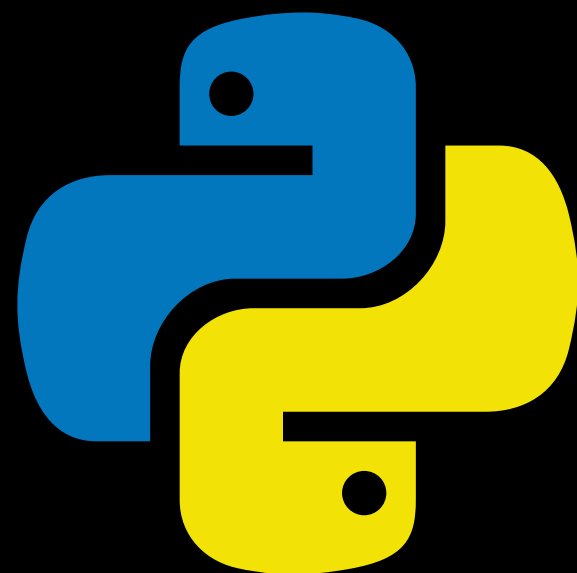


PYTHON



CHEATSHEET

BEGINNERS

MAIN DATATYPES

Integers (int) – an integer number to represent an object such as “3”.

float – we use them to represent floating-point numbers. For example “0.5”.

Complex Numbers – represented by complex class. It is specified as (real part) + (imaginary part)*j*. For example – “2+3*j*”.

Strings – codify a sequence of characters using a string. For example, the word “hello”.

Boolean – Data type with one of the two built-in values, **True** or **False**.

MAIN DATATYPES




List – Python Lists are similar to arrays in C. However, the list can contain data of different types ex `["apple", "banana", "cherry"]`

Tuple – tuple is similar to the list, a read-only data structure as we can't modify the size and value of the items of a tuple, ex `("apple", "banana", "cherry")`

Dictionary – an unordered set of a key-value pair of items. Ex `{"name": "John", "age": 36}`

Set – Set is the unordered collection of the data type. It is iterable, mutable, and has unique elements. Ex `{"apple", "banana", "cherry"}`

HOW TO CREATE A **STRING**



```
1 my_string = "Let's Learn Python  
  with Indian Servers"  
2 another_string = 'It may seem  
  difficult first, but you can do  
  it!'  
3 #IndianServer
```

IMP! Whichever option you choose, you should stick to it and use it consistently within your program.

STRING CONCATENATION & PRINTING

```
1 string_one = "Indian"  
2 string_two = "Servers"  
3 string_three = string_one + string_two  
4 print(string_three)  
5 #IndianServer
```

OUTPUT:- IndianServers

Note: You can't apply + operator to two different data types e.g. string + integer. If you try to do that, you'll get the following Python error:

TypeError: Can't convert 'int' object to str implicitly


STRING REPLICATION

EXAMPLE :- 'Indian Servers ' * 5 'Indian Servers Indian Servers
Indian Servers Indian Servers Indian Servers



```
1 print("Indian Servers " * 5)  
2 #IndianServer
```

OUTPUT:-



```
Indian Servers Indian Servers Indian Servers Indian  
Servers Indian Servers
```

As the name implies, this command lets you repeat the same string several times. This is done using * operator.

MATH OPERATORS

Operators	Operation	Example
**	Exponent	$2 ** 3 = 8$
%	Modulus	$21 \% 2 = 1$
//	Integer Division	$22 / 8 = 2$
/	Division	$21 / 3 = 7$
*	Multiplication	$3 * 3 = 9$
-	Subtraction	$2 - 1 = 1$
+	Addition	$1 + 1 = 2$

BUILT-IN FUNCTIONS

Input() function :- `input()` function is a simple way to prompt the user for some input (e.g. provide their name). All user input is stored as a string



```
1 name = input("Hi! What's your name? ")
2 print("Nice to meet you " + name + "!")
3 age = input("How old are you ")
4 print("So, you are already " + str(age)
5     + " years old, "
6     + name + "!")
6 #IndianServer
```

OUTPUT:- Hi! What's your name? "Indian Servers" Nice to meet you, Indian Servers! How old are you? 13 So, you are already 13 years old, Indian Servers!

DEFINE A FUNCTION

Apart from using in-built functions, Python also allows you to define your own functions for your program.

Syntax:-

```
def function_name(aregument1, argument2,...):  
    statement 1  
    statement 2  
    .....
```

EXAMPLE:-

```
1 def name(): #defining function  
2 | print("What's your name?") #function statement  
3 name() #calling function  
4 #IndianServer
```

OUTPUT:- What's your name?

if, else, elif – used for conditional branching or decision making

from, import – used to import modules into the current namespace

except, raise, try – used with exceptions in Python.

for – for is used for looping

finally – used with try...except block to close up resources or file streams.

global – used to declare that a variable inside the function is global

in – used to test if a sequence (list, tuple, string etc.) contains a value.

is – used in Python for testing object identity

lambda – used to create an anonymous function

nonlocal – used to declare that a variable inside a nested function

return – used inside a function to exit it and return a value

pass – pass is a null statement in Python

while – used for looping in Python.

with – used to wrap the execution of a block of code within methods defined by the context manager.

yield – used inside a function like a return statement.
But it returns a generator.

DEFINE A FUNCTION

True, False – True and False are truth values in Python.

assert – used for debugging purposes.

and, or, not – Logical operators in Python,

as – create an alias while importing a module.

None – Represents the absence of a value or a null value

async, await – used to write concurrent code in Python

del – del is used to delete the reference to an object.

class – used to define a new user-defined class in Python.

def – def is used to define a user-defined function.

break, continue – used inside for and while loops to alter their normal behavior.

if, else, elif – used for conditional branching or decision making

from, import – used to import modules into the current namespace

except, raise, try – used with exceptions in Python.

for – for is used for looping

finally – used with try...except block to close up resources or file streams.

global – used to declare that a variable inside the function is global

in – used to test if a sequence (list, tuple, string etc.) contains a value.

is – used in Python for testing object identity

lambda – used to create an anonymous function

nonlocal – used to declare that a variable inside a nested function

return – used inside a function to exit it and return a value

pass – pass is a null statement in Python

while – used for looping in Python.

with – used to wrap the execution of a block of code within methods defined by the context manager.

yield – used inside a function like a return statement.
But it returns a generator.

PASS KEYWORD ARGUMENTS

A function can also accept **keyword arguments**. In this case, you can use **parameters** in random order as the **Python interpreter** will match the values to the parameters

```
1  # Define function with parameters
2  def product_info(product_name, price):
3      print("Product Name: " + product_name)
4      print("Price " + str(price))
5  # Call function with parameters assigned as above
6  product_info("White T-shirt: ", 15)
7  # Call function with keyword arguments
8  product_info(product_name="jeans", price=45)
9  #IndianServer
```

OUTPUT:-

Product Name: White T-shirt:
Price 15
Product Name: jeans
Price 45

LISTS

Tuples are similar to lists — they allow you to display an ordered sequence of elements. However, they are **immutable** and you can't change the values stored in a tuple. The advantage of using tuples over lists is that the former are slightly **faster**. So it's a nice way to optimize your code.

EXAMPLE:-

```
my_list = [1, 2, 3]
my_list2 = ["a", "b", "c"]
my_list3 = ["4", d, "book", 5]
```

Alternatively, you can use `list()` function to do the same

```
alpha_list = list(("1", "2", "3"))
print(alpha_list)
```

ADD ITEMS TO A LIST

The first one is using **append()** function:

```
1 # creating a list
2 beta_list = ["apple", "android", "python"]
3 # adding item to the list
4 beta_list.append("Indian Servers")
5 print(beta_list)
6 #IndianServers
```

The second option is to **insert()** function to add an item at the specified index:

```
1 # creating a list
2 beta_list = ["apple", "android", "python"]
3 # adding item to the list
4 beta_list.insert(2, "Indian Servers")
5 print(beta_list)
6 #IndianServers
```

OUTPUT:- ['apple', 'android', 'Indian Servers', 'python']

REMOVE AN ITEM FROM A LIST

The first one is using `remove()` function:

```
1  # creating a list
2  beta_list = ["Indian Servers", "python", "android"]
3  # removing item to the list
4  beta_list.remove("android")
5  print(beta_list)
6  #IndianServers
```

The second option is to `pop()` function if no index is specified it will remove the last item

```
1  # creating a list
2  beta_list = ["Indian Servers", "python", "android"]
3  # removing item to the list
4  beta_list.pop()
5  print(beta_list)
6  #IndianServers
```

OUTPUT - ['Indian Servers', 'python']

LIST OPERATIONS

Combine two lists - To mash up two lists use the + operator.

```
1 my_list = [1, 2, 3]
2 my_list2 = ["a", "b", "c"]
3 combo_list = my_list + my_list2
4 print(combo_list)
5 #IndianServers
```


OUTPUT - `combo_list = [1, 2, 3, 'a', 'b', 'c']`

Create Nested list - You can also create a list of your lists when you have plenty of them

```
1 my_nested_list = [my_list, my_list2]
2 print(my_nested_list)
3 #IndianServers
```

OUTPUT - `my_nested_list [[1, 2, 3], ['a', 'b', 'c']]`


Sort a list – Use the `sort()` function to organize all items in your list.



```
1 alpha_list = [34, 23, 67, 100, 88, 2]
2 alpha_list.sort()
3 print(alpha_list)
4 #IndianServers
```

OUTPUT - sorted list = [2, 23, 34, 67, 88, 100]

Slice a List – if you want to call just a few elements from your list



```
1 alpha_list = [34, 23, 67, 100, 88, 2]
2 alpha_list.sort()
3 print(alpha_list[0:4])
4 #IndianServers
```

OUTPUT - sliced list [2, 23, 34, 67]

Change an item of list – You can easily overwrite a value of one list items:

```
1 beta_list = ["Indian Servers", "python", "android"]
2 beta_list[2] = "apple"
3 print(beta_list)
4 #IndianServers
```

OUTPUT - ['Indian Servers', 'python', 'apple']

Loop through a List – Using for loop you can multiply the usage of certain items

```
1 beta_list = ["Indian Servers", "python", "android"]
2 for x in range(1,4):
3     beta_list += ['tech']
4 print(beta_list)
5 #IndianServers
```


OUTPUT - ['Indian Servers', 'python', 'android', 'tech', 'tech', 'tech']

Copy a List – Use the built-in **copy()** function to replicate your data:



```
1 beta_list = ["Indian Servers", "python", "android"]
2 beta_list = beta_list.copy()
3 print(beta_list)
4 #IndianServers
```

Alternatively, you can copy a list with the list() method:



```
1 beta_list = ["Indian Servers", "python", "android"]
2 beta_list = list(beta_list)
3 print(beta_list)
4 #IndianServers
```

OUTPUT - ['Indian Servers', 'python', 'android']

TUPLES

Tuples are similar to lists — they allow you to display an ordered sequence of elements. However, they are **immutable** and you can't change the values stored in a tuple. The advantage of using tuples over lists is that the former are slightly **faster**. So it's a nice way to optimize your code.

How to Create a **Tuple**

```
1 my_tuple = (1, 2, 3, 4, 5)
2 #IndianServers
```

Note: Once you create a tuple, you can't add new items to it or change it in any other way

Slice a tuple – the process is similar to slicing lists.

```
1 numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12)
2 print(numbers[1:11:2])
3 #IndianServers
```

OUTPUT - (1, 3, 5, 7, 9)

Tuple to a List – Since Tuples are immutable, you can't change them. What you can do though is convert a tuple into a list, make an edit and then convert it back to a tuple.

```
1 x = ("Indian Servers", "Python", "Android")
2 y = list(x)
3 y[2] = "iPhone"
4 x = tuple(y)
5 print(x)
6 #IndianServers
```

OUTPUT - ('Indian Servers', 'Python', 'iPhone')

DICTIONARIES

A **dictionary** holds indexes with **keys** that are mapped to certain values. These **key-value** pairs offer a great way of organizing and storing data in Python. They are **mutable**, meaning you can change the stored information

A key value can be either a string, Boolean, or integer. Here's an example **dictionary** illustrating this:

```
customer1= {  
    'username': 'sai-satish',  
    'online': 'false',  
    'friends':100  
}
```

Create Dictionary – Here's a quick example showcasing how to make an empty dictionary.

Option 1: `new_dict = {}`

Option 2: `other_dict= dict()`

And you can use the same two approaches to add values to your dictionary:

```
1 new_dict= {  
2     "brand": "Indian Servers",  
3     "industry": "education",  
4     "year": 2008  
5 }  
6 #IndianServers
```

Access a Value in a Dictionary – You can access any of the values in your dictionary the following way:

```
x = new_dict["brand"]
```

You can also use the following methods to accomplish the same.

- **dict.keys()** isolates keys
- **dict.values()** isolates values
- **dict.items()** returns items in a list format of (key, value) tuple pairs

Change Item Value - To change one of the items, you need to refer to it by its key name:

```
1  #change the year to 2021
2  new_dict= {
3      "brand": "Indian Servers",
4      "industry": "education",
5      "year": 2008
6  }
7  new_dict["year"] = 2021
8  print(new_dict)
9  #IndianServers
```

OUTPUT - {'brand': 'Indian Servers', 'industry': 'education', 'year': 2021}

Loop through a Dictionary – Again to implement looping, use for loop command

```
1 new_dict = {
2     "brand": "Indian Servers",
3     "industry": "Education",
4     "year": 2008
5 }
6 #prints all key names
7 for x in new_dict:
8     print(x)
9 #prints all values
10 for x in new_dict:
11     print(new_dict[x])
12 #prints both
13 for x, y in new_dict.items():
14     print(x, y)
15 #IndianServers
```

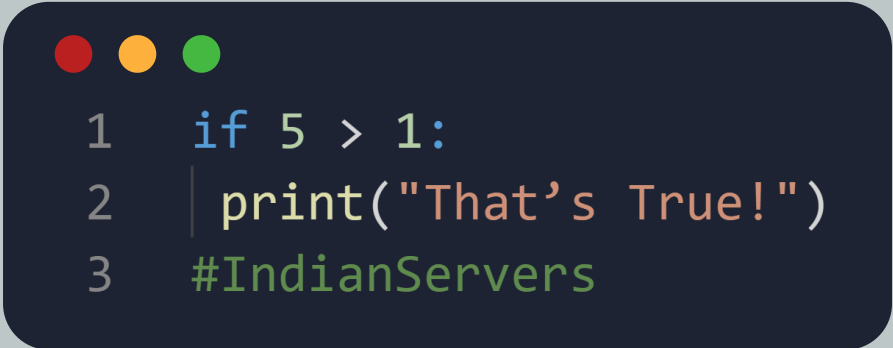

IF STATEMENTS IN PYTHON

Just like other programming languages, Python supports the basic logical conditions from math:

- Equals: $a == b$
- Not Equals: $a != b$
- Less than: $a < b$
- Less than or equal to $a <= b$
- Greater than: $a > b$
- Greater than or equal to: $a >= b$

You can leverage these conditions in various ways. But most likely, you'll use them in "if statements" and loops.

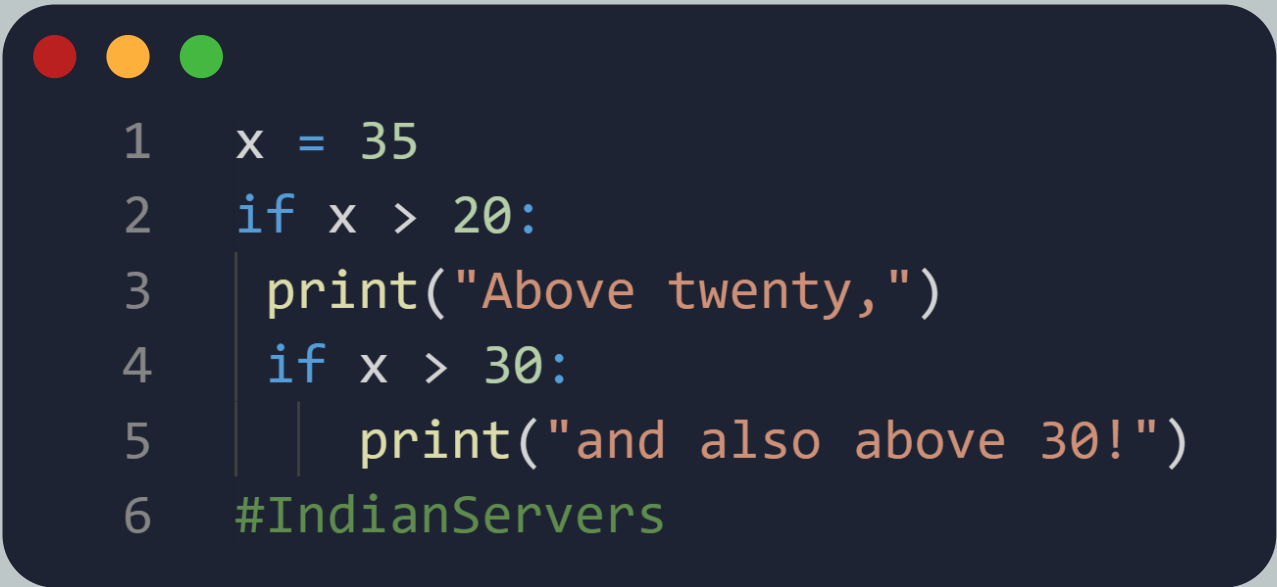
If Statement Example – The goal of a conditional statement is to check if it's True or False.



```
1  if 5 > 1:
2      print("That's True!")
3  #IndianServers
```

OUTPUT - That's True!

Nested If Statements – For more complex operations, you can create nested if statements. Here's how it looks:



```
1  x = 35
2  if x > 20:
3      print("Above twenty,")
4      if x > 30:
5          print("and also above 30!")
6  #IndianServers
```

OUTPUT - Above twenty,
and also above 30!

Elif Statements – **elif** keyword prompts your program to try another condition if the previous one(s) was not true. Here's an example

```
1 a, b = 45, 45
2 if b > a:
3     print("b is greater than a")
4 elif a == b:
5     print("a and b are equal")
6 #IndianServers
```

OUTPUT - a and b are equal

If Else Statements – **else** keyword helps you add some additional filters to your condition clause. Here's how an if-elif-else combo looks:

```
1 print("Enter your age")
2 age = int(input())
3 if age < 4:
4     ticket_price = 0
5 elif age < 18:
6     ticket_price = 10
7 else: ticket_price = 15
8 print(ticket_price)
9 #IndianServers
```


OUTPUT -

Case I:- if age is less than 4 ticket_price = 0

Case II:- if age is greater than 4 but less than 18 then ticket_price = 10

Case III:- if age is greater than 18 ticket_price = 15


If-Not-Statements – Not keyword let's you check for the opposite meaning to verify whether the value is NOT True:



```
1 new_list = [1, 2, 3, 4]
2 x = 10
3 if x not in new_list:
4     print("'x' isn't on the list, so this is
      True!")
5 #IndianServers
```

OUTPUT - That's True!

Pass Statements – If statements can't be empty. But if that's your case, add the pass statement to avoid having an error:



```
1 a, b = 33, 200
2 if b > a:
3     pass
4 #IndianServers
```

PYTHON LOOPS

Python has two simple **loop** commands that are good to know:

- **for loops**
- **while loops**

for loop – for loop is a handy way for iterating over a sequence such as a list, tuple, dictionary, string, etc.

```
1  for letter in 'Python':  
2      | print ('Current Letter :', letter)  
3      #IndianServers
```

OUTPUT - Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n

while loop – While loop enables you to execute a set of statements as long as the condition for them is true.

```
1 x, i = "Indian Servers", 1
2 while i < 8:
3     print(x)
4     i += 1
5 #IndianServers
```

OUTPUT -

Indian Servers
Indian Servers
Indian Servers
Indian Servers
Indian Servers
Indian Servers
Indian Servers

Nested Loop – You can also stop the loop from running even if the condition is met. For that, use the break statement both in while and for loops:

```
1 i = 2
2 while(i < 100):
3     j = 2
4     while(j <= (i/j)):
5         if not(i%j): break
6         j = j + 1
7     if (j > i/j) : print (i, " is prime")
8     i = i + 1
9 #IndianServers
```

OUTPUT -

2 is prime 3 is prime 5 is prime 7
is prime 11 is prime 13 is prime 17
is prime 19 is prime 23 is prime 29
is prime 31 is prime 37 is prime 41
is prime 43 is prime 47 is prime 53
is prime 59 is prime 61 is prime 67
is prime 71 is prime 73 is prime 79
is prime 83 is prime 89 is prime 97
is prime

Break a loop – You can also stop the loop from running even if the condition is met. For that, use the break statement both in while and for loops:

```
1  i = 1
2  while i < 8:
3      print(i)
4      if i == 4:
5          break
6      i += 1
7  #IndianServers
```

OUTPUT -

1
2
3
4

PYTHON - NETWORK PROGRAMMING

Python provides two levels of access to **network** services. At a low level, you can access the basic socket support in the underlying operating system, which allows you to implement clients and servers for both **connection-oriented** and **connectionless** protocols.

Python also has libraries that provide higher-level access to specific application-level network protocols, such as **FTP**, **HTTP**, and so on.

SOCKETS

Sockets are interior endpoints built for **sending** and **receiving** data. A single **network** will have two sockets, one for each communicating device or program. These sockets are a combination of an IP address and a **Port**. A single device can have 'n' number of sockets based on the port number that is being used. Different ports are available for different types of **protocols**.

Protocol	Port	Functions
HTTP	80	Web Pages
FTP	20	File Transfers
NNTP	119	Unsent
SMTP	25	Sending Email
Telnet	23	Command Lines
POP3	110	Fetching Email
Gopher	70	Document Transfer

SOCKET TYPES

SOCK_STREAM – For TCP protocols, Reliable transmission, Packet sequence, Connection-oriented, Bi-directional

SOCK_DGRAM – For UDP protocols, Unreliable transmission, No sequence of packets, Connectionless(UDP), Not Bidirectional

Socket Family – AF_UNIX of AF_INET

import socket #Imports the socket method

socket.**socket**() #Function that creates socket

sock = socket.socket(**socket family, socket type, protocol=value**)

Socket Methods :

- Client socket method – connect()
- Server socket methods – bind(), listen(backlog), accept()
- TCP socket methods –
 - s.recv() #receive TCP packets
 - s.send() #send TCP packets

SERVER


A **server** is either a **program**, a **computer**, or a **device** that is devoted to managing **network resources**. Servers can either be on the same device or computer or **locally** connected to other devices and computers or even **remote**. There are various types of servers such as database servers, network servers, print servers, etc.

Servers commonly make use of methods like **socket.socket()**, **socket.bind()**, **socket.listen()**, etc to establish a connection and bind to the clients.

```
1 import socket
2 s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3 s.bind((socket.gethostname(),1234))
4 #port number can be anything between 0-65535(we usually specify
  non-privileged ports which are > 1023)
5 s.listen(5)
6
7 while True:
8     clt,adr=s.accept()
9     print(f"Connection to {adr}established")
10    #f string is literal string prefixed with f which
11    #contains python expressions inside braces
12    clt.send(bytes("Socket Programming in Python","utf-8 ")) #to send
    info to clientsocket
13 #IndianServers
```

CLIENT

A **client** is either a **computer** or **software** that receives **information** or services from the server. In a client-server module, clients **requests** for services from **servers**. The best example is a web browser such as Google Chrome, Firefox, etc. These web browsers request web servers for the required web pages and services as directed by the user. Other examples include online games, online chats, etc.



```
1  import socket
2  s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3  s.connect((socket.gethostname(), 2346))
4  msg=s.recv(1024)
5  print(msg.decode("utf-8"))
6  #IndianServers
```

SOCKET ERRORS/EXCEPTIONS

exception socket.error – A deprecated alias of OSError, raised when a system function returns a system-related error.

exception socket.herror – raised for address-related errors

exception socket.gaierror – raised for address-related errors by **getaddrinfo()** and **getnameinfo()**

exception socket.timeout – raised when a timeout occurs on a socket which has had timeouts enabled via a prior call to **settimeout()** (or implicitly through **setdefaulttimeout()**)

PYTHON NETWORK PROGRAMMING LIBRARIES

Django – High-level Python Web framework for rapid development and pragmatic

pycos – Python framework for asynchronous, concurrent, network, distributed programming and distributed computing

Diesel – A clean API for writing network clients and servers. TCP and UDP supported. Bundles client for HTTP, DNS, Redis, Riak, and MongoDB

Pulsar – Easy way to build scalable network programs

Twisted – Event-based framework for internet applications: HTTP clients and servers, SSHv2 and Telnet, IRC, XMPP, IMAPv4, POP3, SMTP, etc.

NAPALM – Network Automation and Programmability Abstraction Layer with Multivendor support – For dealing with device vendors

gevent – A coroutine-based Python networking library that uses greenlet to provide a high-level synchronous API on top of the libev or libuv event loop.

PYTHON CLASS

Since Python is an **object-oriented** programming language almost every element of it is an object — with its methods and properties.

Class acts as a blueprint for creating different **objects**. Objects are an instance of a class, where the class is manifested in some program.

create a Class – Let's create a class named TestClass, with one property named z:

```
1 class TestClass:
2     z = 5
3     #IndianServers
```

create an object – As a next step, you can create an object using your class. Here's how it's done:

```
1 p1 = TestClass()
2 print(p1.x)
3     #IndianServers
```

You can assign different attributes and methods to your object.
The example is below:

```
1 class car(object):
2     """docstring"""
3     def __init__(self, color, doors, tires):
4         """Constructor"""
5         self.color = color
6         self.doors = doors
7         self.tires = tires
8         def brake(self):
9             """
10            Stop the car
11            """
12            return "Braking"
13        def drive(self):
14            """
15            Drive the car
16            """
17            return "I'm driving!"
18 #IndianServers
```


Create a Subclass – Every object can be further sub-classified.
Here's an example

```
1  class Car(Vehicle):
2      """
3      The Car class
4      """
5      def brake(self):
6          """
7          Override brake method
8          """
9          return "The car class is breaking slowly!"
10 if __name__ == "__main__":
11     car = Car("yellow", 2, 4, "car")
12     car.brake()
13     'The car class is breaking slowly!'
14     car.drive()
15     "I'm driving a yellow car!"
16     #IndianServers
```

PYTHON EXCEPTIONS (ERRORS)

Python has a list of in-built **exceptions** (errors) that will pop up whenever you make a **mistake** in your code. As a newbie, it's good to know how to fix these.

THE MOST COMMON PYTHON EXCEPTIONS

- **AttributeError** — pops up when an attribute reference or assignment fails.
- **IOError** — emerges when some I/O operation (e.g. an `open()` function) fails for an I/O-related reason, e.g., “file not found” or “disk full”.
- **ImportError** — comes up when an import statement cannot locate the module definition. Also, when a `from... import` can't find a name that must be imported.
- **IndexError** — emerges when a sequence subscript is out of range.
- **KeyError** — raised when a dictionary key isn't found in the set of existing keys.
- **KeyboardInterrupt** — lights up when the user hits the interrupt key (such as Control-C or Delete).

PYTHON EXCEPTIONS (ERRORS)

Python has a list of in-built **exceptions** (errors) that will pop up whenever you make a **mistake** in your code. As a newbie, it's good to know how to fix these.

THE MOST COMMON PYTHON EXCEPTIONS

- **NameError** — shows up when a local or global name can't be found
- **OSError** — indicated a system-related error.
- **SyntaxError** — pops up when a parser encounters a syntax error.
- **TypeError** — comes up when an operation or function is applied to an object of inappropriate type.
- **ValueError** — raised when a built-in operation/function gets an argument that has the right type but not an appropriate value, and the situation is not described by a more precise exception such as `IndexError`.
- **ZeroDivisionError** — emerges when the second argument of a division or modulo operation is zero.

TROUBLESHOOT THE ERRORS

Python has a useful statement, design just for the purpose of handling exceptions — **try/except** statement. Here's a code snippet showing how you can catch **KeyErrors** in a dictionary using this statement:

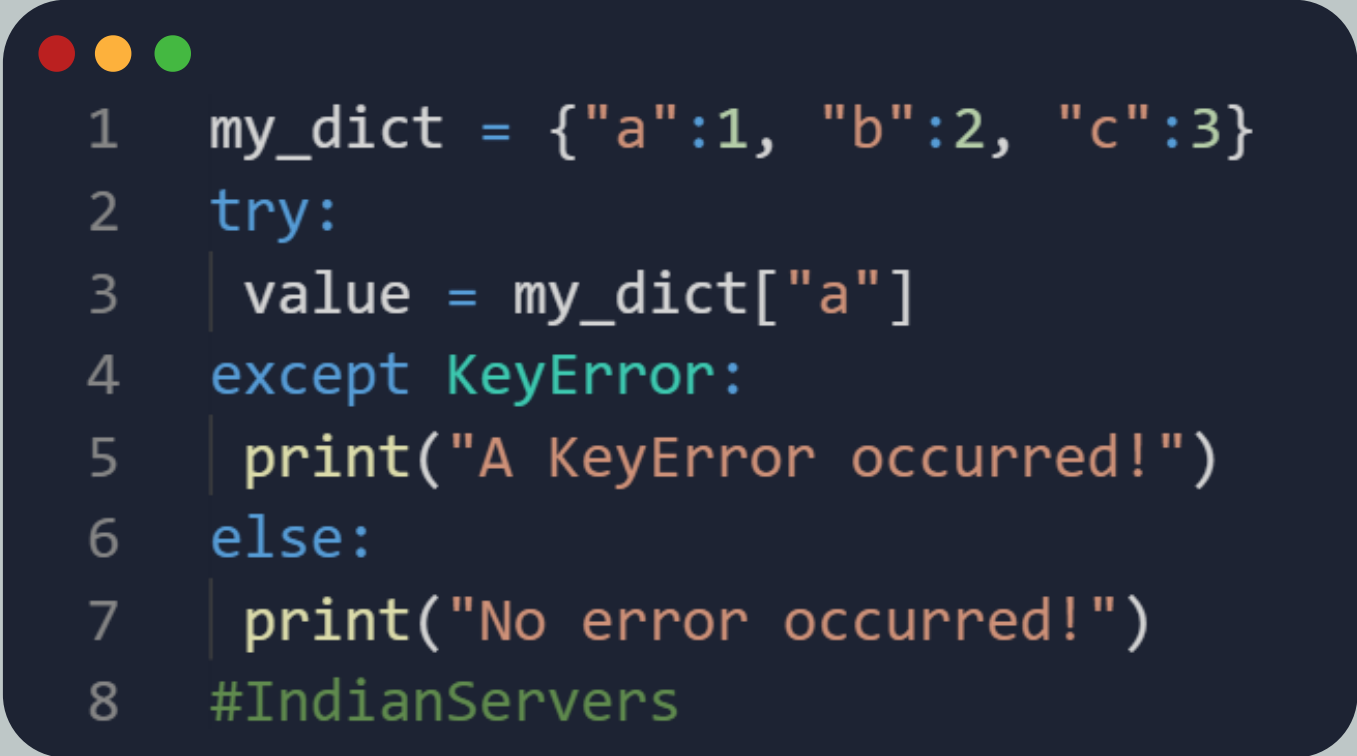
```
1 my_dict = {"a":1, "b":2, "c":3}
2 try:
3     value = my_dict["d"]
4 except KeyError:
5     print("That key does not exist!")
6 #IndianServers
```

You can also detect several exceptions at once with a single statement. Here's an example for that:

```
1 my_dict = {"a":1, "b":2, "c":3}
2 try:
3     value = my_dict["d"]
4 except IndexError:
5     print("This index does not exist!")
6 except KeyError:
7     print("This key is not in the dictionary!")
8 except:
9     print("Some other problem happened!")
10 #IndianServers
```

TRY/EXCEPT WITH ELSE CLAUSE

Adding an else clause will help you confirm that no errors were found:



```
1 my_dict = {"a":1, "b":2, "c":3}
2 try:
3     value = my_dict["a"]
4 except KeyError:
5     print("A KeyError occurred!")
6 else:
7     print("No error occurred!")
8 #IndianServers
```