
import socket
HOST = "127.0.0.1"  # Standard loopback interface address
(localhost)PORT = 65432  # Port to listen on (non-privileged
ports are > 1023)
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()
    conn, addr = s.accept()
    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

Okay, so what exactly is happening in the API call?

socket.socket() creates a socket object that supports the context manager type, so you can use it in a with statement. There's no need to call s.close():

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
  pass  # Use the socket object without calling s.close().
```

The arguments passed to socket() are constants used to specify the address family and socket type. AF_INET is the Internet address family

for IPv4. SOCK_STREAM is the socket type for TCP, the protocol that will be used to transport messages in the network.

The .bind() method is used to associate the socket with a specific network interface and port number:

```
# echo-server.py
# ...
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:    s.bind((HOST, PORT))    # ...
```

The values passed to .bind() depend on the address family of the socket. In this example, you're using socket.AF_INET (IPv4). So it expects a two-tuple: (host, port).

host can be a hostname, IP address, or empty string. If an IP address is used, host should be an IPv4-formatted address string. The IP address 127.0.0.1 is the standard IPv4 address for the loopback interface, so only processes on the host will be able to connect to the server. If you pass an empty string, the server will accept connections on all available IPv4 interfaces.

port represents the TCP port number to accept connections on from clients. It should be an integer from 1 to 65535, as 0 is reserved. Some systems may require superuser privileges if the port number is less than 1024.

Here's a note on using hostnames with .bind():

> "If you use a hostname in the host portion of IPv4/v6 socket address, the program may show a non-deterministic behavior, as Python uses the first address returned from the DNS resolution. The socket address will be resolved differently into an actual IPv4/v6 address, depending on the results from DNS resolution and/or the host configuration. For deterministic behavior use a numeric address in host portion." (Source)

You'll learn more about this later, in Using Hostnames. For now, just understand that when using a hostname, you could see different results depending on what's returned from the name resolution process. These results could be anything. The first time you run your application, you might get the address 10.1.2.3. The next time, you get a different address, 192.168.0.1. The third time, you could get 172.16.7.8, and so on.

In the server example, .listen() enables a server to accept connections. It makes the server a "listening" socket:

```
# echo-server.py
# ...
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))    s.listen()    conn, addr = s.accept()    # ...
```

The .listen() method has a backlog parameter. It specifies the number of unaccepted connections that the system will allow before refusing new connections. Starting in Python 3.5, it's optional. If not specified, a default backlog value is chosen.

If your server receives a lot of connection requests simultaneously, increasing the backlog value may help by setting the maximum length of the queue for pending connections. The maximum value is system dependent. For example, on Linux, see /proc/sys/net/core/somaxconn.

The .accept() method blocks execution and waits for an incoming connection. When a client connects, it returns a new socket object representing the connection and a tuple holding the address of the client. The tuple will contain (host, port) for IPv4 connections or (host, port, flowinfo, scopeid) for IPv6. See Socket Address Families in the reference section for details on the tuple values.

One thing that's imperative to understand is that you now have a new socket object from .accept(). This is important because it's the socket that you'll use to communicate with the client. It's distinct from the listening socket that the server is using to accept new connections:

```
# echo-server.py
# ...
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
  s.bind((HOST, PORT))
  s.listen()  conn, addr = s.accept()  with conn:  print(f"Connected by {addr}")
while True:  data = conn.recv(1024)  if not data:  break
conn.sendall(data)
```

After .accept() provides the client socket object conn, an infinite while loop is used to loop over blocking calls to conn.recv(). This reads whatever data the client sends and echoes it back using conn.sendall().

If conn.recv() returns an empty bytes object, b'', that signals that the client closed the connection and the loop is terminated. The with statement is used with conn to automatically close the socket at the end of the block.

Echo Client

Now let's look at the client:

```
# echo-client.py
import socket
HOST = "127.0.0.1"  # The server's hostname or IP addressPORT = 65432  # The port used by the server
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
  s.connect((HOST, PORT))
```

```
    s.sendall(b"Hello, world")
    data = s.recv(1024)
print(f"Received {data!r}")
```

In comparison to the server, the client is pretty simple. It creates a socket object, uses .connect() to connect to the server and calls s.sendall() to send its message. Lastly, it calls s.recv() to read the server's reply and then prints it.

Running the Echo Client and Server

In this section, you'll run the client and server to see how they behave and inspect what's happening.

Open a terminal or command prompt, navigate to the directory that contains your scripts, ensure that you have Python 3.6 or above installed and on your path, then run the server:

```
$ python echo-server.py
```

Your terminal will appear to hang. That's because the server is blocked, or suspended, on .accept():

```
# echo-server.py
# ...
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.bind((HOST, PORT))
    s.listen()    conn, addr = s.accept()    with conn:
        print(f"Connected by {addr}")
        while True:
            data = conn.recv(1024)
            if not data:
                break
            conn.sendall(data)
```

It's waiting for a client connection. Now, open another terminal window or command prompt and run the client:

```
$ python echo-client.py Received b'Hello, world'
```

In the server window, you should notice something like this:

```
$ python echo-server.py Connected by ('127.0.0.1', 64623)
```

In the output above, the server printed the addr tuple returned from s.accept(). This is the client's IP address and TCP port number. The port number, 64623, will most likely be different when you run it on your machine.