

Python

Thursday, September 5, 2024 8:14 PM

- Download anaconda navigator from "Anaconda Distribution Download"
- Jupyter - notebook - most used notebook - Localhost:8888/tree
- .ipynb extension gets created when you create a new notebook. Its name is - Ipython notebook file
- Jupyter has three types of cells - "code", "raw" and "markdown". Raw is to store any code you don't want to run. Raw cells won't be evaluated by notebook

The screenshot shows a slide titled "9. Python Data Types" with the heading "Data Types". On the left, there's a Python logo and the word "Python". A table lists four data types:

DESCRIPTION	TYPE	EXAMPLE
Boolean	bool	True, False
Integer	int	3, -8, 2508
Floating Point	float	1.56, -2.8
Text String	str	'John', "Sydney"

Everything is object in python programming

Don't specify datatype of variable

The screenshot shows a slide titled "9. Python Data Types" with the heading "Data Types – Advanced". On the left, there's a Python logo and the word "Python". To the right, a list of advanced data types is shown:

1. Sequence Types – list, tuple, range, str
2. Set Types – set, frozenset
3. Mapping Types – dict

The screenshot shows a Jupyter Notebook interface with the following code execution history:

```
In [10]: x = {"milk", "bread", "eggs"}  
In [11]: type(x)  
Out[11]: set  
In [12]: x = frozenset({"milk", "bread", "eggs"})  
In [13]: type(x)  
Out[13]: frozenset
```



myVariable



str Object

Unique ID
Type
Value
Reference Count



Python



Characters Allowed :

- Lowercase letters (a through z)
- Uppercase letters (A through Z)
- Digits (0 through 9)
- Underscore (_)



Mutable vs immutable -

Int, str, float, bool, tuple, frozenset are all immutable

List, set, dictionary are all mutable



Python



Mutable types :

- List
- Set
- Dictionary

Immutable types :

- Boolean
- Integer
- Floating Point
- Text String
- Tuple
- Frozen Set



```
In [24]: help('keywords')
```

Here is a list of the Python keywords. Enter any keyword to get more help.

False	class	from	or
None	continue	global	pass
True	def	if	raise
and	del	import	return
as	elif	in	try
assert	else	is	while
async	except	lambda	with
await	finally	nonlocal	yield
break	for	not	

File Edit View Insert Cell Kernel Widgets Help

9. Python Data Types

[Image of Jupyter Notebook toolbar]

```
In [28]: import sys
x = 59802345
print(sys.getrefcount(x))
```

3

File Edit View Insert Cell Kernel Widgets Help

9. Python Data Types

[Image of Jupyter Notebook toolbar]

```
In [29]: import sys
x = 59802345
print(sys.getrefcount(x))

y = x
print(sys.getrefcount(x))
```

3

4

File Edit View Insert Cell Kernel Widgets Help

9. Python Data Types

[Image of Jupyter Notebook toolbar]

```
In [30]: import sys
x = 59802345
print(sys.getrefcount(x))

y = x
print(sys.getrefcount(x))

x = 2348790
print(sys.getrefcount(y))
```

3

4

3

When reference count of the object becomes zero, the object is ready to be garbage collected.

Mathematical Operators



Types of Operations:

- Mathematical
- Logical
- Relational
- Comparisons

Mathematical Operators

DESCRIPTION	OPERATOR	EXAMPLE
Addition	+	$5 + 2 = 7$
Subtraction	-	$5 - 2 = 3$
Multiplication	*	$5 * 2 = 10$
Floating-point Division	/	$5 / 2 = 2.5$
Integer Truncate Division	//	$5 // 2 = 2$
Modulus	%	$5 \% 2 = 1$
Exponent	**	$5 ** 2 = 25$

```
In [15]: x = 3
y = 5

x = x + y
print(x)
```

8

```
In [16]: x = 3
y = 5

x += y
print(x)
```

8

```
File Edit View Insert Cell Kernel Widgets Help
```

In [17]: x = 3
y = 5

y /= x
print(y)

1.6666666666666667

In [18]: x = 3
y = 5

y = y / x
print(y)

1.6666666666666667

- Python also follows BODMAS rule
- Python is case sensitive
- Python will broaden the data type when the math is done. For ex: here the data type is broadened to float

In [21]: `4 ** 0.5`
Out[21]: 2.0

- In python there is no need of semi colon to end the line. Each lines new line character is understood as line
- And each line is considered as command

Python Program Structure

Lexical Rules :

- Line termination
- Multiple expressions per line
- Line continuation
- Grouping of statements
- Comments

In [15]:
`year = 2020
month = 2
day = 15
hour = 10
minute = 12
second = 35

x = 1900 < year < 2100 and 0 <= month <= 12 and 0 <= day <= 30 and 0 <= hour <= 24 and 0 <= minute <= 60 and 0 <= second <= 60
print(x)`
True

For better readability of the above statements, you can use this by ending it with **backslash** -

In [4]:
`year=2020
month=2
day=15
hour=10
minute=12
second=35

x= 1900 < year < 2100 \
and 1 <= month <= 12 \
and 1 <= day <= 31 \
and 0 <= hour < 24 \
and 0 <= minute < 60 \
and 0 <= second < 60 \

print(x)`
True

Also you can start with the open bracket so that python understands it is not end of the line until it finds the closed bracket

In []: year=2020
month=2
day=15
hour=10
minute=12
second=35

x= (1900 < year < 2100
and 1 <= month <= 12
and 1 <= day <= 31
and 0 < hour < 24
and 0 <= minute < 60
and 0 <= second < 60)

In [11]: # This is a Comment.
a = 100
This is another Comment. ←

Python doesn't have multi line comment but can use the string not assigned

In [16]: # This is a Comment.
a = 100
This is another Comment.

In [17]: a = 200 # This is a Comment too.

In [18]: a = 300
This is a Long comment
which explains the concept
of multi - Line comments.

In [21]: a = 300
'''This is a long comment
which explains the concept
of multi - line comments.'''
print(a)
300

Python If - else

1. If statement
2. If - else statement
3. Nested if - else
4. If - elif

if condition:

Execute only if the condition is true.

Comparison Operators

DESCRIPTION	OPERATOR
Equality	<code>==</code>
Inequality	<code>!=</code>
Less than	<code><</code>
Less than or equal	<code><=</code>
Greater than	<code>></code>
Greater than or equal	<code>>=</code>

Logical Operators

OPERATOR	DESCRIPTION
<code>and</code>	if and only if both the expressions evaluate to true
<code>or</code>	if either side of the expression evaluates to true
<code>not</code>	reverses the state of its operand

These are not present in python.

```
File Edit View Insert Cell Kernel Widgets Help  
In [5]: a = 7  
x = not(a > 3 and a < 5)  
print(x)  
True  
In [6]: x = a > 3 and not a < 5      !>, !<  
print(x)  
True
```

```
In [5]: a = 7  
x = not(a > 3 and a < 5)  
print(x)
```

```
True
```

```
In [6]: x = a > 3 and not a < 5  
print(x)
```

```
True
```

In operator -

```
In [7]: l = 'e'  
isVowel = l == 'a' or l == 'e' or l == 'i' or l == 'o' or l == 'u'  
print(isVowel)
```

```
True
```

```
In [8]: vowels = 'aeiou'  
l = 'e'  
isVowel = l in vowels  
print(isVowel)
```

```
True
```

The screenshot shows a Jupyter Notebook interface with two code cells. Cell 7 contains the code to check if 'e' is a vowel using an OR expression. Cell 8 contains the code to check if 'e' is in the string 'aeiou'. Both cells return 'True'.

```
In [7]: l = 'e'  
isVowel = l == 'a' or l == 'e' or l == 'i' or l == 'o' or l == 'u'  
print(isVowel)  
True  
  
In [8]: vowels = 'aeiou'  
l = 'e'  
isVowel = l in vowels  
print(isVowel)  
True
```

Python Sequence Types

Sequence Type

- List
- Tuple
- Range
- Str

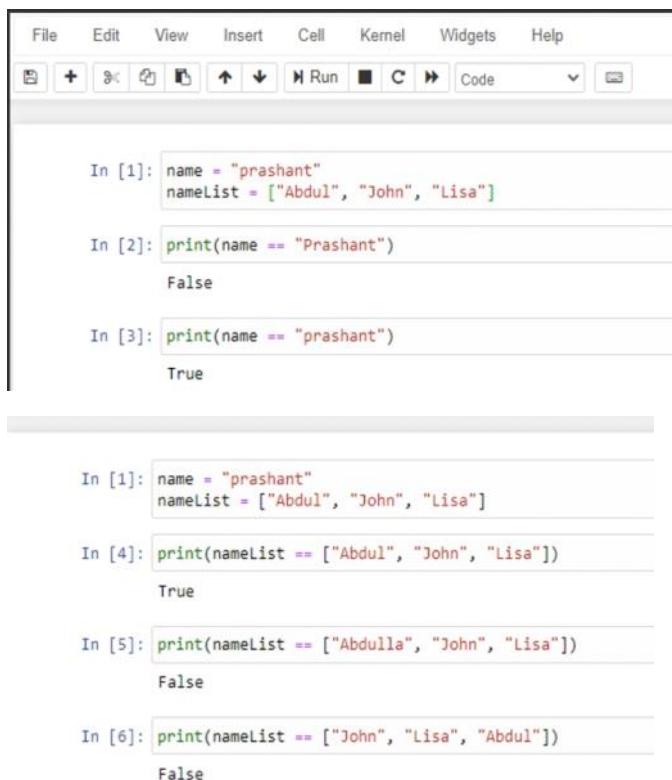
If sequence type is mutable, you can change the content and if the type is immutable then you cannot change the content.

SEQUENCE TYPE	MUTABILITY
List	Mutable
Tuple	Immutable
Range	Immutable
Str	Immutable

Sequence Operations:

1. Comparison
2. Containment Testing
3. Concatenation and Repetition
4. Element Extraction and Slicing
5. Element Index and Count

Sequence is considered same if the type, order and length and elements in the sequence should be same to be marked same.



```

File Edit View Insert Cell Kernel Widgets Help
[+] Run C > Code < >

In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]

In [2]: print(name == "Prashant")
False

In [3]: print(name == "prashant")
True

In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]

In [4]: print(nameList == ["Abdul", "John", "Lisa"])
True

In [5]: print(nameList == ["Abdulla", "John", "Lisa"])
False

In [6]: print(nameList == ["John", "Lisa", "Abdul"])
False

```

Containment testing -
Using in or not in operator

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]

In [2]: print('h' in name)
True

In [3]: print("Abdul" in nameList)
True
```

This kind of search is called as subsequence search -

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]

In [2]: print('ash' in name)
True
```

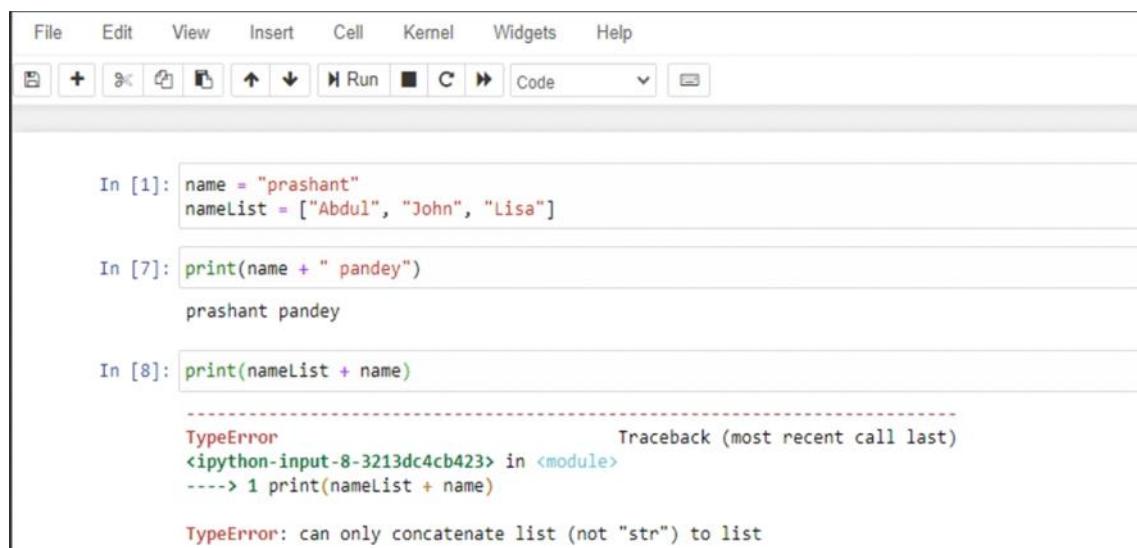
Subsequence search only works in

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]

In [2]: print('ash' in name)
True

In [3]: print(["Abdul", "John"] in nameList)
False
```

Concatenation - which means combining two sequences of same type using + operator



```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]

In [7]: print(name + " pandey")
prashant pandey

In [8]: print(nameList + name)

-----  
TypeError                                     Traceback (most recent call last)  
<ipython-input-8-3213dc4cb423> in <module>  
----> 1 print(nameList + name)

TypeError: can only concatenate list (not "str") to list
```

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]

In [7]: print(name + " pandey")
prashant pandey

In [9]: print(nameList + [name])
['Abdul', 'John', 'Lisa', 'prashant']
```

Repetition -

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]

In [2]: repeatedName = (name + " ") * 3
print("Old Name: " + name)
print("New Name: " + repeatedName)

Old Name: prashant
New Name: prashant prashant prashant

In [3]: repeatedList = nameList * 2
print("Old List: " + str(nameList))
print("New List: " + str(repeatedList))

Old List: ['Abdul', 'John', 'Lisa']
New List: ['Abdul', 'John', 'Lisa', 'Abdul', 'John', 'Lisa']
```

Concatenation and repetition are not applicable to range sequences.

Extraction and slicing -

Python Sequence Types – Element Extraction



- 1. Python Sequences are zero-indexed.
- 2. A negative index is subtracted from the sequence length.
- 3. You cannot go beyond the index boundary.

When you use 2 it is the index 2 we are trying to fetch from the left

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]
```

```
In [2]: print(name[2])
```

```
a
```

```
In [3]: print(name[-2])
```

```
n
```

Similarly when we use -2, we are trying to fetch the second index from last index element.

Counting from the left starts with 0 -

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]
```



Counting from the right starts with 1 -

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]
```



```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]
```

```
In [4]: print(name[9])
```

```
-----  
IndexError Traceback (most recent call last)  
<ipython-input-4-6f52fec778be> in <module>  
----> 1 print(name[9])  
  
IndexError: string index out of range
```

```
In [1]: name = "prashant"
nameList = ["Abdul", "John", "Lisa"]
```

```
In [6]: print(name[-9])
```

```
-----  
IndexError Traceback (most recent call last)  
<ipython-input-6-38923d30fa56> in <module>  
----> 1 print(name[-9])  
  
IndexError: string index out of range
```

So far we did extraction of single element, if we want to extract multiple characters we can use slicing -

While slicing, the stop index is not included -

Python Sequence Types – Slicing Sequences



Python

`s[start_index : stop_index : step_size]`

File Edit View Insert Cell Kernel Widgets **s [start_index : stop_index : step_size]**

In [1]: `name = "prashant kumar pandey"`
`nameList = ["Abdul", "John", "Lisa", "prashant"]`

In [2]: `print(name[0])`
p

In [3]: `print(name[0:7])`
prashan

File Edit View Insert Cell Kernel Widgets **s [start_index : stop_index : step_size]**

In [1]: `name = "prashant kumar pandey"`
`nameList = ["Abdul", "John", "Lisa", "prashant"]`

In [5]: `print(nameList[1:3])`
['John', 'Lisa']

If you don't include the start or stop index, python assumes 0 and full length respectively -

File Edit View Insert Cell Kernel Widgets **s [start_index : stop_index : step_size]**

In [1]: `name = "prashant kumar pandey"`
`nameList = ["Abdul", "John", "Lisa", "prashant"]`

In [6]: `print(name[:8])`
prashant

In [7]: `print(name[9:])`
kumar pandey

If start and stop both are skipped, it gives entire sequence -

```
In [8]: print(name[:])
```

```
prashant kumar pandey
```

Screenshot of Jupyter Notebook showing code execution:

```
In [1]: name = "prashant kumar pandey"
nameList = ["Abdul", "John", "Lisa", "prashant"]

In [11]: print(name[-6:])
pandey
```

The status bar at the top right shows the text **s [start_index : stop_index : step_size]**.

Screenshot of Jupyter Notebook showing code execution:

```
In [1]: name = "prashant kumar pandey"
nameList = ["Abdul", "John", "Lisa", "prashant"]

In [11]: print(name[-6:])
pandey

In [12]: print(name[:-13])
prashant
```

Screenshot of Jupyter Notebook showing code execution:

```
In [1]: name = "prashant kumar pandey"
nameList = ["Abdul", "John", "Lisa", "prashant"]

In [11]: print(name[-6:])
pandey

In [12]: print(name[:-13])
prashant

In [13]: print(name[9:-7])
kumar
```

Python Sequence Types – Slicing Sequences

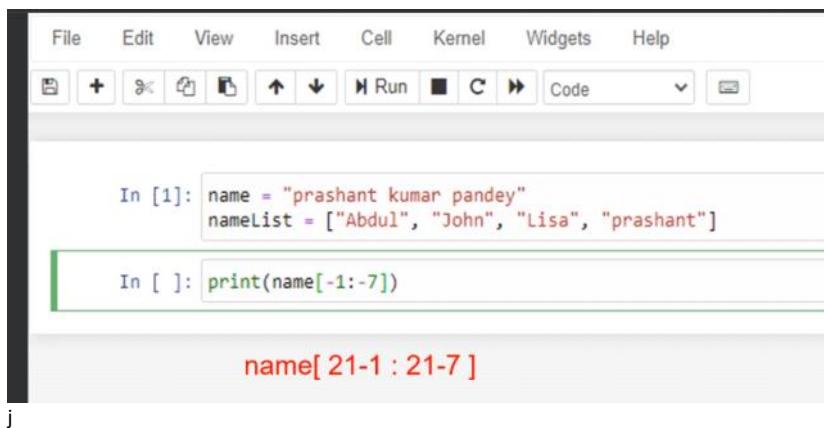


The Python logo consists of two interlocking snakes, one blue and one yellow, forming a stylized 'P' shape. Below the logo, the word "Python" is written in a bold, sans-serif font.

`s[start_index : stop_index : step_size]`

- `start_index < stop_index`

We can check the equivalent position by subtracting the index position from the length -



A screenshot of a Jupyter Notebook interface. The menu bar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help. Below the menu is a toolbar with various icons for file operations. The code cell contains:

```
In [1]: name = "prashant kumar pandey"
nameList = ["Abdul", "John", "Lisa", "prashant"]
```

The next cell is active and contains:

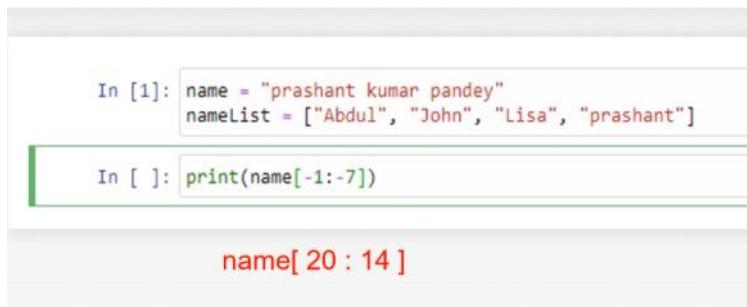
```
In [ ]: print(name[-1:-7])
```

The output cell shows the result:

```
name[ 21-1 : 21-7 ]
```

So this is not possible -

Output isn't printed. You will get null in the output. No exception



A screenshot of a Jupyter Notebook interface. The code cell contains:

```
In [1]: name = "prashant kumar pandey"
nameList = ["Abdul", "John", "Lisa", "prashant"]
```

The next cell is active and contains:

```
In [ ]: print(name[-1:-7])
```

The output cell shows the result:

```
name[ 20 : 14 ]
```

When the step size is not passed, python assumes default as 1.

Python Sequence Types – Slicing Sequences



`s[start_index : stop_index : step_size]`

- `start_index < stop_index`
- • `start <= step < stop`

IMP Slicing doesn't raise exception -

```
File Edit View Insert Cell Kernel Widgets Help
[+] Run C [ ] Code [ ]
```

In [1]: `name = "prashant kumar pandey"`
`nameList = ["Abdul", "John", "Lisa", "prashant"]`

In [15]: `print(name[0:21:50])`

p

Equivalent -

```
File Edit View Insert Cell Kernel Widgets Help
[+] Run C [ ] Code [ ]
```

In [1]: `name = "prashant kumar pandey"`
`nameList = ["Abdul", "John", "Lisa", "prashant"]`

In [15]: `print(name[0:21:50])`

p

In [16]: `print(name[0:21:21])`

p

r

In [17]: `print(name[0:21:3])`

psnkadp

In [25]: `name = "prashant kumar pandey"`
`print(name[-1:-7:-1])`

yednap

In [26]: `print(name[20:14:-1])`

yednap

Step being negative has changed the notion of the rule start \leq step $<$ stop. Because traversal happens in reverse.

So far we are able to pass the index and do the extraction. What if we want to search the index and then perform the extraction. So indexing helps.

This looks for the first occurrence of x in s. It begins the search at start_index and stops at stop_index. Stop_index is again non inclusive. Start and stop are optional.

Python Sequence Types – Index Function



The Python logo is displayed on the left side of the slide. To its right, a dashed box contains the function definition `s.index(x[, start_index[, stop_index]])`.

Jupyter Notebook Interface:

File Edit View Insert Cell Kernel Widgets Help

In [28]: `name = "prashant kumar pandey"`

In [29]: `print(name.index("a", 3, 10))`

5

Arrows point to the indices 3 and 10 in the code `print(name.index("a", 3, 10))`.

Working with Python String

1. String Literals
2. String Functions
3. String Formatting

```
In [1]: quote1 = 'Limit your "Always" and your "Never.''
quote2 = "Time isn't the main thing. It's the only thing."
print(quote1)
print(quote2)

Limit your "Always" and your "Never.''
Time isn't the main thing. It's the only thing.
```

All the spaces are retained when using the multi line string literal

```
In [2]: msg = """Hi!
    How are you doing today?
When can we meet?"""

print(msg)

Hi!
    How are you doing today?
When can we meet?
```

```
In [7]: 'Don\'t run'
```

```
Out[7]: "Don't run"
```

```
In [8]: 'Don't run'
```

```
File "<ipython-input-8-331173d60e9c>", line 1
  'Don't run'
  ^
SyntaxError: invalid syntax
```

Functions in string -

```
In [9]: name = "Learning Journal"
```

```
In [11]: len(name)
```

```
Out[11]: 16
```

```
In [9]: name = "Learning Journal"
```

```
In [12]: name.split()
```

```
Out[12]: ['Learning', 'Journal']
```

```
In [1]: name = "Big Data, Machine Learning, Artificial Intelligence"
name.split(',')
```

```
Out[1]: ['Big Data', 'Machine Learning', 'Artificial Intelligence']
```

Join function is opposite of split function -

```
In [2]: nameList = ["Big Data", "Machine Learning", "Artificial Intelligence"]
nameString = ','.join(nameList)
print(nameString)

Big Data,Machine Learning,Artificial Intelligence
```

Replace function -

```
In [3]: msg = 'Hey there! I am going to the park.'
print(msg.replace('park', 'zoo'))
print(msg)
```

```
Hey there! I am going to the zoo.
Hey there! I am going to the park.
```

```
In [7]: name = " Learning Journal "
```

```
In [8]: name.strip()
```

```
Out[8]: 'Learning Journal'
```

Lstrip and rstrip

```
In [7]: name = " Learning Journal "
In [8]: name.strip()
Out[8]: 'Learning Journal'
In [9]: name.rstrip()
Out[9]: ' Learning Journal'
In [10]: name.lstrip()
Out[10]: 'Learning Journal '
In [ ]:
```

```
In [14]: name = "John"
age = "25"
```

```
In [15]: info = "Hey! It's " + name + ", and I am " + age + " years old."
print(info)
```

```
Hey! It's John, and I am 25 years old.
```

{ } and format in combination can perform the concatenation cleanly -

```
In [14]: name = "John"  
age = "25"  
  
In [17]: info = "Hey! It's {}, and I am {} years old.".format(name, age)  
print(info)  
  
Hey! It's John, and I am 25 years old.
```

String formatting -

```
In [14]: name = "John"  
age = "25"  
  
In [20]: info = f"Hey! It's {name}, and I am {age} years old."  
print(info)  
  
Hey! It's John, and I am 25 years old.
```

Working with Python List and Tuple



Tuple – (0, 1, 2, 3)

List – [0, 1, 2, 3]

Tuple – (0, 1, 2, 3) : Immutable

List – [0, 1, 2, 3] : Mutable

```
In [1]: myTuple = ('Big Data', 'Machine Learning')  
myList = ['Artificial Intelligence', 'Data Analytics']  
print(myTuple)  
print(myList)
```

```
('Big Data', 'Machine Learning')  
['Artificial Intelligence', 'Data Analytics']
```

```
In [2]: myList2 = ['Artificial Intelligence', 'Data Analytics', 2, 3.6]  
print(myList2)
```

```
['Artificial Intelligence', 'Data Analytics', 2, 3.6]
```

We can use append function to add elements to the list.

```
In [3]: myTuple = ('Big Data', 'Machine Learning')

In [4]: myTuple.append('Data Science')

-----
AttributeError                                     Traceback (most recent call last)
<ipython-input-4-c62fb5822749> in <module>
----> 1 myTuple.append('Data Science')

AttributeError: 'tuple' object has no attribute 'append'
```

We can use list of different functions that can be applied to variety of datatypes.

The screenshot shows the Python Built-in Functions documentation page. The left sidebar includes links for Home Page, Select or create a notebook, mynotebook - Jupyter Notebook, Built-in Functions — Python 3.8.5 documentation, Previous topic (Introduction), Next topic (Built-in Constants), This Page, Report a Bug, and Show Source. The main content area is titled "Built-in Functions" and contains a note: "The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order." Below this is a large table titled "Built-In Functions" with two columns of function names:

	Built-In Functions
abs()	delattr()
all()	dict()
any()	dir()
ascii()	divmod()
bin()	enumerate()
bool()	eval()
breakpoint()	exec()
bytearray()	filter()
bytes()	float()
callable()	format()
chr()	frozenset()
classmethod()	getattr()
compile()	globals()
complex()	hasattr()
	hash()
	help()
	hex()
	id()
	input()
	int()
	isinstance()
	iter()
	len()
	list()
	locals()
	map()
	max()
	memoryview()
	min()
	next()
	object()
	oct()
	open()
	ord()
	pow()
	print()
	property()
	range()
	repr()
	reversed()
	round()
	set()
	setattr()
	slice()
	sorted()
	staticmethod()
	str()
	sum()
	super()
	tuple()
	type()
	vars()
	zip()
	__import__()

```
In [1]: myTuple = ('Big Data', 'Machine Learning')
myList = ['Artificial Intelligence', 'Data Analytics', 'Data Science']
print(len(myTuple))
print(len(myList))

2
3
```

```
In [1]: list1 = [23, 48, 3, 17]
tuple1 = (25, 52, 7, 12)
print(max(list1))
print(min(list1))
print(max(tuple1))
print(min(tuple1))

48
3
52
7
```

```
In [1]: list1 = [23, 48, 3, 17]

In [2]: list1.pop()
Out[2]: 17

In [3]: list1
Out[3]: [23, 48, 3]

In [4]: list1.append(8)
print(list1)

[23, 48, 3, 8]
```

```
In [5]: list1.insert(0, 11)
list1
```

```
Out[5]: [11, 23, 48, 3, 8]
```

Python Loops, Iterators and Range

- 1. While Loops
- 2. For Loops

- 1. While Loop
- 2. For Loop
- 3. List Comprehension

Includes break and continue -

```
In [1]: a = 0

while a < 5:
    print(a)
    a += 1
```

0
1
2
3
4

```
In [2]: name = "Learning Journal"
i = 0

while True:
    if name[i] == 'J':
        break
    print(name[i])
    i += 1

print("Outside While Loop.")

L
e
a
r
n
i
n
g
```

Outside While Loop.

```
In [3]: num = 5  
        while num > 0:  
            num -= 1  
            if num == 2:  
                continue  
            print(num)  
  
        print("Outside While Loop.")
```

```
4  
3  
1  
0  
Outside While Loop.
```

```
In [4]: grocery_list = ['eggs', 'milk', 'bread', 'veggies']  
  
for food in grocery_list:  
    print(food)
```

```
eggs  
milk  
bread  
veggies
```

```
In [7]: myRange = range(10)
```

```
In [8]: print(myRange)
```

```
range(0, 10)
```

Start is always inclusive and endpoint is exclusive.

```
In [9]: for i in myRange:  
        print(i)
```

```
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

```
In [10]: for i in range(5, 10):  
        print(i)
```

```
5  
6  
7  
8  
9
```

You can use step in range and slice.

You can use step size too -

```
In [11]: for i in range(5, 15, 2):  
        print(i)
```

```
5  
7  
9  
11  
13
```

Can also use negative step size too -

```
In [12]: for i in range(15, 5, -2):
    print(i)
```

```
15
13
11
9
7
```

List comprehension -

We have to create a new list by transforming existing list.

```
In [14]: rectangles = [(l, b)
```

```
In [14]: rectangles = [(15, 10), (3, 5), (5, 6)]
```

```
In [17]: areas = [150, 15, 30]
```

```
In [18]: rectangle_areas = [(l, b, l*b)
```

```
In [18]: rectangle_areas = [(15, 10, 150), (3, 5, 15), (5, 6, 30)]
```

Traditional approach using for loop -

```
In [19]: rectangles = [(15,10), (3,5), (5,6)]
```

```
In [20]: areas = []
```

```
for (x, y) in rectangles:
    areas.append(x*y)
```

```
print(areas)
```

```
[150, 15, 30]
```

```
In [26]: rectangles = [(15,10), (3,5), (5,6)]
```

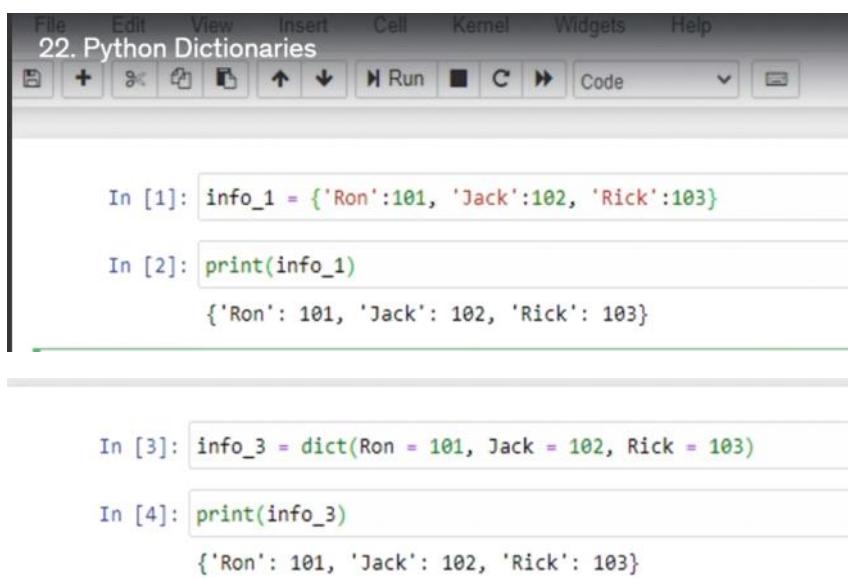
```
In [27]: rectangle_areas = [ (x, y, x*y) for (x, y) in rectangles]
```

```
In [28]: print(rectangle_areas)
```

```
[(15, 10, 150), (3, 5, 15), (5, 6, 30)]
```

Python Dictionaries

{Key : Value}



The screenshot shows a Jupyter Notebook interface with the title "22. Python Dictionaries". The toolbar includes File, Edit, View, Insert, Cell, Kernel, Widgets, and Help, along with various icons for file operations and cell execution.

```
In [1]: info_1 = {'Ron':101, 'Jack':102, 'Rick':103}
In [2]: print(info_1)
{'Ron': 101, 'Jack': 102, 'Rick': 103}

In [3]: info_3 = dict(Ron = 101, Jack = 102, Rick = 103)
In [4]: print(info_3)
{'Ron': 101, 'Jack': 102, 'Rick': 103}
```

Usually dict function is used to convert the list or tuple with key, value pair to dictionary.

```
[41]: mydict = dict(abc = 101, ade = 102, alla=149)
print(mydict)
{'abc': 101, 'ade': 102, 'alla': 149}
```

```
In [7]: my_tuple = (('Tom', 104), ('John', 105))
In [8]: info_5 = dict(my_tuple)
print(info_5)
{'Tom': 104, 'John': 105}
```

We can notice that the value associated with the Ron got updated. As well we can add the item to the dictionary too.

```
In [10]: info_1 = {'Ron':101, 'Jack':102, 'Rick':103}
In [11]: info_1['Jack']
Out[11]: 102
In [12]: info_1['Ron'] = 108
print(info_1)
{'Ron': 108, 'Jack': 102, 'Rick': 103}
In [13]: info_1['Siri'] = 110
print(info_1)
{'Ron': 108, 'Jack': 102, 'Rick': 103, 'Siri': 110}
```

Keys, values, items methods can be used to access the keys, values and the key.value pairs i.e. items from the dictionary.

```
In [15]: info_1.keys()
Out[15]: dict_keys(['Ron', 'Jack', 'Rick', 'Siri'])
In [16]: info_1.values()
Out[16]: dict_values([108, 102, 103, 110])
In [17]: info_1.items()
Out[17]: dict_items([('Ron', 108), ('Jack', 102), ('Rick', 103), ('Siri', 110)])
```

```
In [8]: for i, j in info_1.items():
    print(i, j)
```

```
Ron 108
Jack 102
Rick 103
Siri 110
```

We can combine two dictionaries using the double asterisk on each of the dictionary.

```
In [9]: info_1 = {'Ron': 101, 'John': 102, 'Rick': 103}
info_2 = {'Tom': 104, 'Harry': 105, 'Rick': 105}
In [10]: info_1_2 = {**info_1, **info_2}
print(info_1_2)
{'Ron': 101, 'John': 102, 'Rick': 105, 'Tom': 104, 'Harry': 105}
```

We can use update function to update the current dictionary with latest from dictionary 2.

```
In [9]: info_1 = {'Ron': 101, 'John': 102, 'Rick': 103}
info_2 = {'Tom': 104, 'Harry': 105, 'Rick': 105}

In [12]: info_1.update(info_2)

In [13]: print(info_1)
{'Ron': 101, 'John': 102, 'Rick': 105, 'Tom': 104, 'Harry': 105}
```

We can delete certain elements in the dictionary by using del function.

```
In [14]: info_1 = {'Ron': 101, 'John': 102, 'Rick': 103}

In [16]: len(info_1)
Out[16]: 3

In [17]: del info_1['Ron']

In [18]: info_1
Out[18]: {'John': 102, 'Rick': 103}
```

You can delete the dictionary -

```
In [21]: info_1 = {'Ron': 101, 'John': 102, 'Rick': 103}

In [22]: del info_1

In [23]: info_1
-----
NameError                                 Traceback (most recent call last)
<ipython-input-23-83eaab0eeb3b> in <module>
      1 info_1
      2
      3     NameError: name 'info_1' is not defined
```

If you don't want to delete the dictionary where as want to clear all the elements in the dictionary, you can use dict.clear() function

```
In [24]: info_1 = {'Ron': 101, 'John': 102, 'Rick': 103}

In [25]: info_1.clear()

In [26]: info_1
Out[26]: {}
```

In operator checks keys in dictionary -

```
In [30]: info_1 = {'Ron': 101, 'John': 102, 'Rick': 103}
```

```
In [31]: 'John' in info_1
```

```
Out[31]: True
```

```
In [32]: 'Harry' not in info_1
```

```
Out[32]: True
```

```
In [33]: info_1 = {'Ron': 101, 'John': 102, 'Rick': 103}  
info_2 = {'Ron': 101, 'John': 102, 'Rick': 103}
```

```
In [34]: info_1 == info_2
```

```
Out[34]: True
```

```
In [35]: info_1 != info_2
```

```
Out[35]: False
```

Shallow copy of the dictionary to another -

```
In [37]: info_1 = {'Ron': 101, 'John': 102, 'Rick': 103}
```

```
In [38]: info = info_1.copy()
```

```
print(info)
```

```
{'Ron': 101, 'John': 102, 'Rick': 103}
```

```
In [40]: msg = {'A':'Sleep', 'B':'Eat', 'X':['Play', 'Study']}
```

```
In [42]: inform = msg.copy()
```

```
print(inform)  
print(msg)
```

```
{'A': 'Sleep', 'B': 'Eat', 'X': ['Play', 'Study']}  
{'A': 'Sleep', 'B': 'Eat', 'X': ['Play', 'Study']}
```

```
In [43]: inform ['X'][1] = 'Cook'
```

```
print(inform)
```

```
{'A': 'Sleep', 'B': 'Eat', 'X': ['Play', 'Cook']}
```

```
In [44]: print(msg)
```

```
{'A': 'Sleep', 'B': 'Eat', 'X': ['Play', 'Cook']}
```

If we want to utilize the functionality of deep copy, we can do so by using the copy module. This will not affect changes made to cloned dictionary

```
In [48]: import copy  
  
inform = copy.deepcopy(msg)  
  
print(inform)  
print(msg)  
  
{'A': 'Sleep', 'B': 'Eat', 'X': ['Play', 'Study']}  
{'A': 'Sleep', 'B': 'Eat', 'X': ['Play', 'Study']}
```

```
In [49]: inform['X'][1] = 'Cook'  
  
print(inform)  
  
{'A': 'Sleep', 'B': 'Eat', 'X': ['Play', 'Cook']}
```

```
In [50]: print(msg)  
  
{'A': 'Sleep', 'B': 'Eat', 'X': ['Play', 'Study']}
```

Dictionary comprehension -

We know that the comprehension is used to transform an iterable object to another iterable object.

```
In [51]: square = {x : x**2 for x in range(0, 5)}  
  
print(square)  
  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Python sets -

Set is an unordered collection of elements.

You can think of set as dictionary with only keys and no values.

```
In [1]: mySet = {0, 1, 2, 3}  
  
In [2]: print(mySet)  
  
{0, 1, 2, 3}
```

Primary objective of set is to eliminate duplicate elements or test for the membership.

```
In [1]: mySet = {0, 1, 2, 3}  
  
In [2]: print(mySet)  
  
{0, 1, 2, 3}  
  
In [3]: mySet = {0, 0, 1, 2, 2, 2}  
  
print(mySet)  
  
{0, 1, 2}
```

```
In [4]: myList = ['Ram', 'Mohan', 'Abdul', 'John', 'Ram']  
  
print(myList)  
  
['Ram', 'Mohan', 'Abdul', 'John', 'Ram']
```

You can convert the list to set using set function.

Side note - Single asterisk can be used to unpack the list, tuple.

```
In [4]: myList = ['Ram', 'Mohan', 'Abdul', 'John', 'Ram']
print(myList)
['Ram', 'Mohan', 'Abdul', 'John', 'Ram']
```

```
In [5]: mySet = set(myList)
print(mySet)
{'Ram', 'Mohan', 'Abdul', 'John'}
```

We have created a non empty set of set. How to create an empty set of set. It is not curly braces. Because we create dictionary using empty curly braces.

```
In [5]: mySet = {0, 0, 1, 2, 2}
```

```
In [6]: myEmptySet = {}
```

```
In [7]: type(myEmptySet)
```

```
Out[7]: dict
```

```
In [8]: myEmptySet = set()
```

```
In [9]: type(myEmptySet)
```

```
Out[9]: set
```

Major use of using set is to eliminate duplicates. We were just able to create a set by having both upper and lower case letters. But if we want to eliminate duplicates, we will have to either upper / lower case the letters first before applying the set on it.

```
In [10]: set('Learning Journal')
```

```
Out[10]: {' ', 'J', 'L', 'a', 'e', 'g', 'i', 'l', 'n', 'o', 'r', 'u'}
```

```
In [10]: set('Learning Journal')
```

```
Out[10]: {' ', 'J', 'L', 'a', 'e', 'g', 'i', 'l', 'n', 'o', 'r', 'u'}
```

```
In [11]: set('Learning Journal'.lower())
```

```
Out[11]: {' ', 'a', 'e', 'g', 'i', 'j', 'l', 'n', 'o', 'r', 'u'}
```

Converting different data structures to set -

Notice the dictionary conversion just printing its values.

```
In [12]: set(['Literature', 'Science', 'Maths'])  
Out[12]: {'Literature', 'Maths', 'Science'}  
  
In [13]: set(('Eat', 'Sleep', 'Repeat'))  
Out[13]: {'Eat', 'Repeat', 'Sleep'}  
  
In [14]: set({1:'East', 2:'South', 3:'North', 4:'West'})  
Out[14]: {1, 2, 3, 4}
```

Set has add function -

However, it can just add one element to the add function

```
In [15]: mySet = {0, 1, 2, 3, 4, 5}  
  
In [17]: mySet.add(6)  
  
In [18]: mySet  
Out[18]: {0, 1, 2, 3, 4, 5, 6}
```

If we want to add list to the set, then we can use the update method -

```
In [15]: mySet = {0, 1, 2, 3, 4, 5}  
  
In [20]: mySet.update([7, 8, 9])  
  
In [21]: mySet  
Out[21]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

You can also remove element from the set -

```
In [22]: mySet  
Out[22]: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}  
  
In [23]: mySet.remove(4)  
  
In [24]: mySet  
Out[24]: {0, 1, 2, 3, 5, 6, 7, 8, 9}
```

You can also perform union, intersection and

```
In [25]: set1 = {0, 1, 2}
set2 = {2, 3, 4, 5}
```

```
In [26]: set1.union(set2)
```

```
Out[26]: {0, 1, 2, 3, 4, 5}
```

```
In [27]: set1 | set2
```

```
Out[27]: {0, 1, 2, 3, 4, 5}
```

Intersection can be performed using below. Also can use ampersand -

```
In [25]: set1 = {0, 1, 2}
set2 = {2, 3, 4, 5}
```

```
In [28]: set1.intersection(set2)
```

```
Out[28]: {2}
```

```
In [29]: set1 & set2
```

```
Out[29]: {2}
```

Also perform difference -

```
In [25]: set1 = {0, 1, 2}
set2 = {2, 3, 4, 5}
```

```
In [32]: set1.difference(set2)
```

```
Out[32]: {0, 1}
```

```
In [33]: set2.difference(set1)
```

```
Out[33]: {3, 4, 5}
```

```
In [34]: set1 - set2
```

```
Out[34]: {0, 1}
```

```
In [35]: set2 - set1
```

```
Out[35]: {3, 4, 5}
```

You can also use symmetric difference. Present in set1, set2 but not in both -

Anything other than overlap

```
In [25]: set1 = {0, 1, 2}
set2 = {2, 3, 4, 5}
```

```
In [36]: set1.symmetric_difference(set2)
```

```
Out[36]: {0, 1, 3, 4, 5}
```

```
In [37]: set1 ^ set2
```

```
Out[37]: {0, 1, 3, 4, 5}
```

```
In [38]: mySet  
Out[38]: {0, 1, 2, 3, 5, 6, 7, 8, 9}
```

```
In [39]: 5 in mySet  
Out[39]: True
```

```
In [40]: 4 in mySet  
Out[40]: False
```

```
In [41]: 101 not in mySet  
Out[41]: True
```

```
In [42]: set1 = {0, 1}  
set2 = {0, 1, 2, 3}
```

```
In [43]: set1.issubset(set2)  
Out[43]: True
```

```
In [44]: set1 <= set2  
Out[44]: True
```

```
In [42]: set1 = {0, 1}  
set2 = {0, 1, 2, 3}
```

```
In [45]: set2.issuperset(set1)  
Out[45]: True
```

```
In [46]: set2 >= set1  
Out[46]: True
```

Think of set as mathematical set to perform it's operations -

```
In [3]: mySet = {1, 3, 5, 8}  
In [4]: mySet{2}  
File "<ipython-input-4-bca429599148>", line 1  
    mySet{2}  
          ^  
SyntaxError: invalid syntax
```

If you want to access elements in the set like in case of tuple or list like below -

```
In [3]: mySet = {1, 3, 5, 8}

In [4]: mySet[2]
        File "<ipython-input-4-bca429599148>", line 1
                mySet[2]
                  ^
SyntaxError: invalid syntax
```

*** So extraction and slicing can't be performed on the set as it is an unordered structure.

You can access using iterator though. Notice unordered retrieval of elements -

```
In [1]: mySet = {1,3,5,8}

In [2]: for i in mySet:
        print(i)

8
1
3
5
```

To create a frozen set. i.e. immutable set.

```
In [3]: myFrozenSet = frozenset((0, 2, 4))

In [4]: print(myFrozenSet)
frozenset({0, 2, 4})

In [5]: myFrozenSet.add(6)
        AttributeError                                Traceback (most recent call last)
<ipython-input-5-6d977a8eb510> in <module>
----> 1 myFrozenSet.add(6)

AttributeError: 'frozenset' object has no attribute 'add'
```

Python doesn't have a standard module for date and time. We must import the datetime module from python library.

Datetime module offers primarily the below classes. All the objects of this class are immutable.
However you can manipulate them and create a new object

Python Datetime Module

1. date
2. time
3. datetime
4. timedelta

The screenshot shows a Jupyter Notebook interface with three cells. Cell 1 imports the datetime module. Cell 2 creates a date object for August 21, 1990, and prints it. Cell 3 attempts to create a date object for April 31, 1990, which results in a ValueError because April only has 30 days.

```
In [1]: from datetime import date
In [2]: dob = date(1990, 8, 21)
print(dob)
1990-08-21
```

```
In [1]: from datetime import date
In [4]: date(1990, 3, 31)
Out[4]: datetime.date(1990, 3, 31)

In [5]: date(1990, 4, 31)
-----
ValueError                                     Traceback (most recent call last)
<ipython-input-5-9dc9cce8a5d> in <module>
----> 1 date(1990, 4, 31)

ValueError: day is out of range for month
```

```
In [1]: from datetime import date
In [6]: date.today()
Out[6]: datetime.date(2020, 10, 7)

In [1]: from datetime import date
In [7]: dob = date(1990, 8, 21)
In [8]: print(dob.day)
21
In [9]: print(dob.month)
8
In [10]: print(dob.year)
1990
```

Explore these methods -

- `weekday()`
- `replace()`
- `strftime()`

`d.ctime()` is equivalent to:

```
time.ctime(time.mktime(d.timetuple()))
```

on platforms where the native C `ctime()` function (which `time.ctime()` invokes, but which `date.ctime()` does not invoke) conforms to the C standard.

`date.strftime(format)`

Return a string representing the date, controlled by an explicit format string. Format codes referring to hours, minutes or seconds will see 0 values. For a complete list of formatting directives, see `strftime()` and `strptime()` Behavior.

`date.__format__(format)`

Same as `date.strftime()`. This makes it possible to specify a format string for a `date` object in formatted string literals and when using `str.format()`. For a complete list of formatting directives, see `strftime()` and `strptime()` Behavior.

Examples of Usage: date

Example of counting days to an event:

```
>>> import time
>>> from datetime import date
>>> today = date.today()
>>> today
datetime.date(2007, 12, 5)
>>> today == date.fromtimestamp(time.time())
True
>>> my_birthday = date(today.year, 6, 24)
```

```
In [1]: from datetime import date
```

```
In [12]: dob = date(1990, 8, 21)
```

```
In [13]: dob.strftime("%A %d - %b %Y")
```

```
Out[13]: 'Tuesday 21 - Aug 1990'
```

strftime() and strptime() Format Codes

The following is a list of all the format codes that the 1989 C standard requires, and these work on all platforms with a standard C implementation.

Directive	Meaning	Example	Notes
%a	Weekday as locale's abbreviated name.	Sun, Mon, ..., Sat (en_US); So, Mo, ..., Sa (de_DE)	(1)
%A	Weekday as locale's full name.	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)	(1)
%w	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday	0, 1, ..., 6	
%d	Day of the month as a zero-padded decimal number.	01, 02, ..., 31	(9)
%b	Month as locale's abbreviated name	Jan, Feb, ..., Dec (en_US); Jan, Feb, ..., Dez (de_DE)	(1)
%B	Month as locale's full name.	January, February, ..., December (en_US); Januar, Februar, ..., Dezember (de_DE)	(1)

```
In [16]: from datetime import time
```

```
In [17]: meet = time(11, 33, 48, 900000)
```

```
In [18]: print(meet)
```

```
11:33:48.900000
```

It does validation of the values supplied -

```
In [16]: from datetime import time  
  
In [19]: meet = time(24, 33, 48, 900000)  
  
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-19-cd970a5f1059> in <module>  
----> 1 meet = time(24, 33, 48, 900000)  
  
ValueError: hour must be in 0..23
```

```
In [16]: from datetime import time
```

```
In [21]: print(meet)
```

```
23:33:48.900000
```

```
In [25]: print(meet.hour)
```

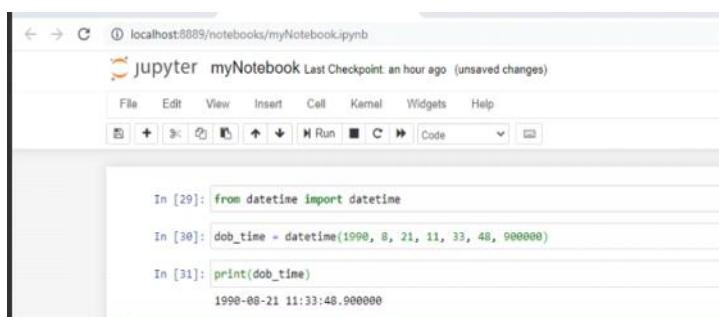
```
23
```

```
In [26]: print(meet.minute)
```

```
33
```

```
In [27]: print(meet.second)
```

```
48
```



The screenshot shows a Jupyter Notebook interface with the following details:

- URL: localhost:8889/notebooks/myNotebook.ipynb
- Kernel: jupyter myNotebook (Last Checkpoint: an hour ago (unsaved changes))
- Toolbar buttons: File, Edit, View, Insert, Cell, Kernel, Widgets, Help.
- Code cells:
 - In [29]: from datetime import datetime
 - In [30]: dob_time = datetime(1990, 8, 21, 11, 33, 48, 900000)
 - In [31]: print(dob_time)
- Output:

```
1990-08-21 11:33:48.900000
```

Python Datetime Module – datetime function



1. year
2. month
3. date
4. hour
5. minute
6. second
7. microsecond

```
In [29]: from datetime import datetime
```

```
In [31]: print(dob_time)
```

```
1990-08-21 11:33:48.900000
```

```
In [35]: print(dob_time.month)
```

```
8
```

```
In [36]: print(dob_time.day)
```

```
21
```

```
In [37]: print(dob_time.hour)
```

```
11
```

```
In [39]: from datetime import timedelta
```

```
In [40]: duration = timedelta(days = 2, hours = 3, minutes = 45)
```

```
In [41]: print(duration)
```

```
2 days, 3:45:00
```

Time addition doesn't make sense -

```
In [45]: from datetime import datetime
```

```
In [46]: my_dob = datetime(1991, 5, 17)
your_dob = datetime(1990, 8, 21)
```

```
In [47]: my_dob + your_dob
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-47-6b4d4a9273a4> in <module>  
----> 1 my_dob + your_dob
```

```
TypeError: unsupported operand type(s) for +: 'datetime.datetime' and 'datetime.datetime'
```

Date subtraction can show age differences -

```
In [45]: from datetime import datetime
```

```
In [46]: my_dob = datetime(1991, 5, 17)
your_dob = datetime(1990, 8, 21)
```

```
In [48]: my_dob - your_dob
```

```
Out[48]: datetime.timedelta(days=269)
```

If you still want to add the date, then we can use duration and add it to date.

```
In [45]: from datetime import datetime  
  
In [51]: from datetime import timedelta  
  
In [52]: my_dob = datetime(1991, 5, 17)  
         your_dob = datetime(1990, 8, 21)  
  
In [53]: duration = timedelta(days = 2)  
  
In [54]: my_dob + duration  
  
Out[54]: datetime.datetime(1991, 5, 19, 0, 0)
```

```
In [45]: from datetime import datetime  
  
In [51]: from datetime import timedelta  
  
In [52]: my_dob = datetime(1991, 5, 17)  
         your_dob = datetime(1990, 8, 21)  
  
In [61]: duration = timedelta(hours = 1, seconds = 15)  
  
In [62]: my_dob - duration  
  
Out[62]: datetime.datetime(1991, 5, 16, 22, 59, 45)
```

datetime + datetime = Not allowed
datetime - datetime = datetime
datetime + timedelta = datetime
datetime - timedelta = datetime

```
In [68]: from datetime import time  
  
In [69]: my_time = time(8, 15, 10)  
         your_time = time(11, 33, 48)  
  
In [70]: my_time - your_time  
  
-----  
TypeError                                         Traceback (most recent call last)  
<ipython-input-70-291014de0b9f> in <module>  
----> 1 my_time - your_time  
  
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.time'
```

```
In [1]: from datetime import time, timedelta  
  
In [8]: my_time = time(10, 0)  
        duration = timedelta(hours = 1)  
  
In [10]: my_time - duration  
  
-----  
TypeError                                         Traceback (most recent call last)  
<ipython-input-10-1eb5252eb978> in <module>  
----> 1 my_time - duration  
  
TypeError: unsupported operand type(s) for -: 'datetime.time' and 'datetime.timedelta'
```

This is not allowed because, time is always associated with date. If there isn't date associated with time, there isn't reason for adding duration

time + timedelta = Not allowed
time - timedelta = Not allowed
time + time = Not allowed
time - time = Not allowed

Date BUT NO Time → date()

Date AND Time → datetime()

Functions -

Functions are either built in or user defined.

Some of the built in are as below -

```
In [1]: tuple_1 = (5, 10, 15, 20)
In [2]: min(tuple_1)
Out[2]: 5
In [3]: max(tuple_1)
Out[3]: 20
In [4]: len(tuple_1)
Out[4]: 4
In [5]: print(tuple_1)
(5, 10, 15, 20)
```

The screenshot shows a web browser displaying the Python documentation at docs.python.org/3/library/functions.html. The page title is "Built-in Functions". On the left, there's a sidebar with navigation links: "Previous topic", "Introduction", "Next topic", "Built-in Constants", "This Page", "Report a Bug", and "Show Source". The main content area contains a brief introduction followed by a large table titled "Built-in Functions". The table has five columns and lists various built-in functions. Below the table, there's a code block showing the syntax for defining a function:

Built-in Functions				
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
complex()	hasattr()	max()	round()	

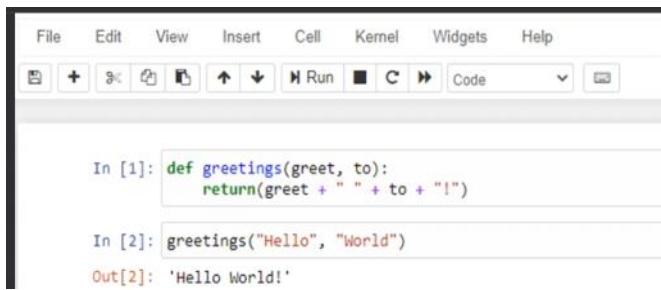
```
def function_name (parameter_1, parameter_2, ..):
    statements
    ....
```

```
In [1]: def even_or_odd(a):
    if a % 2 == 0:
        print('Even')
    else:
        print('Odd')
```

```
In [2]: even_or_odd(5)
```

```
Odd
```

```
In [3]: def even_or_odd(a):
    if a % 2 == 0:
        return('Even')
    else:
        return('Odd')
```



```
In [1]: def greetings(greet, to):
    return(greet + " " + to + "!")

In [2]: greetings("Hello", "World")
Out[2]: 'Hello World!'
```

If you send more than the given number of parameters to the function or any less it will give an error missing 1 required positional argument -

```
In [1]: def greetings(greet, to):
    return(greet + " " + to + "!")

In [2]: greetings("Hello", "World")
Out[2]: 'Hello World!'

In [3]: greetings("Hello")
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-3-90cba574d163> in <module>
      1 greetings("Hello")
----> 1 greetings("Hello")

TypeError: greetings() missing 1 required positional argument: 'to'
```

We should have a feasibility to allow python to perform this -

```
In [4]: def greetings(greet, to = "World"):
    return(greet + " " + to + "!")

In [5]: greetings("Hello", "Friends")
Out[5]: 'Hello Friends!'

In [6]: greetings("Hello")
Out[6]: 'Hello World!'
```

```
File Edit View Insert Cell Kernel Widgets Help  
In [4]: def greetings(greet, to = "World"):  
         return(greet + " " + to + "!")  
  
In [5]: greetings("Hello", "Friends")  
Out[5]: 'Hello Friends!'  
  
In [6]: greetings("Hello")  
Out[6]: 'Hello World!'  
  
In [ ]:
```

1. Mandatory Parameters 2. Optional Parameters

Parameters are either mandatory or optional. If you want to make it optional, you need to provide a default value to it.

```
In [7]: def greetings(greet = "Hello", to = "World"):  
         return(greet + " " + to + "!")  
  
In [8]: greetings()  
Out[8]: 'Hello World!'  
  
In [9]: greetings("Hey")  
Out[9]: 'Hey World!'  
  
In [10]: greetings("Hi", "Friends")  
Out[10]: 'Hi Friends!'
```

You cannot just pass the second argument alone thinking first argument already defaults. To fix this problem we have keyword argument passing.

```
File Edit View Insert Cell Kernel Widgets Help  
In [7]: def greetings(greet = "Hello", to = "World"):  
         return(greet + " " + to + "!")  
  
In [11]: greetings("Hello", "Prashant")  
Out[11]: 'Hello Prashant!'  
  
In [12]: greetings("Prashant")  
Out[12]: 'Prashant World!'
```

1. greetings()
2. greetings("Hey")
3. greetings("Hello", "Prashant")

Keyword argument gives flexibility to change the arguments order

```
File Edit View Insert Cell Kernel Widgets Help  
In [13]: def greetings(greet="Hello", to="World"):  
         return(greet + " " + to + "!")  
  
In [14]: greetings(to = 'Prashant')  
Out[14]: 'Hello Prashant!'
```

```
In [13]: def greetings(greet="Hello", to="World"):
    return(greet + " " + to + "!")-->

In [15]: greetings(to = 'Prashant', greet = 'Hi')
Out[15]: 'Hi Prashant!'

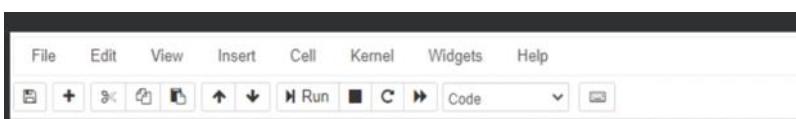
In [ ]:
```

1. Mandatory Parameter
2. Optional Parameter
3. Keyword Parameter

You can't have the mandatory arguments followed by optional arguments.

```
In [37]: def configure(os = "Windows", cpu, ram, disk):
    return "Operating System = " + os \
           + ", Processor = " + cpu \
           + ", Memory = " + ram \
           + ", Storage = " + disk

File "<ipython-input-37-cf167e1abcf>", line 1
  def configure(os = "Windows", cpu, ram, disk):
           ^
SyntaxError: non-default argument follows default argument ←
```



```
In [39]: def configure(cpu, ram, disk = "512 GB SSD", os = "Windows"):
    return "Operating System = " + os \
           + ", Processor = " + cpu \
           + ", Memory = " + ram \
           + ", Storage = " + disk
```

While calling all the keyword arguments should be at the end

```
In [39]: def configure(cpu, ram, disk = "512 GB SSD", os = "Windows"):
    return "Operating System = " + os \
           + ", Processor = " + cpu \
           + ", Memory = " + ram \
           + ", Storage = " + disk

In [40]: configure("i5", "8GB")
Out[40]: 'Operating System = Windows, Processor = i5, Memory = 8GB, Storage = 512 GB SSD'

In [41]: configure(cpu = "i5", "8GB")
File "<ipython-input-41-b2b567a1d793>", line 1
  configure(cpu = "i5", "8GB")
           ^
SyntaxError: positional argument follows keyword argument
```

```
In [39]: def configure(cpu, ram, disk = "512 GB SSD", os = "Windows"):
    return "Operating System = " + os \
           + ", Processor = " + cpu \
           + ", Memory = " + ram \
           + ", Storage = " + disk

In [42]: configure(cpu = "i5", ram = "8GB")
Out[42]: 'Operating System = Windows, Processor = i5, Memory = 8GB, Storage = 512 GB SSD'
```

```
In [39]: def configure(cpu, ram, disk = "512 GB SSD", os = "Windows"):
    return "Operating System = " + os \
           +", Processor = " + cpu \
           +", Memory = " + ram \
           +", Storage = " + disk

In [45]: configure("i5", ram = "8GB", "256GB SSD")
          File "<ipython-input-45-512a401a480a>", line 1
              configure("i5", ram = "8GB", "256GB SSD")
                                         ^
SyntaxError: positional argument follows keyword argument
```

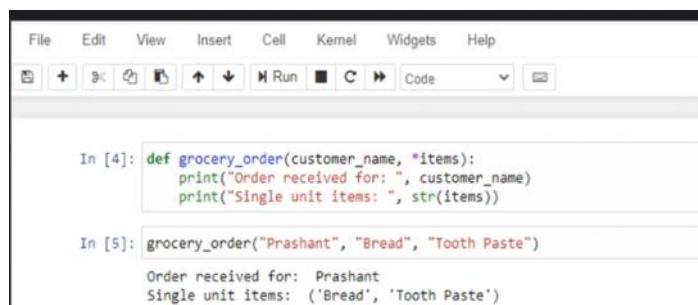
So everything after this ram, has to be a key, value pair.

```
In [1]: def grocery_order(customer_name, items):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))

In [3]: grocery_order("Prashant", ["Bread", "Tooth Paste"])

Order received for: Prashant
Single unit items: ['Bread', 'Tooth Paste']
```

Keeping star in front of a list will make it accept the rest of the arguments after the mandatory argument



```
In [4]: def grocery_order(customer_name, *items):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))

In [5]: grocery_order("Prashant", "Bread", "Tooth Paste")

Order received for: Prashant
Single unit items: ('Bread', 'Tooth Paste')
```

Default arguments can also be used -

** Always keep the default value arguments to the end.

```
In [6]: def grocery_order(customer_name, *items, delivery_type = "Home"):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))
    print("Delivery Type: ", delivery_type)

In [7]: grocery_order("Prashant", "Bread", "Tooth Paste")

Order received for: Prashant
Single unit items: ('Bread', 'Tooth Paste')
Delivery Type: Home
```

If the order changes, it will complain about the keyword argument -

```
In [9]: def grocery_order(customer_name, delivery_type = "Home", *items):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))
    print("Delivery Type: ", delivery_type)

In [11]: grocery_order("Prashant", delivery_type = "Home", "Bread", "Tooth Paste")
          File "<ipython-input-11-e269160f1f2e>", line 1
              grocery_order("Prashant", delivery_type = "Home", "Bread", "Tooth Paste")
                                         ^
SyntaxError: positional argument follows keyword argument
```

*** So always keep the default values to the end.

You want something like below which accepts the customer name, item and its quantity, those which are single items might not need number passed.

```
grocery_order("Prashant", "Bread", "Tooth Paste", "eggs" = 12, "milk" = "1 ltr")
```

*** This way of syntax is wrong. Because after prashant, it will expect the other arguments as variable length arguments. It can't understand that the dict goes to items_with_qty.

```
In [16]: def grocery_order(customer_name, *items, items_with_qty, delivery_type = "Home"):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))
    print("Other Items: ", str(items_with_qty))
    print("Delivery Type: ", delivery_type)

    if 'eggs' in items_with_qty:
        print("Note: Handle with care.")
```

```
In [19]: grocery_order("Prashant", "Bread", "Tooth Paste", dict(eggs = 12, milk = "1 ltr"))
```

```
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-19-f0700d5ddd2c> in <module>
----> 1 grocery_order("Prashant", "Bread", "Tooth Paste", dict(eggs = 12, milk = "1 ltr"))

TypeError: grocery_order() missing 1 required keyword-only argument: 'items_with_qty'
```

Using Key/Value syntax, will eliminate the problem.

```
In [16]: def grocery_order(customer_name, *items, items_with_qty, delivery_type = "Home"):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))
    print("Other Items: ", str(items_with_qty))
    print("Delivery Type: ", delivery_type)

    if 'eggs' in items_with_qty:
        print("Note: Handle with care.")
```

```
In [20]: grocery_order("Prashant", "Bread", "Tooth Paste", items_with_qty = dict(eggs = 12, milk = "1 ltr"))
```

```
Order received for: Prashant
Single unit items: ('Bread', 'Tooth Paste')
Other Items: {'eggs': 12, 'milk': '1 ltr'}
Delivery Type: Home
Note: Handle with care.
```

If there are no items passed you can mark that argument as "None"

```
In [21]: def grocery_order(customer_name, *items, items_with_qty = None, delivery_type = "Home"):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))
    print("Other Items: ", str(items_with_qty))
    print("Delivery Type: ", delivery_type)
```

```
In [22]: grocery_order("Prashant", "Bread", "Tooth Paste")
```

```
Order received for: Prashant
Single unit items: ('Bread', 'Tooth Paste')
Other Items: None
Delivery Type: Home
```

Passing the dictionary argument can be easily be resolved using this syntax -

You have to make the double asterisk as the last argument of your function. This double asterisk variable will be able to receive dictionary.

```
In [24]: def grocery_order(customer_name, *items, delivery_type = "Home", **items_with_qty):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))
    print("Other Items: ", str(items_with_qty))
    print("Delivery Type: ", delivery_type)

    if 'eggs' in items_with_qty:
        print("Note: Handle with care")
```

Now we are not passing the keyword name of the argument, instead we are passing the comma separated values of key/value pairs. Python will pack these key/value pairs into the dictionary.

```
In [24]: def grocery_order(customer_name, *items, delivery_type = "Home", **items_with_qty):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))
    print("Other Items: ", str(items_with_qty))
    print("Delivery Type: ", delivery_type)

    if 'eggs' in items_with_qty:
        print("Note: Handle with care")
```

```
In [25]: grocery_order("Prashant", "Bread", "Tooth Paste", eggs = 12, milk = "1 ltr", delivery_type = "Pickup")
```

```
Order received for: Prashant
Single unit items: ('Bread', 'Tooth Paste')
Other Items: {'eggs': 12, 'milk': '1 ltr'}
Delivery Type: Pickup
Note: Handle with care
```

- Python will accept the variable length tuple and variable length dictionary

```
: def grocery_order(customer_name, *items, delivery_type = "Home", **items_with_qty):
    print("Order received for: ", customer_name)
    print("Single unit items: ", str(items))
    print("Other Items: ", str(items_with_qty))
    print("Delivery Type: ", delivery_type)

    if 'eggs' in items_with_qty:
        print("Note: Handle with care")
```

```
: grocery_order("Prashant", "Bread", "Tooth Paste", eggs = 12, milk = "1 ltr", delivery_type = "Pickup")
```

```
Order received for: Prashant
Single unit items: ('Bread', 'Tooth Paste')
Other Items: {'eggs': 12, 'milk': '1 ltr'}
Delivery Type: Pickup
Note: Handle with care
```

1. Variable Length - Tuple 2. Variable Length - Dictionary

You can place the tuple argument anywhere in the method, where as you need the dictionary type argument to be passed as the last argument.

What if you have this function and you need to pass the array to the *items parameter. Then we can first unpack the list of items to comma separated values and then pass it as an argument.

```
[5]: def book_groceries(customer, *items, type="Delivery", **item_dict):
    print("customer", customer)
    print("items", items)
    print("type", type)
    print("item_dict", item_dict)

    item_args = ["Curd", "Milk"]

book_groceries("Vishnu", *item_args, type="Pickup", soaps=2, brushes = 5)
customer Vishnu
items ('Curd', 'Milk')
type Pickup
item_dict {'soaps': 2, 'brushes': 5}

def book_groceries(customer, *items, type="Delivery", **item_dict):
    print("customer", customer)
    print("items", items)
    print("type", type)
    print("item_dict", item_dict)

    item_args = ["Curd", "Milk"]
    dict_args = dict(soaps=2, brushes=5)

book_groceries("Vishnu", *item_args, type="Pickup", **dict_args)
customer Vishnu
items ('Curd', 'Milk')
type Pickup
item_dict {'soap': 2, 'brushes': 5}
```

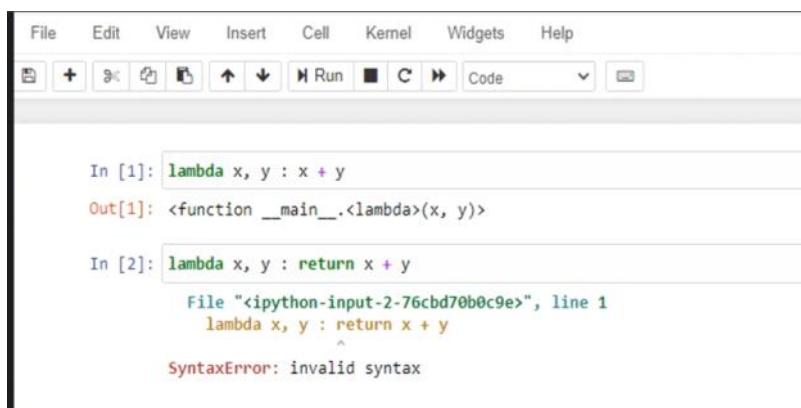
We can also pass the dictionary first followed by optional arguments too.

```
def book_groceries(customer, *items, type="Delivery", **item_dict):
    print("customer", customer)
    print("items", items)
    print("type", type)
    print("item_dict", item_dict)

    item_args = ["Curd", "Milk"]
    dict_args = dict(soaps=2, brushes=5)

book_groceries("Vishnu", *item_args, **dict_args, type="Pickup")
customer Vishnu
items ('Curd', 'Milk')
type Pickup
item_dict {'soap': 2, 'brushes': 5}
```

Lambda functions -



```
In [1]: lambda x, y : x + y
Out[1]: <function __main__.lambda>(x, y)

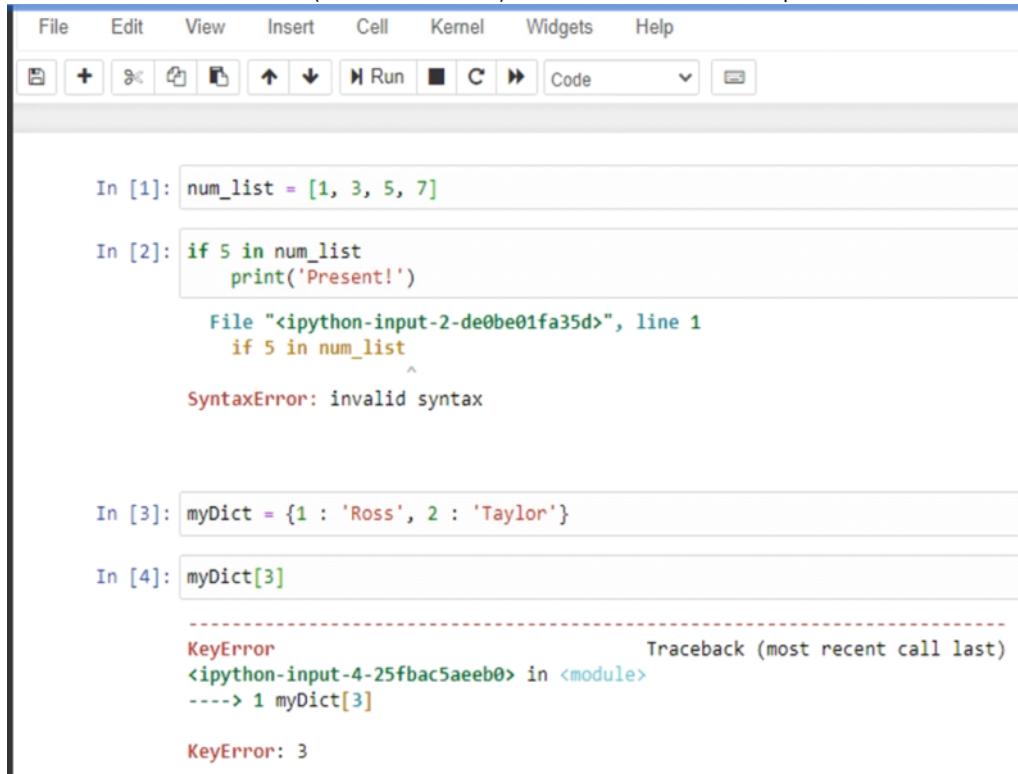
In [2]: lambda x, y : return x + y
        ^
File "<ipython-input-2-76cbd70b0c9e>", line 1
      lambda x, y : return x + y
                  ^
SyntaxError: invalid syntax
```

```
In [6]: def apply_ops(*args, ops):
         return ops(args)

In [7]: apply_ops(2, 3, 7, 9, ops = sum)
Out[7]: 21
```

Errors -

In first error there is no traceback (most recent call last). Where as in second error it is present -



```
In [1]: num_list = [1, 3, 5, 7]

In [2]: if 5 in num_list
         print('Present!')

      File "<ipython-input-2-de0be01fa35d>", line 1
            if 5 in num_list
                  ^
      SyntaxError: invalid syntax

In [3]: myDict = {1 : 'Ross', 2 : 'Taylor'}

In [4]: myDict[3]

-----
      KeyError                                                 Traceback (most recent call last)
      <ipython-input-4-25fbac5aeeb0> in <module>
      ----> 1 myDict[3]

      KeyError: 3
```

Errors are classified into two -

1. Syntax errors
2. Exceptions

```
In [8]: a = 3
        b = 2

        print(a / b))

      File "<ipython-input-8-031d60405199>", line 4
            print(a / b))
                  ^
      SyntaxError: unmatched ')'
```

Syntax errors are clearly shown as where and what we missed.

These logical errors are known as exceptions. This doesn't have syntax error on the error it showed. This exception is called zero division error

```
In [8]: a = 3
b = 0

print(a / b)

-----
ZeroDivisionError                                Traceback (most recent call last)
<ipython-input-8-a7d9afa154f8> in <module>
      2 b = 0
      3
----> 4 print(a / b)

ZeroDivisionError: division by zero
```

```
In [9]: a = 2
b = "Hi"

print(a + b)

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-9-ab830ab2ec8e> in <module>
      2 b = "Hi"
      3
----> 4 print(a + b)

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



There are two types of Exceptions -

1. Built-in Exception 2. User-defined Exception

These are predefined/ built in exceptions

Where as python also allows to create user-defined exceptions too

[36. Introduction to Errors and Exceptions](https://docs.python.org/3/library/exceptions.html#exception-hierarchy)

← → ⌂ doc.python.org/3/library/exceptions.html#exception-hierarchy

Exception hierarchy

The class hierarchy for built-in exceptions is:

```
BaseException
+-- SystemExit
+-- KeyboardInterrupt
+-- GeneratorExit
+-- Exception
    +-- StopIteration
    +-- StopAsyncIteration
    +-- ArithmeticError
        +-- FloatingPointError
        +-- OverflowError
        +-- ZeroDivisionError
    +-- AssertionError
    +-- AttributeError
    +-- BufferError
    +-- EOFError
    +-- ImportError
        +-- ModuleNotFoundError
    +-- LookupError
        +-- IndexError
        +-- KeyError
    +-- MemoryError
    +-- NameError
        +-- UnboundLocalError
    +-- OSError
        +-- BlockingIOError
        +-- ChildProcessError
```

Now this is the exception that is not something caused by me. But this is coming because of passing incorrectly passing string a instead of numbers.

```
In [3]: n = int(input('Please enter a number: '))
print('You entered: ' + str(n))

Please enter a number: a

-----
ValueError                                Traceback (most recent call last)
<ipython-input-3-6e62128bde98> in <module>
----> 1 n = int(input('Please enter a number: '))
      2
      3 print('You entered: ' + str(n))

ValueError: invalid literal for int() with base 10: 'a'
```

Try except blocks are good to take care of any unexpected exceptions. This approach is called exception handling.

```
In [9]: my_file = open('A:\demo\myfile.txt', "r")
for line in my_file:
    print(line)

Python 3 Foundation Course
Learning Journal
```

1. Open the file and read the numbers given in the file.
2. Divide each number with the divisor and sum the result.
3. Finally, return the sum in the end.

```
In [1]: def file_divide(file, divisor):
    my_file = open(file, "r")
    sum = 0

    for num in my_file:
        sum += int(num)/divisor

    my_file.close()
    return sum
```

```
In [1]: def file_divide(file, divisor):
    my_file = open(file, "r")
    sum = 0

    for num in my_file:
        sum += int(num)/divisor

    my_file.close()
    return sum

In [2]: file_divide('A:\\demo\\data.txt', 10)

Out[2]: 3.0
```

1. What if the file does not exist at a given location?
2. What if the caller passed a zero for the divisor?
3. What if the data file contains some non-numeric values?

Always have a wildcard exception to the end of the exception order -
Finally always gets executed

```
def file_divide(file, divisor):
    try:
        my_file = open(file, "r")
    except FileNotFoundError:
        print("File not found - Make sure to place " + file)
        return

    sum = 0
    try:
        for num in my_file:
            sum += int(num)/divisor
    except ZeroDivisionError:
        print('Cannot divide by zero.')
        return
    except ValueError:
        print('Invalid value! Check your data file.')
        return
    except:
        print('Something went wrong!')
        return
    finally:
        my_file.close()
    return sum
```

```
In [1]: def advance_by(die_input):
    print('move you by ' + str(die_input))

In [2]: advance_by(7)
```

move you by 7

```
In [3]: def advance_by(die_input):
    if die_input <= 0:
        print('Not a valid number. Enter a number greater than 0.')
    elif die_input > 6:
        print('Not a valid number. Enter a number less than or equal to 6.')
    else:
        print('moved you by ' + str(die_input))
```

If we do the return, then the caller will have hard time to compare the possible outcomes and determine what is the exception.

```
In [7]: def advance_by(die_input):
    if die_input <= 0:
        return('Not a valid number. Enter a number greater than 0.')
    elif die_input > 6:
        return('Not a valid number. Enter a number less than or equal to 6.')
    else:
        print('moved you by ' + str(die_input))
```

```
In [7]: def advance_by(die_input):
    if die_input <= 0:
        return('Not a valid number. Enter a number greater than 0.')
    elif die_input > 6:
        return('Not a valid number. Enter a number less than or equal to 6.')
    else:
        print('moved you by ' + str(die_input))
```

```
In [8]: result = advance_by(7)

if result == 'Not a valid number. Enter a number greater than 0.':
    print('die cannot give <= 0')
elif result == 'Not a valid number. Enter a number less than or equal to 6.':
    print('die cannot give > 6')

die cannot give > 6
```

This is good, however if this application is being developed and this error has to be communicated to the caller, then we need to prepare a customized exception so the caller can manage these errors using exception handling. So it is better we handle these errors using try, except block -

```
In [9]: def advance_by(die_input):
    if die_input <= 0:
        raise ValueError('Not a valid number. Enter a number greater than 0.')
    elif die_input > 6:
        raise ValueError('Not a valid number. Enter a number less than or equal to 6.')
    else:
        print("moved you by " + str(die_input))
```

```
In [10]: try:
    advance_by(7)
except ValueError:
    print('die cannot give <= 0 or > 6')

die cannot give <= 0 or > 6
```

```
In [9]: def advance_by(die_input):
    if die_input <= 0:
        raise ValueError('Not a valid number. Enter a number greater than 0.')
    elif die_input > 6:
        raise ValueError('Not a valid number. Enter a number less than or equal to 6.')
    else:
        print('moved you by ' + str(die_input))
```

```
In [11]: try:
    advance_by(7)
except ValueError as err:
    print(str(err))

Not a valid number. Enter a number less than or equal to 6.
```

In this case, we are raising same valueerror exception to identify whether the number > 0 or <=6

We defined base class as DieInputError and extended it from base class Exception.

```
class DieInputError(Exception):
    pass
    ↓
class DieInputLowerBoundError(DieInputError):
    pass

class DieInputUpperBoundError(DieInputError):
    pass|
```

```
In [1]: class DieInputError(Exception):
         pass
```

```
class DieInputLowerBoundError(DieInputError):
    pass
```

```
class DieInputUpperBoundError(DieInputError):
    pass
```

```
In [2]: def advance_by(die_input):
         if die_input <= 0:
             raise DieInputLowerBoundError('Not a valid number. Enter a number greater than 0.')
         elif die_input > 6:
             raise DieInputUpperBoundError('Not a valid number. Enter a number less than or equal to 6.')
         else:
             print('moved you by ' + str(die_input))
```

```
In [3]: try:
         advance_by(0)
except DieInputLowerBoundError:
    print('Value <= 0')
except DieInputUpperBoundError:
    print('Value > 6')
```

```
Value <= 0
```

Classes -

```
In [1]: class MyClass:
         my_num = 1234
→ def say_hello(name):
         print('Hello ' + name)
```

```
In [1]: class MyClass:
         my_num = 1234
         def say_hello(name):
             print('Hello ' + name)
```

```
In [2]: MyClass.my_num
```

```
Out[2]: 1234
```

```
In [3]: MyClass.my_num = 4567
```

```
In [4]: MyClass.my_num
```

```
Out[4]: 4567
```

```
In [5]: MyClass.say_hello('Prashant')
```

```
Hello Prashant
```

```
class class_name:
    # Class Body
```

`__init__` is the constructor and it should have first argument as "self". Primary purpose of the constructor is to create an object and set the initial state of object.

```
In [1]: class Car:  
    def __init__(self, name, brand):  
        self.model_name = name  
        self.model_brand = brand  
  
    def launch(self):  
        print(f"{self.model_name} by {self.model_brand} launched.")
```

```
In [ ]:
```

- 1. Initializer or Constructor
- 2. Instance Variables
- 3. Class Methods

```
In [1]: class Car:  
    def __init__(self, name, brand):  
        self.model_name = name  
        self.model_brand = brand  
  
    def launch(self):  
        print(f'{self.model_name} by {self.model_brand} launched.')
```

```
In [2]: car1 = Car('Hector', 'MG')
```

```
In [3]: car1.model_name
```

```
Out[3]: 'Hector'
```

```
In [1]: class Car:  
    def __init__(self, name, brand):  
        self.model_name = name  
        self.model_brand = brand  
  
    def launch(self):  
        print(f'{self.model_name} by {self.model_brand} launched.')
```

```
In [2]: car1 = Car('Hector', 'MG')
```

```
In [5]: car1.model_name = 'Hector Plus'
```

```
In [6]: car1.model_name
```

```
Out[6]: 'Hector Plus'
```

```
In [1]: class Car:  
    def __init__(self, name, brand):  
        self.model_name = name  
        self.model_brand = brand  
  
    def launch(self):  
        print(f"{self.model_name} by {self.model_brand} launched.")  
  
In [2]: car1 = Car('Hector', 'MG')  
  
In [7]: car2 = Car('i10', 'Hyundai')  
  
In [10]: car1.model_name  
Out[10]: 'Hector Plus'  
  
In [11]: car2.model_name  
Out[11]: 'i10'
```

First argument is always self for all the instance methods.

```
In [1]: class Car:  
    def __init__(self, name, brand):  
        self.model_name = name  
        self.model_brand = brand  
  
    def launch(self):  
        print(f"{self.model_name} by {self.model_brand} launched.")  
  
In [2]: car1 = Car('Hector', 'MG')  
  
In [7]: car2 = Car('i10', 'Hyundai')  
  
In [12]: car1.launch()  
Hector Plus by MG launched.  
  
In [13]: car2.launch()  
i10 by Hyundai launched.
```

Python will accept any other name instead of self. Like in below -

```
In [13]: class Car:  
    def __init__(self, name, brand):  
        self.model_name = name  
        self.model_brand = brand  
  
    def launch(this):  
        print(f"{this.model_name} by {this.model_brand} launched.")
```

```
class Car:
    def __init__(self):
        self.current_state = "Standing"
        self.current_speed = 0

    def start(self):
        self.current_state = "Running"
        self.current_speed = 10

    def stop(self):
        self.current_state = "Standing"
        self.current_speed = 0

    def speed_up(self, speed):
        if self.current_state == "Standing":
            self.start()
        self.current_speed += speed

    def speed_down(self, speed):
        if self.current_state == "Running":
            self.current_speed -= speed

    def show_state(self):
        print(f"{self.current_state} and current speed is {self.current_speed}.")
```

```
In [4]: class Hector(Car):
    pass
```

```
In [5]: h1 = Hector()
```

```
In [6]: h1.show_state()
```

```
Standing and current speed is 0.
```

```
def stop(self):
    self.current_state = "Standing"
    self.current_speed = 0

def speed_up(self, speed):
    if self.current_state == "Standing":
        self.start()
    self.current_speed += speed

def speed_down(self, speed):
    if self.current_state == "Running":
        self.current_speed -= speed

def show_state(self):
    print(f"{self.current_state} and current speed is {self.current_speed}.")
```

```
In [7]: class Hector(Car):
    def show_state(self):
        print(f"Hector is {self.current_state} and current speed is {self.current_speed}.")
```

```

        self.current_speed += speed
43. Class Inheritance
    def speed_down(self, speed):
        if self.current_state == "Running":
            self.current_speed -= speed

    def show_state(self):
        print(f"{self.current_state} and current speed is {self.current_speed}.")

```

```

In [7]: class Hector(Car):
    def show_state(self):
        print(f"Hector is {self.current_state} and current speed is {self.current_speed}.")

In [8]: h1 = Hector()

In [9]: h1.show_state()
Hector is Standing and current speed is 0.

```

```

In [11]: h1 = Hector()

In [12]: h1.start()
h1.show_state()

Hector is Running and current speed is 10.

In [13]: h1.speed_up(40)
h1.show_state()

Hector is Running and current speed is 50.

In [14]: h1.speed_down(30)
h1.show_state()

Hector is Running and current speed is 20.

```

Left part is Class variable and Class Methods where as right ones are instance variables and instance methods as we are referring them using self.

Class Methods vs Instance Methods

```

class Car:
    brand = "MG Hector"

    def launch_year():
        return 2020

```

How to access attributes?
Car.brand
Car.launch_year()

```

class Car:
    def __init__(self):
        self.current_state = "Standing"
        self.current_speed = 0

    def show_state(self):
        print(f"{self.current_state} and current
speed is {self.current_speed}")

```

How to access attributes?
car1 = Car()
car1.start()

```
In [1]: class Car:
    brand = "MG Hector"

    def launch_year():
        return 2020

    def __init__(self):
        self.current_state = "Standing"
        self.current_speed = 0

    def show_state(self):
        print(f"{self.brand} is {self.current_state} and current speed is {self.current_speed}.")
```

The screenshot shows a Jupyter Notebook interface with the following content:

```
In [1]: class Car:
    brand = "MG Hector"

    def launch_year():
        return 2020

    def __init__(self):
        self.current_state = "Standing"
        self.current_speed = 0

    def show_state(self):
        print(f"{self.brand} is {self.current_state} and current speed is {self.current_speed}.")
```

```
In [2]: car1 = Car()
```

```
In [3]: car2 = Car()
```

```
In [4]: car1.show_state()
MG Hector is Standing and current speed is 0.
```

```
In [1]: class Car:
    brand = "MG Hector"

    def launch_year():
        return 2020

    def __init__(self):
        self.current_state = "Standing"
        self.current_speed = 0

    def show_state(self):
        print(f"{self.brand} is {self.current_state} and current speed is {self.current_speed}.")
```

```
In [8]: Car.brand = "MG Hector Plus"
```

```
In [9]: car1.show_state()
MG Hector Plus is Running and current speed is 10.
```

```
In [8]: Car.brand = "MG Hector Plus"
```

```
In [9]: car1.show_state()
MG Hector Plus is Running and current speed is 10.
```

```
In [10]: car2.show_state()
MG Hector Plus is Standing and current speed is 0.
```

Don't try to change the class variable using object. It is not the right way to change.

You can call the class variable using the object where as not the class method.

```
In [13]: Car.launch_year()
Out[13]: 2020
```

```
In [14]: car1.launch_year()
-----
TypeError                                 Traceback (most recent call last)
<ipython-input-14-9d57fb8e1da2> in <module>
----> 1 car1.launch_year()

TypeError: launch_year() takes 0 positional arguments but 1 was given
```

However you can achieve calling the method using the object by decorating the method with `@classmethod` and passing the `cls` argument. `cls` argument is similar to `self` in instance methods.

```
In [15]: class Car:
    brand = "MG Hector"

    @classmethod
    def launch_year(cls):
        return 2020

    def __init__(self):
        self.current_state = "Standing"
        self.current_speed = 0

    def show_state(self):
        print(f"{self.brand} is {self.current_state} and current speed is {self.current_speed}.")
```

```
all_123.. Class Car:
    brand = "MG Hector"

    @classmethod
    def launch_year(cls):
        return 2020

    def __init__(self):
        self.current_state = "Standing"
        self.current_speed = 0

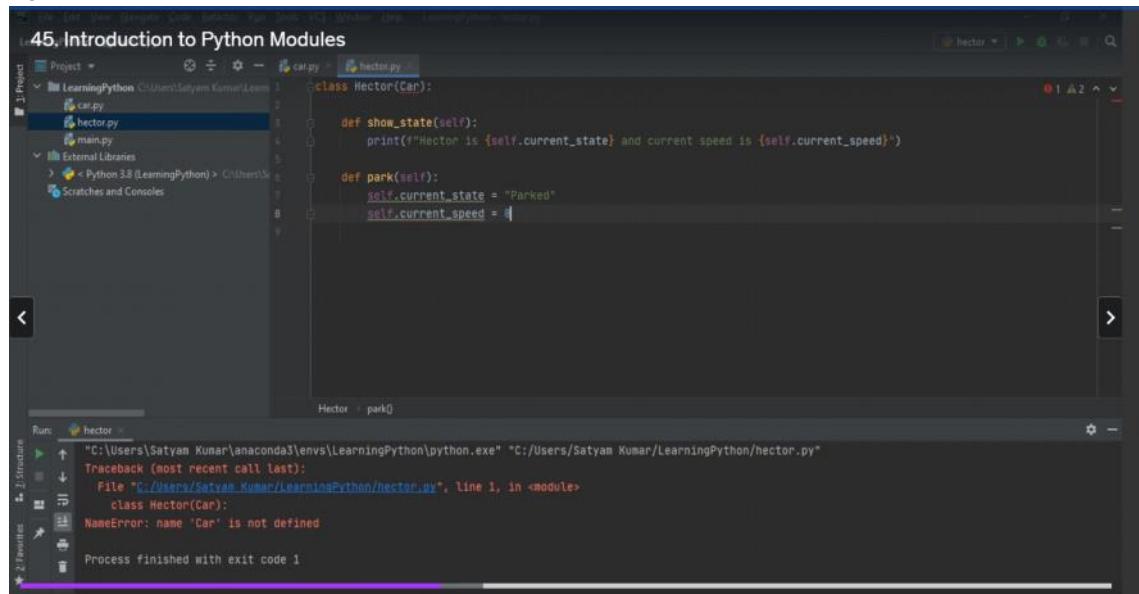
    def show_state(self):
        print(f"{self.brand} is {self.current_state} and current speed is {self.current_speed}.")
```

```
In [16]: Car.launch_year()
Out[16]: 2020
```

```
In [18]: car1.launch_year()
Out[18]: 2020
```

Python modules -

We can see the Hector class is not able to figure out the Car class as the Car class is defined in different file.



45. Introduction to Python Modules

Project

- LearningPython
 - car.py
 - hector.py
 - main.py
- External Libraries
- > Python 3.8 (LearningPython) > Children
- Scratches and Consoles

```
class Car:
    brand = "MG Hector"

    @classmethod
    def launch_year(cls):
        return 2020

    def __init__(self):
        self.current_state = "Standing"
        self.current_speed = 0

    def show_state(self):
        print(f"{self.brand} is {self.current_state} and current speed is {self.current_speed}.")
```

```
class Hector(Car):
    def show_state(self):
        print(f"Hector is {self.current_state} and current speed is {self.current_speed}.")
```

Run: hector

```
"C:\Users\Satyam Kumar\anaconda3\envs\LearningPython\python.exe" "C:/Users/Satyam Kumar/LearningPython/hector.py"
Traceback (most recent call last):
  File "C:/Users/Satyam Kumar/LearningPython/hector.py", line 1, in <module>
    class Hector(Car):
NameError: name 'Car' is not defined

Process finished with exit code 1
```

After adding from car import Car it started working

```
car.py x hector.py x
1 from car import Car
2
3
4 class Hector(Car):
5
6     def show_state(self):
7         print(f"Hector is {self.current_state} and current speed is {self.current_speed}")
8
9     def park(self):
10        self.current_state = "Parked"
11        self.current_speed = 0
```

Each .py file in python is also called as python module.

We can notice after importing Car class from car module (car.py file). It started working.

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help LearningPython - hector.py
LearningPython hector.py
Project LearningPython C:\Users\Satyam Kumar\Learni...
  car.py
  hector.py
  main.py
External Libraries > Python 3.8 (LearningPython) > C:\Users\S...
Scratches and Consoles
```

```
1 from car import Car
2
3
4 class Hector(Car):
5
6     def show_state(self):
7         print(f"Hector is {self.current_state} and current speed is {self.current_speed}")
8
9     def park(self):
10        self.current_state = "Parked"
11        self.current_speed = 0
```

```
45. Introduction to Python Modules
File Edit View Navigate Code Refactor Run Tools VCS Window Help LearningPython - hector.py
LearningPython hector.py
Project LearningPython C:\Users\Satyam Kumar\Learni...
  car.py
  hector.py
  main.py
External Libraries > Python 3.8 (LearningPython) > C:\Users\S...
Scratches and Consoles
```

```
1 from car import Car
2
3
4 class Hector(Car):
5
6     def show_state(self):
7         print(f"Hector is {self.current_state} and current speed is {self.current_speed}")
8
9     def park(self):
10        self.current_state = "Parked"
11        self.current_speed = 0
12
13
14 h1 = Hector()
15 h1.start()
16 h1.show_state
```

```
Run: hector
"C:\Users\Satyam Kumar\anaconda3\envs\LearningPython\python.exe" "C:/Users/Satyam Kumar/LearningPython/hector.py"
Running and current speed is 10
Hector is Running and current speed is 10
Process finished with exit code 0
```

We can see in addition to the hector code, we are also seeing the code from Car module is also getting executed -

45. Introduction to Python Modules

```
Project : LearningPython C:\Users\Satyam Kumar\Learn
File : hector.py
15      self.current_speed = 0
16
17  def stop(self):
18      self.current_state = "Standing"
19      self.current_speed = 0
20
21  def speed_up(self, speed):
22      if self.current_state == "Standing":
23          self.start()
24      self.current_speed += speed
25
26  def speed_down(self, speed):
27      if self.current_state == "Running":
28          self.current_speed -= speed
29
30  else:
31      def show_state(self):
32          print(f"{self.current_state} and current speed is {self.current_speed}")
33
34  car1 = Car()
35  car1.start()
36  car1.show_state()
```

`__name__` is set to `__main__` when the module is run from inside of the module.

45. Introduction to Python Modules

```
Project : LearningPython C:\Users\Satyam Kumar\Learn
File : hector.py
16      self.current_speed = 0
17
18  def stop(self):
19      self.current_state = "Standing"
20      self.current_speed = 0
21
22  def speed_up(self, speed):
23      if self.current_state == "Standing":
24          self.start()
25      self.current_speed += speed
26
27  def speed_down(self, speed):
28      if self.current_state == "Running":
29          self.current_speed -= speed
30
31  else:
32      def show_state(self):
33          print(f"{self.current_state} and current speed is {self.current_speed}")
34
35  car1 = Car()
36  car1.start()
37  car1.show_state()
38
39 if __name__ == "__main__":
40     car1 = Car()
41     car1.start()
42     car1.show_state()
```

Now after the change, lets run hector module -

The screenshot shows the PyCharm IDE interface. The project structure on the left includes files car.py, hector.py, and main.py. The hector.py file is open in the editor, displaying Python code that defines a class Hector that inherits from Car. It contains methods show_state and park, and a main block that creates an instance of Hector, starts it, and then calls its show_state method. The run output window at the bottom shows the command "C:\Users\Satyam Kumar\anaconda3\envs\LearningPython\python.exe" "C:/Users/Satyam Kumar/LearningPython/hector.py" and the resulting output: "Hector is Running and current speed is 10".

```
from car import Car

class Hector(Car):
    def show_state(self):
        print(f"Hector is {self.current_state} and current speed is {self.current_speed}")

    def park(self):
        self.current_state = "Parked"
        self.current_speed = 0

h1 = Hector()
h1.start()
h1.show_state()
```

This screenshot shows the same PyCharm environment as above, but with a modification to the hector.py code. A new line has been added at the bottom: "if __name__ == '__main__':". This is a common idiom in Python to ensure that the code in the module runs only when it is executed directly, and not when it is imported as a module.

```
from car import Car

class Hector(Car):
    def show_state(self):
        print(f"Hector is {self.current_state} and current speed is {self.current_speed}")

    def park(self):
        self.current_state = "Parked"
        self.current_speed = 0

if __name__ == '__main__':
    h1 = Hector()
    h1.start()
    h1.show_state()
```

We can have another py (module) to create the objects and run from there. So classes exist in individual classes where as the execution happens from this main module

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help LearningPython - CarApp.py
1: Project 1: LearningPython C:\Users\Satyam Kumar\Learnin...
  LearningPython
    car.py
    CarApp.py
    hector.py
    main.py
  External Libraries
  Scratches and Consoles
  Python 3.8 (LearningPython) > C:\Users\Satyam Kumar\Learnin...
  Run: CarApp
  "C:\Users\Satyam Kumar\anaconda3\envs\LearningPython\python.exe" "C:/Users/Satyam Kumar/LearningPython/CarApp.py"
  Hector is Running and current speed is 40
  Hector is Running and current speed is 10
  Process finished with exit code 0
```

Python enables to club hundred's of modules into one module

Create a package and organize the modules (py files) init. A module is a directory with a `__init__.py` file init.

```
File Edit View Navigate Code Refactor Run Tools VCS Window Help merit.py - __init__.py
LearningPython > vehicles > __init__.py
1: Project 1: LearningPython C:\Users\Satyam Kumar\Learnin...
  LearningPython
    vehicles
      __init__.py
      bike.py
      car.py
      CarApp.py
      hector.py
  External Libraries
  Scratches and Consoles
```

I'll leave the main module in the project root and rest of the files will be moved under "vehicles" folder.

PyCharm IDE showing the `hector.py` file in the `LearningPython` project. The code defines a `Hector` class that inherits from `Car`. It has methods `show_state` and `park`. The `__main__` block creates an instance of `Hector`, starts it, and prints its state.

```
from vehicles.car import Car

class Hector(Car):
    def show_state(self):
        print(f"Hector is {self.current_state} and current speed is {self.current_speed}")

    def park(self):
        self.current_state = "Parked"
        self.current_speed = 0

if __name__ == "__main__":
    h1 = Hector()
    h1.start()
    h1.show_state()
```

PyCharm IDE showing the `LearningPython` project structure. The `LearningPython` folder contains a `vehicles` directory which is expanded to show `fourwheelers` and `twowheelers` sub-directories, each containing `__init__.py`, `car.py`, `hector.py`, and `bike.py`.

PyCharm IDE showing the `CarApp.py` file in the `LearningPython` project. It imports `Hector` from `fourwheelers.hector`. The `__main__` block creates two instances of `Hector`, starts them, and calls their `speed_up` and `show_state` methods.

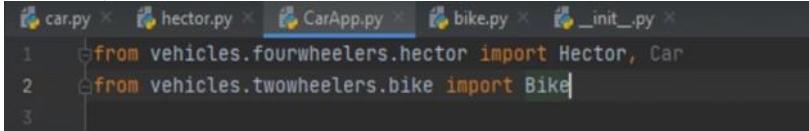
```
from vehicles.fourwheelers.hector import Hector

if __name__ == "__main__":
    hector1 = Hector()
    hector2 = Hector()

    hector1.start()
    hector2.start()

    hector1.speed_up(30)
    hector1.show_state()
    hector2.show_state()
```

You can import like below -



```
car.py × hector.py × CarApp.py × bike.py × __init__.py ×
1 from vehicles.fourwheelers.hector import Hector, Car
2 from vehicles.twowheelers.bike import Bike
3
```

A screenshot of a code editor window showing five tabs at the top: car.py, hector.py, CarApp.py (which is the active tab), bike.py, and __init__.py. Below the tabs, there are three lines of Python code. Line 1 imports Hector and Car from the vehicles.fourwheelers.hector module. Line 2 imports Bike from the vehicles.twowheelers.bike module.