DATA SCIENCE AT LIGHT SPEED

BY KHALID MOHAMED

Khalid Mohamed

The FASTEST Way to Start Your Data Science Journey...

Information is the oil of the 21st century, and analytics is the combustion engine - Peter Sondergaard

Contents

1. Python data types, values and Identifiers

Variables in python are reserved spaces in memory that can hold some value, so whenever a variable is created some space is reserved in memory to hold the value of that variable. Every variable in python is a specific data type which tells the python interpreter how much memory should be reserved for the variable and what type of values the variable can hold.

In python variables don't need to be declared to assign values to them, so values can be assigned to variable names without explicit declaration of that variable beforehand,the declaration happens when you assign values to them.

1.1 Assigning values to variables

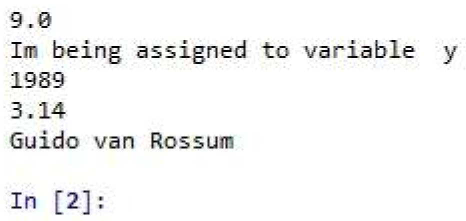
Assignments in python creates references not copies.Variables hold references to objects and do not hold the object itself. Names don't have intrinsic types objects have types, the type of reference is determined by the type of object assigned to the name. Remember the words "Name", "Variable" and "Identifier" refer to the same thing.

A name is created the first time it appears on the left hand side of an assignment. The code below shows some examples.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Feb 20 15:42:18 2019

author: khalid
"""
x = 3 + 12 / 2
y = "I'm being assigned to the variable y"

a = 1989
b = 3.14
```

```python
c = "Guido van Rossum"

print(x)
print(y)

print(a)
print(b)
print(c)

```

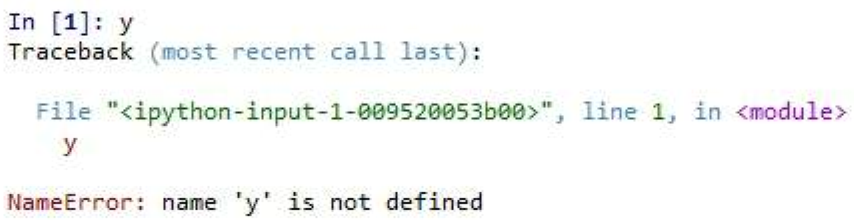
This python code when executed gives the following result.



```
9.0
Im being assigned to variable  y
1989
3.14
Guido van Rossum

In [2]:
```

The garbage collector deletes reference after any names that are bound to the reference have passed out of scope. When the reference is deleted the object being referenced is deleted.

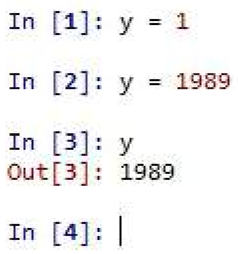
In the case where a non existent name(a variable that doesn't exist yet) is accessed an error is raised. Take a look at the example.



```
In [1]: y
Traceback (most recent call last):

  File "<ipython-input-1-009520053b00>", line 1, in <module>
    y

NameError: name 'y' is not defined
```

If a name is accessed after assignment it returns the value the name was last assigned.

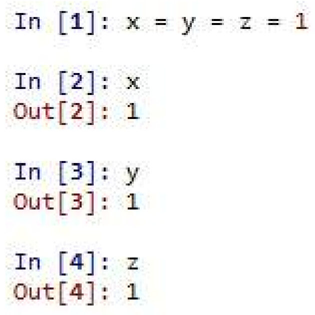
Take a look at the example given below.



```
In [1]: y = 1

In [2]: y = 1989

In [3]: y
Out[3]: 1989

In [4]: |
```

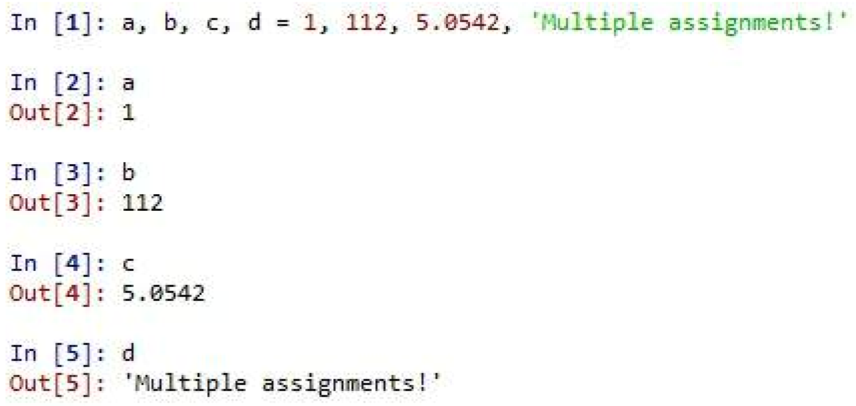
1.2 Multiple assignment

Python allows assigning a single value to several names(variables) simultaneously. For example:



```
In [1]: x = y = z = 1

In [2]: x
Out[2]: 1

In [3]: y
Out[3]: 1

In [4]: z
Out[4]: 1
```

Memory is reserved for the integer object with the value 1, and all three variables refer to the same object. Python also allows assigning multiple objects to multiple variables like the example below.



```
In [1]: a, b, c, d = 1, 112, 5.0542, 'Multiple assignments!'

In [2]: a
Out[2]: 1

In [3]: b
Out[3]: 112

In [4]: c
Out[4]: 5.0542

In [5]: d
Out[5]: 'Multiple assignments!'
```

Here two integer objects are assigned to the names ‘a’ and ‘b’ , a float object is assigned to the name ‘c’ and a string object is assigned to the name ‘d’.

1.3 Naming rules

Names are case sensitive they can contain letters, numbers and underscores but they cant start with a number.

Examples of valid name instances:

```
Var, VAR, Var1223, V123Bob, c020b
```

Examples of invalid instances:

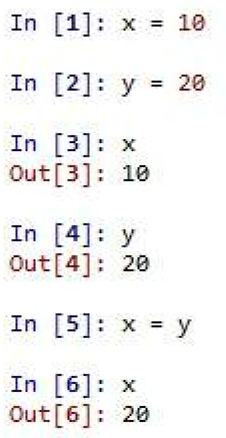
```
1Var,  2a00
```

Reserved words cant be used as a variable name, so the following cant be used as names.

```
and, assert, break, class, continue, def, del, elif, else, except,
finally, for, from, global, if, import, in, is, lambda, not, or,
pass, raise, return, try, while
```

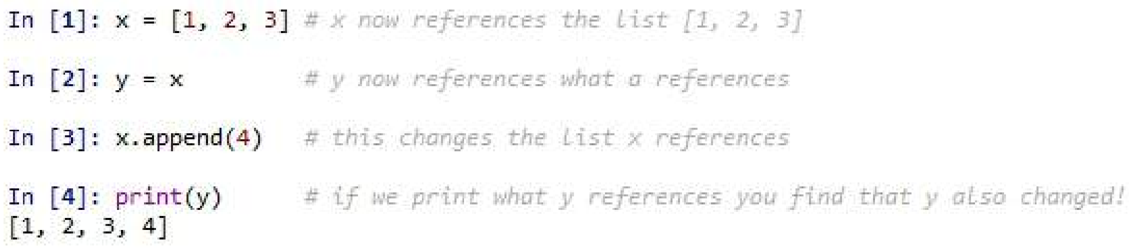
1.4 Understanding reference semantics

This may be different from the language that you used to work with but in python an assignment operation changes references instead of making a new copy of the object. This can be better understood with an example.



```
In [1]: x = 10

In [2]: y = 20

In [3]: x
Out[3]: 10

In [4]: y
Out[4]: 20

In [5]: x = y

In [6]: x
Out[6]: 20
```

Here `x = y` doesn't make a copy of the object `y` references, instead it makes `x` reference the object(integer 20) that y refers to.This can prove to be very useful but we have to be careful for example take a loo at the code below.



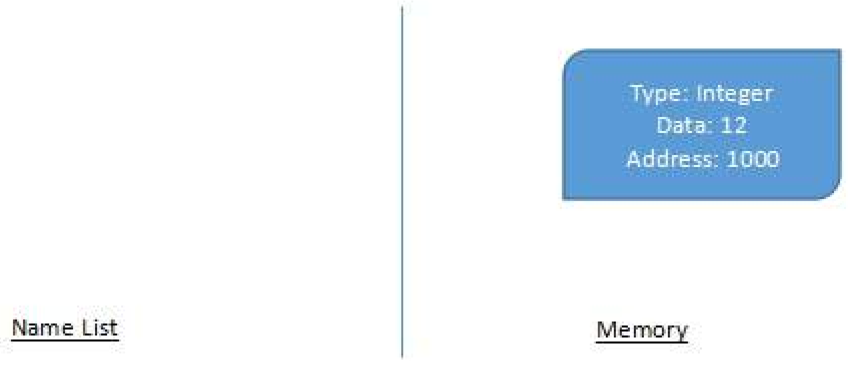
```
In [1]: x = [1, 2, 3] # x now references the list [1, 2, 3]

In [2]: y = x         # y now references what a references

In [3]: x.append(4)   # this changes the list x references

In [4]: print(y)      # if we print what y references you find that y also changed!
[1, 2, 3, 4]
```

If this is confusing it's important to first understand what happens when an assignment operation(like x = 3) is performed.

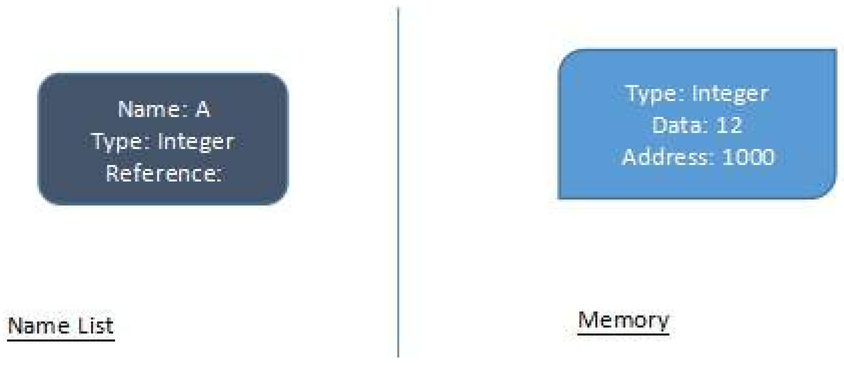
There is a lot that happens in an assignment. For example this is what happens when the below assignment expression is executed

```
A = 12
```

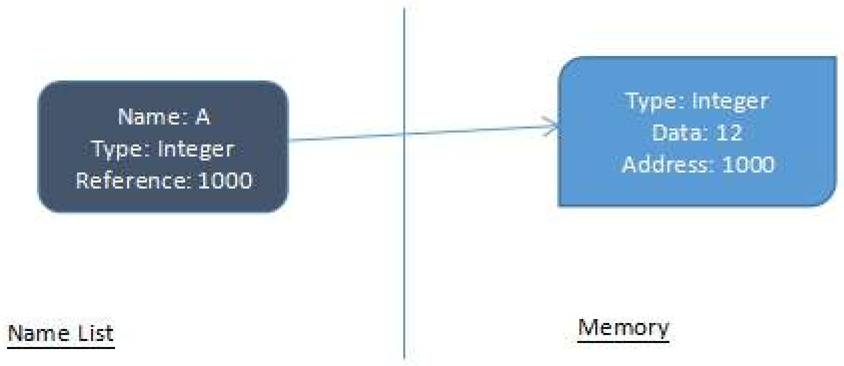
1. First an integer object 12 is created and is stored in memory.



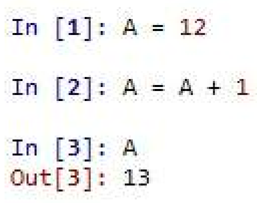
2. A name 'A' is created.



3. A reference to the memory location where the integer object 12 is stored at is now assigned to the name ‘`A`’



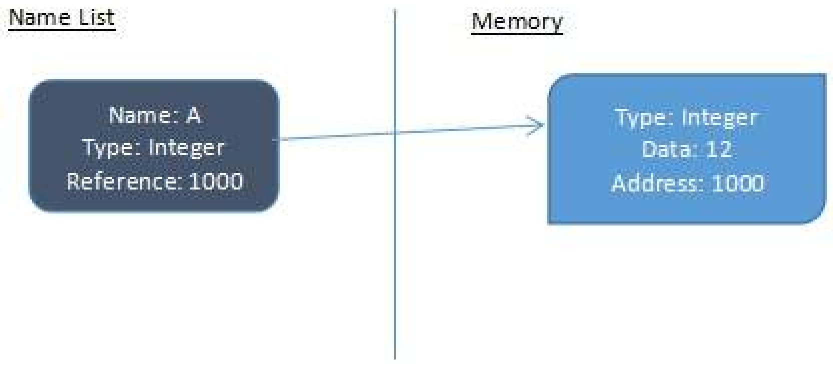
The data 12 that was just created is of type integer. In python the data types integer, float, string and tuple are immutable(which means that object cant be changed). This does not mean that we cant change the value of the variable ‘`A`’, it can still be reassigned to another integer object i.e what ‘`A`’ refers to can be changed. For example we could still increment ‘`A`’ even though its data type is immutable.



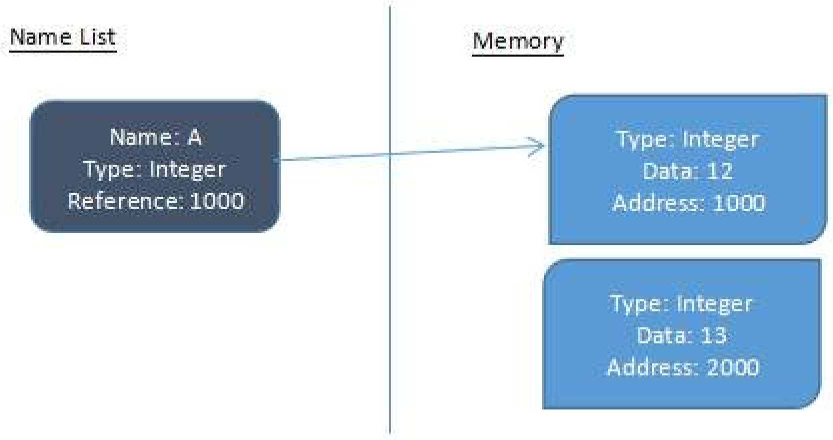
```
In [1]: A = 12

In [2]: A = A + 1

In [3]: A
Out[3]: 13
```

Lets look at what’s really happening when ‘`A`’ is incremented(`A = A + 1`).

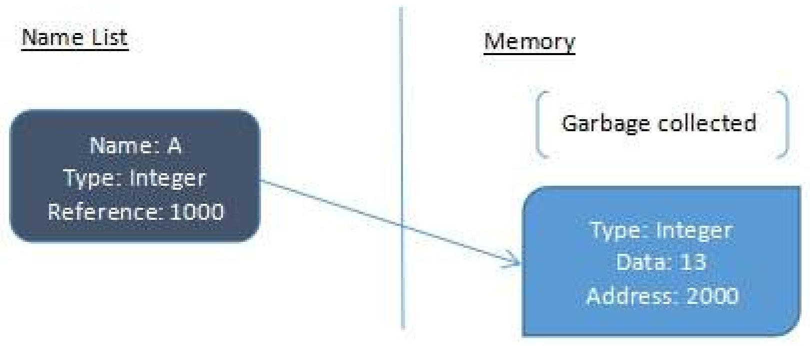
1. The reference of ‘`A`’ is looked up.
2. The value of the integer object at that reference is retrieved.



3. The 12 + 1 increment operation is performed and the result integer object 13 is stored in a new memory location



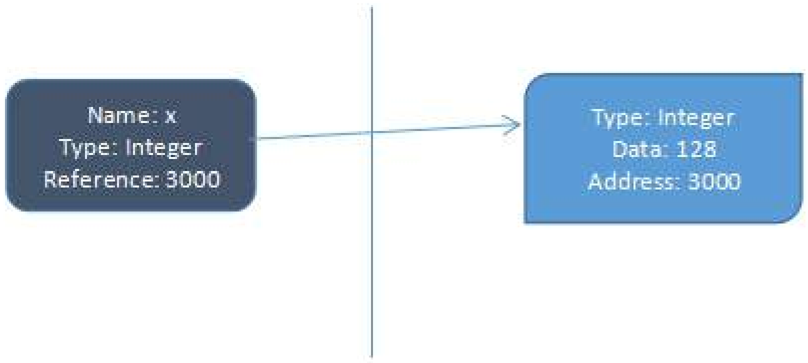
4. The variable 'A' is changed to point to this new integer object.while the previuos integer object '12' is deleted by the garbage collector as there are no variables referring to it.



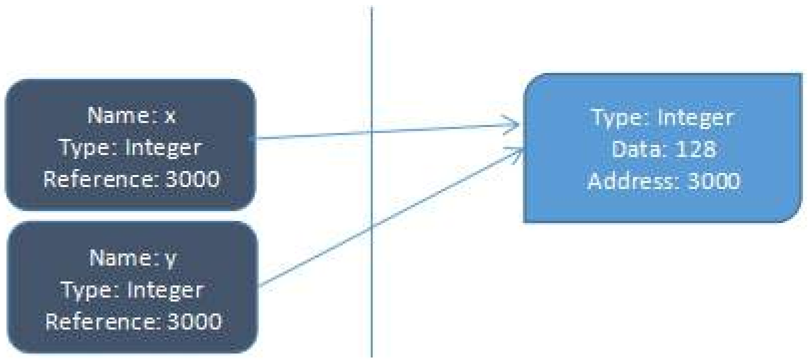
So for pythons basic data types assignments behaves just as you expect them to.

Take another example.

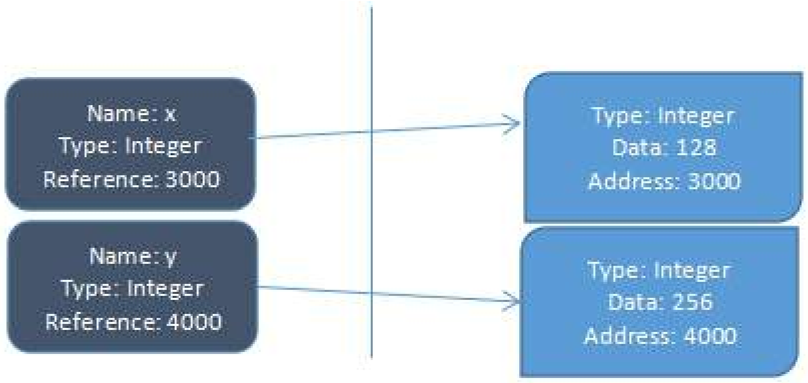
```
In [1]: x = 128
```

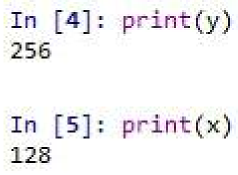


```
In [2]: y = x
```



```
In [3]: y = 256
```

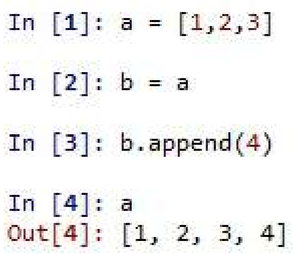




```
In [4]: print(y)
256

In [5]: print(x)
128
```

These data types that we've worked with till now are immutable(the values of the objects cant be changed). Assignments in python works differently when its done with mutable objects.

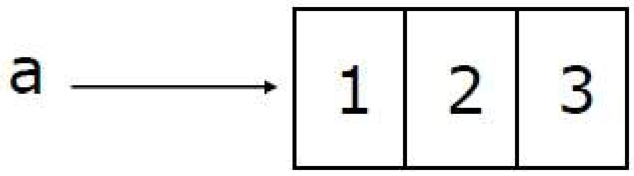
Data types like lists, dictionaries and user defined types are mutable. When mutable data is changed the change happens in place. This means they are not copied to a new memory location every time they are changed. If you type "`y = x`", this changes the value of '`y`' with the value of '`x`'. Remember a new copy is not created to make a change, the change is performed on the same object(unlike for immutable objects) so all variables referring to this object has its value changed as well. We'll go through an example on this just to make it clear.



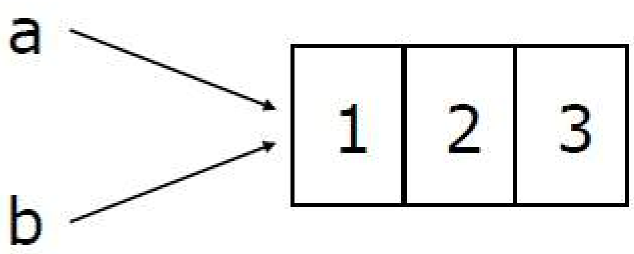
```
In [1]: a = [1,2,3]

In [2]: b = a

In [3]: b.append(4)

In [4]: a
Out[4]: [1, 2, 3, 4]
```

This is what happens when the above code is executed.

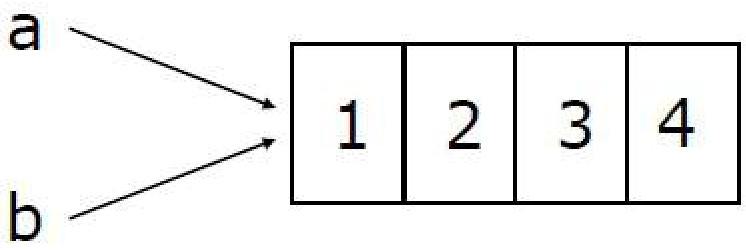
```
In [1]: a = [1,2,3]
```



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



```
In [3]: b.append(4)
```



Lets look at each of python standard data types briefly one by one.

1.5 Python's standard data types

If you have experience in other programming languages you would already know that the value that can be stored in variables are of different types. Having different data types is of course a requirement of programmers in general. Each data type defines the different operations possible and it's storage mechanism.

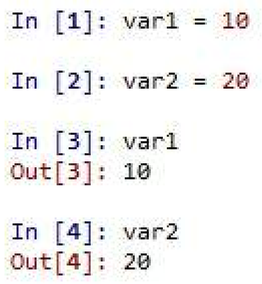
Python has five standard data types:

- Integers
- Strings
- List
- Tuple
- Dictionary

1.5.1 Python Integer(Number)

Integer objects in python are immutable,so the value of the objects can’t be changed so a new object is created every time a change is made to the variable

Number objects are created when a value is assigned to them.

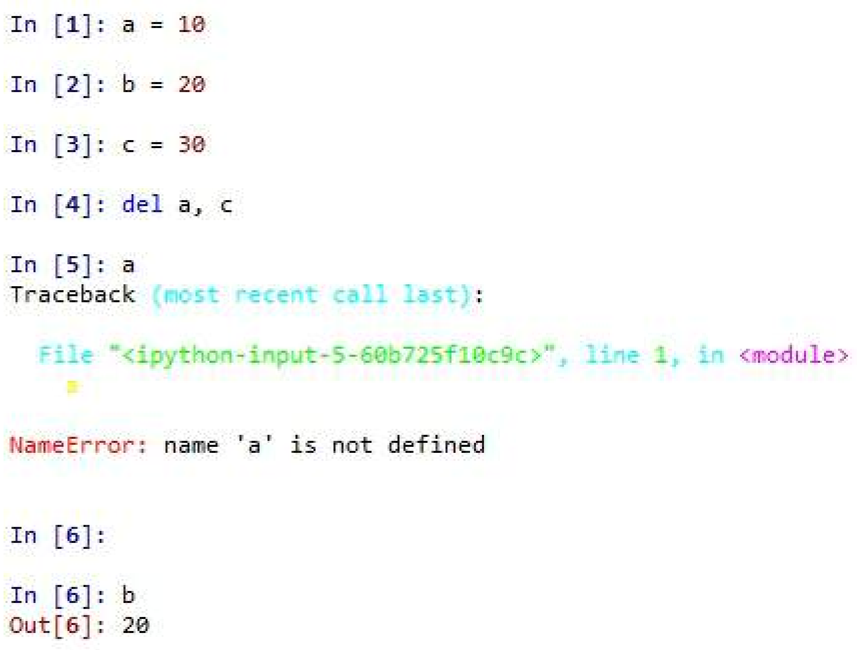


```
In [1]: var1 = 10

In [2]: var2 = 20

In [3]: var1
Out[3]: 10

In [4]: var2
Out[4]: 20
```

These objects can be deleted using the ‘`del`’ keyword. The syntax for the ‘`del`’ command is as follows.

```
del var1[,var2[,var3[....,varN]]]]
```

With ‘`del`’ objects can be deleted one at a time or many at a time. In the following code three variables(a, b, c) are created then ‘`a`’ and ‘`c`’ are deleted using ‘`del`’ command



```
In [1]: a = 10

In [2]: b = 20

In [3]: c = 30

In [4]: del a, c

In [5]: a
Traceback (most recent call last):

  File "<ipython-input-5-60b725f10c9c>", line 1, in <module>


NameError: name 'a' is not defined


In [6]:

In [6]: b
Out[6]: 20
```

There are four different numerical types in Python:

1. int : These are standard signed integers. There is no upper limit on the number of bits that an int object can take up so they can take as much as the memory can offer. This can be tested out by running the following script

```
# A python script to show that we can store

# large numbers in python


a = 20000000000000000000000000000000000000001

a = a + 11

print(a)

```

Output:

```
20000000000000000000000000000000000000011
```

In python 3 there is only one int type but in python 2.7 there are two int types, one is ‘int’ which is of 32 bit length and the other is ‘long int’ which is the same as the Python 3.x int type(it can store arbitrarily large numbers).

When the following code is run on python 3.x and python 2.7 outputs in each would be as follows

```python
# This program shows that there are two types in
# Python 2.7: int and long int
# and in python 3 there is only one type: int

a = 200
print(type(a))

b = 2000000000000000000000000000000000000000000
print(type(b))

```

Output in python 3.x :

```
<class 'int'>
<class 'int'>
```

Output in python 2.7 :

```
<type 'int'>
<type 'long'>
```

2. float (floating point real values): floats can be used to represent real numbers with a whole number part and a fractional part.
3. complex (complex numbers): represented by the formula a + bJ, where a and b are real numbers(float), and J is the square root of -1 (the result of which is an imaginary number). Complex numbers aren't that often used by python programmers.

Examples of each numeric type:

int	Long int	float	complex
1242	5193488661L	0.0	3.14j
-0009743	-0x19323L	-21.9	45.j
-0x3344	0162L	42.3+e18	4e+26J
1	-05236456727123L	80.2-E12	-.6985+0J

0x958	-47223598762L	-92.	4.76e-7j

Remember ‘long int’ is only in python 2.7 which is the same as ‘int’ in python 3.

1.5.2 Sequence types(Lists, Tuples and Strings)

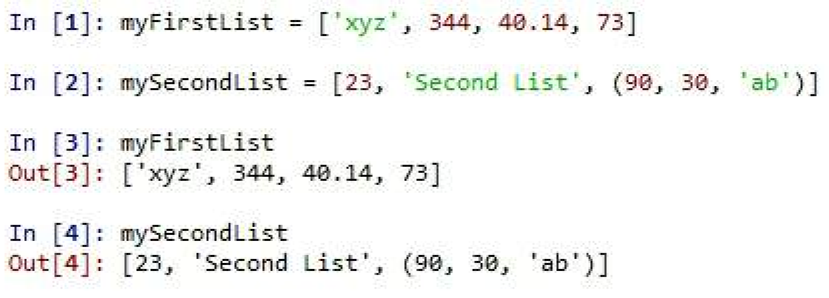
Among the standard types in python there are three sequence types Lists, Tuples and Strings.

List:

A List is a simple ordered sequence of items, these items can be of different types and can also include other sequence types. Lists are mutable so to the value of list objects can be changed in place with out having to create a copy of the object that has the changed value.

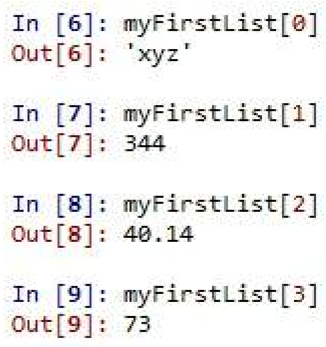
List is the most versatile of the collection types in python. They contain a sequence of items separated by commas(,) and enclosed within square brackets([ ]) This might seem similar to arrays in c, but here items of the list can be of different types.

A list variable in python is created when a list object is assigned to a name as follows.



```
In [1]: myFirstList = ['xyz', 344, 40.14, 73]

In [2]: mySecondList = [23, 'Second List', (90, 30, 'ab')]

In [3]: myFirstList
Out[3]: ['xyz', 344, 40.14, 73]

In [4]: mySecondList
Out[4]: [23, 'Second List', (90, 30, 'ab')]
```

List are defined using square brackets and commas. Individual members of a list can be accessed using the square bracket notation, the same way arrays are accessed in c as follows.

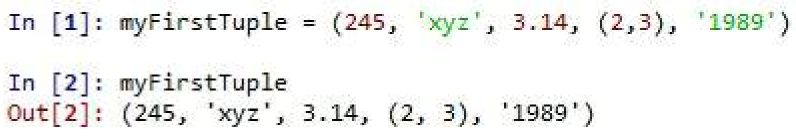


```
In [6]: myFirstList[0]
Out[6]: 'xyz'

In [7]: myFirstList[1]
Out[7]: 344

In [8]: myFirstList[2]
Out[8]: 40.14

In [9]: myFirstList[3]
Out[9]: 73
```

Tuple:

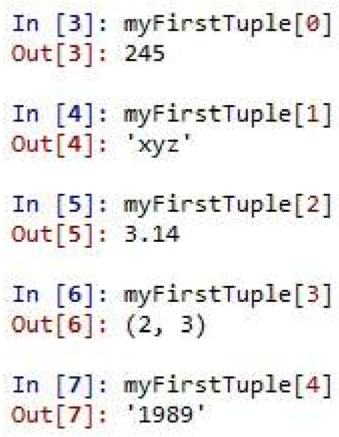
A Tuple is a simple ordered sequence of items, these items can be of different types and can also include sequence types. They are immutable so to change the content of a tuple object a new object with the changed value is created and the tuple variable is made to point to this new object.

Python tuples are similar to lists except that unlike lists they are enclosed within parentheses and tuple objects are immutable. Tuples can be thought of as read only lists.Creating a tuple variable is similar to the creation of a list variable but with parentheses instead of square brackets.



```
In [1]: myFirstTuple = (245, 'xyz', 3.14, (2,3), '1989')

In [2]: myFirstTuple
Out[2]: (245, 'xyz', 3.14, (2, 3), '1989')
```

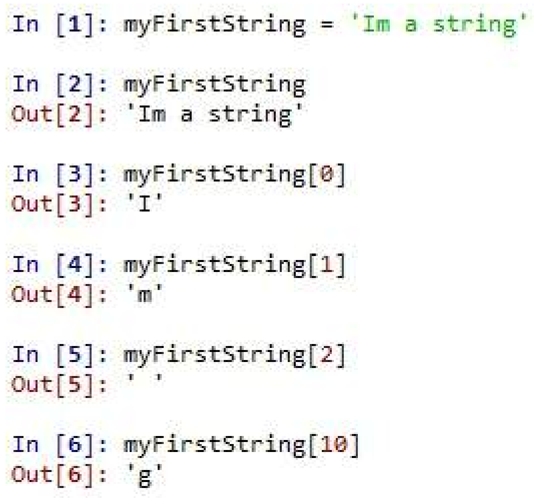
And just like lists, individual members of a tuple can be accessed using the square bracket notation, the same way arrays are accessed in c .



```
In [3]: myFirstTuple[0]
Out[3]: 245

In [4]: myFirstTuple[1]
Out[4]: 'xyz'

In [5]: myFirstTuple[2]
Out[5]: 3.14

In [6]: myFirstTuple[3]
Out[6]: (2, 3)

In [7]: myFirstTuple[4]
Out[7]: '1989'
```

String:

Strings in python are a continuous set of characters that are enclosed within quotation marks. Python allows using either single quotes(‘’) or double quotes(“”).Strings are immutable like tuples, so the value of a string object cant be changed in place to change the value of a string object a new object is created with the changed value. Declaration of a string and accessing individual elements of a string are the same as in lists and tuples. Examples of string declaration and access are given below.



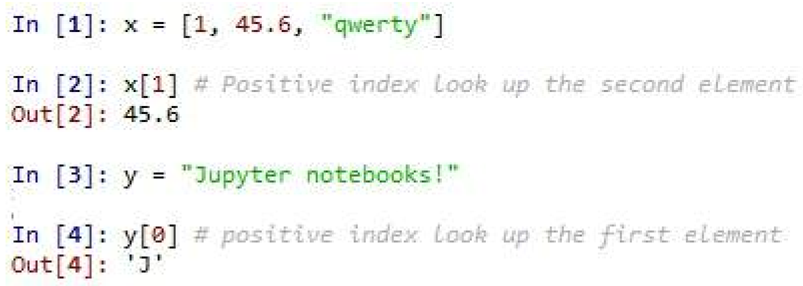
```
In [1]: myFirstString = 'Im a string'

In [2]: myFirstString
Out[2]: 'Im a string'

In [3]: myFirstString[0]
Out[3]: 'I'

In [4]: myFirstString[1]
Out[4]: 'm'

In [5]: myFirstString[2]
Out[5]: ' '

In [6]: myFirstString[10]
Out[6]: 'g'
```

1.5.3 Operations on sequence types

The operations shown here can be applied to all sequence types. Examples are shown for clarity on how they work. Most examples show the operation performed only on one or two of the sequence types but remember that these operations can be performed on all of the sequence types.

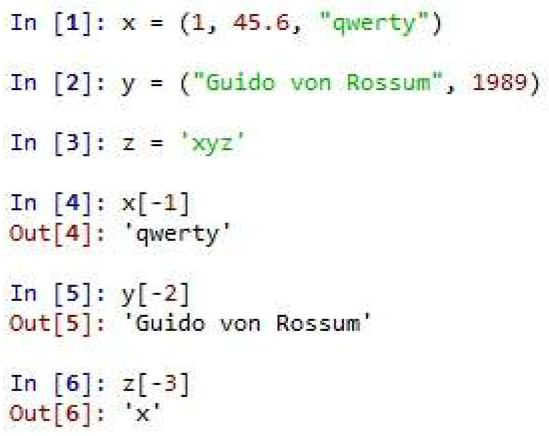
Positive and Negative index look up:

Positive index look up counts from the left and starts the count from 0.



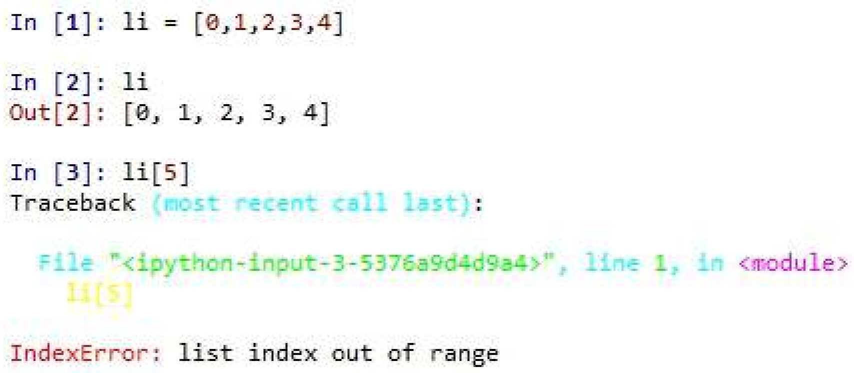
```
In [1]: x = [1, 45.6, "qwerty"]

In [2]: x[1] # Positive index Look up the second element
Out[2]: 45.6

In [3]: y = "Jupyter notebooks!"

In [4]: y[0] # positive index Look up the first element
Out[4]: 'J'
```

Negative index look up counts from the left and starts the count from -1.



```
In [1]: x = (1, 45.6, "qwerty")

In [2]: y = ("Guido von Rossum", 1989)

In [3]: z = 'xyz'

In [4]: x[-1]
Out[4]: 'qwerty'

In [5]: y[-2]
Out[5]: 'Guido von Rossum'

In [6]: z[-3]
Out[6]: 'x'
```

When an element that is out of range is accessed the python interpreter gives an index error, an example is shown below.



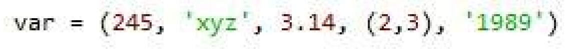
```
In [1]: li = [0,1,2,3,4]

In [2]: li
Out[2]: [0, 1, 2, 3, 4]

In [3]: li[5]
Traceback (most recent call last):

  File "<ipython-input-3-5376a9d4d9a4>", line 1, in <module>
    li[5]

IndexError: list index out of range
```

Slicing:

Slicing can be used to return a copy subset of the sequence typed object. The general form of a slicing operation is as follows

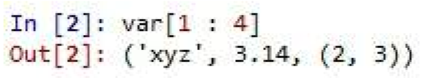
```
<Variable Name>[x : y]
```

Where ‘x’ and ‘y’ are the first and second indices of the slicing operation. The container is copied from index ‘x’ and the copying is discontinued from index y. Lets go through some examples so its made clear how the slicing operation actually works. We’ll work with the list ‘li’ given below.



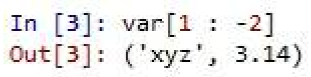
```
var = (245, 'xyz', 3.14, (2,3), '1989')
```

The below slicing operation on ‘var’ returns a copy of a subset of ‘var’. Copying starts from the first index(1) and discontinues from the second index(4).



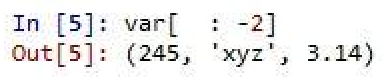
```
In [2]: var[1 : 4]
Out[2]: ('xyz', 3.14, (2, 3))
```

Negative indices can also be used when slicing like below.



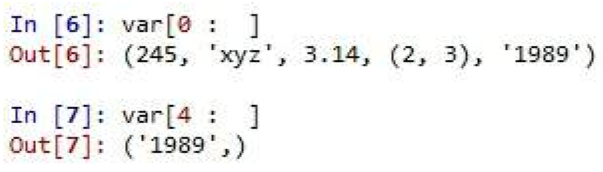
```
In [3]: var[1 : -2]
Out[3]: ('xyz', 3.14)
```

The first index can be left blank to return a copy of a subset of the that starts copying from the first element of the container.



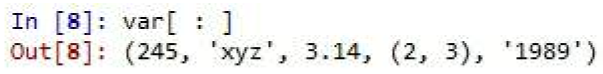
```
In [5]: var[  : -2]
Out[5]: (245, 'xyz', 3.14)
```

The second index can be left blank to start copying from the first index and continue copying till the last element.



```
In [6]: var[0 :  ]
Out[6]: (245, 'xyz', 3.14, (2, 3), '1989')

In [7]: var[4 :  ]
Out[7]: ('1989',)
```

To return a copy of the entire container both indices can be left blank



```
In [8]: var[ : ]
Out[8]: (245, 'xyz', 3.14, (2, 3), '1989')
```

From the above examples it could be understood that for any sequence typed variable say ‘A’

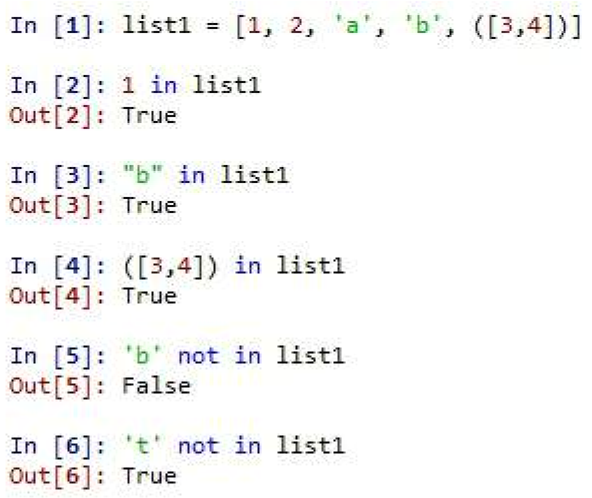
```
A[ : ] = A[0 : ] != A[0 : -1]
```

Since lists are mutable sequence types, a change in the objects value happens in place without creating a new copy of the object, but wen the slicing operator is used a new object is created and returned. Take a look at the example given below.

```
list1 = [1, 2, 3, 4]

list2 = list1          # 2 names referring to the same object
                       # When one name is changed the other changes as well

list2 = list1[ : ]      # Name list2 refers to a new object which is a
                       # copy of a subset list1

```

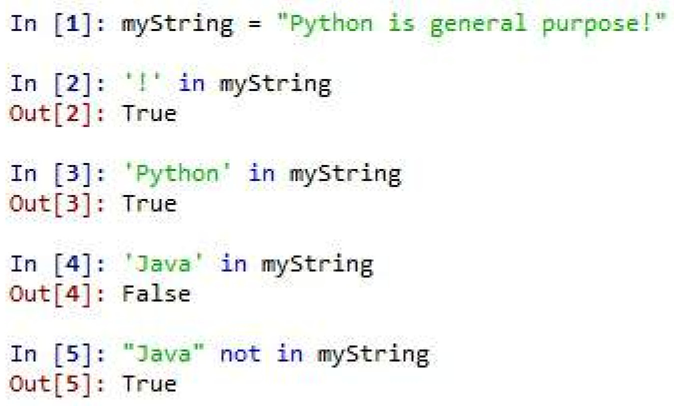
The “in” operator:

The “in” operator can be used as a Boolean test to check whether some value is inside a container object.



```
In [1]: list1 = [1, 2, 'a', 'b', ([3,4])]

In [2]: 1 in list1
Out[2]: True

In [3]: "b" in list1
Out[3]: True

In [4]: ([3,4]) in list1
Out[4]: True

In [5]: 'b' not in list1
Out[5]: False

In [6]: 't' not in list1
Out[6]: True
```

For strings the “in” operator can be used to test whether the given sub string exists in the string like in the examples given below.

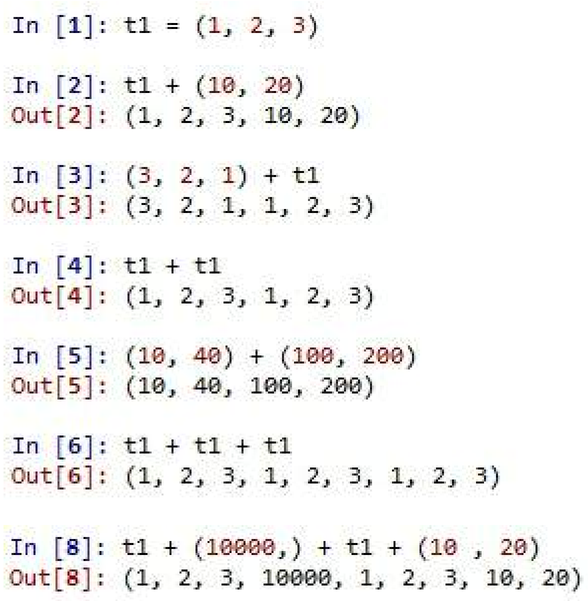


```
In [1]: myString = "Python is general purpose!"

In [2]: '!' in myString
Out[2]: True

In [3]: 'Python' in myString
Out[3]: True

In [4]: 'Java' in myString
Out[4]: False

In [5]: "Java" not in myString
Out[5]: True
```

The ‘+’ operator:

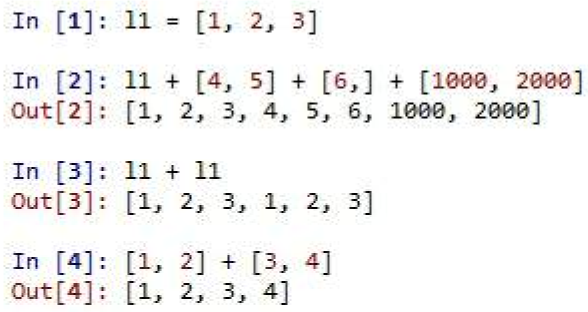
This operator is used to concatenate two tuples, strings or lists together the result obtained after use is the new object obtained after concatenation. Concatenation can not be performed with different container types. Here are some examples of the use of the ‘+’ operator with tuples, lists and strings.

With tuples:



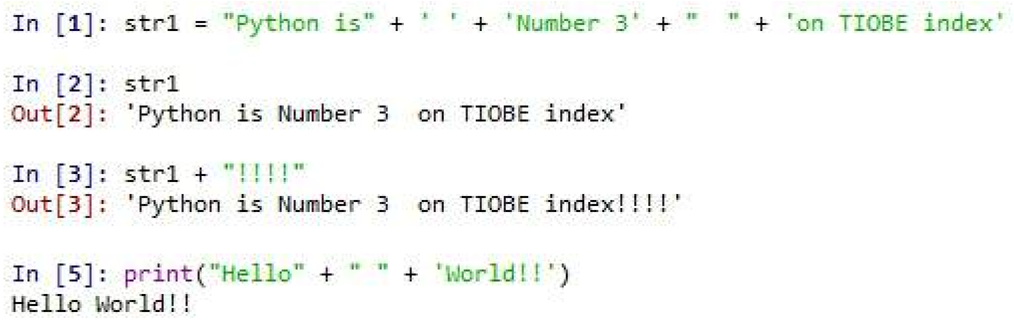
```
In [1]: t1 = (1, 2, 3)

In [2]: t1 + (10, 20)
Out[2]: (1, 2, 3, 10, 20)

In [3]: (3, 2, 1) + t1
Out[3]: (3, 2, 1, 1, 2, 3)

In [4]: t1 + t1
Out[4]: (1, 2, 3, 1, 2, 3)

In [5]: (10, 40) + (100, 200)
Out[5]: (10, 40, 100, 200)

In [6]: t1 + t1 + t1
Out[6]: (1, 2, 3, 1, 2, 3, 1, 2, 3)

In [8]: t1 + (10000,) + t1 + (10 , 20)
Out[8]: (1, 2, 3, 10000, 1, 2, 3, 10, 20)
```

With lists:(A new object is returned as the result)



```
In [1]: l1 = [1, 2, 3]

In [2]: l1 + [4, 5] + [6,] + [1000, 2000]
Out[2]: [1, 2, 3, 4, 5, 6, 1000, 2000]

In [3]: l1 + l1
Out[3]: [1, 2, 3, 1, 2, 3]

In [4]: [1, 2] + [3, 4]
Out[4]: [1, 2, 3, 4]
```

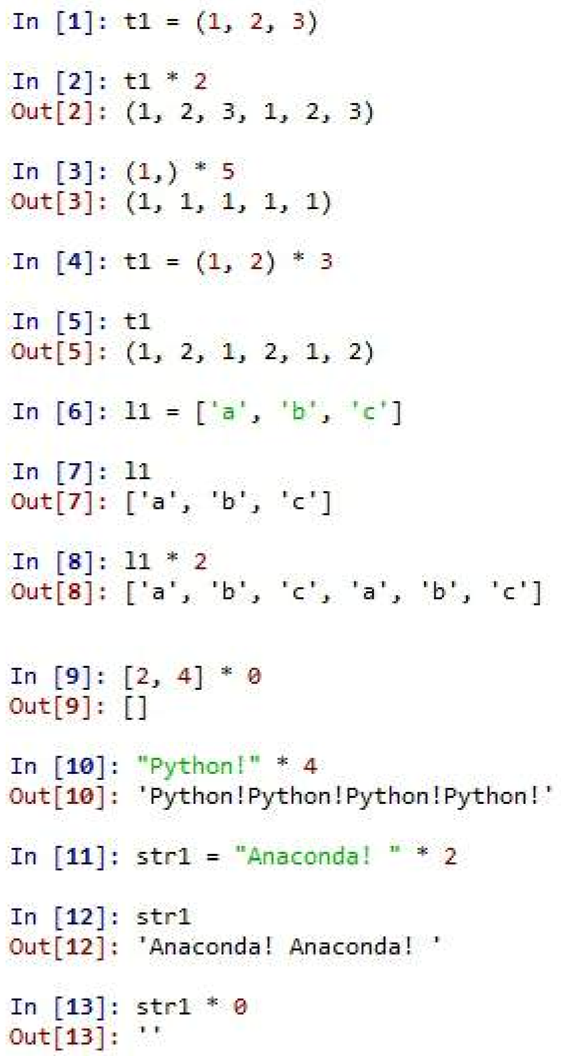
With strings:



```
In [1]: str1 = "Python is" + ' ' + 'Number 3' + "  " + 'on TIOBE index'

In [2]: str1
Out[2]: 'Python is Number 3  on TIOBE index'

In [3]: str1 + "!!!!"
Out[3]: 'Python is Number 3  on TIOBE index!!!!'

In [5]: print("Hello" + " " + 'World!!')
Hello World!!
```

The ‘*’ operator:

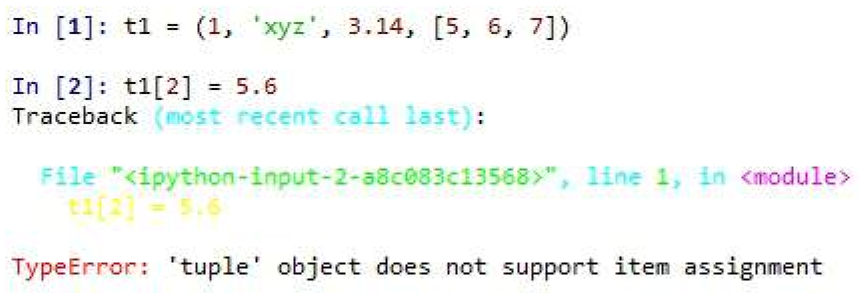
Produces a new tuple, list or string that has the original value repeated a number of times, the number of repeats is given as an operand to this operator and the other operand is the sequence itself. Lets take a look at some examples.



```
In [1]: t1 = (1, 2, 3)

In [2]: t1 * 2
Out[2]: (1, 2, 3, 1, 2, 3)

In [3]: (1,) * 5
Out[3]: (1, 1, 1, 1, 1)

In [4]: t1 = (1, 2) * 3

In [5]: t1
Out[5]: (1, 2, 1, 2, 1, 2)

In [6]: l1 = ['a', 'b', 'c']

In [7]: l1
Out[7]: ['a', 'b', 'c']

In [8]: l1 * 2
Out[8]: ['a', 'b', 'c', 'a', 'b', 'c']


In [9]: [2, 4] * 0
Out[9]: []

In [10]: "Python!" * 4
Out[10]: 'Python!Python!Python!Python!'

In [11]: str1 = "Anaconda! " * 2

In [12]: str1
Out[12]: 'Anaconda! Anaconda! '

In [13]: str1 * 0
Out[13]: ''
```

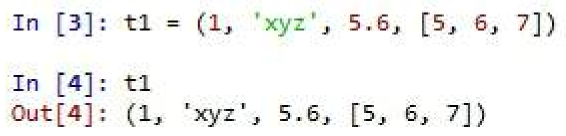
1.5.4 Lists and Tuples (Mutability vs immutability)

Tuple is an immutable type so the value/content of a tuple object can’t be changed the only way to change the value of the a tuple variable is to create a new tuple object with the changed value and assign the new reference to the tuple variable. The value/content of a list can be changed in place as they are mutable, but when a value of a mutable object that is referred to by more than one variable is changed the value is changed for all variables referring to that object.Take a look at the example below.



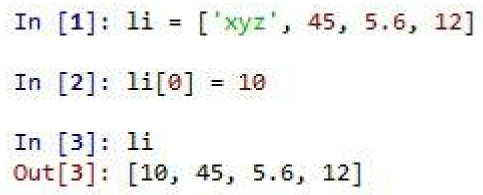
```
In [1]: t1 = (1, 'xyz', 3.14, [5, 6, 7])

In [2]: t1[2] = 5.6
Traceback (most recent call last):

  File "<ipython-input-2-a8c083c13568>", line 1, in <module>
    t1[2] = 5.6

TypeError: 'tuple' object does not support item assignment
```

Tuples can’t be changed but instead make a fresh tuple and assign the reference of it to an already existing name like below.



```
In [3]: t1 = (1, 'xyz', 5.6, [5, 6, 7])

In [4]: t1
Out[4]: (1, 'xyz', 5.6, [5, 6, 7])
```

Unlike tuples, lists are mutable and so they can be changed in place without having to change the reference to a new object



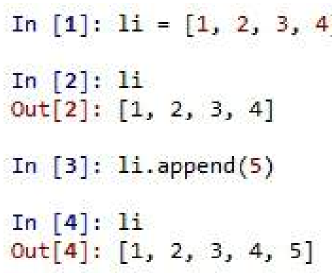
```
In [1]: li = ['xyz', 45, 5.6, 12]

In [2]: li[0] = 10

In [3]: li
Out[3]: [10, 45, 5.6, 12]
```

Like in the above example lists can be changed in place. Name “li” still points to the same memory reference when the changed is done. The mutability of lists means that they aren’t as fast as tuples.

Since lists are mutable there are some operations that are applicable only to lists and not tuples. lets go through those operations one by one. The list class has some in built methods that can be used to change the value of the list object or return a value based on the current state of the list object. This is also our first exposure to pythons method syntax.

append( ) :

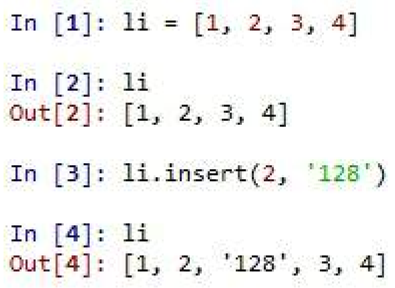
The list class has an append method that can be used to insert a new list element at the end of the list. In the example below a new name “li” is assigned a reference to a list object, then the append method of the list class is called to add the element passed as an argument to method at the end of the list object.



```
In [1]: li = [1, 2, 3, 4]

In [2]: li
Out[2]: [1, 2, 3, 4]

In [3]: li.append(5)

In [4]: li
Out[4]: [1, 2, 3, 4, 5]
```

insert( ) :

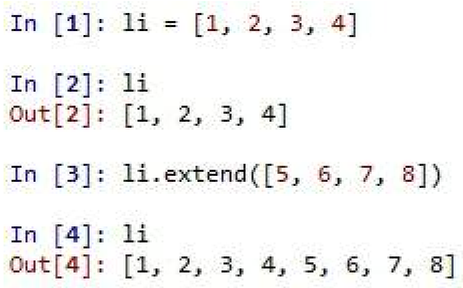
The list class has an insert method that can be used to insert an new element at any location the list. The insert method takes two parameter's, first one is the location in the list to which the the new element is to be inserted and the second is the element itself. In the below example a new element is inserted at location 2 in the list.



```
In [1]: li = [1, 2, 3, 4]

In [2]: li
Out[2]: [1, 2, 3, 4]

In [3]: li.insert(2, '128')

In [4]: li
Out[4]: [1, 2, '128', 3, 4]
```

extend( ) :

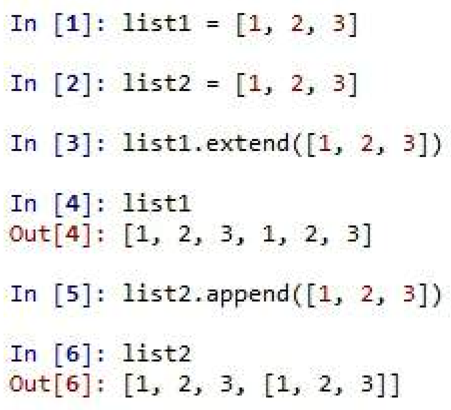
Some times instead of appending single elements at a time we may want to append a list of elements to a list, the list class has a method to do that, the extend method. The extend method takes a list as an argument and appends that list to the list object to which it was called upon. An example is given below.



```
In [1]: li = [1, 2, 3, 4]

In [2]: li
Out[2]: [1, 2, 3, 4]

In [3]: li.extend([5, 6, 7, 8])

In [4]: li
Out[4]: [1, 2, 3, 4, 5, 6, 7, 8]
```

It may look like the extend method does exactly what the ‘+’ operator does for sequence types but the difference is that the ‘+’ operator creates a new list with a new memory reference, and extend operates on the list object it’s called upon in place.

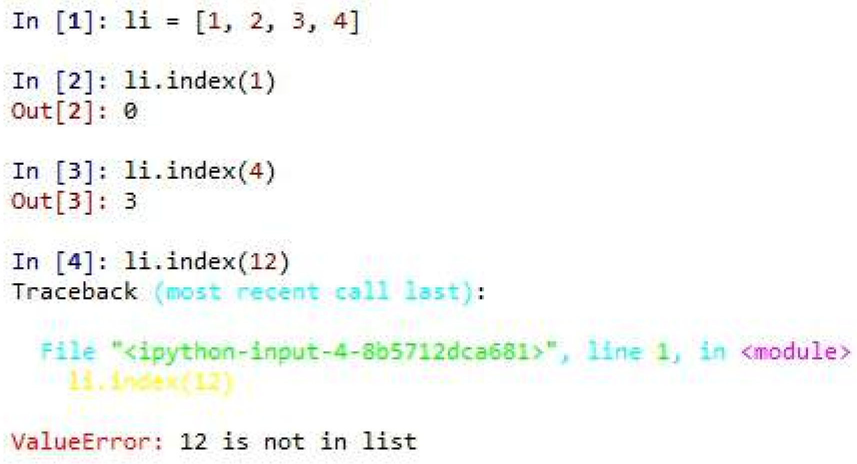
Extend method takes a list as an argument and append method takes a single object as an argument. Take a look at the example below for more clarity.



```
In [1]: list1 = [1, 2, 3]

In [2]: list2 = [1, 2, 3]

In [3]: list1.extend([1, 2, 3])

In [4]: list1
Out[4]: [1, 2, 3, 1, 2, 3]

In [5]: list2.append([1, 2, 3])

In [6]: list2
Out[6]: [1, 2, 3, [1, 2, 3]]
```

index( ) :

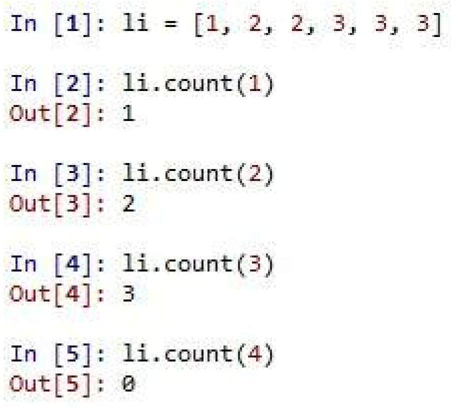
List class has an index method that returns the index of the first occurrence of the value passed as an argument to the method. If the value passed does not exist in the list then a “value error” is raised by the python interpreter. Examples are shown below.



```
In [1]: li = [1, 2, 3, 4]

In [2]: li.index(1)
Out[2]: 0

In [3]: li.index(4)
Out[3]: 3

In [4]: li.index(12)
Traceback (most recent call last):

  File "<ipython-input-4-8b5712dca681>", line 1, in <module>
    li.index(12)

ValueError: 12 is not in list
```

count( ) :

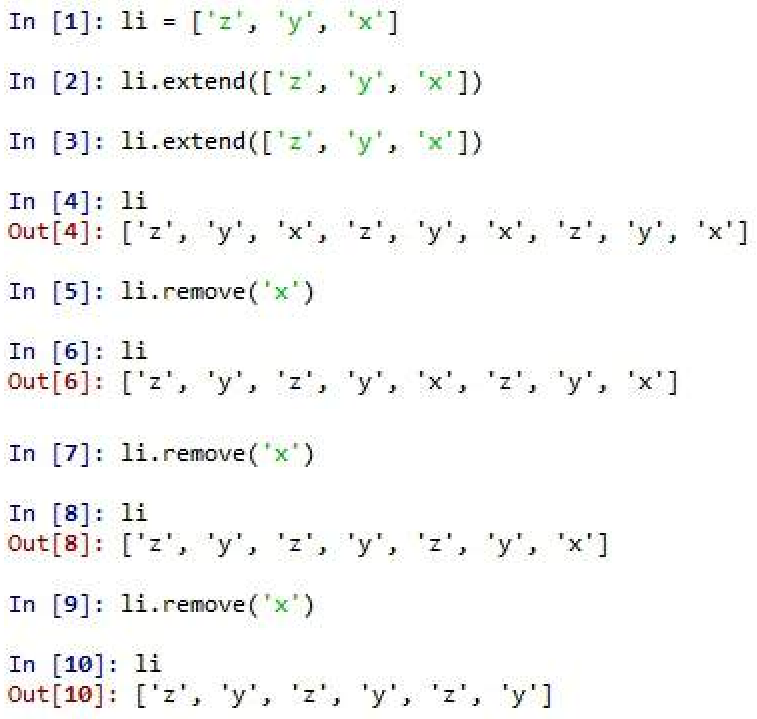
When a value is passed as an argument to the count method it returns the number of occurrences of that value in the list object on which it was called upon.



```
In [1]: li = [1, 2, 2, 3, 3, 3]

In [2]: li.count(1)
Out[2]: 1

In [3]: li.count(2)
Out[3]: 2

In [4]: li.count(3)
Out[4]: 3

In [5]: li.count(4)
Out[5]: 0
```

remove( ) :

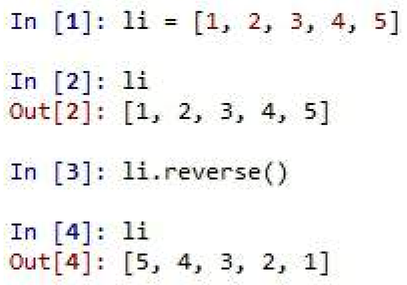
The remove method of the list class can be used to remove the first occurrence of the value passed as the argument to the method.examples are shown below.



```
In [1]: li = ['z', 'y', 'x']

In [2]: li.extend(['z', 'y', 'x'])

In [3]: li.extend(['z', 'y', 'x'])

In [4]: li
Out[4]: ['z', 'y', 'x', 'z', 'y', 'x', 'z', 'y', 'x']

In [5]: li.remove('x')

In [6]: li
Out[6]: ['z', 'y', 'z', 'y', 'x', 'z', 'y', 'x']

In [7]: li.remove('x')

In [8]: li
Out[8]: ['z', 'y', 'z', 'y', 'z', 'y', 'x']

In [9]: li.remove('x')

In [10]: li
Out[10]: ['z', 'y', 'z', 'y', 'z', 'y']
```

reverse( ) :

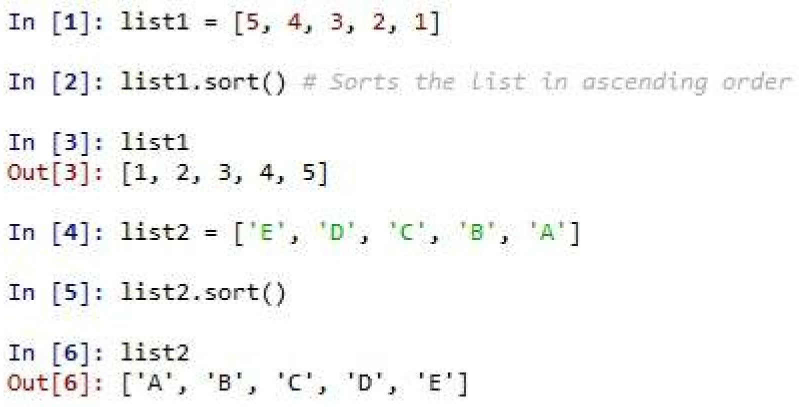
When the reverse method of the list class is called on a list object it reverses the list object in place in the memory. In the below example when the reverse method is called for a list object, the order of its content are reversed.



```
In [1]: li = [1, 2, 3, 4, 5]

In [2]: li
Out[2]: [1, 2, 3, 4, 5]

In [3]: li.reverse()

In [4]: li
Out[4]: [5, 4, 3, 2, 1]
```

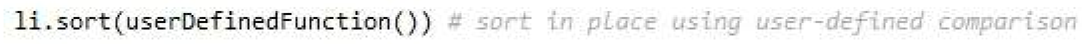
sort( ) :

The sort method of the list class can be used to sort the contents of the list in place in memory



```
In [1]: list1 = [5, 4, 3, 2, 1]

In [2]: list1.sort() # Sorts the list in ascending order

In [3]: list1
Out[3]: [1, 2, 3, 4, 5]

In [4]: list2 = ['E', 'D', 'C', 'B', 'A']

In [5]: list2.sort()

In [6]: list2
Out[6]: ['A', 'B', 'C', 'D', 'E']
```

The sort function of the list class can be passed a user defined function with it’s own approach to sorting a list.

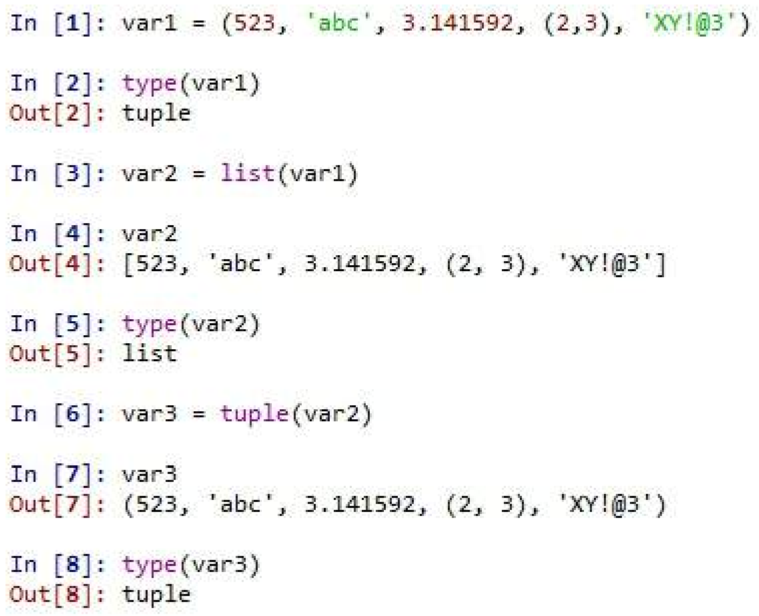


```
li.sort(userDefinedFunction()) # sort in place using user-defined comparison
```

Although lists are slower than tuples, they are much more powerful than tuples and are the most versatile types in python. Lists unlike tuples are mutable, they can be modified and they have a lot of operations that can be performed on them that make them very handy while programming. Tuples are immutable, they can’t be modified and they have much lesser features than lists, although they are faster than lists.

list( ) and tuple( ) :

Python provides the list method and tuple method to convert between lists and tuples. The list method takes a tuple and returns a new instance of a list object. The tuple method takes a list and returns a new instance of a tuple object. Some examples are shown below.



```
In [1]: var1 = (523, 'abc', 3.141592, (2,3), 'XY!@3')

In [2]: type(var1)
Out[2]: tuple

In [3]: var2 = list(var1)

In [4]: var2
Out[4]: [523, 'abc', 3.141592, (2, 3), 'XY!@3']

In [5]: type(var2)
Out[5]: list

In [6]: var3 = tuple(var2)

In [7]: var3
Out[7]: (523, 'abc', 3.141592, (2, 3), 'XY!@3')

In [8]: type(var3)
Out[8]: tuple
```

1.5.5 Dictionary

Python dictionaries are like a hash table,They store a mapping between key's and value's. They consist of key value pairs, Where the key can be any of python's immutable types, although they are usually numbers or strings. Values on the other hand can be any python object. Dictionaries are enclosed within curly braces({ }) and values can be assigned and accessed using square braces([ ]). A single dictionary can store values of different types.

Dictionaries have many operations that can be performed on them. It's possible to define, view, look up and delete the key value pairs in a dictionary object.

Declaration :

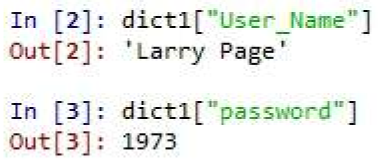
Dictionaries are enclosed in curly braces({ }), each key value pair is separated by commas(,) and a colon(:) between the key and value of each key and value pair.

The keys can be any immutable type in in python and values can be any standard or user defined type in python. Dictionary variables are declared when a dictionary object reference is assigned to a name as in the example below.

```
In [1]: dict1 = {'User_Name':'Larry Page', 'password':1973}
```

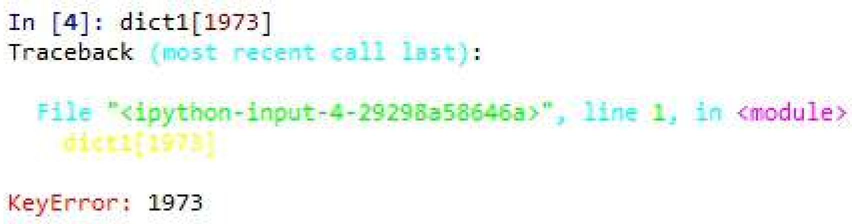
Accessing Values :

Dictionary values can be accessed using the keys enclosed in square braces like how they are accessed in tuples and lists except in the case of dictionaries the subscript is the key.Some examples are shown below.



```
In [2]: dict1["User_Name"]
Out[2]: 'Larry Page'

In [3]: dict1["password"]
Out[3]: 1973
```

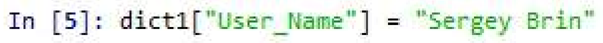
Python doesn’t support look ups with the values if a key that doesn’t exist in th e dictionary is looked up a “key error” is raised.



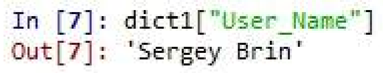
```
In [4]: dict1[1973]
Traceback (most recent call last):

  File "<ipython-input-4-29298a58646a>", line 1, in <module>
    dict1[1973]

KeyError: 1973
```

Changing values :

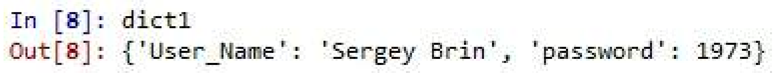
Dictionaries are mutable types so the values of the dictionary objects are changed in place in memory. A value corresponding to a dictionary key can be changed as follows



```
In [5]: dict1["User_Name"] = "Sergey Brin"
```



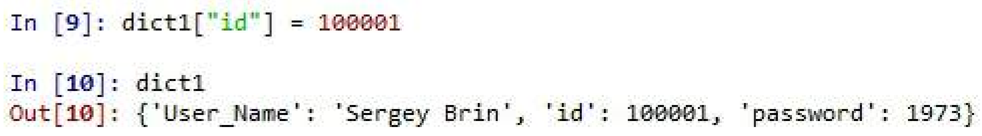
```
In [7]: dict1["User_Name"]
Out[7]: 'Sergey Brin'
```



```
In [8]: dict1
Out[8]: {'User_Name': 'Sergey Brin', 'password': 1973}
```

Inserting a new key-value pair :

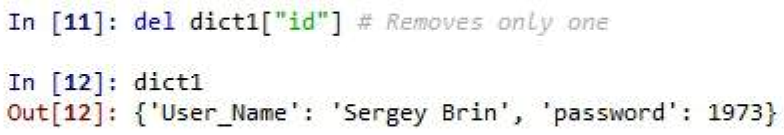
A new key-value pair can be inserted into the dictionary by assigning a value to a dictionary key that doesn't exist yet. An example is shown below.



```
In [9]: dict1["id"] = 100001

In [10]: dict1
Out[10]: {'User_Name': 'Sergey Brin', 'id': 100001, 'password': 1973}
```

`del` command :

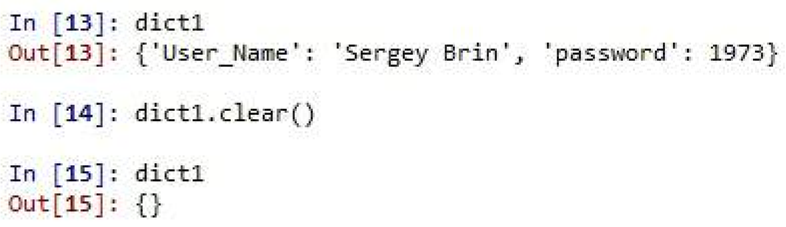
The `del` command is used to remove a single key value pair from the dictionary object, only the specified key value pair is deleted the rest of the dictionary is left intact. An example of using the `del` command is shown below.



```
In [11]: del dict1["id"] # Removes only one

In [12]: dict1
Out[12]: {'User_Name': 'Sergey Brin', 'password': 1973}
```

clear( ) :

The dictionary class has a "clear" method which when called for a dictionary object deletes all the key-value pairs in that dictionary object. It is the same as applying the del command to all keys in the dictionary

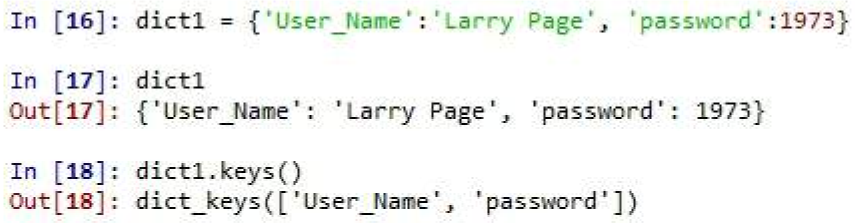


```
In [13]: dict1
Out[13]: {'User_Name': 'Sergey Brin', 'password': 1973}

In [14]: dict1.clear()

In [15]: dict1
Out[15]: {}
```

Let's discuss some of the other methods that the python dictionary class offers.

keys( ) :

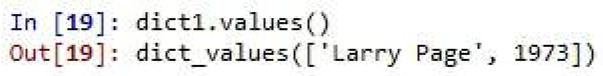
The “keys” method of python’s dictionary class when called for a method returns a list containing all the keys of that dictionary object. An example is shown below.



```
In [16]: dict1 = {'User_Name':'Larry Page', 'password':1973}

In [17]: dict1
Out[17]: {'User_Name': 'Larry Page', 'password': 1973}

In [18]: dict1.keys()
Out[18]: dict_keys(['User_Name', 'password'])
```

values( ) :

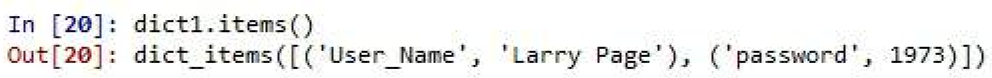
The “values” method of the dictionary class when called for a dictionary object returns a list containing all the values in the dictionary.



```
In [19]: dict1.values()
Out[19]: dict_values(['Larry Page', 1973])
```

items( ) :

Python’s dictionary method has a items class which when called on an object returns a list of all the key value pairs as tuples. An example is shown below.



```
In [20]: dict1.items()
Out[20]: dict_items([('User_Name', 'Larry Page'), ('password', 1973)])
```

1.6 Data type conversion

There may come situations where it’s need to convert between python’s built in data type, to convert between built in data types the type nae can be used as a function.

There are several built in functions to perform a conversion from one type to another type, these functions return a fresh object in memory with the converted value.

Function	Description
`int(x [,base])`	Converts x to an integer. base specifies the base if x is a string.
`long(x [,base] )`	Converts x to a long integer. base specifies the base if x is a string.
`float(x)`	Converts x to a floating-point number.
`complex(real [,imag])`	Creates a complex number.
`str(x)`	Converts object x to a string representation.
`repr(x)`	Converts object x to an expression string.
`eval(str)`	Evaluates a string and returns an object.
`tuple(s)`	Converts s to a tuple.
`list(s)`	Converts s to a list.
`set(s)`	Converts s to a set.
`dict(d)`	Creates a dictionary. d must be a sequence of (key,value) tuples.
`frozenset(s)`	Converts s to a frozen set.
`chr(x)`	Converts an integer to a character.
`unichr(x)`	Converts an integer to a Unicode character.
`ord(x)`	Converts a single character to its integer value.
`hex(x)`	Converts an integer to a hexadecimal string.
`oct(x)`	Converts an integer to an octal string.

In the next chapter we'll look into the basic python operator's.

1.7 Exercises

1. Will the following lines of code print the same thing? Explain why or why not.

```
x = 6
print(6)
print("6")
```

2. Will the following lines of code print the same thing? Explain why or why not.

```
x = 7
print(x)
print("x")
```

3. What is the largest floating-point value available on your system?

4. What is the smallest floating-point value available on your system?

5. What happens if you attempt to use a variable within a program, and that variable has not been assigned a value?

6. What is wrong with the following statement that attempts to assign the value ten to variable x?

```
10 = x
```

2. Once a variable has been properly assigned can its value be changed?

3. In Python can you assign more than one variable in a single statement?

9. Classify each of the following as either a legal or illegal Python identifier:

(a) fred

(b) if

(c) 2x

(d) -4

(e) sum_total

(f) sumTotal

(g) sum-total

(h) sum total

(i) sumtotal

(j) While

(k) x2

(l) Private

(m) public

(n) $16

(o) xTwo

(p) _static

(q) _4

(r) ___

(s) 10%

(t) a27834

(u) wilma's

10. What can you do if a variable name you would like to use is the same as a reserved word?

11. How is the value 2.45×10−5 expressed as a Python literal?

12. How is the value 0.0000000000000000000000000449 expressed as a Python literal?

13. How is the value 5699234120000000000000000000000000000 expressed as a Python literal?

14. Can a Python programmer do anything to ensure that a variable’s value can never be changed after its initial assignment?

15. Is "i" a string literal or variable?

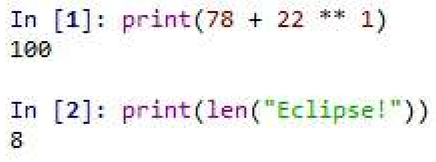
2. Statements, Expressions and operators

In this chapter we'll get understand the what a statement, expression and an operator in python is. We'll discuss the different basic operators in python and the operator precedence in python. Let's get started.

2.1 Statements and Expressions

Statements are instructions that the python interpreter could execute. So far only the assignment, del and method invocation statement has been discussed but there are other statements like "for" statements, "if" statements, "while" statements, "import" statements and others.

Expressions, they are a combination of variables, values, function calls and operators. Expressions in python need to be evaluated before they can be used. When an expression is the right hand side of an assignment statement the expression is first evaluated before it is assigned to name. When an expression is passed as the parameter to pythons "print" function the expressions is first evaluated then the result is displayed. In the below examples the expressions are first evaluated then are used.



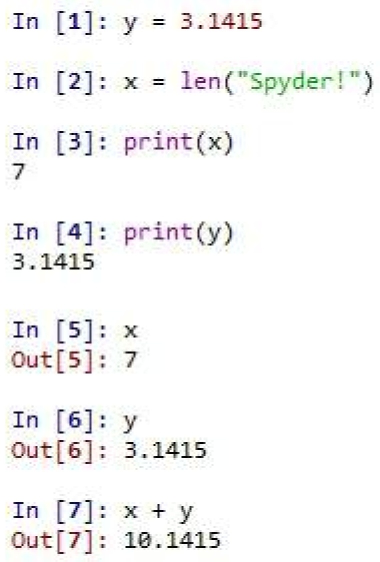
```
In [1]: print(78 + 22 ** 1)
100

In [2]: print(len("Eclipse!"))
8
```

Here len( ) is a built in python function that returns the length of the string passed as an argument. In the above code "78 + 2 ** 1" and "len("Eclipse!")" are both expressions they are evaluated before they are used by the print function.

Evaluating an expression results in a value, and that's the reason why expressions can be on the right hand side of an assignment statement. A value or variable or a function call that return a value are all simple expressions. Evaluating a variable gives the value the variable refers to. It's important to remember that expressions in python returns a value. Take a look at the code below.

```python
y = 3.1415
x = len("Spyder!")
print(x)
print(y)
```

We can see one of the differences between expressions and assignments when the above code is run in the python shell.



```
In [1]: y = 3.1415

In [2]: x = len("Spyder!")

In [3]: print(x)
7

In [4]: print(y)
3.1415

In [5]: x
Out[5]: 7

In [6]: y
Out[6]: 3.1415

In [7]: x + y
Out[7]: 10.1415
```

We can see that for assignments only the prompt is returned and no value. That's because assignments statements do not return any values. The assignment statement "`y = 3.1415`" doesn't return any value so only the prompt is returned, but the result of the simple expression "`3.1415`" is returns a value(i.e `3.1415`) which creates a new float object in memory and assigns it's reference to '`y`'. Assignment statements are simply executed and don't return any value.

When the print statement is called on '`y`' we can see the value that '`y`' is referring to as print function returns a value, also when '`y`' is entered into the shell by itself '`y`' is evaluated and returned.

2.2 What are operators and operands

Operands the tokens in python represent some kind of computation and the computation happens between the operand or operands that are given to the operator to work on.

The following are some legal expressions using arithmetic operator's.

```python
# Assignments
x = 10
y = 20
z = 30

# Valid expressions using arithmetic operators
25 + 22
x - 1
y * 60 + z
x / 60
5 ** 2
(5 + 13) * (55 + 77)
```

The operators that python supports are listed below.

- Arithmetic operators
- Relational operators(comparison operators)
- Assignment operators
- Logical operators
- Bitwise operators
- Membership operators
- Identity operators

We'll go through each of these operators one by one.

2.3 Arithmetic operators

The seven arithmetic operators supported by python are listed and described below briefly. We'll then go through some examples on these arithmetic operators.

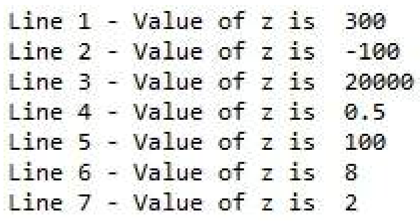
Operator	Description
+	Addition - Adds values on either side of the operator
_	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand and returns remainder
**	Exponent - Performs exponential (power) calculation on operators
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed.

Now lets look at some example code that demonstrates how the arithmetic operators work in python.

```
#=======================================
# Assignments
#=======================================
x = 100
y = 200
z = 500

#=======================================
# Using arithmetic operators
#=======================================
z = x + y
print("Line 1 - Value of z is ", z)
z = x - y
```

```python
print("Line 2 - Value of z is ", z)
z = x * y
print("Line 3 - Value of z is ", z)
z = x / y
print("Line 4 - Value of z is ", z)
z = x % y
print("Line 5 - Value of z is ", z)
x = 2
y = 3
z = x ** y
print("Line 6 - Value of z is ", z)
x = 10
y = 5
z = x//y
print("Line 7 - Value of z is ", z)
```

The above code gives the output given below.



```
Line 1 - Value of z is  300
Line 2 - Value of z is  -100
Line 3 - Value of z is  20000
Line 4 - Value of z is  0.5
Line 5 - Value of z is  100
Line 6 - Value of z is  8
Line 7 - Value of z is  2
```

2.4 Relational operators

The six relational operators that python supports are listed and described in the table below.

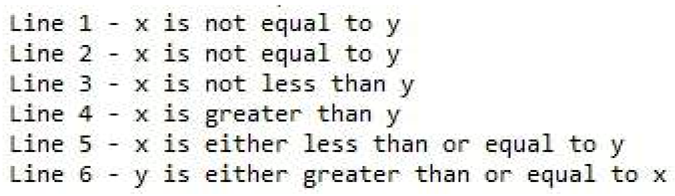
Operators	Description
==	Checks if the value of two operands is equal or not, if yes then condition becomes true.
!=	Checks if the value of two operands is equal or not, if values are not equal then condition becomes true.
>	Checks if the value of left operand is greater than the value of

	right operand, if yes then condition becomes true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.

The example code below demonstrates the use of pythons relational operators. As relational operators give a Boolean result of either true or false, if-else statements are used to demonstrate the the behaviour of relational operators in python .

```python
#======================================
# Assignments
#======================================
x = 10
y = 5
z = 0

#======================================
# Using relational operators
#======================================
if( x == y ):
    print("Line 1 - x is equal to y")
else:
    print("Line 1 - x is not equal to y")
if( x != y ):
    print("Line 2 - x is not equal to y")
else:
    print("Line 2 - x is equal to y")
if( x < y ):
    print("Line 3 - x is less than y")
else:
    print("Line 3 - x is not less than y")
if( x > y ):
    print("Line 4 - x is greater than y")
else:
    print("Line 4 - x is not greater than y")

x = 5
y = 20

if( x <= y ):
    print("Line 5 - x is either less than or equal to y")
else:
    print("Line 5 - x is neither less than nor equal to y")
if( y >= x ):
    print("Line 6 - y is either greater than or equal to x")
else:
    print("Line 6 - y is neither greater than nor equal to x")
```

The above python code is ran it gives the output shown below.



```
Line 1 - x is not equal to y
Line 2 - x is not equal to y
Line 3 - x is not less than y
Line 4 - x is greater than y
Line 5 - x is either less than or equal to y
Line 6 - y is either greater than or equal to x
```

2.5 Assignment operators

Python's most basic assignment operator is '=' which assigns a reference to the value on the right hand of the operator to the variable name on the left hand side of the operator. Python also has several other assignment operators that perform an arithmetic operation before the assignment is made. The nine different assignment operators in python are listed and described below.

Operator	Description
=	Simple assignment operator, Assigns values from right side

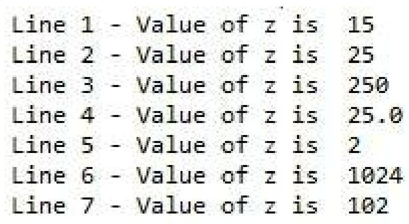
	operands to left side operand
+=	Add AND assignment operator, It adds right operand to the left operand and assigns the result to left operand
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assigns the result to left operand
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assigns the result to left operand
/=	Divide AND assignment operator, It divides left operand with the right operand and assigns the result to left operand
%=	Modulus AND assignment operator, It takes modulus using two operands and assigns the result to left operand
**=	Exponent AND assignment operator, Performs exponential (power) calculation on operators and assigns value to the left operand
//=	Floor Division and assigns a value, Performs floor division on operators and assigns value to the left operand

Example code is is give below that explains the working of the different assignment operators in python.

```
#=======================================
# Assignments
#=======================================
x = 10
y = 5
z = 0

#=======================================
# Using assignment operators
#=======================================
z = x + y
print("Line 1 - Value of z is ", z)
z += x
print("Line 2 - Value of z is ", z)
z *= x
print("Line 3 - Value of z is ", z)
```

```
z /= x
print("Line 4 - Value of z is ", z)
z = 2
z %= x
print("Line 5 - Value of z is ", z)
z **= x
print("Line 6 - Value of z is ", z)
z //= x
print("Line 7 - Value of z is ", z)
```

The above code gives the below output when executed.



```
Line 1 - Value of z is  15
Line 2 - Value of z is  25
Line 3 - Value of z is  250
Line 4 - Value of z is  25.0
Line 5 - Value of z is  2
Line 6 - Value of z is  1024
Line 7 - Value of z is  102
```

2.6 Bitwise operators

Python's bitwise operators are applied to the binary form of the values that python's numeric data types can hold.

Bitwise operators are performed bit by bit. For example when a bitwise operation is performed on "`x = 10`" and "`y = 20`", the binary equivalent of of the integer objects `10` and `20` are used instead of the decimal values(`10, 20`) that were assigned to '`x`' and '`y`', so the operands to the bitwise operation are the binary values "`1010`"(`10`) and "`10100`"(`20`). Operation is performed bit by bit.

```
X = 63 (base - 10) = 0011 1100 (base - 2)

Y = 13 (base - 10) = 0000 1101 (base - 2)

X & y = 0000 1100

X & y = 0011 1101

X & y = 0011 0001
```

The six bitwise operators that python supports are listed and briefly described in the below table.

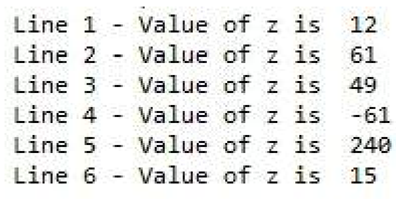
Operator	Description
&	Binary AND Operator copies a bit to the result if it exists in both operands.
|	Binary OR Operator copies a bit if it exists in either operand.
^	Binary XOR Operator copies the bit if it is set in one operand but not both.
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.

The following example code makes use of each of bitwise operations that python supports.

```
#======================================
# Assignments
#======================================

#-----In Decimal------In Binary--------
#--------------------------------------
x =      60        # 0011 1100
y =      13        # 0000 1101
z =       0        # 0000 0000

#======================================
# Using bitwise operators
#======================================

z = x & y     # 12 = 0000 1100
print("Line 1 - value of z is ", z)

z = x | y     # 61 = 0011 1101
print("Line 2 - value of z is ", z)
```

```python

z = x ^ y     # 49 = 0011 0001
print("Line 3 - value of z is ", z)

z = ~x        #-61 = 1100 0011
print("Line 4 - value of z is ", z)

z = x << 2    #240 = 1111 0000
print("Line 5 - value of z is ", z)

z = x >> 2    # 15 = 0000 1111
print("Line 6 - value of z is ", z)
```

When the above python script is executed it displays the following output.



```
Line 1 - Value of z is  12
Line 2 - Value of z is  61
Line 3 - Value of z is  49
Line 4 - Value of z is  -61
Line 5 - Value of z is  240
Line 6 - Value of z is  15
```

2.7 Logical operators

The logical operators and, or and not are supported by python these logical operators are described below. Unlike languages like c, logical operators are written in plain in English.

Operators	Description
`and`	Called logical AND operator. If both the operands are true, then the condition becomes true.
`or`	Called logical OR Operator. If any of the two operands are non zero, then the condition becomes true.
`not`	Called logical NOT Operator. Used to reverse the logical state of its operand. If a condition is true, then Logical NOT

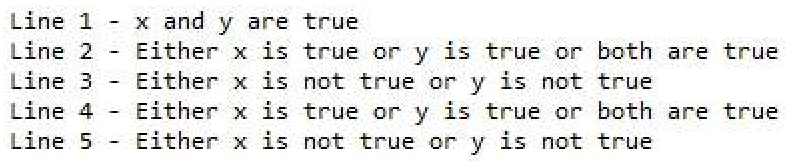
	operator will make it false.

Any non zero numerical value is considered true and the numerical value zero(0) is considered false. The example coed below works with python's logical operators.

```python
#====================================================================
# Assignments
#====================================================================
x = 5
y = 10

#====================================================================
# Using logical operators
#====================================================================
if( x and y ):
    print("Line 1 - x and y are true")
else:
    print("Line 1 - Either x is not true or y is not true")

if( x or y ):
    print("Line 2 - Either x is true or y is true or both are true")
else:
    print("Line 2 - Neither x is true nor y is true")


x = 0

if( x and y ):
    print("Line 3 - x and y are true")
else:
    print("Line 3 - Either x is not true or y is not true")

if( x or y ):
    print("Line 4 - Either x is true or y is true or both are true")
else:
    print("Line 4 - Neither x is true nor y is true")
```

```python

if not( x and y ):
    print("Line 5 - Either x is not true or y is not true")
else:
    print("Line 5 - x and y are true")
```

The output when the above code is executed is provided below.



```
Line 1 - x and y are true
Line 2 - Either x is true or y is true or both are true
Line 3 - Either x is not true or y is not true
Line 4 - Either x is true or y is true or both are true
Line 5 - Either x is not true or y is not true
```

2.8 Membership operators

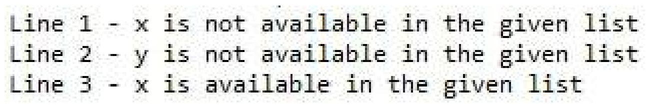
Python’s two membership operators enable testing whether a given element exists in a sequence(lists, tuples or strings). The two membership operators that python support are listed and described in the table below.

Operators	Description
`in`	Evaluates to true if it finds a variable in the specified sequence and false otherwise.
`not in`	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

The example code below may give a better understanding on what the python membership operators do.

```python
#======================================================================
# Assignments
#======================================================================
x = 6
y = 7
li = [1, 2, 3, 4, 5 ];

#======================================================================
# Using membership operators
#===================================================================
if ( x in li ):
    print("Line 1 - x is available in the given list")
else:
    print("Line 1 - x is not available in the given list")

if ( y not in li ):
    print("Line 2 - y is not available in the given list")
else:
    print("Line 2 - y is available in the given list")

x = 4

if ( x in li ):
    print("Line 3 - x is available in the given list")
else:
    print("Line 3 - x is not available in the given list")
```

The above code gives the below output.



```
Line 1 - x is not available in the given list
Line 2 - y is not available in the given list
Line 3 - x is available in the given list
```

2.9 Identity operators

Python's identity operators can be used to check whether two names points to/references the same object in memory. There are two identity operators supported by python, they are described in the following table.

Operators	Description
`is`	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.
`is not`	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

The code below demonstrates the use of identity operators in python. When you try this out in your python shell you can see how the python interpreter assigns objects to variables.

The `id( )` method takes a variable as parameter and returns the identity of the object that name references. An identity of an object is constant and is retained by the object as long as it is alive. The output of the of the above code is given below.

```python
#================================================================
# Assignments
#================================================================
x = 10
y = 10

#================================================================
# Using identity operators
#================================================================
if ( x is y ):
    print("Line 1 - x and y have same identity")
else:
    print("Line 1 - x and y do not have same identity")
if ( id(x) == id(y) ):
    print("Line 2 - x and y have same identity")
else:
    print("Line 2 - x and y do not have same identity")

x = 30

if ( x is y ):
    print("Line 3 - x and y have same identity")
else:
    print("Line 3 - x and y do not have same identity")
if ( x is not y ):
    print("Line 4 - x and y do not have same identity")

```

```
else:
    print("Line 4 - x and y have same identity")
```

```
Line 1 - x and y have same identity
Line 2 - x and y have same identity
Line 3 - x and y do not have same identity
Line 4 - x and y do not have same identity
```

2.10 Precedence and associativity of python operators

It's already been discussed that an expression in python and most other programming languages are a combination of values, variables, function calls and operators. Lets take some instances of simple expressions in python.

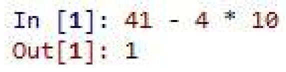
```
# Assignments
var1 = 10
var2 = 20
var3 = "Ipython"

# Expressions with a single or no operator
var1
var2
var1 + var2
var1 % 3
len(var3) ** var1
1 + 4.67
```

Remember a valid expression always returns a value and an expression can not be used until its evaluated. In the above code every expression has either one or no operator so evaluation is very straight forward but in the cases where multiple operators are involved in the expression what is the order of evaluation to be followed? Valid expressions of this case are given in the code below.

```
# Assignments
var1 = 10
var2 = 20
var3 = "Ipython"

```

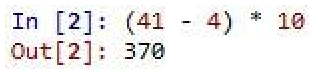
```python
# Expressions with multiple operators
var1 + var2 ** (2 % 7) // var1
var1 % 3 ** (var1 + var2)
len(var3) ** var1 % 100000
1 + 4.67 * 5 + (len(var3) * id(var2))
```

Python specifies the order of precedence of operators and associativity(in the case that there are more than one operator with the same precedence level). For example example multiplication has a higher order than subtraction.



```
In [1]: 41 - 4 * 10
Out[1]: 1
```

But we can change this order of evaluation by using parentheses(( )). parentheses have a higher precedence level.



```
In [2]: (41 - 4) * 10
Out[2]: 370
```

Python operator precedence rule (from highest precedence to lowest)

Operator	Meaning
`()`	Parentheses
`**`	Exponent
`+x, -x, ~x`	Unary plus, Unary minus, Bitwise NOT
`*, /, %, //`	Multiplication, Division, Floor division, Modulus
`+, -`	Addition, Subtraction
`<<, >>`	Bitwise shift operators
`&`	Bitwise AND
`^`	Bitwise XOR
`|`	Bitwise OR
`==, !=, >, >=, <, <=, in, not in, is, is not,`	Comparisons, Identity, Membership operators
`not`	Logical NOT
`and`	Logical AND

`or`	Logical OR

Python operator associativity

In the above table there are certain precedence levels that have multiple operators, this means that all those operators have the same level of precedence. When there are multiple operators of the same precedence in an expression the appropriate associativity rule is followed. For almost all of pythons operators the associativity rule is left to right. Take a look at the example below.

```python
# Left to right associativity
# Output: 3
print(5 * 2 // 3)         # *, // have equal precedence
                         # so follow it's associativity rule

# Shows left to right associativity
# Output: 0
print(5 * (12 // 3) % 2)   # parentheses has the highest precedence
                         # *, //, % have equal precedence
                         # so follow it's associativity rule
```

The exponent operator(**) has right to left associativity in python, so if there are multiple exponent operators they are evaluated from right to left after operators of higher precedence have already been evaluated.

```python
#===============================================================
# Right to left associativity of the ** operator
#===============================================================

# Output: 512
print(2 ** 3 ** 2)

# Output: 64
print((2 ** 3) ** 2) # Parentheses has the highest precedence
```

Non associative operators(Assignment and comparison operators)

There are some python operators that do not follow any associativity rule, the assignment and comparison operators. There are some special rules followed when multiple operators of this type are used in a sequence.

Comparison operators:

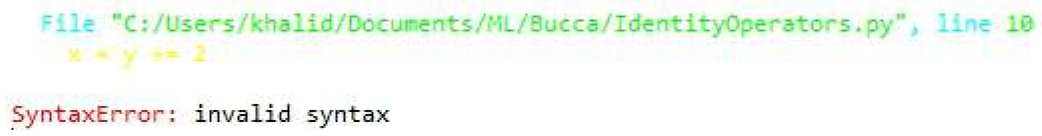
Take a conditional expression "`x < y < z`" this expression is not equivalent to "`(X < y) < z`" or "`X < (y < z)`", but instead is equivalent to "`X < y and y < z`" and this is evaluated from left to right.

Assignment operators:

Assignments of the form "`x = y = z`" are valid and are evaluated from right to left, but assignments of the form "`x = y += z`" are not considered to be a valid syntax.

```
# Initializing x, y, z
x = 1
y = 2
z = 3
# The below assignment
# expression is invalid
# (Non-associative operators)
# SyntaxError: invalid syntax

x = y += 2
```

So when the python interpreter encounters such syntax it raises a syntax error.



```
  File "C:/Users/khalid/Documents/ML/Bucca/IdentityOperators.py", line 10
    x = y += 2

SyntaxError: invalid syntax
```

2.11 Exercises

1. Is the literal 4 a valid Python expression?

2. Is the variable x a valid Python expression?

3. Is x + 4 a valid Python expression?

4. What affect does the unary + operator have when applied to a numeric expression?

5. Sort the following binary operators in order of high to low precedence: +, -, *, //, /, %, =.

6. Given the following assignment:

   `x = 2`

   Indicate what each of the following Python statements would print.

   (a) `print("x")`

   (b) `print(’x’)`

   (c) `print(x)`

   (d) `print("x + 1")`

   (e) `print(’x’ + 1)`

   (f) `print(x + 1)`

7. Given the following assignments:

   ```
   i1 = 2
   i2 = 5
   i3 = -3
   d1 = 2.0
   d2 = 5.0
   d3 = -0.5;
   ```

   Evaluate each of the following Python expressions.

   (a) i1 + i2

   (b) i1 / i2

   (c) i1 // i2

   (d) i2 / i1

   (e) i2 // i1

   (f) i1 * i3

(g) d1 + d2

(h) d1 / d2

(i) d2 / d1

(j) d3 * d1

(k) d1 + i2

(l) i1 / d2

(m) d2 / i1

(n) i2 / d1

(o) i1/i2*d1

(p) d1*i1/i2

(q) d1/d2*i1

(r) i1*d1/d2

(s) i2/i1*d1

(t) d1*i2/i1

(u) d2/d1*i1

(v) i1*d2/d1

8. What is printed by the following statement:

   #print(5/3)

9. Given the following assignments:

```
i1 = 2
i2 = 5
i3 = -3
d1 = 2.0
d2 = 5.0
d3 = -0.5
```

   Evaluate each of the following Python expressions.

   (a) i1 + (i2 * i3)

   (b) i1 * (i2 + i3)

(c) i1 / (i2 + i3)

(d) i1 // (i2 + i3)

(e) i1 / i2 + i3

(f) i1 // i2 + i3

(g) 3 + 4 + 5 / 3

(h) 3 + 4 + 5 // 3

(i) (3 + 4 + 5) / 3

(j) (3 + 4 + 5) // 3

(k) d1 + (d2 * d3)

(l) d1 + d2 * d3

(m) d1 / d2 - d3

(n) d1 / (d2 - d3)

(o) d1 + d2 + d3 / 3

(p) (d1 + d2 + d3) / 3

(q) d1 + d2 + (d3 / 3)

(r) 3 * (d1 + d2) * (d1 - d3)

10. What symbol signifies the beginning of a comment in Python?

11. How do Python comments end?

12. Which is better, too many comments or too few comments?

13. What is the purpose of comments?

14. Why is human readability such an important consideration?

15. Consider the following program which contains some errors. You may assume that the comments within the program accurately describe the program's intended behavior.

```
# Get two numbers from the user
n1, n2 = eval(input()) # 1
# Compute sum of the two numbers
print(n1 + n2) # 2
# Compute average of the two numbers
print(n1+n2/2) # 3
# Assign some variables
d1 = d2 = 0 # 4
# Compute a quotient
print(n1/d1) # 5
# Compute a product
n1*n2 = d1 # 6
# Print result
print(d1) # 7
```

For each line listed in the comments, indicate whether or not an interpreter error,run-time exception, or logic error is present. Not all lines contain an error.

16. Write the shortest way to express each of the following statements.

(a) x = x + 1

(b) x = x / 2

(c) x = x - 1

(d) x = x + y

(e) x = x - (y + 7)

(f) x = 2*x

(g) number_of_closed_cases = number_of_closed_cases + 2*ncc

17. What is printed by the following code fragment?

```
x1 = 2
x2 = 2
x1 += 1
x2 -= 1
print(x1)
print(x2)
```

Why does the output appear as it does?

18. Consider the following program that attempts to compute the circumference of a circle given the radius entered by the user. Given a circle's radius, r, the circle's circumference, C is given by the formula:

C = 2pr

```
r = 0
PI = 3.14159
# Formula for the area of a circle given its radius
C = 2*PI*r
# Get the radius from the user
r = eval(input("Please enter the circle's radius: ")
# Print the circumference
print("Circumference is", C)
```

(a) The program does not produce the intended result. Why?

(b) How can it be repaired so that it works correctly?

19. Write a Python program that ...

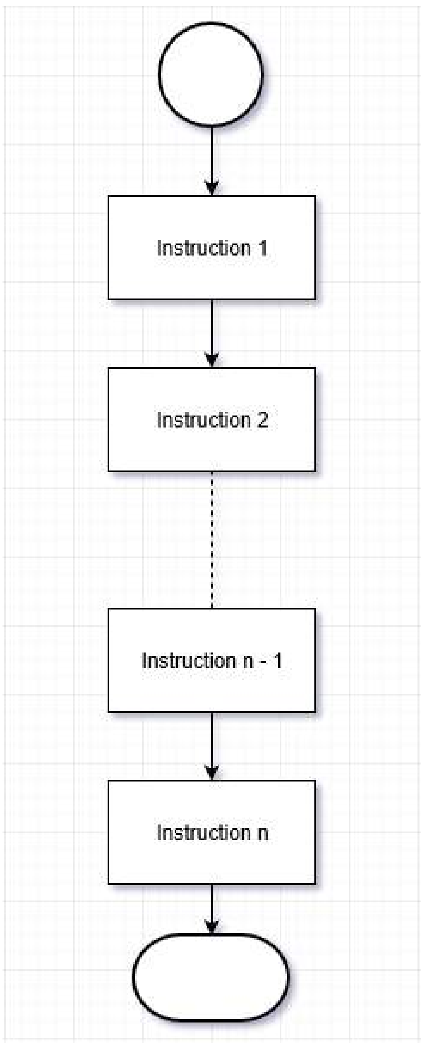
20. Write a Python program that ...

3. Pythons flow control tools

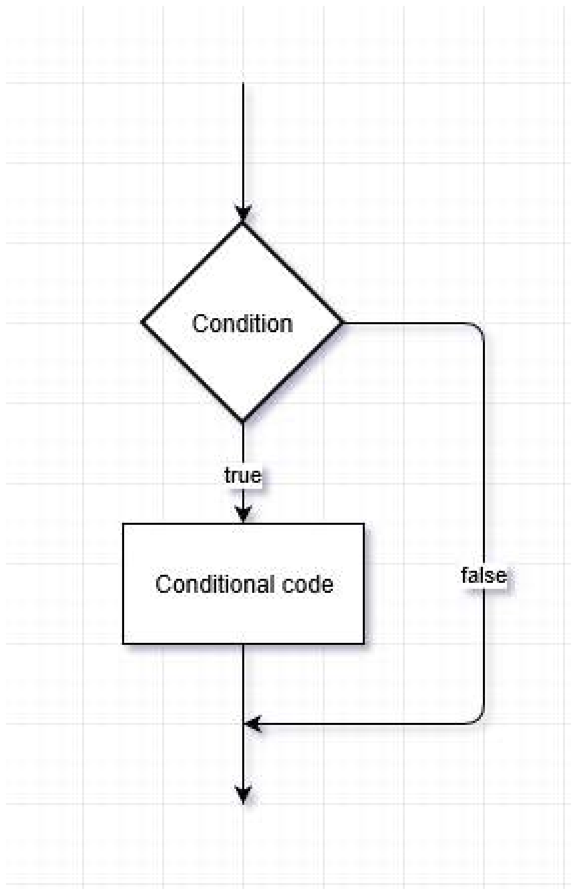
Most the code examples that were discussed prior to this chapter followed a sequential execution of statements, from start to finish(from line 1 to that last line of code) except some examples that had to involve the “if” and “if-else”for the purpose of explanation of the content being discussed. We’ll now go through each of the tools that that python offers to change the normal sequential flow of control that python programs follow.

These control flow statements can also be referred to as “branching statements”. There are two types of branching statements in python “conditional branching” and “unconditional branching”. conditional branching statements change the control flow of programs based on the result of conditional expression, unconditional branching statements change the control flow of the program when they are encountered and regardless of any condition. Unconditional branching instructions are almost always used along with conditional branching statements.

The sequential control of a python program is shown below in the flow diagram.



The following flowchart portrays the general decision making structure in almost all programming languages.



In python the numerical value “`0`” and “`null`” are considered to be “`false`” any value that is not “`0`” or “`null`” are considered to be “`true`”. The types of decision making statements in python are listed and briefly described below, we’ll go into details in the coming sections.

Statement	Description
if statements	An if statement consists of a conditional expression followed by one or more statements.
if-else statements	An if statement can be followed by an optional else statement, which executes when the conditional expression is false.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).

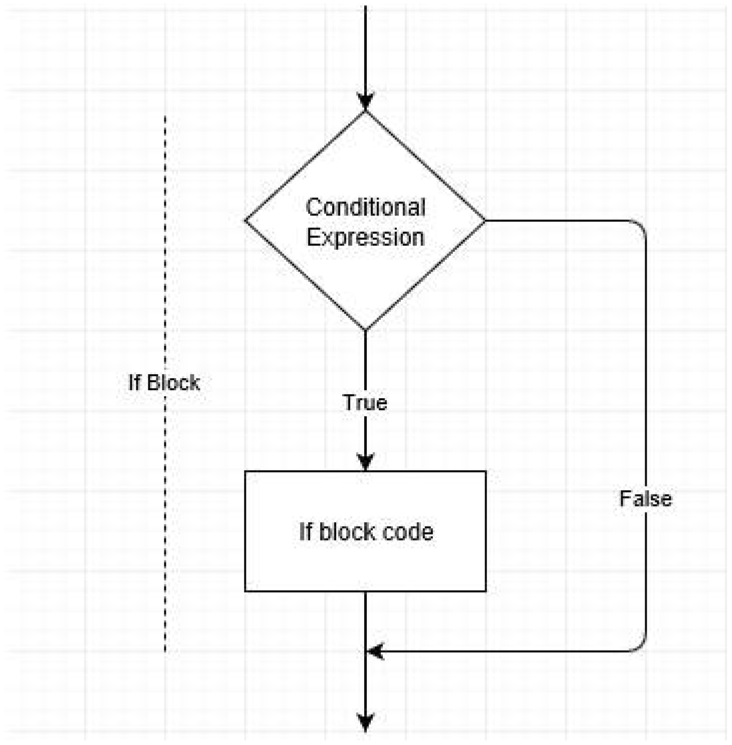
Decision making is needed in cased where a block of code should be executed only when certain condition is true, for these kind of cases that require decision making we can use “`if..elif..else`” statements.

3.1 if statements

The “if statement” has a conditional expression, when the result of the conditional expression is true the body of the “if statement” is executed, in the case that it is false body of the “if statement” is skipped. The general syntax of the if statement is given below.

```
if <Boolean Expression> :
      <if block Statement(s)>
```

In python the body of the “if statement” is specified by indentation. The body starts with an indentation and the first line without the indentation marks the end and is outside the body of the “if block”. The flowchart of pythons “if statement” is shown below.



The below example code demonstrates how the “if statement” works in python.

```
#=================================================================
# Demonstrating the if statement.
# If the number is positive, then print the appropriate message.
#=================================================================
num = 3
if num > 0:                                # if statements condition
    print(num, "is a positive number.")  # indentation marks the if block
    print("still in if block")            # indentation marks the if block
print("outside the if block.")           # outside the if block

num = -1

if num > 0:                                # if statements condition
    print(num, "is a positive number.")  # indentation marks the if block
print("outside the if block.")           # outside the if block

```

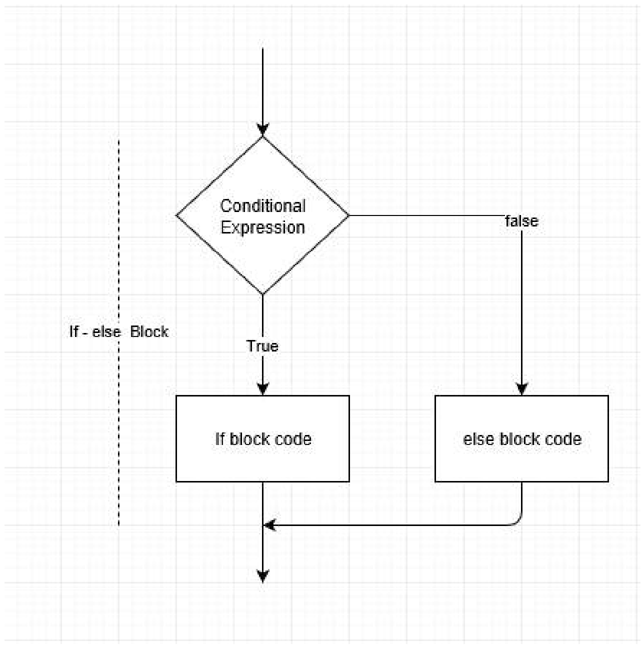
The body of the "if statement" is executed when the condition "`num > 0`" is evaluated to be true. If the condition is evaluated to be "`true`" the body of the "if statement"(marked by the indentation) gets executed, else it is skipped. In the above example code the condition of the first "if statement" is true so it's body is executed and the condition of the second "if statement" is false so it's body is skipped. The output when the above program is executed is given below.

```
3 is a positive number.
still in if block
outside the if block.
outside the if block.
```

3.2 if-else statements

The "if-else statement" has a conditional expression, when the result of the conditional expression is true the body of the "if" is executed, in the case that it is false body of the "else" is executed. The general syntax of the if-else statement is given below.

```
if <Boolean Expression>:

        <if block Statement(s)>

else:
    <else block Statement(s)>
```

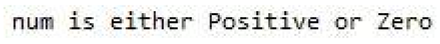


In the given flow chart of the flow chart of the "if-else statement" we can see that when the Boolean expression is evaluated to be false the the body of the else block is executed instead.

The example code below shows an instance of using the "if-else statement", take a look for more clarity.

```python
#================================================================
# Demonstrating the if..elif..else Decision chain.
# Print the appropriate message for the different values of num
#================================================================

num = 3

# try these two variations your self
# num = -12
# num = 0

if num >= 0:
    print("num is either Positive or Zero")
else:
    print("num is a Negative number")
```

The output of the above program is given below, and try the above two variations program yourself.



```
num is either Positive or Zero
```

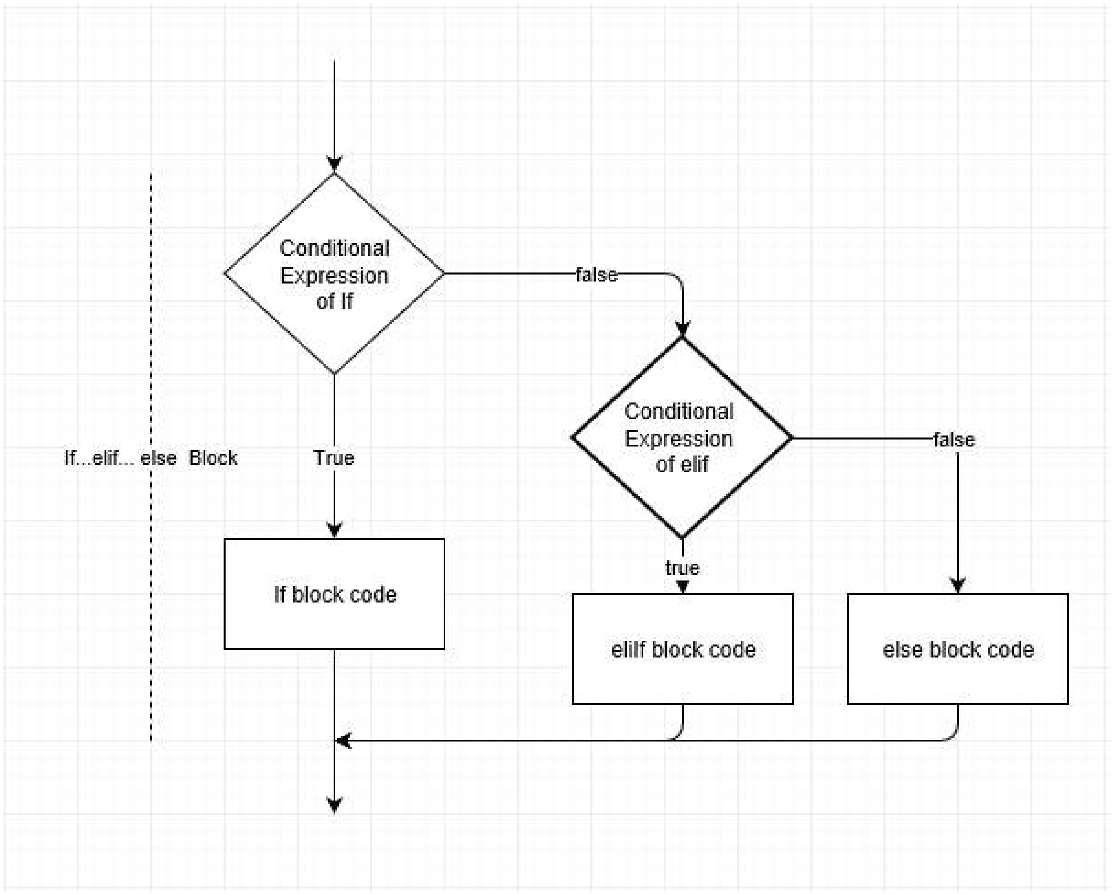
In the above program the condition of the "if statement"(`num > 0`) is true so the "if block" is executed and the "else" block code is skipped. If the value of `num` is changed to "0" the condition is still evaluated to be true so again the "if" block is executed instead. If the value of `num` is changed to "-12" the "else" block is executed instead.

3.3 if..elif..else statements

The general syntax of "if..elif..else" is given below. First the Boolean expression of "if" is evaluated, if it is evaluated to be false then the Boolean expression of elif is evaluated, if it is evaluated to be true then the body of that "elif" is executed if it is false then it's body is skipped and control goes to the next "elif" or "else". It is allowed to have more than one "elif" in the "if..elif..else" chain.

```
if <Boolean Expression>:

       <if block Statement(s)>

elif <Boolean Expression>:

       <elif block Statement(s)>

else:

       <else block Statement(s)>
```

The flowchart of the if..elif..else chain is shown below. Remember there can be more than one "elif" in the "if..elif..else" chain.



Lets take a look at a code example to better understand how the if..elif..else decision chain works in python.

```python
#================================================================
# Demonstrating the if..elif..else decision chain.
# Print the appropriate message for the different values of num
#================================================================

num = 34

# Try these two variations yourself
# num = 0
# num = -205

if num >= 0:
    print("You've entered a positive number!")
elif num == 0:
    print("You've entered zero!")
```

```
else:
    print("You've entered a negative number!")
```

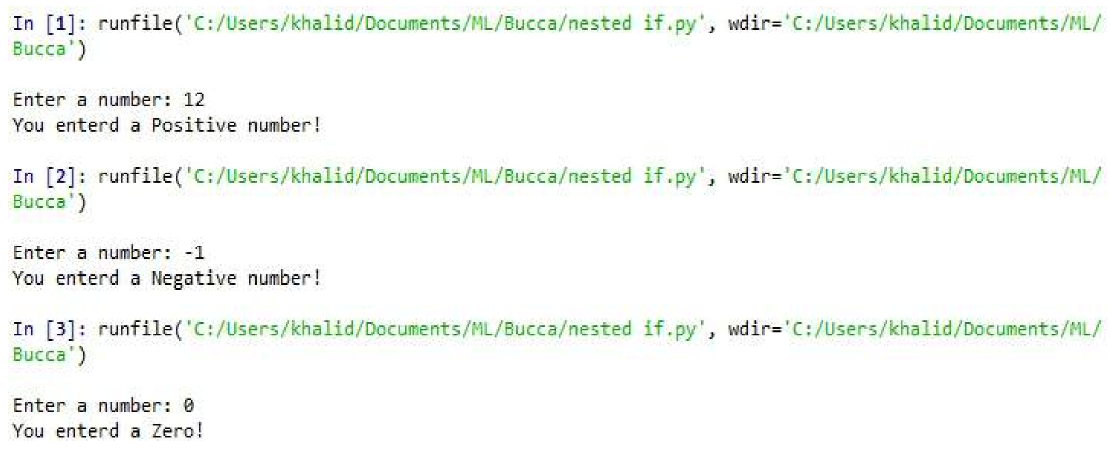
In the above program if “num” is positive then “positive number”Since the Boolean expression of “if” evaluates to be true, the if block code is executed, after execution the control goes out of the current “if..elif.else” decision chain. Try the above two variations yourself. The output of the above program is given below.

```
Positive number
```

Python permits nesting any number “if..elif..else” chains within “if..elif..else” decision chains. The only way to differentiate one chain from the other is by the indentation and so in very complex cases gets confusing to read, This is better avoided to maintain the readability of the program. Lets take a look at a code example that uses “nested if statements”.

```
#=============================================================
# Demonstrating the nested if statements.
# Print the appropriate message for the different values of num
#=============================================================

num = int(input("Enter a number: "))   # Input from the user
                                        # convert it to the type int

if num >= 0:
    if num == 0:
        print("You've entered zero!")
    else:
        print("You've entered a positive number!")
else:
    print("You've entered a negative number!")

```

The output of the above program for each possible case is given below. In the above python code the input function waits for the user to enter a value in the python shell and returns this value as a string. In this program it is expected that the user enters a numerical value, this value entered by the user is returned by the input function as a string which is converted into an int type using the int( ) method.



```
In [1]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/
Bucca')

Enter a number: 12
You enterd a Positive number!

In [2]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/
Bucca')

Enter a number: -1
You enterd a Negative number!

In [3]: runfile('C:/Users/khalid/Documents/ML/Bucca/nested if.py', wdir='C:/Users/khalid/Documents/ML/
Bucca')

Enter a number: 0
You enterd a Zero!
```

There are some statements in python that allow iterating the execution over a certain part of the code until some condition is met, these types of statements are called looping statements. Python provides two looping statements

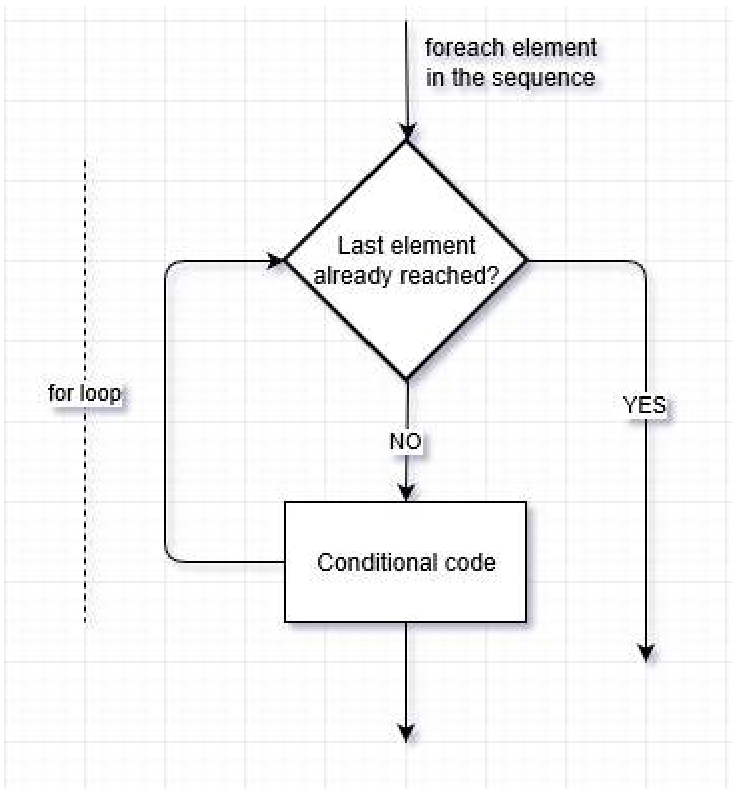
- for loop
- while loop

3.4 Pythons for loop

For loops in are very handy they are used iterate over all the elements of a sequence typed object(lists, tuples, strings) or an object that is iterated through. They are similar to the "foreach statement" which is available in languages like C#. The body of the for loop is executed once for each element of the container object. The general syntax of pythons for loop is given below.

```
for <variable> in <iterable typed object>:

        <Statements in body of for loop>
```

Here body of the for loop is identified by indentation. `<variable>` can be used in the body of the for loop and it holds the value of the element in the sequence typed object for each iteration. The loop continues until the last element in the sequence is reached. The flow chart of the for loop is shown below.


foreach element in the sequence
Last element already reached?
for loop
YES
NO
Conditional code

Lets look at a program that uses a "for loop" in it. In the example below a for loop is used to add the elements of a list and print the output. "numbers" is a a hard coded list of integers and variable "sum" is initialized to 0. In every iteration the current element of the sequence of that iteration is added to the variable "sum".

```python
#================================================================
# Program to find the sum of all numbers in a list
#================================================================


# List of integers
numlist = [10, 20, 30, 40, 50, 60, 70, 80, 90]

# variable to store the sum of elements
sum = 0

# iterate over the list
for num in numlist:
    sum = sum + num

```

```python

# Output: The sum is 450
print("The sum of all elements is ", sum)
```

The output of the above code is given below.

```
The sum of all elements is  450
```

3.4.1 The range( ) method

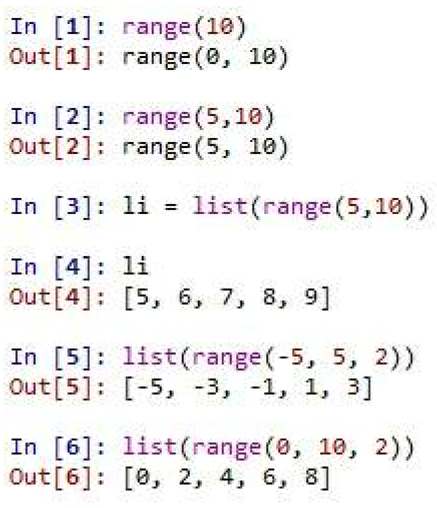
Python provides a range method that can be used to generate a list of a sequence of numbers. For example if "`range(10)`" is called it can generate a list of numbers from 0 to 10.

```
In [1]: list(range(10))
Out[1]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The number to start from, the number to end at and the step size can be specified in the range function by passing them as parameters as "`range(start, stop, step size)`". The step size has a default value of "`1`" if not specified. The start index defaults to "`0`" if not specified. The range function can be called in the following ways.

- `range(stop)`
- `range(start, stop)`
- `range(start, stop, step size)`

Here `start` is the number to start from, `stop` is the number to stop at(not generated by range function). In python 3.7 the range function returns an object of type "`range`", which can be converted to a list using the `list( )` method. The following examples can clarify more on the `range( )` function.

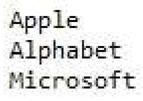


```
In [1]: range(10)
Out[1]: range(0, 10)

In [2]: range(5,10)
Out[2]: range(5, 10)

In [3]: li = list(range(5,10))

In [4]: li
Out[4]: [5, 6, 7, 8, 9]

In [5]: list(range(-5, 5, 2))
Out[5]: [-5, -3, -1, 1, 3]

In [6]: list(range(0, 10, 2))
Out[6]: [0, 2, 4, 6, 8]
```

The `range( )` function is being discussed with "for loops" because the `range( )` function is mostly used with for loops. Then can be used to generate the sequence of items that the "for loop" iterates over. In the example below the `range( )` function is combined with the `len( )` is used in a for loop to iterate through a list of items using index.

```python
#=========================================================================
# iterate through a list using indexing
#=========================================================================
Titans = ['Apple', 'Alphabet', 'Microsoft']

# Iterate through the list using the index
for i in range(len(Titans)):
    print(Titans[i])        # Accessing the i-th element of the list Titans


```

Here `len(Titans)` returns "3" which is passed as a parameter to the `range( )` function. The `range( )` method returns an range object "`range(0, 3)`" which can be iterated through by the for loop. This range can be used as the index for the "for loop". The above code generates the below output.



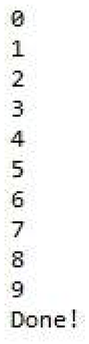
```
Apple
Alphabet
Microsoft
```

3.4.2 for / else statement

The “for loop” can have an optional “else” block, which is executed after all the elements in the sequence have been iterated through. In the case that a “break” statement was encountered during the execution of the “for loop” which causes an abrupt exit from the loop the “else” part is ignored and so the else part of the “for loop” is executed if there is no abrupt exit from the loop(if a break statement is not encountered while in the loop). An example that involves the “else” part of the “for loop’ is given below.

```python
#================================================================
# for / else loops example
#================================================================
temp = list(range(10))

for i in temp:
    print(i)
else:
    print("Done!")  # Executed after all iterations through
                    # the list are completed.

```

The output of the above program is given below.



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

3.5 Pythons while loop

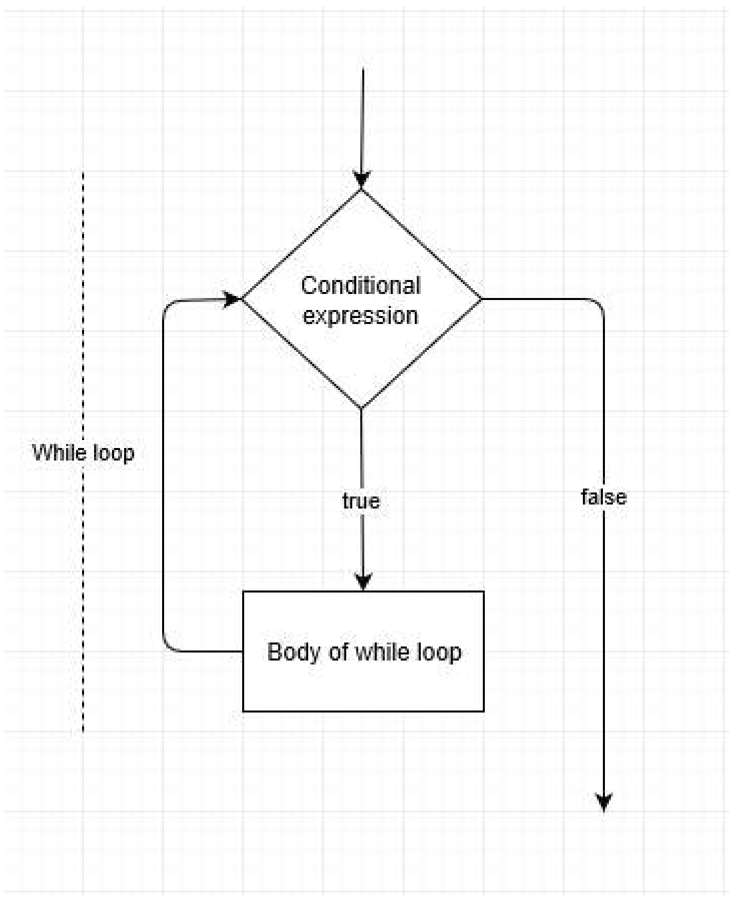
Pythons provides flow control tool, “while” statement which is similar to the “for” statement. The while loop in python allows looping/iterating over a block of code as long as it's conditional expression(condition) evaluates to be true. The difference between this loop and the "for loop" is that the for loop iterates over the elements in a sequence, the while loop on the other hand continue looping as long as its condition is true, so the while loop is mostly used in cased where the number of times to loop is unknown. The syntax of the while loop is given below.

```
while <Boolean expression>:

      <while block code>
```

Here as long as the `<boolean expression>` is evaluated to be true the `<while bloc code>` is executed. The step by step execution of the while loop is shown below.

1. The conditional expression is evaluated.
2. Executed the body of the while loop if the conditional expression is evaluated to be true.
3. Repeat steps 1 and 2 until the conditional expression is evaluated to be false.

Remember that any value other than "`0`" and "`null`" are considered to be true. The flowchart of the "while loop" is given below.



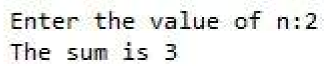
The conditional expression in the “while loop” usually contains one or many variables that is changed every time in the “while” block. This is important because without this change the “while loop” may end up looping without any end as it’s condition is always true, so its important to have one or more variables in the conditional expression of the “while loop” that approaches the terminating condition. Lets look at an example that uses the “while loop”.

```
#============================================================
# This program adds the first "n" natural numbers
#============================================================

n = int(input("Enter the value of n: "))

# Initialize sum and counter variables
i = 1
sum = 0

while i <= n:
    sum = sum + i
```

```python
    i = i + 1     # Update counter

print("The sum is: ", sum)

```

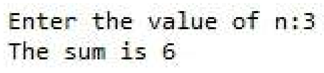
The above python script adds the first ‘n’ natural numbers(1, 2, 3, …), where “n” is a user provided input to add until. Here ‘i’ is the counter variable, it’s value is incremented in the body of the loop so the next natural number can be added in the next iteration of the loop. The conditional expression of the while loop returns false when the value of ‘i’ equals “n + 1”, this is when the control leaves the while loop. Some of the possible outputs for the above program are given below.

When “n = 2” the output of the above program is given below.



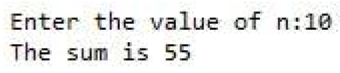
```
Enter the value of n:2
The sum is 3
```

When “n = 3” the output of the above program is



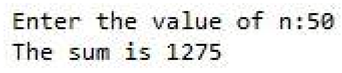
```
Enter the value of n:3
The sum is 6
```

When “n = 10” the output of the above program is



```
Enter the value of n:10
The sum is 55
```

When “n = 50” the output of the above program is



```
Enter the value of n:50
The sum is 1275
```

3.5.1 while / else statement

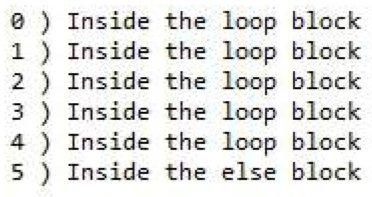
The just like the “for loop” the“while loop” can have an optional “else” block, which is executed after all the elements in the sequence have been iterated through. In the case that a “break” statement was encountered during the execution of the “while loop” which causes an abrupt exit from the loop the “else” part is ignored and so the else part of the “while loop” is executed if there is no abrupt exit from the loop.

An example that uses the “else” part of the “while loop’ is given below.

```python
#===============================================================
# This program illustrates the else statement with a while loop
#===============================================================

i = 0  # Initializing the counter variable for the while loop

while i < 5:
    print(i, ") Inside the loop block")
    i = i + 1
else:
    print(i, ") Inside the else block")

```

In the above python script the conditional expression for the while loop "`i < 5`" is `true` until "`i = 5`" which is when the else block code is executed

The output for the above code example is given below.



```
0 ) Inside the loop block
1 ) Inside the loop block
2 ) Inside the loop block
3 ) Inside the loop block
4 ) Inside the loop block
5 ) Inside the else block
```

3.5.2 Infinite loops

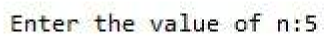
As we have already seen the "while loops" conditional expression commonly contain one or more "counter variables" that have their value changed on every iteration through the loop this allows the conditional expression to tend towards the terminating condition, which when reached, the control exits from the loop.

In the previously discussed example program for counting first n natural numbers the "while" block has a statement that increments the value of '`i`', this is very important (and sometimes forgotten) because if missed the loop executes forever with an end, this is called an infinite loop. Try the "counting the first n natural numbers" program yourself without the statement to increment the value of the counter variable.

```python
#============================================================
# This program adds the first "n" natural numbers
#============================================================

n = int(input("Enter the value of n: "))
```

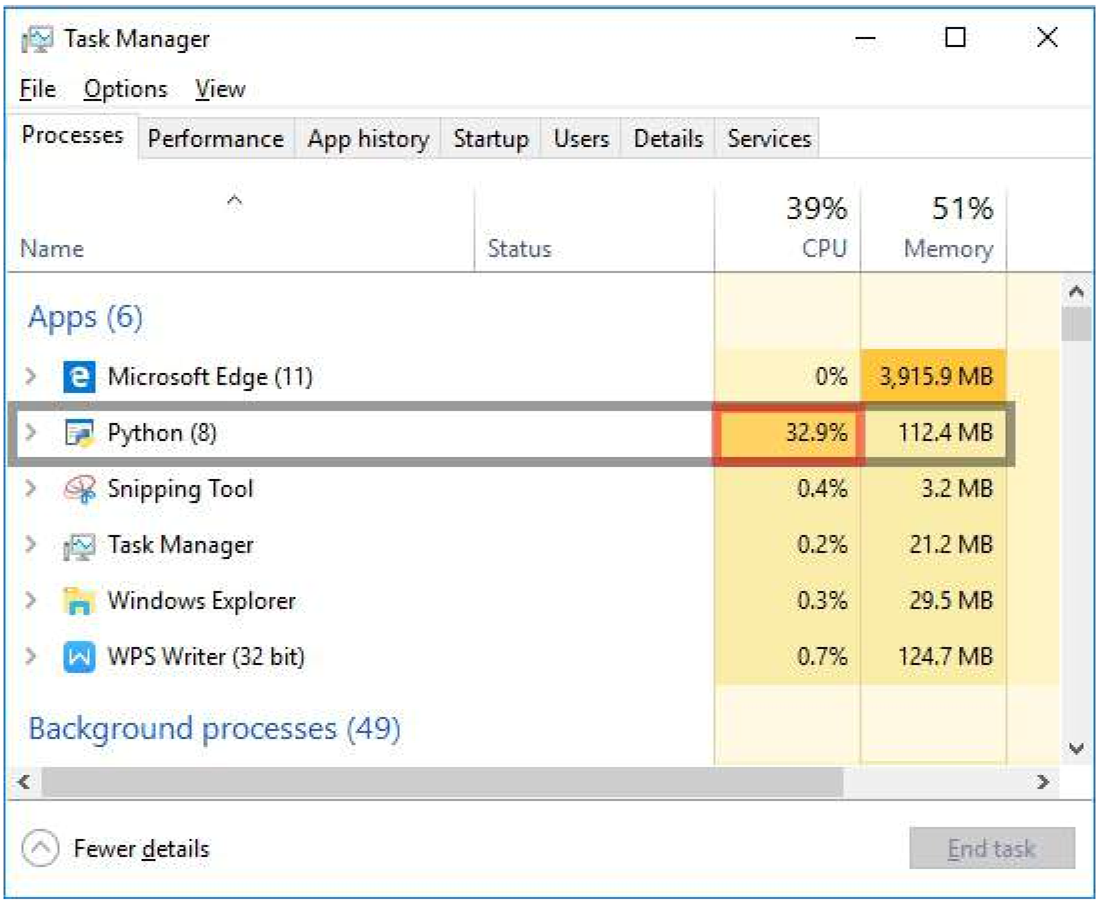
```python

# Initialize sum and counter variables
i = 1
sum = 0

while i <= n:
    sum = sum + i
    i = i + 1     # Update to the counter variable is skipped

print("The sum is: ", sum)

```

When the above program is executed, it print the output given below.



```
Enter the value of n:5
```

As you can see that after the value of ‘n’ s provided, the sum doesn’t get printed, this is because the control is still in the while loop which is an infinite loop. It’s possible to see that the program is still in execution by checking the “task manager”(ctrl + shift + esc on windows).


Task Manager
File Options View
Processes Performance App history Startup Users Details Services
Fewer details
End task

Name	Status	39% CPU	51% Memory
Apps (6)			
Microsoft Edge (11)		0%	3,915.9 MB
Python (8)		32.9%	112.4 MB
Snipping Tool		0.4%	3.2 MB
Task Manager		0.2%	21.2 MB
Windows Explorer		0.3%	29.5 MB
WPS Writer (32 bit)		0.7%	124.7 MB
Background processes (49)			

Lets take a look at some more examples of infinite loops with while loops.

```
# -*- coding: utf-8 -*-
"""
Created on Sat Mar  2 15:41:08 2019

@author: Mohamed Khalid
"""

#================================================================
# Examples of loops that terminate and infinite loops
#================================================================

# Assignments
var1 = 1
var2 = 2
var3 = 3
```

```python

#---------------------------------------- Loop is never executed
while (0 + 1 - var1):
    print(var1)
    var1 += 1

#---------------------------------------------- Terminating loop

while (var1 <= 10):
    print(var1)
    var1 += 1

#---------------------------------------------- Terminating loop
while (var2 >= -10):
    print(var2)
    var1 -= 5

#------------------------------------------------- Infinite loop
while (var3 != 10):
    print(var3)
    var3 += 6

#------------------------------------------------- Infinite loop
while (1):
    print(var1)
```

3.6 Break and Continue statements

Python provides two statements for abnormal termination of loops, the `break` and `continue` statements. Break and continue statements in python are used to change the control flow in a loop. Loop are used to iterate over a certain portion of the code, break statement is used to abruptly exit the loop entirely and continue is used to skip a specific iteration of the loop. Let's discuss more details about the break and continue statement.

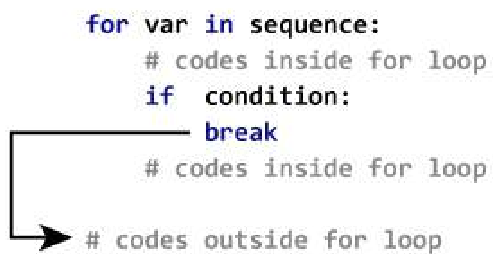
3.6.1 The break statement

The `break` statement when encountered terminates the loop that contains it. When the `break` statement is encountered by the the python interpreter passes the control to the statement immediately after the body of the loop.

In the case of nested loops(loops embedded in other loops) the the `break` statement when encountered, terminates the loop that it was encountered at. As the `break` statement changes the control flow regardless of any condition it is an unconditional branching statement. The syntax of the break statement is given below.

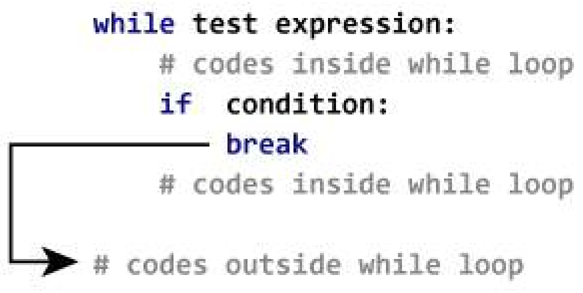
```
break
```

The general code given below illustrates what the `break` statement does when it is encountered in a "for loop".



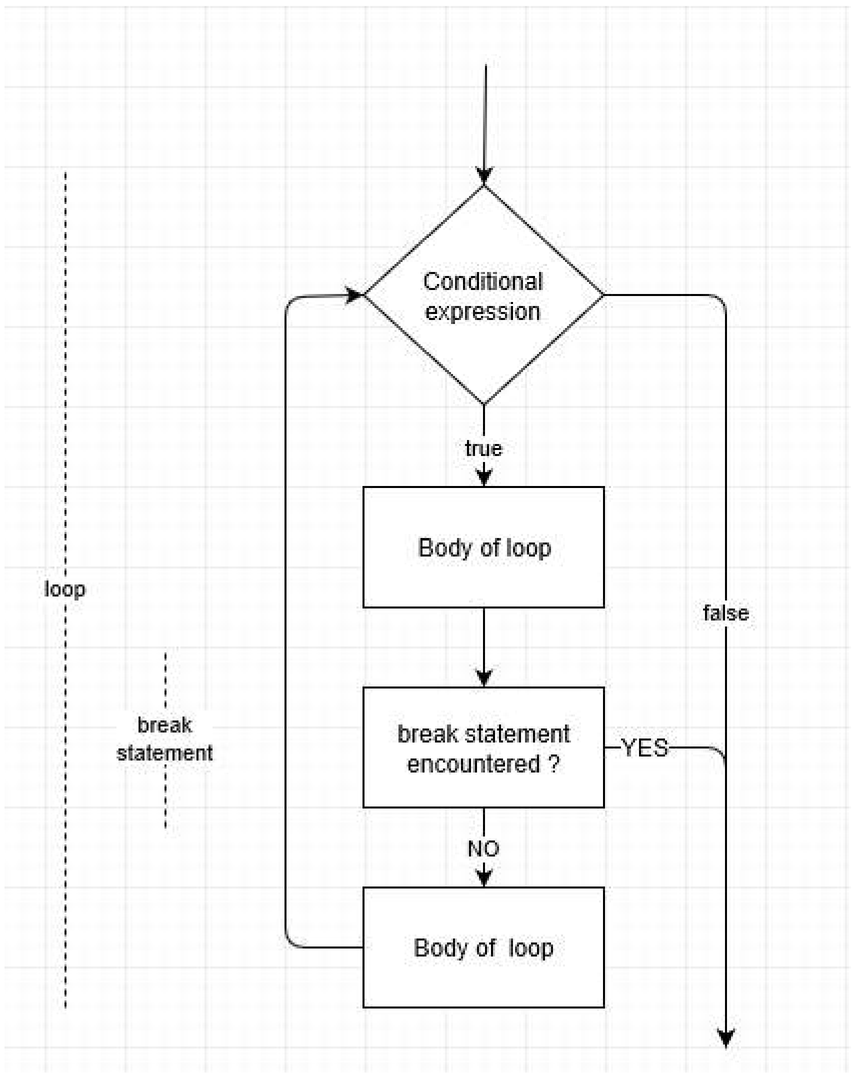
```
for var in sequence:
    # codes inside for loop
    if  condition:
        break
    # codes inside for loop

# codes outside for loop
```

The general code given below illustrates what the `break` statement does when it is encountered in a "while loop".



```
while test expression:
    # codes inside while loop
    if  condition:
        break
    # codes inside while loop

# codes outside while loop
```

The flow chart of the `break` statement is shown below.

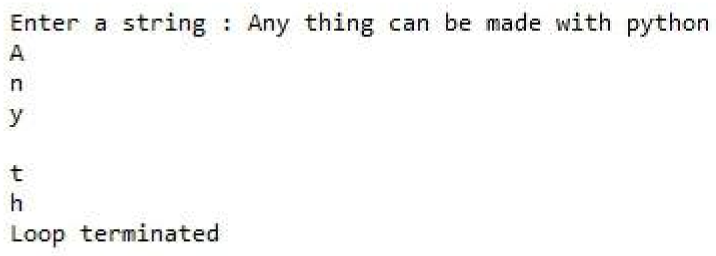


Lets look at an example that uses the `break` statement in a loop. In this python program the loop is abruptly terminated when the a specific letter is found.

```python
#=================================================================
# Print all characters of a string until the first "i" is found
#=================================================================

userString = input("Enter a string : ")

for val in userString:
    if val == "i":
        print("Loop terminated")
        break
    print(val)
else:
    print("The letter i was not found in your string")
```

In the above python script a “for loop” is used to iterate through each character of the the user provided string. On each iteration the current character is checked if it is the letter ‘i’ in the case it is not, then this character is printed, if the character is the letter ‘i’ then the python interpreter encounters the `break` statement which causes the control to exit the loop and passes the control outside the “for loop”(“else” is not executed)

The output of the above program is given below.



```
Enter a string : Any thing can be made with python
A
n
y

t
h
Loop terminated
```

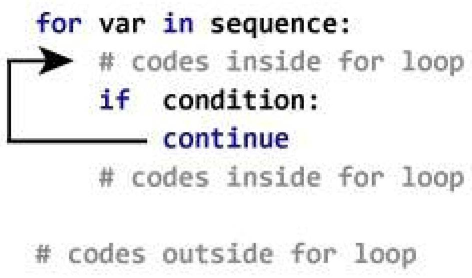
3.6.2 The continue statement

The `continue` statement when encountered abruptly terminates the current iteration loop that contains it. When the `continue` statement is encountered by the the python interpreter passes the control to the statement immediately after the body of the loop.

In the case of nested loops(loops embedded in other loops) the the `continue` statement when encountered, terminates the current iteration of the loop that it was encountered at. As the `continue` statement changes the control flow regardless of any condition it is an unconditional branching statement. The syntax of the `continue` statement is given below

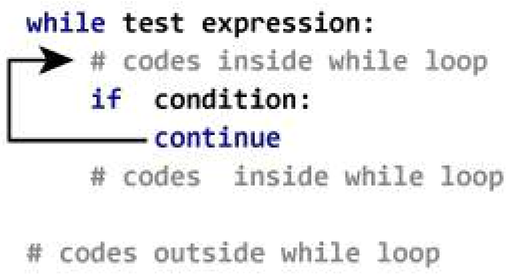
```
continue
```

The general code given below illustrates what the `continue` statement does when it is encountered in a “for loop”.



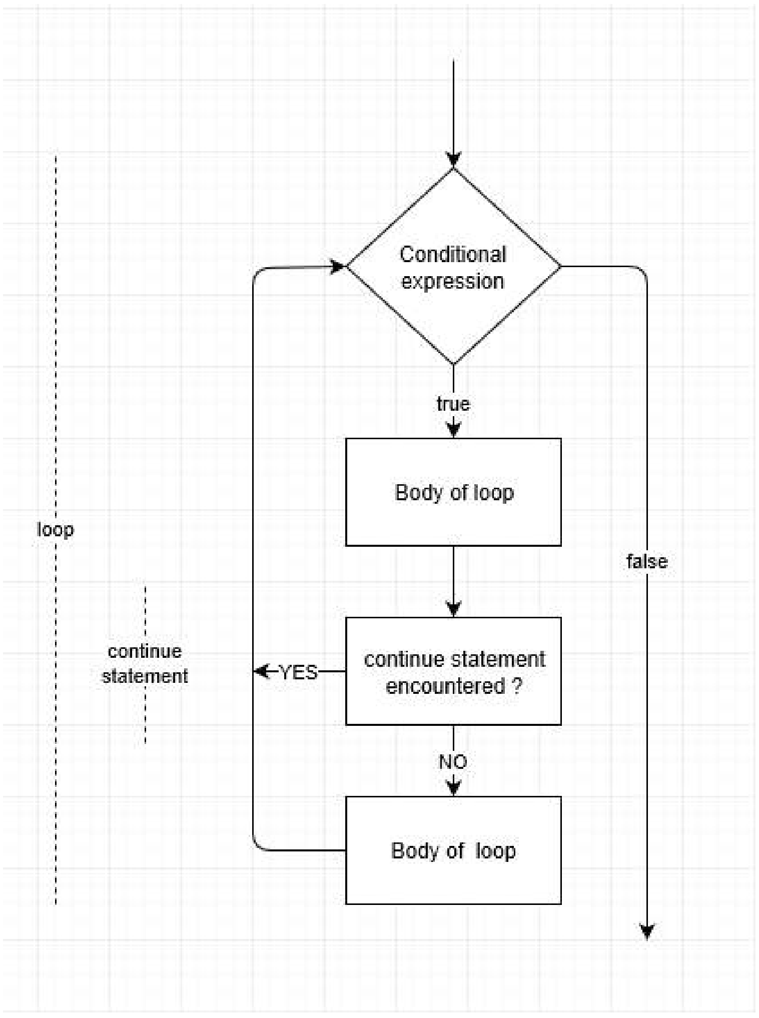
```
for var in sequence:
    # codes inside for loop
    if  condition:
        continue
    # codes inside for loop

# codes outside for loop
```

The general code given below illustrates what the `continue` statement does when it is encountered in a “while loop”.



```
while test expression:
    # codes inside while loop
    if  condition:
        continue
    # codes  inside while loop

# codes outside while loop
```

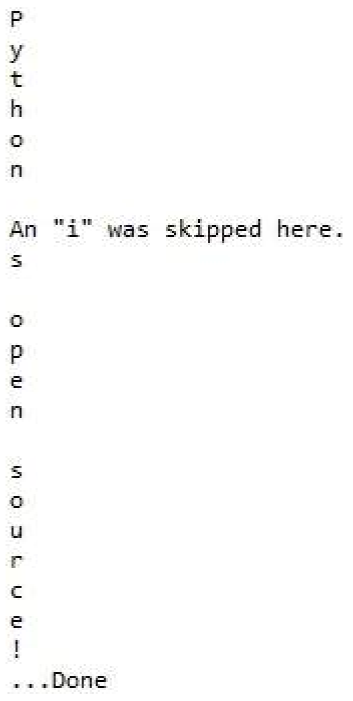
The flowchart of the `continue` statement is shown below.



Lets look at an example that uses the `continue` statement in a loop. In this python program the an iteration that contains a specific letter is skipped.

```python
#================================================================
# Print all characters of a given string except i's
#================================================================

userString = input("Enter a string : ")

for val in userString:
    if val == "i":
        print("An \"i\" was skipped here.")
        continue
    print(val)
else:
    print("...Done")
```

In the above python script a “for loop” is used to iterate through each character of the the user provided string. On each iteration the current character is checked if it is the letter ‘i’ in the case it is not, then this character is printed, if the character is the letter ‘i’ then the python interpreter encounters the `continue` statement which causes the control to exit the current iteration of the loop and passes the control to the next iteration of that loop.



```
P
y
t
h
o
n

An "i" was skipped here.
s

o
p
e
n

s
o
u
r
c
e
!
...Done
```

3.7 Pythons pass statement

Python provides the `pass` statement, which basically does not do anything, it is a null statement. It is not completely ignored by the python interpreter like the comments in python are, they simply don’t do anything and are mostly used as place holders.

Nothing happens on the execution of a `pass` statement, it’s execution results in a no operation(NOP). The syntax of the `pass` statement in python is given below.

```
pass
```

Let’s take a look at an example that shows the use of the pass statement in python. Suppose that there is a loop or a function definition who’s bodies haven’t been implemented yet, python does not allow loops or function definitions to have empty bodies, so instead they can contain a pass statement that serves as a place holder as it is not ignored by the python interpreter nor does it do any operation The example below shows how the pass statement can be used as a place holder in loops.

```python
#============================================================
# Using the pass statement as a place holder
#============================================================

# Variable declaration
mySequence = list(range(10))

for item in mySequence:
    pass            # pass is acts as a placeholder for
                   # the functionality to be added later.

```

The pass statement can also be used as a placeholder in function definitions as shown below.

```python
def myFunction():
    pass            # pass is acts as a placeholder for
                   # the function definition to be
                   # added later.

```

The pass statement can also be used as a placeholder in class definitions as shown below.

```python
class myClass:
    pass            # pass is acts as a placeholder for
                   # the class definition to be
                   # added later.

```

3.8 Exercises

1. What possible values can a Boolean expression have?

2. Where does the term Boolean originate?

3. What is an integer equivalent to True in Python?

4. What is the integer equivalent to False in Python?

5. Is the value -16 interpreted as True or False?

6. Given the following definitions:

x, y, z = 3, 5, 7

evaluate the following Boolean expressions:

(a) x == 3

(b) x < y

(c) x >= y

(d) x <= y

(e) x != y - 2

(f) x < 10

(g) x >= 0 and x < 10

(h) x < 0 and x < 10

(i) x >= 0 and x < 2

(j) x < 0 or x < 10

(k) x > 0 or x < 10

(l) x < 0 or x > 10

7. Given the following definitions:

b1, b2, b3, b4 = true, false, x == 3, y < 3

evaluate the following Boolean expressions:

(a) b3

(b) b4

(c) not b1

(d) not b2

(e) not b3

(f) not b4

(g) b1 and b2

(h) b1 or b2

(i) b1 and b3

(j) b1 or b3

(k) b1 and b4

(l) b1 or b4

(m) b2 and b3

(n) b2 or b3

(o) b1 and b2 or b3

(p) b1 or b2 and b3

(q) b1 and b2 and b3

(r) b1 or b2 or b3

(s) not b1 and b2 and b3

(t) not b1 or b2 or b3

(u) not (b1 and b2 and b3)

(v) not (b1 or b2 or b3)

(w) not b1 and not b2 and not b3

(x) not b1 or not b2 or not b3

(y) not (not b1 and not b2 and not b3)

(z) not (not b1 or not b2 or not b3)

8. Express the following Boolean expressions in simpler form; that is, use fewer operators. x is an integer.

(a) not (x == 2)

(b) x < 2 or x == 2

(c) not (x < y)

(d) not (x <= y)

(e) x < 10 and x > 20

(f) x > 10 or x < 20

(g) x != 0

(h) x == 0

9. What is the simplest tautology?

10. What is the simplest contradiction?

11. Write a Python program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print ”OK;” otherwise, do not print anything.

12. Write a Python program that requests an integer value from the user. If the value is between 1 and 100 inclusive, print ”OK;” otherwise, print ”Out of range.”

13. Write a Python program that allows a user to type in an English day of the week (Sunday, Monday, etc.). The program should print the Spanish equivalent, if possible.

14. Consider the following Python code fragment:

```
# i, j, and k are numbers
if i < j:
if j < k:
i = j
else:
j = k
else:
if j > k:
```

```
j = i
else:
i = k
print("i =", i, " j =", j, " k =", k)
```

What will the code print if the variables i, j, and k have the following values?

(a) i is 3, j is 5, and k is 7

(b) i is 3, j is 7, and k is 5

(c) i is 5, j is 3, and k is 7

(d) i is 5, j is 7, and k is 3

(e) i is 7, j is 3, and k is 5

(f) i is 7, j is 5, and k is 3

15. Consider the following Python program that prints one line of text:

```
val = eval(input())
if val < 10:
if val != 5:
print("wow ", end='')
else:
val += 1
else:
if val == 17:
val += 10
else:
print("whoa ", end='')
print(val)
```

What will the program print if the user provides the following input?

(a) 3

(b) 21

(c) 5

(d) 17

(e) -5

16. Write a Python program that requests five integer values from the user. It then prints the maximum and minimum values entered. If the user enters the values 3, 2, 5, 0, and 1, the program would indicate that 5 is the maximum and 0 is the minimum. Your program should handle ties properly; for example, if the user enters 2, 4 2, 3 and 3, the program should report 2 as the minimum and 4 as maximum.

17. Write a Python program that requests five integer values from the user. It then prints one of two things: if any of the values entered are duplicates, it prints "DUPLICATES"; otherwise, it prints "ALL UNIQUE".

18. Write a Python program that ...

19. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
print('*', end='')
a += 1
print()
```

20. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
print('*', end='')
print()
```

21. How many asterisks does the following code fragment print?

```
a = 0
while a > 100:
print('*', end='')
```

```
a += 1
print()
```

22. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
b = 0;
while b < 55:
print('*', end='')
b += 1
print()
a += 1
```

23. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
if a % 5 == 0:
+= 1
print()
```

24. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
b = 0
while b < 40:
if (a + b) % 2 == 0:
print('*', end='')
b += 1
print()
```

```
a += 1
```

25. How many asterisks does the following code fragment print?

```
a = 0
while a < 100:
b = 0
while b < 100:
c = 0
while c < 100:
print('*', end='')
c++;
b += 1
a += 1
print()
```

26. How many asterisks does the following code fragment print?

```
for a in range(100):
print('*', end='')
print()
```

27. How many asterisks does the following code fragment print?

```
for a in range(20, 100, 5):
print('*', end='')
print()
```

28. How many asterisks does the following code fragment print?

```
for a in range(100, 0, -2):
print('*', end='')
print()
```

29. How many asterisks does the following code fragment print?

```
for a in range(1, 1):
print('*', end='')
print()
```

30. How many asterisks does the following code fragment print?

```
for a in range(-100, 100):
print('*', end='')
print()
```

4. Functions in python

Function syntax has already been introduced in the previous chapters, we have already worked with some of functions from python s standard library like `print()`, `list()`, `tuple()`, `len()`, `id()` and some other functions. The python library has several functions that can be used for common programming tasks.

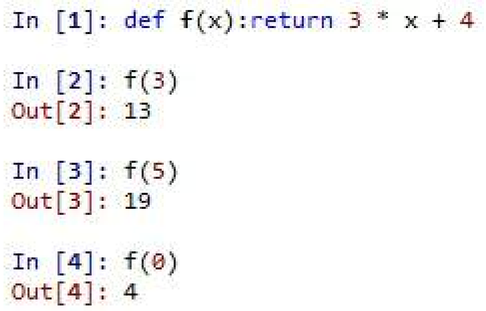
Functions in mathematics are of the form `f(x) = y`. Here function 'f' when given the input 'x' performs operations on 'x' and gives 'y' as the output. An example of function a definition in mathematics is

```
f(x) = 3x + 4
```

From this function definition we can compute the values of `f(x)` for the different values of 'x' as below.

```
f(3) = 13

f(5) = 19

f(0) = 4
```

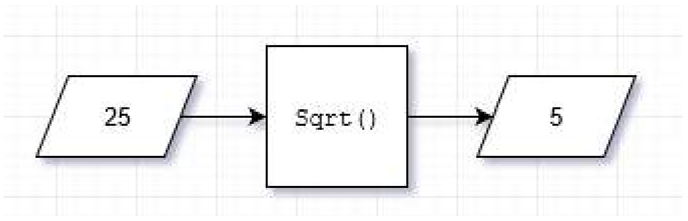
Functions in python is a block of code that is given a name(like the name of above function is 'f'). This block of code can be called any where in the program, when called control is shifted to the body of the called function. The above function 'f' can be written in python as follows.



```
In [1]: def f(x):return 3 * x + 4

In [2]: f(3)
Out[2]: 13

In [3]: f(5)
Out[3]: 19

In [4]: f(0)
Out[4]: 4
```

One example of a function in pythons standard library is the square root function, this function is named `sqrt`. The square root functions takes numeric as parameter and returns a float value, like square root of "25" is '5'.

Functions should be thought of as tools that perform a certain operation, so it's important to know "what a function does" than "how it does it". Thinking of functions this way allows the programmer to see functions as a black box that performs the desired operation. In the case of the square root function it allows the programmer to focus on the task at hand rather than focusing on the algorithm to calculate the square root of a number, like for example to make a program that return the square root of a number provided by the user, the `sqrt()` function can be used to avoid the complex logic of computing the square root of a number. The code example below returns the square root of a number provided by the user by using the `sqrt()` function instead of implementing the logic for square root calculation.



```python
from math import sqrt

# ask user for input
num = eval(input("Enter number: "))

# Compute the square root with sqrt()
# sqrt() is responsible for square root calculation,
# allowing the programmer to focus on whats important
root = sqrt(num);

# print the result
print("Square root of", num, "=", root)
```

Output for the above program is below.

```
Enter number: 25
Square root of 25 = 5.0
```

Here `sqrt(num)` is function invocation or a function call(a call to the `sqrt()` function). A function provides a service to it's the the caller(invoker of the service). The above code that we have to compute square root of a number is the "calling code" or the "client code" to the `sqrt()` function which acts like a service that can be invoked

```
root = sqrt(num)
```

The above statement invokes `sqrt()` passing it the value of num. The expression `sqrt(num)` the square root of the value of the variable "num".

Unlike the small collection of functions(like `type, str, int, range, id`) that are always available in python programs, the `sqrt()` function is available in the "math" module and can not be used unless the math module is imported, hence the purpose of the "import" statement(the first line of code)

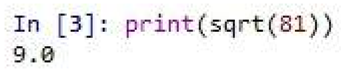
```
from math import sqrt
```

The above code imports the `sqrt()` function from the "math" module. A module is a collection of code that can be imported and used in other python programs. The "math" module is very rich in the functions it provides to perform math operations, it provides functions for trigonometric, logarithmic and several other math operations.

A function has a name followed by a set of parentheses which contain the information that the function needs to perform it's task.

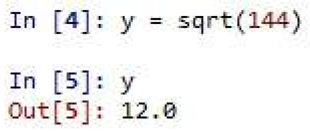
```
Sqrt( num ) # sqrt needs a parameter to return the square root of
```

`num` is the value that the `sqrt()` function operates on. The values that are passed to a function are called the parameters or arguments of the function. Functions after they perform the computations it was defined to perform can return a value back to the caller(client invoking it) like in the case of the `sqrt()` function which returns the square root of the parameter passed as a float type to its invoker. This value that is returned to the client can be used by the client just like any other value.



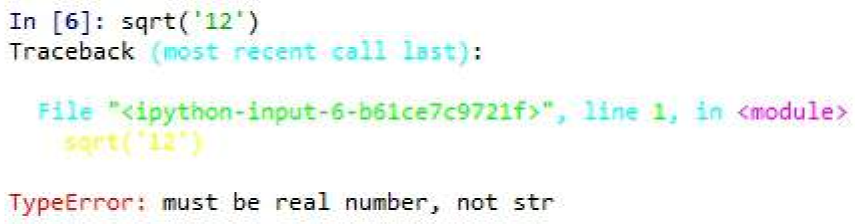
```
In [3]: print(sqrt(81))
9.0
```

The `print()` function above can use the value returned by the `sqrt()` function.



```
In [4]: y = sqrt(144)

In [5]: y
Out[5]: 12.0
```

Like in the above assignment statement, the value returned by a function can be assigned to a name. When the client code attempts to pass an argument to the function it's invoking that has a type that is different then what is expected by the function, then the interpreter raises an error.



```
In [6]: sqrt('12')
Traceback (most recent call last):

  File "<ipython-input-6-b61ce7c9721f>", line 1, in <module>
    sqrt('12')

TypeError: must be real number, not str
```

A function can be called any number of times anywhere in the program. Remember to he the client of a function, the function is like a labeled black box, the knows client knows what the function bing called does and is not concerned with how the task is performed. All functions can be treated like black boxes. A service that a function provides can be used without being concerned of the internal details.

The only way a behaviour of a function can be altered is through the functions parameters(the arguments that are passed to a function can be different). When immutable types(int, float, string, tuple) are passed as arguments to a function, a copy of the objects are passed instead of a reference to the original objects, therefore immutable objects when passed to a function as an argument can not be changed by the called function. In the case that the arguments passed are mutable objects a reference to those objects are passed instead of a new copy of those objects, therefore mutable objects when passed to a function as an argument can be changed by the invoked function.

As we already seen before functions can take more than one parameter like the `range()` function that can take one, two or three parameters.

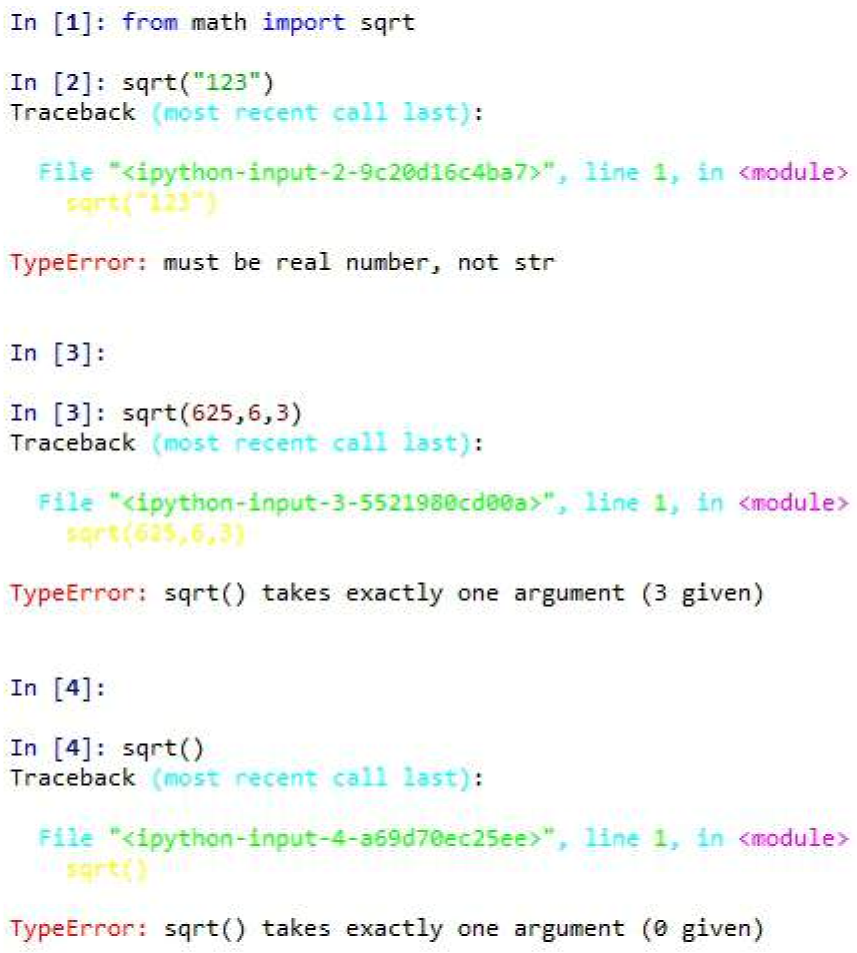
From the view of the client/caller, a function has three parts. Lets discuss eac of them in brief.

Name

A function's name is the handle that is used to identify the code to be executed when it is invoked. The same rules that apply to variable names apply to function names. Function names, like variable names are also identifiers.

Parameters

Functions are called with zero or more parameters. The types of each argument passed in the invocation of a function must be the same as the types of parameters in the function definition. If more are or less parameters are passed than what is expected the interpreter raises an error. Some examples of this these two kinds of errors are illustrated in the shell execution below.

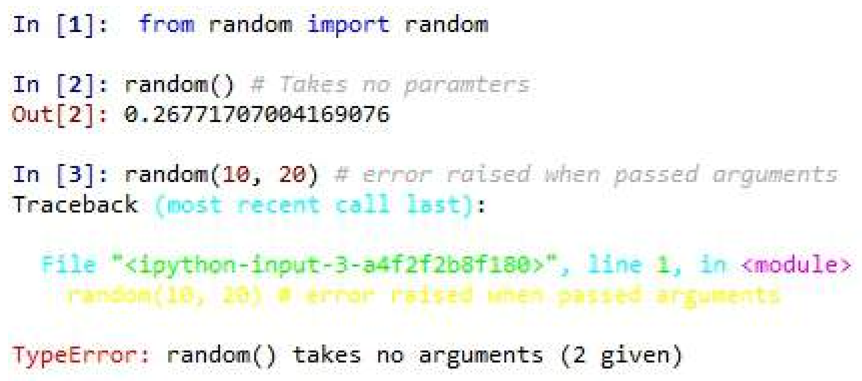


```
In [1]: from math import sqrt

In [2]: sqrt("123")
Traceback (most recent call last):

  File "<ipython-input-2-9c20d16c4ba7>", line 1, in <module>
    sqrt("123")

TypeError: must be real number, not str


In [3]:

In [3]: sqrt(625,6,3)
Traceback (most recent call last):

  File "<ipython-input-3-5521980cd00a>", line 1, in <module>
    sqrt(625,6,3)

TypeError: sqrt() takes exactly one argument (3 given)


In [4]:

In [4]: sqrt()
Traceback (most recent call last):

  File "<ipython-input-4-a69d70ec25ee>", line 1, in <module>
    sqrt()

TypeError: sqrt() takes exactly one argument (0 given)
```

Return type

A function always returns a value to it's caller. The client code should be compatible with the type of the object returned by called function. The return type and the parameter types are not related. Even in the case that the function does not specifically return any value, a special "none" value is return to the client.

There are functions that take no parameter like the `random()` function. The `random()` function returns a random floating point value and it belongs to the "random" module. Again if the arguments passed does not match with the number of parameters and the type of each parameter in the function definition then an error is raised by the interpreter. You can see in the below code that if a parameter is passed to the `random()` function an error is raised.



```
In [1]:  from random import random

In [2]: random() # Takes no paramters
Out[2]: 0.26771707004169076

In [3]: random(10, 20) # error raised when passed arguments
Traceback (most recent call last):

  File "<ipython-input-3-a4f2f2b8f180>", line 1, in <module>
    random(10, 20) # error raised when passed arguments

TypeError: random() takes no arguments (2 given)
```

One of the ways that functions can be categorized is as,

- **Functions without side effects:** These types of functions just perform the required task and return the result to the client caller and that is all it does, it makes no other changes like printing to the display, changing the variables of the caller, etc. This type of functions always return a result(not none) to the invoking client
- **Functions with side effects:** These types of variables perform the required task which may also require to change it's environment like printing to the display or changing the value of the caller, etc. This type of functions may sometimes return a special value "none" to the invoking client.

4.1 Standard mathematical functions

The "math" module provides almost all of the functionality of a scientific a scientific calculator. Some of he function of the "math" module a re listed below.

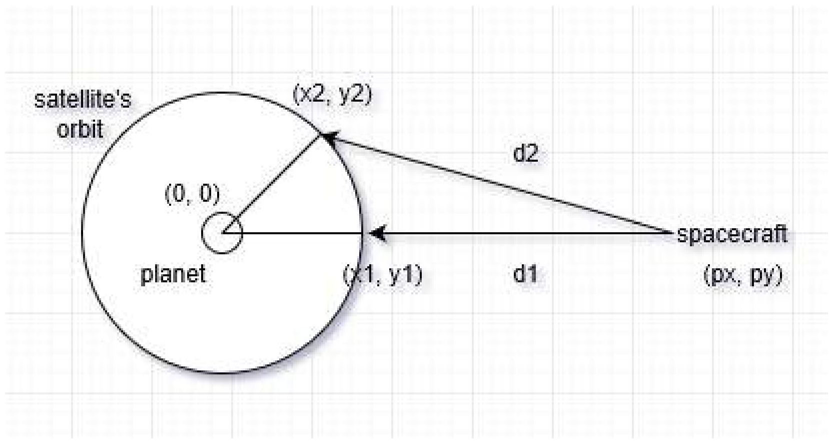
Function	description
`sqrt`	Computes the square root of a number.
`exp`	Computes e raised a power
`log`	Computes the natural logarithm of a number.
`log10`	Computes the common logarithm of a number.
`cos`	Computes the cosine of a value specified in radians: cos(x) = cos x; other trigonometric functions include sine, tangent, arc cosine, arc sine, arc tangent, hyperbolic cosine, hyperbolic sine, and hyperbolic tangent
`pow`	Raises one number to a power of another.
`degrees`	Converts a value in radians to degrees.

`radians`	Converts a value in degrees to radians.
`fabs`	Computes the absolute value of a number.

The parameters passed to the function by the calling client code is called the actual parameters(arguments) and the parameters that are specified in the function definition are the formal parameters. When a function is invoked the first actual parameter is assigned to the first formal parameter, then the second actual parameter is assigned to the second formal parameter, so the assignment of actual parameters to formal parameters happens from left to right. The order of passing the assignment matters, for example calling the `pow()` function as `pow(10,2)` returns 100 but calling it as `pow(2, 10)` returns 1024.

Lets try to solve a trigonometry problem with the functions provided by the math module. The problem is as follows.

Suppose that there is a spacecraft at some distance from a plane. A satellite orbits this planet, the problem is to find how much farther a way the satellite would be away from the spacecraft if the the satellite moves 10 degrees along it's orbital path. The spacecraft is on the plane of the satellites orbit.



Let the center of the planet be the origin(0, 0) of the coordinate system, this is also the center of the satellites orbit. In this problem the spacecraft is located at the point (p_x, p_y), the satellite is initially at point (x_1, y_1) then moves to point (x_2 , y_2). Here the the points (x_1, y_1) and (p_x, p_y) are given and the problem is to find the the difference between d_1 and d_2. This problem can be solved in two steps as follows.

Step 1

Calculate the point (x_1, y_1). For any given point (x_1, y_1) a rotation of theta degrees gives a new point (x_2, y_2), where

$$\begin{aligned} x_2 &= x_1\cos\theta - y_1\sin\theta \\ y_2 &= x_1\sin\theta + y_1\cos\theta \end{aligned}$$

Step 2

The distance d_1 between the two points (p_x, p_y) and (x_1, y_1) is given by,

$$d_1 = \sqrt{(x_1 - p_x)^2 + (y_1 - p_y)^2}$$

The same way, distance d_2 between the two points (p_x, p_y) and (x_2, y_2) is given by,

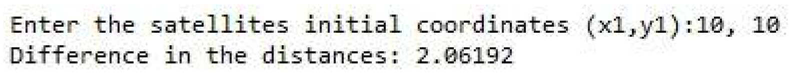
$$d_2 = \sqrt{(x_2 - p_x)^2 + (y_2 - p_y)^2}$$

The python program below solves the orbital distance problem for the spacecraft at location (100, 0) and user provided starting location for the satellite. You can see that the standard math problem was used in the program to com at the solution.

```python
#======================================================================
# Python program for the orbital distance problem
#======================================================================
from math import sqrt, cos, sin, pi

# The orbiting point is (x,y)
# A fixed point is always (100, 0),
# (p_x, p_y). Change them as necessary.
p_x = 100
p_y = 0

# number of radians in 10 degrees
radians = 10 * pi/180

# The cosine and sine of 10 degrees
cos10 = cos(radians)
sin10 = sin(radians)

# ask the satellites starting point from the user
x1, y1 = eval(input("Enter the satellites initial coordinates (x1,y1):"))

# calculate the initial distance d1
d1 = sqrt((p_x - x1)**2 + (p_y - y1)**2)

```

```python
# calculate the new location (x2, y2) of the satellite after movement
x2 = x1*cos10 - y1*sin10

y2 = x1*sin10 + y1*cos10

# Compute the new distance d2
d2 = sqrt((p_x - x2)**2 + (p_y - y2)**2)

# Print the difference in the distances
print("Difference in the distances: %.5f" % (d2 - d1))

```

The output when the above program is executed for the starting location (10, 10) is,



```
Enter the satellites initial coordinates (x1,y1):10, 10
Difference in the distances: 2.06192
```

4.2 Time functions

Pythons “time” module provides a lot of functions that involve time. In this section we will take a look at two of the functions that are provided by this module the `clock()` and `sleep()` functions.

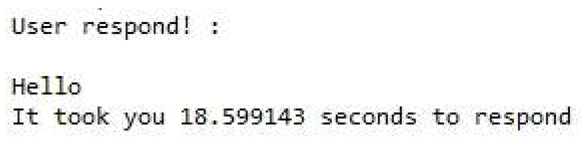
The clock function allows measuring time in seconds and it can be used anywhere in a program allowing the measure of time at any part of a program. The clock function works differently based on the operating system that it is being executed on. On unix like operating systems like linux and mac OS, the clock() function returns the number number of seconds elapsed from the start of the programs execution. On Microsoft windows the clock() function returns the number of seconds elapsed from the previous call to the clock() function. The return the number of seconds elapsed as a floating point number. The difference between the returned values of two clock function calls can be used to find the time period between the two calls.

The below program returns the amount of time it took the user to respond.

```python
#=====================================================================
# This program prints the time it took the user to respond in seconds
#=====================================================================
from time import clock # import the clock method
```

```python

print("User respond! : ")
startTime = clock()
input()
endTime = clock()
print("It took you %f seconds to respond " % (endTime - startTime))


```

An output a very slow user might get is,



```
User respond! :

Hello
It took you 18.599143 seconds to respond
```

Here is a another program that displays the amount of time it took to add the first ten million natural numbers.

```python
#=================================================================
# Time taken to add the first million natural numbers
#=================================================================
from time import clock # import the clock method

sum = 0 # initialize sum as 0

print("started adding")
startTime = clock()

for i in range(10000000):
    sum += i

endTime = clock()
print("It took %f seconds " % (endTime - startTime))

```

The output of this program is given below.

```
started adding
It took 1.363418 seconds
```

The sleep function can be used to suspend a program for a period of time in seconds. The number of seconds to suspend the program for is passed as an argument to the sleep function. The below program counts down to zero from three with one second intervals.

```
#=====================================================================
# Three second count down timer!
#=====================================================================
from time import sleep    # Import the sleep method

timer = int(3)            # Initialize the timer to a predefined value

print("Count down starts... %i" %timer)

while timer > 0:
    sleep(1)
    timer -= 1
    print(timer)

sleep(1)

```

The proves to be very useful when in comes to controlling the speed in graphical animations. This program when ran produces the following output.

```
count down starts... 3
2
1
0
```

4.3 Random functions

The package random provides some functions that deal with randomness. Applications that require the behaviour of randomness need these kind of functions. Almost all random number generating algorithms produce “pseudo random” numbers which means they are not truly random. pseudo random number generators when repeated long enough produce results that start repeating the exact same sequence of numbers. This not a serious problem as the length of this sequence is long enough to be considered safe for almost all practical applications.

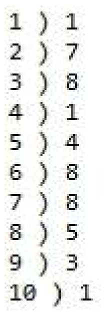
Pythons “random” module consists of a number of functions that enable the use of pseudo random numbers in programs. Few of the commonly used functions of this module are described below.

Function	Description
`random()`	Returns a pseudo random floating-point number x in the range `0 <= x < 1`
`randrange()`	Returns a pseudo random integer value within a specified range.
`seed()`	Sets the random number seed.

The `seed()` sets the value from which the sequence of the pseudo random numbers are generated. Every time the functions `random()` and `randrange()` are called they return the next value in the sequence of pseudo random values. The program below, prints ten random numbers in the range of one to ten.

```python
from random import randrange, seed

seed(10)

for i in range(10):
    print(i+1,")",randrange(1, 10))

```

This program produces the output shown below.

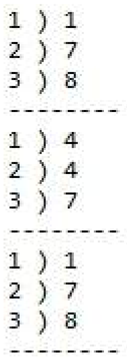


```
1 ) 1
2 ) 7
3 ) 8
4 ) 1
5 ) 4
6 ) 8
7 ) 8
8 ) 5
9 ) 3
10 ) 1
```

The function seed() sets the “seed number” that is used to generate the pseudo random number sequence and as there is a dependency if the seed value is changed then then sequence of pseudo random random numbers will also change. This dependency is in one way a problem because if the above program is run again it produces the exact same sequence of random numbers. This is illustrated in the example program shown below.

```python
from random import randrange, seed

seed(10) # The seed is 10

for i in range(3):
    print(i + 1, ")", randrange(1, 10))
print("----------")

seed(14) # Change the seed to something else

for i in range(3):
    print(i + 1, ")", randrange(1, 10))
print("----------")

seed(3)

for i in range(3):
    print(i + 1, ")", randrange(1, 10))
print("----------")
```

From this program you can see that the same sequence of pseudo random numbers are generated for the same value of the seed. Having constant value of the seed isn't a safe way to generate random numbers, but it can be useful while testing a program. When the value of the seed is not specifically mentioned, then the system time determines the seed. This allows the pseudo random numbers generated to have a more random behaviour. The out put of the above program are shown below.

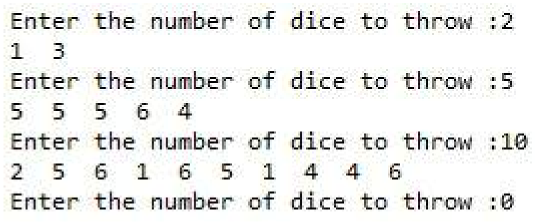


```
1 ) 1
2 ) 7
3 ) 8
--------
1 ) 4
2 ) 4
3 ) 7
--------
1 ) 1
2 ) 7
3 ) 8
--------
```

Now lets write a program that simulates the rolling of a six sided die. A die has six sides labeled one to six with one of the sides that can land on top when the die is rolled. We want to user to provide information on the number of dice to be thrown. The random number generated should lie between zero and six, so the randrange(1, 6) will be used for this, and it will be called for each die thrown.Lastly this entire setup should be placed in a loop that continue indefinitely until the user provides the number zero as input.

```python
from random import randrange

# skipping the seed setup

sides = 7

while 1:
    diceCount = int(input("Enter the number of dice to throw :"))
    for i in range(diceCount):
        dieValue = randrange(1,sides)
        print("%i " % (dieValue), end = " " )

```

An output for the above program is given below.



```
Enter the number of dice to throw :2
1  3
Enter the number of dice to throw :5
5  5  5  6  4
Enter the number of dice to throw :10
2  5  6  1  6  5  1  4  4  6
Enter the number of dice to throw :0
```

4.4 Why write your own functions

All the programs that we have seen till now where very small and come no where near the size of practical applications. When the programs size increase the complexity of the program increases as well and it starts to get difficult to manage this complexity. Programs like the ones we have gone through are “monolithic code”, the program execution flows from start to finish, the these types of programs are like one big block of code,when the size of a monolithic program grows it’s complexity starts to get out of hand. The solution to this is to divide the code into several almost independent smaller pieces with very specific roles. This division of a programs complexity into smaller pieces can be done using functions. A program can be made up of multiple functions, each one having the task of addressing a part of the problem and when used together solves the problem as a whole.

One big and complex monolithic code is avoided for several reasons.

- **Difficulty writing correctly:** When writing any statement, the details of the entire code must be considered. This becomes unmanageable for programmers beyond a certain point of complexity.
- **Difficulty debugging:** The larger a code block is, the harder it is to find the source of an error or an exception in that block of code. The effect of an erroneous statement might not be know until later in the program a correct statement uses the result of the erroneous statement.
- **Difficulty in modification:** Before a code sequence can be modified to achieve a different result, this code sequence needs to be understood, which becomes a very heavy task as the size of code sequence increases.

Our code can be made more manageable by dividing our code into parts and rewriting these parts of our code as functions, calling them in the program when and as needed. Using this divide and conquer strategy, a complicated code block can now be be broken down into simpler code blocks each being handled by a function. The programmer can now accomplish the same original task by delegating tasks to functions, this way a results in more manageable code blocks that are easy to read, write, debug and modify. Besides these advantages of structuring the code, functions provide several other advantages like reusing same code block for the same kind of task(reusability). Functions that provides similar functionality can be bundled together into reusable parts.

Although it’s time saving to use library functions, it is sometimes necessary to write our own functions for custom behaviour. Fortunately python allows programmers to create their own custom functions and use them.

If the purpose of the custom functions are general enough and written in a proper manner then this function can even be used in other programs. In the following sections we’ll take the first steps in writing functions in python.

4.5 Functions basics

Every python function has

1. **Function definition:** A functions definition describes every detail about that function including how many parameters and what type of parameters it takes, what the function does and also what the function returns.
2. **Function invocation(call):** After a function is defined, it can be called from anywhere in the program by passing the appropriate parameters to the function. A function can have only one definition but it can be invoked any number of times.

A function definition has three parts, they are listed below.

- **Name:** The name of a function is what is used to identify the block of code that this function contains. Function names are also identifiers, so they follow all the rules of python identifiers. One of the best ways to name a function is to give it the name of the task it is designed to do.
- **Parameters:** The parameters that the function accepts from callers is specified in the function definition. They appear with a set of parentheses(( )) in a comma(,) separated list. The list of parameters can be left empty if the functions definition does not specify parameters to passed, these types of function do not require information from the caller to perform the task it was designed to carry out.
- **Body:** The block of indented statements that follow after the functions header is the functions body. This is what gets executed when the function call is made. If the function is required to produce a result and return it to the client, the `return` statement can be used to return the result back to the caller.

The syntax of a function definition is given below.

```
def <function_name>(<parameter list>):

    """<docstring>"""

    <statement(s)>
```

The above function definition syntax has the following the components.

- `def` is the keyword that signifies the start of the header of a function
- `<function_name>` is the unique name that identifies the the body of the function
- `<parameter list>` is used to pass arguments to the function. They are optional.

- “`:`” signifies the end of the function header
- `<"""docstring""">` is the documentation string that can be used to describe the purpose and functionality of the function
- `<statement(s)>` are the valid python statements that make up the body of the function. All the statements in the function body have the same indentation usually four spaces(“   ”) are used.
- A `return` statement is optional and it is used to return the result back to the caller if any

The example code below illustrates a simple function definition and calling this defined function.

```
def greet(name):
    """This function prints a hello and doesn't return anything"""
    print("Hello " + name)

greet("khalid")

```

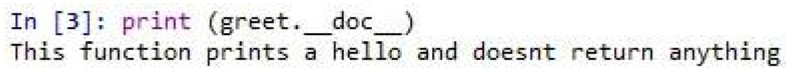
In the above code example a function “greet” is defined that takes a single string(name) as a parameter and prints a specified string onto the prompt. Then the defined function is immediately called after it’s definition, this executes the body of the greet function with the passed arguments.

4.5.1 Docstring

The first line after the function header optionally has a string called the “docstring” that is sort for “document string”. The “docstring” is used for explaining what the function does.

If you have already programmed in other languages you would be aware how programmers tend to forget what their code does, so it’s always a best practice to document your code. Even if you may not use the documentation to refresh your memory on what the function does, it can help others who will be working with your code.

The docstring of a function can be accessed with “.__doc__” after a function name. So if the following code is entered into the python shell it returns,



```
In [3]: print (greet.__doc__)
This function prints a hello and doesnt return anything
```

4.5.2 The return statement

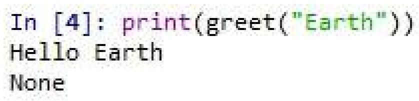
The return statement for functions is like how the break statement is for loops in a way that they both takes the control out of the block they were encountered at. The return statement exits the function, passes the control back to the place it was called from the calling code additionally the return statement contains followed by an expression list that is evaluated and returned back to the caller.

The syntax of the return statement is given below.

```
return <expression_list>
```

Here the expression in the `return` statement is evaluated and returned back to the client. In the case that there is no expression provided or the `return` statement itself is skipped then a special object `none` is returned back to the calling client.

For example try the following code in the python shell. Print the value that is return by the greet function.



```
In [4]: print(greet("Earth"))
Hello Earth
None
```

In the above call to print, which is passed a call to the greet function as the parameter. First the greet method is called which prints to the prompt then the greet method returns a “none” object back to the print function, the print function then prints the “none” object

Lets take a look at an example that illustrates the `return` statement. The program below defines and uses a function that returns the absolute value of an number that’s passed to it. You can see how the return evaluates the expression given to it and returns the result as well as the control back to the caller. Remember that an expression is any combination of values, variables, operators, function calls. Expressions always return an object. If there is no return statement in a function the execution of the function ends when the last statement of the functions code block has been executed and the “none” object as well as the control is returned back to the caller.

```
#===============================================================
# Evaluate absolute value of a number
#===============================================================
def absolute_value(num):
    """This function returns the absolute
```

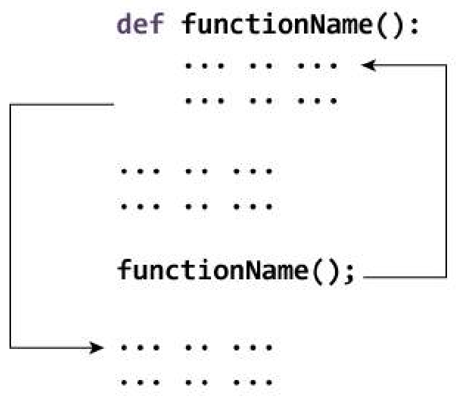
```python
    value of the number passed as an argument"""

    if num >= 0:
        return num
    else:
        return (-num)

userInput = eval(input("Enter a number: ")) # evaluate users input

print("It's absolute value is " + str(absolute_value(userInput)))

```

The output for the above program for the value "-56" is shown below.

```
Enter a number: -56
It's absolute value is 56
```

4.5.3 Flow of control at function call and return

The below figure illustrates the control flow when a function call is made and when the the execution of the functions code block completes.



```
def functionName():
    ... .. ...
    ... .. ...

... .. ...
... .. ...

functionName();

... .. ...
... .. ...
```

When a function call is made the control shifts to the body of the called function, the only two ways that control can exit from a function is by the return statement or by executing the last statement in the body of the function.

4.5.4 The scope and lifetime of variables

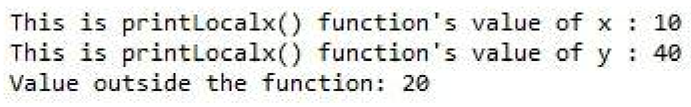
Scope of a variable in a program is the part of the program that can recognize and use the object referred to by that variable. Parameters of a function are considered to be local to that function(local variables) and so they are in the scope of that functions code block but outside the scope of the rest of the program. A functions local variables can't be used by the parts of the program outside that functions code block hence they have local scope.

Lifetime of a variable is the period through which the variable exists in the memory. The lifetime of variables inside functions(i.e local variables) is as long as the function executes. After the function completes it's execution all it's variables are deleted and their associated objects are garbage collected, and because of this functions don't have any memory between two function calls, they don't know what the value of local variable was in the previous call.

The example code below illustrates the scope of a variable inside a function.

```
1  #=======================================================================
2  # This program illustrates the scope of a variable inside a function
3  #=======================================================================
4  def printLocalx():
5      x = 10     # Local variable, scope lines 2 - 4
6      print("This is printLocalx() function's value of x: ", x)
7      print("This is printLocalx() function's value of y: ", y)
8      # Local variable x is deleted once the control flow reaches this point
9
10 x = 20         # Global variable, scope lines 1 - 13
11 y = 40         # Global variable, scope lines 1 - 13
12 printLocalx()
13 print("Value outside the function:", x)
14
```

The above program produces the below output when executed. Observe that even though the the global variable `x` is in the scope of `printLocalx()` function, the function prefers the value of it's local variable when there is a match.



```
This is printLocalx() function's value of x : 10
This is printLocalx() function's value of y : 40
Value outside the function: 20
```

Even though function `printLocalx()` changes the value of the the variable x, it does not affect the global variable `x`. This is because variable `x` inside the `printLocalx()` function is different from the one outside of it, so remember that they are two different variables with different scopes.

In Python the `global` keyword allows modification variables that are outside the current scope. The can be used to create global variables and modify it in the local context.

Things to remember when using the global keyword.

1. A variable is by default a local to a function when it is declared inside the function
2. A variable is by default a global variable when its declared outside a function. It makes no difference using the global keyword on variables defined outside functions.
3. The global keyword can be used inside functions to read and write global variables inside functions.

Lets take a look at some examples that illustrate the use and working of the global keyword.

```
x = 20        # x is a global variable

def printx():
    print(x) # Since there is no local variable x, the global variable x
             # can be accessed

printx()

```

This example code produces the below output.

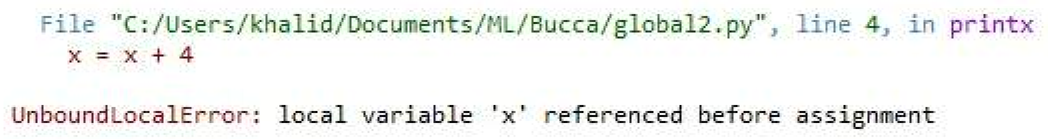
```
20
```

Here, notice that as the function `printx()` doesn't have a local variable 'x', it can read the value of the global variable 'x', but the interpreter raises an error when we try to write to the same variable.

```
x = 20        # x is a global variable

```

```python
def printx():
    x = x + 4 # Try using the value of a global variable.
              # Local variable x is referenced before
              # it's assignment
    print(x)

printx()

```

When the above program is ran it raises the error below,because the value of a global variable is being modified in the function.without using the global keyword it is only possible to read the value of a global variable inside a function



```
  File "C:/Users/khalid/Documents/ML/Bucca/global2.py", line 4, in printx
    x = x + 4

UnboundLocalError: local variable 'x' referenced before assignment
```

The `global` keyword can be used to overcome this kind of a problem where the modifications to global variables from within a function is required. The example below shows how a global keyword can be used to modify a global variable from within a function. To modify a global variable from inside a function this variable has to first to be declared as `global` inside the function it needs to be used in. This is illustrated in the example below.

```python
x = 20       # x is a global variable

def printx():
    global x  # Now the global variable x can be modified
    x = x + 4 # Try using the value of the global variable.
             # you'll see there is no error raised this time
    print("This is the value of x inside function printx(): %i" %x)

printx()
print("This is the value of x outside function printx(): %i" %x)

```

The output of the above program is given below. notice that the value of the variable ‘x’ is the same inside and outside the `printx()` function. This is because variable ‘x’ inside the `printx()` function now refers to the same object as the variable ‘x’ outside this function.

```
This is the value of x inside printx: 24
This is the value of x outside printx: 24
```

More on the global and local scopes will be discussed in the following sections.

4.5.5 Arguments

Functions may take any number of parameters(No upper or lower bound) to perform the task that it was designed for. Before calling a user defined function, this function must be defined and it’s function definition must specify the parameters it needs. When a function is called by a client code, the client caller must provide the same number of arguments as specified in the function definition. if there is any mismatch in the number of arguments passed then what was expected then the python interpreter raises an error. The examples below executed in the python shell show the errors raised when there are issues with the arguments passed to a function call

```
In [1]: def square(x): return x * x  # Takes 1 integer and returns it's square

In [2]: square(9, 8 , 0)            # Number of arguments don't match
Traceback (most recent call last):

  File "<ipython-input-2-50d9f47ac714>", line 1, in <module>
    square(9, 8 , 0)            # Number of arguments don't match

TypeError: square() takes 1 positional argument but 3 were given


In [3]:

In [3]: square(9)                   # A valid call
Out[3]: 81

In [4]: square("Can I be passed?")  # Valid call, error raised from function body
Traceback (most recent call last):

  File "<ipython-input-4-6f0de8c10498>", line 1, in <module>
    square("Can I be passed?")  # Valid call, error raised from function body


  File "<ipython-input-1-b50f9e4cba73>", line 1, in square
    def square(x): return x * x  # Takes 1 integer and returns it's square

TypeError: can't multiply sequence by non-int of type 'str'
```

Observe that the number of arguments to pass are compulsory but the types of the arguments are not. Although passing an argument of a different type than what is expected would most likely end up raising an error.

Python also allows it's users to define functions that take a variable number of parameters. Functions that take a variable number of parameters can be defined with the following types of arguments. We'll discuss each of them in the following section.

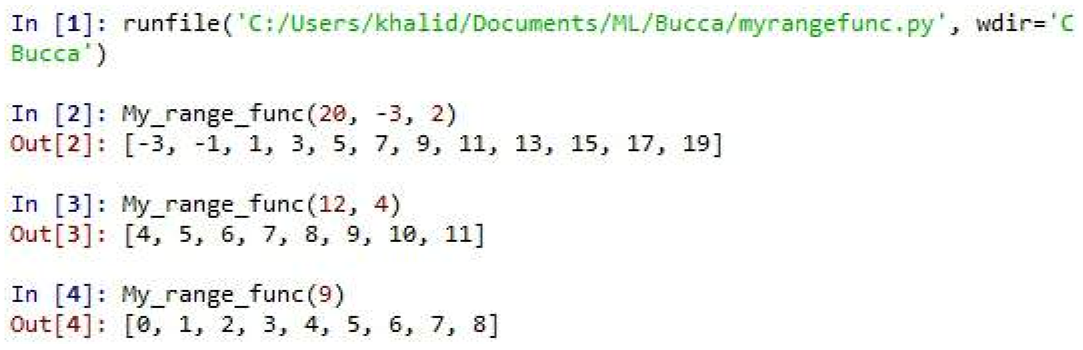
1. Default arguments
2. Keyword arguments
3. Arbitrary arguments

Default and non-default arguments

Python allows to assign default values for function parameters A default value for a parameter can be given with the "=" operator. When a parameter is assigned a default value, its argument can be skipped in the function call in which case the default value provided in the function definition gets assigned instead. In the example below we create our own range function that in turn calls python's standard range function. This example illustrates python's default arguments.

```python
def My_range_func(stop, start = 0, step = 1):
    """
    This function in turn calls
    the standard range function
    and returns it's result as
    a list
    """

    li = list(range(start, stop, step))
    return li

```

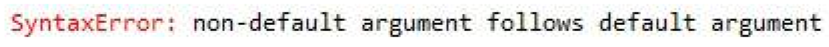
Now when this program is executed, try making calls to this function with different parameters from the python shell. Observe that the function call can be made with one, two or three parameters, this is because the second and third arguments are default arguments are default arguments and when their values are not provided they assume their default values. Here it is necessary to pass a value for parameter `stop` and failing to do so would raise an error. For illustration there are some function calls to the `My_range_func()` given below. These function calls where made from the python shell.



```
In [1]: runfile('C:/Users/khalid/Documents/ML/Bucca/myrangefunc.py', wdir='C
Bucca')

In [2]: My_range_func(20, -3, 2)
Out[2]: [-3, -1, 1, 3, 5, 7, 9, 11, 13, 15, 17, 19]

In [3]: My_range_func(12, 4)
Out[3]: [4, 5, 6, 7, 8, 9, 10, 11]

In [4]: My_range_func(9)
Out[4]: [0, 1, 2, 3, 4, 5, 6, 7, 8]
```

Here the parameter `start` does not have a default value and is mandatory to provide in a call. Default values `0` and `1` are provided for parameters `start` and `step` and so they're value can be skipped in a function call but if provided they overwrite the default values. Any number of parameters in a function can have a default value but it must be made sure that all default values are pushed to the right of the parameter list. This is needed as without it there would be difficulty differentiating between default and compulsory arguments in a function call. To ensure this the python interpreter raises an error when it encounters a function header that doesnt have all default parameters pushed to the right. An example is given below.

```python
def My_range_func( start = 0, stop, step = 1): # This raises an error

```



```
SyntaxError: non-default argument follows default argument
```

Keyword and positional arguments

When a function is called by passing arguments to it, the arguments are assigned to the parameters based on it's position in the argument list which is from left to right. For instance in the above function `My_range_func()` when `(20, -3, 2)` is passed as the argument list, the assignment occurs in the following order.

1. `20` gets assigned to variable `stop`

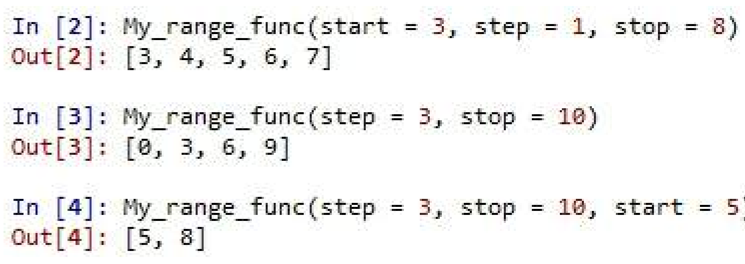
2. `-3` gets assigned to variable `start`

3. `2` gets assigned to variable `step`

These arguments are positional arguments as the position of the arguments matter in the function call. Python allows re-ordering of the arguments that are passed if the arguments are passed as keyword arguments. When function call is made with keyword arguments the arguments can be passed in any order. The general syntax of argument list when passing arguments as keyword arguments is given below.

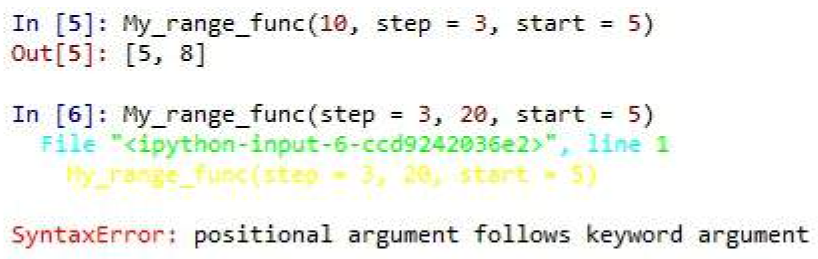
```
<function_name>(<ParamterA> = <value>, <ParameterB> = <value>,...)
```

The following are calls to the previously defined `My_range_func()` function and they are all valid.



```
In [2]: My_range_func(start = 3, step = 1, stop = 8)
Out[2]: [3, 4, 5, 6, 7]

In [3]: My_range_func(step = 3, stop = 10)
Out[3]: [0, 3, 6, 9]

In [4]: My_range_func(step = 3, stop = 10, start = 5)
Out[4]: [5, 8]
```

A function call can be made with a mixture of positional and keyword arguments but in this case the keyword arguments must be pushed to the right side of the argument list which means there should not be any positional arguments that follow keyword arguments in the argument list of a function call. The example below show a valid and invalid instance of making function calls with combination of both positional and keyword arguments.



```
In [5]: My_range_func(10, step = 3, start = 5)
Out[5]: [5, 8]

In [6]: My_range_func(step = 3, 20, start = 5)
  File "<ipython-input-6-ccd9242036e2>", line 1
    My_range_func(step = 3, 20, start = 5)

SyntaxError: positional argument follows keyword argument
```

Arbitrary arguments

There are many cases where the number of arguments that will be passed during a function call is unknown. An example of this case is a function that simply prints every argument it receives,but the number of arguments it is passed is arbitrary. An arbitrary parameter can be specified in the function definition by adding a '*' before the parameter which says that this parameter is a tuple, this tuple will contain the arbitrary number of arguments that will be passed. Lets take a look at an example that illustrates arbitrary arguments in python.

```python
def sayHiTo(*people):
   """This function says hi
   to everyone in the people tuple."""

   # names is a tuple that contains all the arguments
   for person in people:
       print("Hi " + person + "!")

sayHiTo("khalid", "guido von rossum")

```

The above program defines a function that takes an arbitrary number of arguments that is available to the function through the "people" tuple. "people" tuple is a tuple that consists of all the arbitrary arguments that are passed to the function when it's called. "Hi <person name>!" is printed for each element of the "people" tuple. The output of the above program is given below.

```
Hi khalid!
Hi guido von rossum!
```

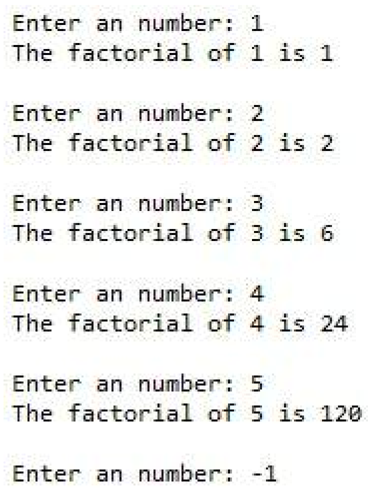
4.5.6 Recursive functions

We already know that functions in python can call other functions, which means there also can be a case where a function calls itself. These kind of functions are called as "recursive functions" as they are defined in terms of themselves. A very common example that is used to explain recursive functions is the factorial program(program that returns the factorial of a given number). The factorial of a number is the product of all integers from one to itself

For example factorial of five represented as "`5!`" is equal to "`1 * 2 * 3 * 4 * 5`". Lets take a look at an example that defines a recursive function to return the factorial of a given number.

```python
#==========================================================
# Python program to calculates the factorial of a number
#==========================================================

def factorial(x):
    """This is a recursive function
    that calculates the factorial of
    an integer"""

    if x == 1: # recursive calls terminates when this condition is met
        return 1
    else:
        return (x * factorial(x-1)) # The function calls itself

while 1:
    num = eval(input("Enter an number: "))
    if num == -1:
        break

    print("The factorial of", num, "is", factorial(num))
```

In this example the factorial() function is recursive as it makes a call to itself in the function body. Whenever the factorial function is called, it recursively calls itself decrementing the number passed as an argument each time until the terminating condition is met that is “`x == 1`” The results are continually evaluated from the last call to the first and the result of each function call is returned to it’s client caller until the initial call is reached. The a possible output of the above program is given below.

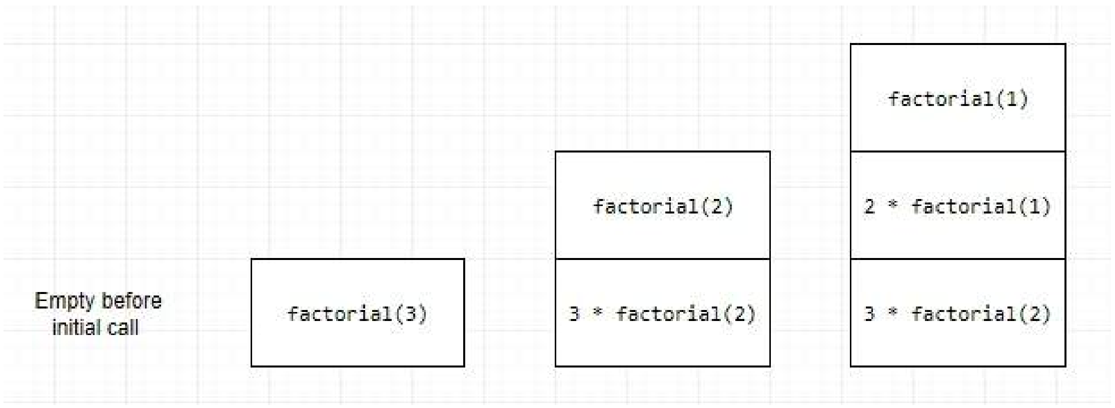


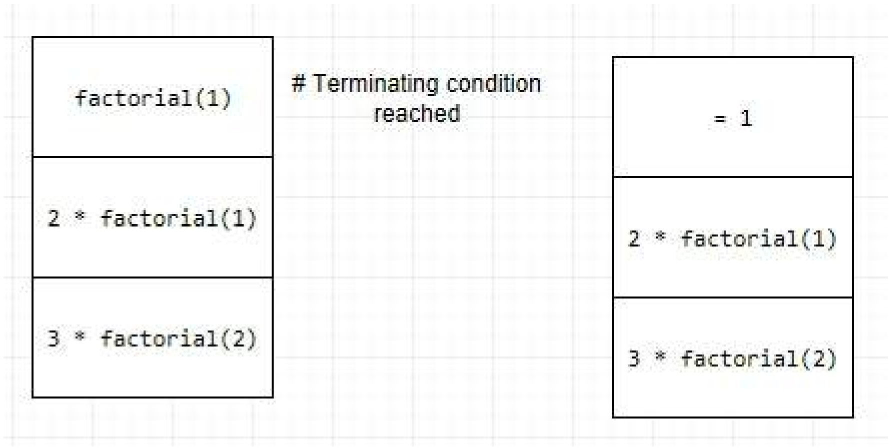
```
Enter an number: 1
The factorial of 1 is 1

Enter an number: 2
The factorial of 2 is 2

Enter an number: 3
The factorial of 3 is 6

Enter an number: 4
The factorial of 4 is 24

Enter an number: 5
The factorial of 5 is 120

Enter an number: -1
```

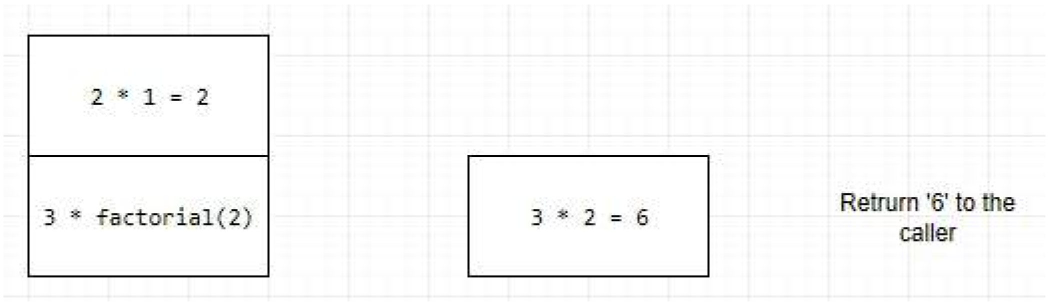
The following illustration shows the method of evaluation when the `factorial()` function is called with `5` passed as the argument.

```
factorial(4)                # initial call with 4

4 * factorial(3)           # second call with 3

4 * 3 * factorial(2)       # third call with 2

4 * 3 * 2 * factorial(1)    # fourth call with 1

4 * 3 * 2 * 1               # fourth call returns 1

4 * 3 * 2                  # third call returns 2 * 1 = 2

4 * 6                      # second call returns 3 * 2 = 6

24                         # return from initial call 4 * 6 = 24

                           # 24 is the value returned by factorial(4)
```

Recursive function calls can be easily understood in terms of pushing and popping elements from a stack, where each call to a recursive function is pushed to the top of the stack. After the last insertion to the stack has been made the elements of the stack are popped one by one till all the elements have been popped. Before popping an element at the top of the stack the function is evaluated and the if there is a any value returned by that function it is returned to it’s caller in the next top of the stack. This order of evaluation continues till there are no more elements in the stack and the if any value is returned by the last element of the stack it is returned to the initial caller of the recursive function. Lets try to visualize this with the function call `factorial(3)`.







Advantages of Recursion:

- The use of recursive functions make the code more elegant and readable.
- Recursive functions can be used to break a complex problem down to simpler problems of a repetitive kind.
- Generating a sequence is easier using recursion than using nested iteration.

Disadvantages of Recursion:

- Sometimes the logic of recursive code can be hard to follow or understand.
- Recursive calls take up more memory than it's iterative version.
- Harder to debug
-

4.5.7 Lambda functions

In this section we'll take a look at what lambda functions are in python and some examples using lambda functions.

Python allows creating functions without a name called lambda functions. Like normal functions are defined using the `def` keyword lambda functions are defined using the `lambda` keyword. Lambda functions are also called as "anonymous functions". The syntax of lambda functions in python are given below.

```
Lambda <arguments>: <Expression>
```

Lambda functions are allowed to have any number of arguments but can have only one expression as it's body. When a lambda function is called the expression is evaluated and returned. Lambda functions are used whenever function objects are needed. Let's take a look at an example that defines and uses a lambda function.

```python
# Program that defines and uses a lambda function that
# returns the cube of number

cube = lambda a: a * a * a # Defining the lambda function

num = eval(input("Enter a number: "))
print(cube(num))          # Using the lambda function

```

In the above code `lambda a: a * a * a` where "a" is the argument and "a * a * a" is the expression that is evaluated and returned. This lambda function has no name that's why its called an anonymous function. The identifier cube returns the function object that is assigned to it, so it can be called a normal function.

The statement

```
cube = lambda a: a * a * a
```

Has the same meaning as the below function definition

```
def cube(a):

   return a * a * a
```

Lambda functions are usually used when a nameless function is needed for a short time. They are generally used as an argument to functions that take other functions as parameters(higher order functions). Lets go through two examples that show the practical use of lambda functions in python.

Using lambda functions in the `filter()` function

The `filter()` function in python takes a function and a list for parameters. The provided function is called for all the elements of the provided list and a new list is returned with all the elements that returned true when passed to the provided function.

The example below shows the calls the “filter” function to filters out all the zero’s from the provided list and returns a new list object with only non zero elements in the provided list.

```
#==========================================================
# Program to filter out zero's from a list
#==========================================================
my_list = [10, 0, 7, 76, 0, 0, 39, 87]

li = list(filter(lambda a: a != 0 , my_list))

print(li)

```

The above program prints the output given below when executed.

```
[10, 7, 76, 39, 87]
```

Using lambda functions in the `map()` function

pythons map() function takes in a function object and a list object as it’s arguments and returns the a list object containing all the values returned by the provided function when it’s called on every element in the provided list.

The example code below makes a call to the map() function to return the square of all the elements in the provided list.

```
#==========================================================
# Program to square all the elements of a given list
```

```python
#============================================================
my_list = [1, 2, 4, 8, 16, 32, 64, 128]

li = list(map(lambda x: x * x, my_list))

print(li)

```

The output of this program is shown below.

```
[1, 4, 16, 64, 256, 1024, 4096, 16384]
```

4.5.8 Global, local and non local variables

Lifetime and scope of variables has already been introduces in the previous chapters, now let’s take a look at global, local and non local variables in detail.

Global variables

The parts of a python program that exists at the surface level i.e outside of any internal code blocks like “if block”, “while block”, “function definitions”, etc are said to be the global scope and any variables that are created at that level are called global variables. Variables declared in the global scope can be accessed from any part of the program as long as they are alive. In this example, ‘x’ is declared in the global scope so it’s a global variable and can be accessed from any part of the program this ability of global variable x is shown by by defining a function that uses the value of ‘x’ without it having an ‘x’ as parameter nor as a variable declared inside the function.

```python
#----------------------------------------------------Global scope
x = 10     # Global variable
y = 20     # Global variable
z = 30     # Global variable
def foo(): # foo's definition, part of the global scope
#----------------------------------------------------Global scope
    #------------------foo's local scope starts
    a = 0 # foo's local variable
    print(a, x, y, z, b) # foo can access global variables
    #------------------foo's local scope ends
```

```python

#---------------------------------------------------Global scope
b = 40      # Global variable
foo()       # making a call to function foo

#---------------------------------------------------Global scope

```

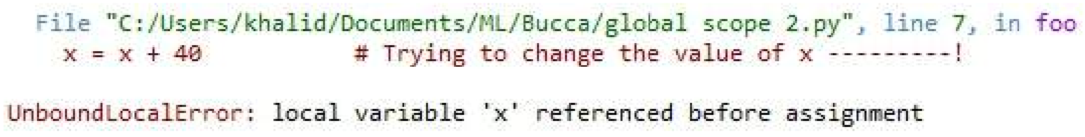
This program prints the output below. Note that the function foo can read the global variables.

```
0 10 20 30 40
```

Take a look at an interesting case where the python interpreter raises an error because it misunderstands what is actually meant.

```python
#-------------------------------------------------------Global scope
x = 10      # Global variable
def foo():  # foo's definition, part of the global scope
#-------------------------------------------------------Global scope
    #-----------------------------------foo's local scope starts
    x = x + 40          # Trying to change the value of x ---------!
    a = 0              # foo's local variable
    print(a, x)         # foo can access global variables
    #-----------------------------------foo's local scope ends
#-------------------------------------------------------Global scope
foo()       # Calling foo
#-------------------------------------------------------Global scope

```

When the above code is executed the interpreter raises an error because, in the normal case when an assignment statement is encountered by the interpreter in a local scope, the name on the right hand side of the assignment is considered to be local variable of that local scope. So when the interpreter encounters the statement "`x = x + 40`", it checks the left hand side, creates a new name 'x' in the local scope of the function "foo" then checks the expression on the right hand side of the assignment statement tries to access the non existent object associated to the local variable 'x'. it raises an error because 'x' is referenced in an expression before it's assigned a value. When the above code is executed, it gives the following output.



```
  File "C:/Users/khalid/Documents/ML/Bucca/global scope 2.py", line 7, in foo
    x = x + 40           # Trying to change the value of x ---------!

UnboundLocalError: local variable 'x' referenced before assignment
```

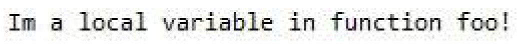
To solve this problem of not being able to modify a global variable, python provides a global variable which will be discussed in the sections that follow.

Local Variables

A variable that is declared inside a local scope is a local variable, a variable local to the scope that it was created in. A local variable exists only in the scope that it was defined in and can not be accessed outside of that scope. Local variable are created just like global variables the only difference being that local variables are declared inside functions or are arguments of a function call and global variables are declared at the global scope of a python program. The example below creates a local variable inside function “foo”.

```
def foo():  # foo's definition
    #------------------------------------foo's local scope starts
    a = "I’m a local variable in function foo!" # foo's local variable
    print(a)
    #------------------------------------foo's local scope ends

foo()

```

The output prints the output given below.



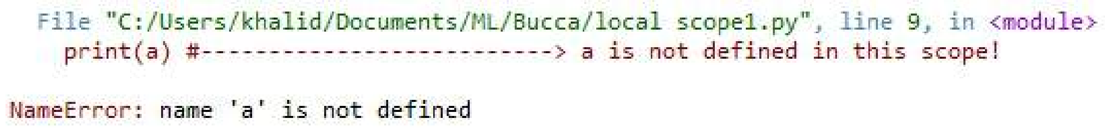
```
Im a local variable in function foo!
```

In the example below the variable ‘a’ that is local to the function “foo” is accessed outside of the scope of the variable which results in an error.

```
def foo():  # foo's definition
    #------------------------------------foo's local scope starts
    a = 0              # foo's local variable
    print(a)
    #------------------------------------foo's local scope ends

```

```python
#-------------------------------------------------Global scope

print(a) #---------------------------> a is not defined in this scope!

#-------------------------------------------------Global scope

```

The following error is raised when the above program is executed as the variable ‘a’ is defined in the local scope of function “foo” and is available for use only there, it can’t be accessed by the upper levels. This program raises the error shown below.



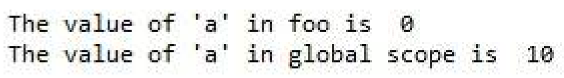
```
  File "C:/Users/khalid/Documents/ML/Bucca/local scope1.py", line 9, in <module>
    print(a) #--------------------------> a is not defined in this scope!

NameError: name 'a' is not defined
```

Global and local variables with the same name

When a variable is declared in the local scope of a function that has the same name as a variable in the global scope, which variable gets accessed? In such cases the python prefers the local variable over the global variable, so when a variable name in the local scope is accessed that has the same name as a variable in the global scope the object associated to that name in the local scope is returned. The code example below shows an instance of this case.

```python
#--------------------------------------------------------Global scope
a = 10          # Global variable
def foo():       # foo's definition, part of the global scope
#--------------------------------------------------------Global scope
    #-------------------------------------foo's local scope starts
    a = 0        # foo's local variable
    return a     # The local variable a is accessed (a = 0)
    #-------------------------------------foo's local scope ends

#--------------------------------------------------------Global scope
print("The value of 'a' in foo is ",foo())
print("The value of 'a' in global scope is ", a)
#--------------------------------------------------------Global scope

```

In the above code, the same name ‘a’ is used both in the global scope as well as in the local scope of function “foo”. When the values of both variables are printed we get different values as the local variables preferred over global if there is a name match



```
The value of 'a' in foo is  0
The value of 'a' in global scope is  10
```

Lets look back at the previous example where we try to change the value of a global variable inside a function. This causes an error to be raised by the python interpreter, and it’s already been mentioned that the solution python provides for this is the “global” keyword

In Python the `global` keyword allows modification variables that are outside the current scope. They can be used to create global variables and modify it in the local context.

Here are some things to remember while using the `global` keyword.

1. A variable is by default a local to a function when it is declared inside the function

2. A variable is by default a global variable when its declared outside a function. It makes no difference using the global keyword on variables defined outside functions.

3. The global keyword can be used inside functions to read and write global variables inside functions.

Lets take a look at some examples that illustrate the use and working of the global keyword.

```python
x = 20       # x is a global variable

def printx():
    print(x)  # Since there is no local variable x, the global variable x
              # can be accessed

printx()

```

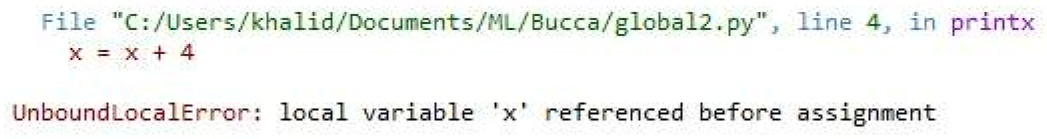
This example code produces the below output.

```
20
```

Here, notice that as the function `printx()` doesn't have a local variable 'x', it can read the value of the global variable 'x', but the interpreter raises an error when we try to write to the same variable.

```python
x = 20 # x is a global variable

def printx():
    x = x + 4  # try using the value of the global variable
    print(x)

printx()

```

When the above program is ran it raises the error below,because the value of a global variable is being modified in the function.without using the global keyword it is only possible to read the value of a global variable inside a function

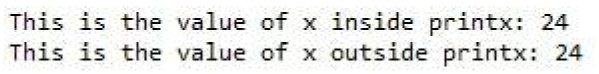


```
  File "C:/Users/khalid/Documents/ML/Bucca/global2.py", line 4, in printx
    x = x + 4

UnboundLocalError: local variable 'x' referenced before assignment
```

The `global` keyword can be used to overcome this kind of a problem where the modifications to global variables from within a function is required. The example below shows how a global keyword can be used to modify a global variable from within a function. To modify a global variable from inside a function this variable has to first to be declared as `global` inside the function it needs to be used in. This is illustrated in the example below.

```python
x = 20        # x is a global variable

def printx():
    global x   # Now the global variable x can be modified
    x = x + 4  # No error raised
    print("This is the value of x inside printx: %i" %x)

printx()
print("This is the value of x outside printx: %i" %x)
```

The output of the above program is given below. notice that the value of the variable 'x' is the same inside and outside the `printx()` function. This is because variable 'x' inside the `printx()` function now refers to the same object as the variable 'x' outside this function.

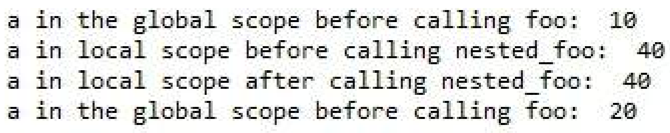


```
This is the value of x inside printx: 24
This is the value of x outside printx: 24
```

The `global` keyword can also be used in nested functions, let's take a look at an example of using the `global` keyword in a nested function.

```python
a = 10
print("a in the global scope before calling foo: ", a)

def foo():
    a = 40
    print("a in local scope before calling nested_foo: ", a)

    def nested_foo():
        global a
        a = 20

    nested_foo()
    print("a in local scope after calling nested_foo: ", a)

foo()
print("a in the global scope before calling foo: ", a)

```

When the global keyword is used on the variable 'a' in the `nested_foo()` function, any change that this function makes to the variable 'a' in it's scope affects the variable at the global scope. The output of the above program is given below.



```
a in the global scope before calling foo:  10
a in local scope before calling nested_foo:  40
a in local scope after calling nested_foo:  40
a in the global scope before calling foo:  20
```

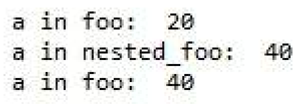
Non local variables

Non local variables are used in nested functions where the local scope is not defined, so the variable can't be in either the local nor the global scope.

The keyword `nonlocal` is used to declare a non local variable in python.let's take a look at an example that explains non local variables in python.

```python
def foo():
    #------------------------------------foo's local scope starts
    a = 20          # foo's local variable
    print("a in foo: ", a)
    #------------------------------------foo's local scope ends
    def nested_foo():
        nonlocal a   # Now 'a' is a non local variable
        a = a + 20
        print("a in nested_foo: ", a)
    nested_foo()
    print("a in foo: ", a)
#-------------------------------------------------Global scope
foo()
#-------------------------------------------------Global scope
```

The output of the above program is given below.



```
a in foo:  20
a in nested_foo:  40
a in foo:  40
```

Here the function `nested_foo()` is defined in the local scope of the function `foo()`. The `nonlocal` keyword is used to declare 'a' to be a non local variable. Observe that when a nonlocal variable is changed, this change also reflects on the local variable 'a' of function `foo()`. This is because the variable 'a' which is in the local scope of function `foo()` is now changed by function `nested_foo()`.

4.6 Exercises

1. Suppose you need to compute the square root of a number in a Python program. Would it be a good idea to write the code to perform the square root calculation? Why or why not?

2. Which of the following values could be produced by the call random.randrange(0, 100) function?

4.5 **34** **-1** **100** **0**
99

3. Classify each of the following expressions as legal or illegal. Each expression represents a call to a standard Python library function.

   (a) math.sqrt(4.5)

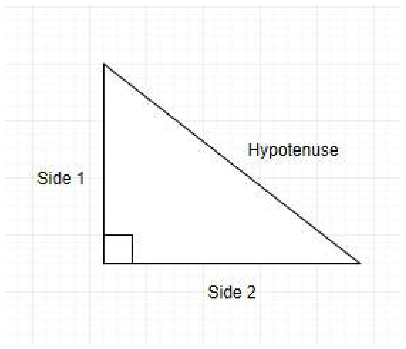
   (b) math.sqrt(4.5, 3.1)

   (c) random.rand(4)

   (d) random.seed()

   (e) random.seed(-1)

4. From geometry: Write a computer program that, given the lengths of the two sides of a right triangle adjacent to the right angle, computes the length of the hypotenuse of the triangle. (See Figure ??.) If you are unsure how to solve the problem mathematically, do a web search for the Pythagorean theorem.



5. Write a guessing game program in which the computer chooses at random an integer in the range 1. . . 100. The user's goal is to guess the number in the least

number of tries. For each incorrect guess the user provides, the computer provides feedback whether the user's number is too high or too low.

6. Extend Problem 5 by keeping track of the number of guesses the user needed to get the correct answer. Report the number of guesses at the end of the game.

7. Extend Problem 6 by measuring how much time it takes for the user to guess the correct answer. Report the time and number of guesses at the end of the game.

8. Is the following a legal Python program?

```
def proc(x):
return x + 2
def proc(n):
return 2*n + 1
def main():
x = proc(5)
main()
```

9. Is the following a legal Python program?

```
def proc(x):
return x + 2
def main():
x = proc(5)
y = proc(4)
main()
```

10. Is the following a legal Python program?

```
def proc(x):
print(x + 2)
```

```
def main():
x = proc(5)
main()
```

11. Is the following a legal Python program?

```
def proc(x):
print(x + 2)
def main():
proc(5)
main()
```

12. Is the following a legal Python program?

```
def proc(x, y):
return 2*x + y*y
def main():
print(proc(5, 4))
main()
```

13. Is the following a legal Python program?

```
def proc(x, y):
return 2*x + y*y
def main():
print(proc(5))
main()
```

14. Is the following a legal Python program?

```
def proc(x):
return 2*x
def main():
print(proc(5, 4))
main()
```

15. Is the following a legal Python program?

```
def proc(x):
print(2*x*x)
def main():
proc(5)
main()
```

16. The programmer was expecting the following program to print 200. What does it print instead? Why does it print what it does?

```
def proc(x):
x = 2*x*x
def main():
num = 10
proc(num)
print(num)
main()
```

17. Is the following program legal since the variable x is used in two different places (proc and main)? Why or why not?

```
def proc(x):
return 2*x*x
def main():
x = 10
```

```
print(proc(x))
main()
```

18. Is the following program legal since the actual parameter has a different name from the formal parameter (y vs. x)? Why or why not?

```
def proc(x):
return 2*x*x
def main():
y = 10
print(proc(y))
main()
```

19. Complete the following distance function that computes the distance between two geometric points (x1,y1) and (x2,y2):

```
def distance(x1, y1, x2, y2):
...
```

Test it with several points to convince yourself that is correct.

20. What happens if a client passes too many parameters to a function?

5. Pandas

Pandas is one of the most popular python libraries that is used for data analysis and manipulation. It gives a very highly optimized performance. In this chapter we focus on getting a proper understanding of the basics of this python library.It's recommended that you try the examples in this chapter yourself.

5.1 Creating, reading and writing

In this section you will learn how to create your own data and also how to work with already existing data. To use this pandas library in a python program import the pandas module. Remember before installing make sure you have the pandas module installed. The pandas module module is imported and renamed with the line of code shown below.

```python
import pandas as pd

```

5.1.1 Creating data

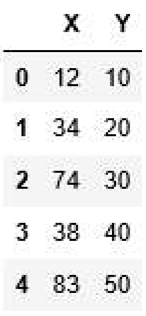
There are two main object types in Pandas and they are "DataFrame" and "Series". Lets go through both of them.

DataFrame

A DataFrame in Pandas can be considered a table. It has an array of entries, each one having a value that corresponds to a column and a record. Let's take a look at a single data frame shown below.

```python
pd.DataFrame({'X': [12, 34, 74, 38, 83], 'Y': [10, 20, 30, 40, 50]})
```

The DataFrame constructor that is invoked in the above code returns the table that is shown below.

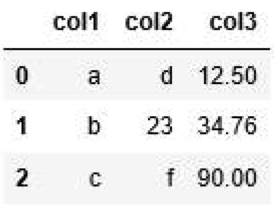


	X	Y
0	12	10
1	34	20
2	74	30
3	38	40
4	83	50

For clarification, the entry at (0, "X") is 12 and the entry at (0, "Y") is 10. The entries of a DataFrame object are not just limited to numeric types they can also be strings like in the example shown below.

```
pd.DataFrame({'col1': ["a", "b", "c"], 'col2': ["d", 23, "f"], 'col3': [12.5,
34.76, 90]})
```

The DataFrame constructor that is invoked in the above code returns the table that is shown below.



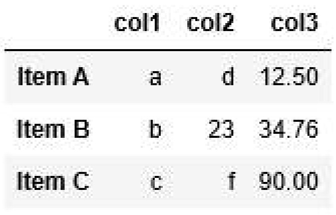
	col1	col2	col3
0	a	d	12.50
1	b	23	34.76
2	c	f	90.00

In the above lines of code we generate the DataFrames using the constructor pd.DataFrame(). This constructor takes a dictionary with keys as columns and the value of the keys as a list of entries, as an argument. This is the standard method for generating DataFrames using pandas and it is the only one that you will most likely use.

The dictionary keys are used to name the columns of the DataFrame.The list of rows labels(also called indexes) are in ascending order from 0 as 0, 1, 2, ans so on. This method of labeling the indexes is fine in many cases but there may be cases where a user defined index is provided in which case the index optional parameter can be used as shown below.

```
pd.DataFrame({'col1': ["a", "b", "c"], 'col2': ["d", 23, "f"], 'col3': [12.5,
34.76, 90]}, index = ["Item A", "Item B", "Item C"])
```

The DataFrame constructor that is invoked in the above code returns the table that is shown below.



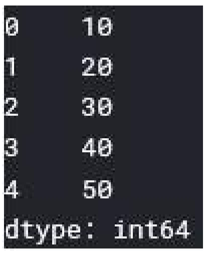
	col1	col2	col3
Item A	a	d	12.50
Item B	b	23	34.76
Item C	c	f	90.00

Series

A series object is simply put, a sequence of values. If a DataFrame object is considered to be a table then a Series object can be considered to be a list. A series object can be created using the Series constructor as shown below.

```
x = pd.Series([10, 20, 30, 40, 50])
x
```

The Series constructor that is invoked in the above code returns the Series object that is shown below.

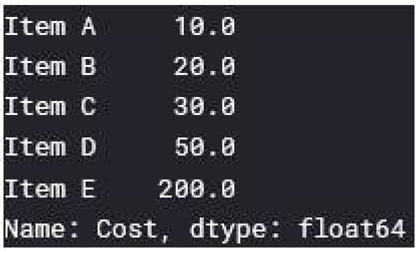


```
0     10
1     20
2     30
3     40
4     50
dtype: int64
```

A Series object is similar to one column of a DataFrame object and therefore row values can be assigned the using same way used in DataFrames, with the index optional parameter. A series object doesnt have column name   but the entire Series object can be given a single name. Take a look at the instruction shown below.

```
x = pd.Series([10, 20, 30, 50.0, 200], index = ["Item A", "Item B", "Item C", "Item D" , "Item E"], name = "Cost")
x
```

The Series constructor that is invoked in the above code returns the Series object that is shown below.



```
Item A     10.0
Item B     20.0
Item C     30.0
Item D     50.0
Item E    200.0
Name: Cost, dtype: float64
```

Series objects are very much related to DataFrames as DataFrames can be considered as a collection of Series.

5.1.2 Reading and writing data files

Creating DataFrames and Series yourself is handy but data scientists rarely work with data that they created. The usual cases is working with data that already exists. Structured data can be stored in many different file formats but one of the most common file formats data scientists use is the csv file which stands for "comma separated values" lets look at an example of how a csv file looks like

```
Object A,Object B,Object C,

78,45,12,

65,13,73,

90,15,23
```

Observe that all the values are separated by a comma. The first line consists of all the column names. The remaining rows are the values of each record and every record has entries for every column specified in the first line. You may also encounter csv files with the column list skipped, in which case the csv file only consists of a list of records

Lets now leave these data sets that we have made aside and start working with real world data sets. The data set that we will be using here can be downloaded through this link https://www.kaggle.com/khalidative/crimeanalysis.

If you have downloaded the dataset, then you will have available to you a set of csv files named

- crimes_by_district
- crimes_by_district_rt
- crimes_by_state
- crimes_by_state_rt

The csv we will be working with is the fourth one named "crimes_by_state_rt.csv". The table below describes the fields of this data set that we will be working with.

Feature Name	Feature Type	Feature Description
STATE/UT	String	Name of the state or union territory(In upper case)
Year	Integer	The year
Murder	Integer	Number of "Murder" cases
Assault on women	Integer	Number of "Assault in women" cases
Kidnapping and Abduction	Integer	Number of "Kidnapping and Abduction" cases
Dacoity	Integer	Number of "Dacoity" cases
Robbery	Integer	Number of "Robbery" cases
Arson	Integer	Number of "Arson" cases
Hurt	Integer	Number of "Hurt" cases
Prevention of atrocities (POA) Act	Integer	Number of cases of violating the "Prevention of atrocities" act
Protection of Civil Rights (PCR) Act	Integer	Number of cases of violating the "Protection of civil rights act" act
Other Crimes Against SCs	Integer	Number of other crimes committed against scheduled castes

Lets start by loading the data in the csv file to a DataFrame object we can do that using the read_csv( ) function of the Pandas library as shown below. This function takes as an argument the name of the csv file.

```
>>> dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv")
```

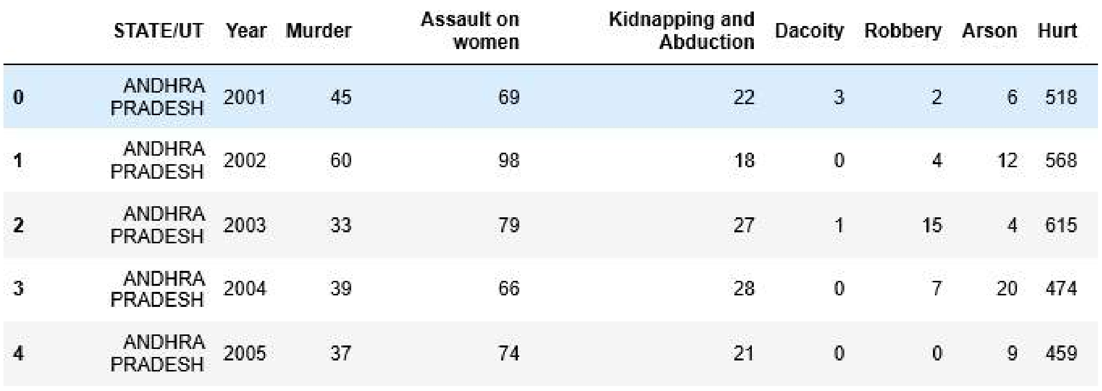
The shape attribute of the data frame can be used to know the size of the data frame lets find out the size of the DataFrame that we just imported.

```
>>> dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv")

>>> dataset.shape
```

Here the value of dataset.shape is a tuple (420, 12) which means that the data frame has 420 rows and 12 columns. The contents of the DataFrame we now have can be examined using the head function, which fetches the first five records of the DataFrame.

```
>>> dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv")

>>> dataset
```

Here dataset.head( ) return the first five lines of the DataFrame shown below.



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
1	ANDHRA PRADESH	2002	60	98	18	0	4	12	568
2	ANDHRA PRADESH	2003	33	79	27	1	15	4	615
3	ANDHRA PRADESH	2004	39	66	28	0	7	20	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	9	459

The read_csv( ) function that Pandas provides has several optional parameters one of which is index_col(takes a numeric value indicating which column). It can be used to provide a column number(starting form 0) to be used as the index column of the dataset instead of using the standard 0, 1, 2, …

Writing a DataFrame to a file

Pandas provides a function `to_csv("<file name>")` to write a DataFrame to the disk as csv file. To write the DataFrame object to the disk we use the to_csv function as shown below.

```
>>> dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv")

>>> dataset.shape

>>> dataset.head()

>>> dataset.to_csv("saved_dataframe.csv")
```

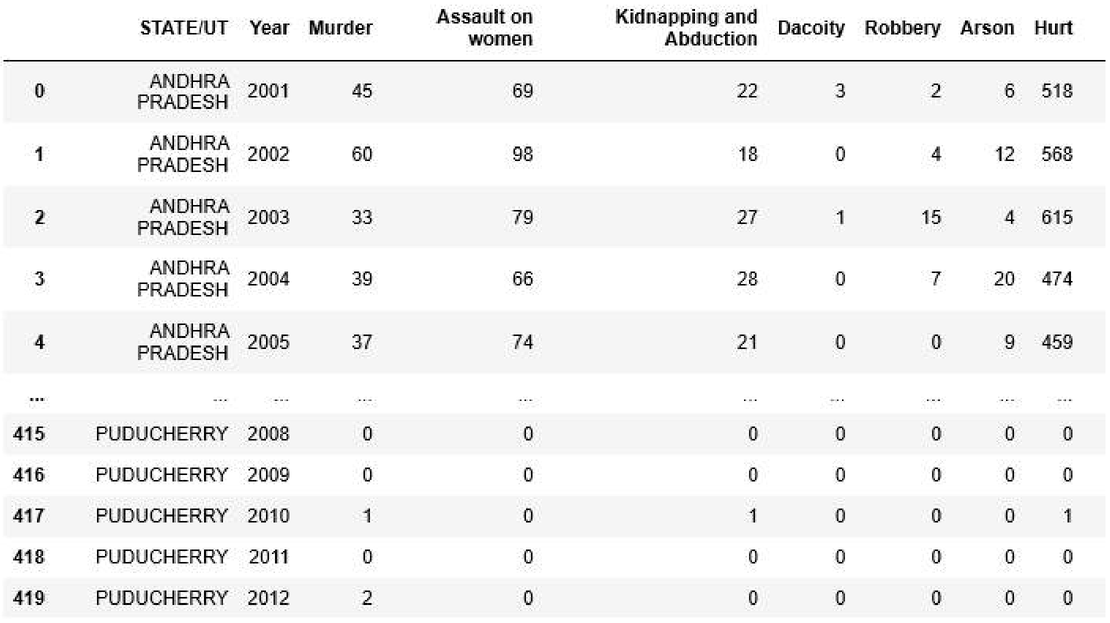
5.2 Indexing, selecting and Assigning

Selecting specific columns, rows, or specific data points from a DataFrame or a Series is a very common step before performing the planned operation, so what you'll learn here is how to select data points that are relevant to your operation easily and effectively.

5.2.1 Naive approach

Python provides its native approach to accessing python objects and this methods are also applicable to instance objects of the Pandas library. Take the DataFrame that we created in the previous section.

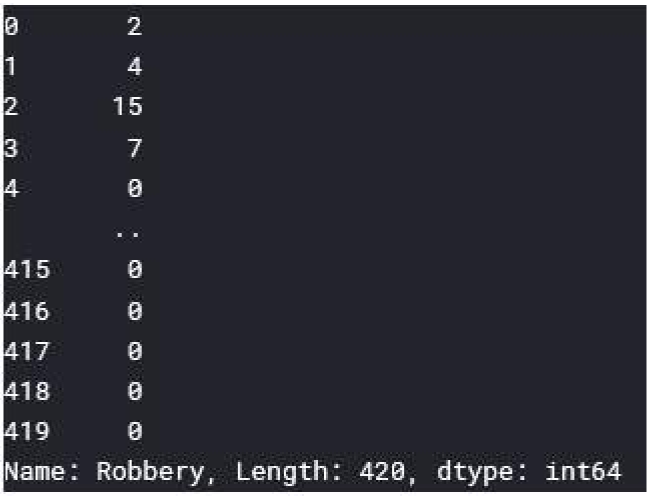
```
>>> dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv")

>>> dataset
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
1	ANDHRA PRADESH	2002	60	98	18	0	4	12	568
2	ANDHRA PRADESH	2003	33	79	27	1	15	4	615
3	ANDHRA PRADESH	2004	39	66	28	0	7	20	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	9	459
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	0	0
416	PUDUCHERRY	2009	0	0	0	0	0	0	0
417	PUDUCHERRY	2010	1	0	1	0	0	0	1
418	PUDUCHERRY	2011	0	0	0	0	0	0	0
419	PUDUCHERRY	2012	2	0	0	0	0	0	0

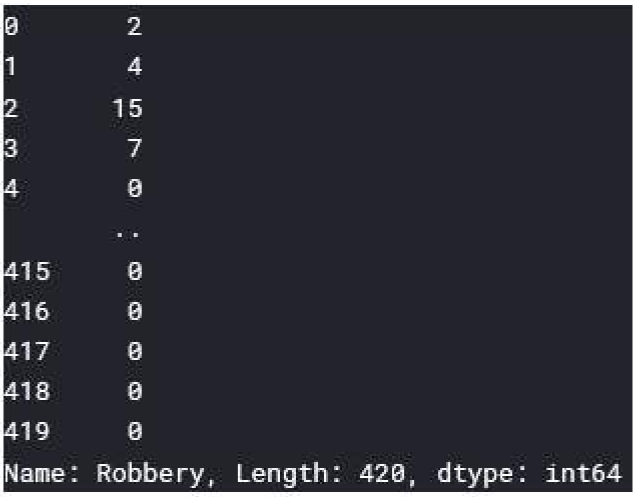
Python allows accessing properties of objects using the standard method to access object attribute. An example of this is to access the "name" attribute of the object "student" we use "`student.name`". Columns of a pandas DataFrame can be accessed in the same way. Now from our DataFrame the Robbery column can be accessed as follows.

```
>>> dataset.Robbery
```


```
0       2
1       4
2      15
3       7
4       0
       ..
415     0
416     0
417     0
418     0
419     0
Name: Robbery, Length: 420, dtype: int64
```

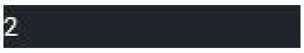
The values of python dictionary can be accessed using the indexing operator([ ]). This method can also be used to access the columns of a DataFrame object as shown below

```
>>> dataset["Robbery"]
```

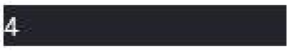

```
0       2
1       4
2      15
3       7
4       0
       ..
415     0
416     0
417     0
418     0
419     0
Name: Robbery, Length: 420, dtype: int64
```

Observe that this returns the same result and this method to retrieve a column is better than the previous method of accessing a column as an attribute, because column names with a space character can't be accessed this way. The pandas Series is like a dictionary as specific values can be accessed using another indexing operator as follows.

```
>>> dataset["Robbery"][0]
```


```
2
```

```
>>> dataset["Robbery"][0]
```


```
4
```

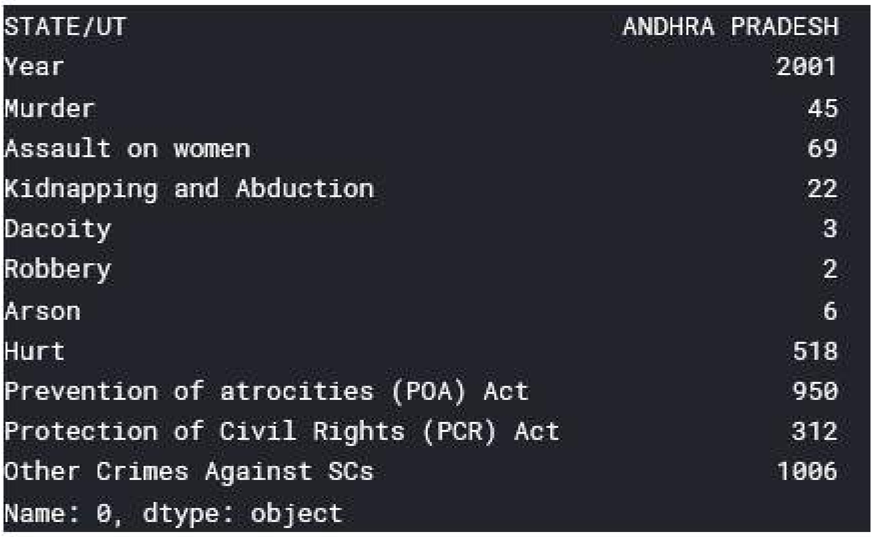
Both these methods of selection, the attribute selection and indexing operator are easy for beginners familiar to the python syntax to pickup and use, but these methods are not sufficient when it comes to more advanced indexing problems. Pandas provides it's own set of access operators and they are,

1. `iloc` (Index based selection)
2. `loc` (label based selection)

5.2.2 Index based selection

In this indexing paradigm that Pandas provides, data selection is performed based on the numerical position of the data in the DataFrame or Series. iloc is the implementation of this indexing paradigm. Now lets use iloc to select the first row of our DataFrame object.

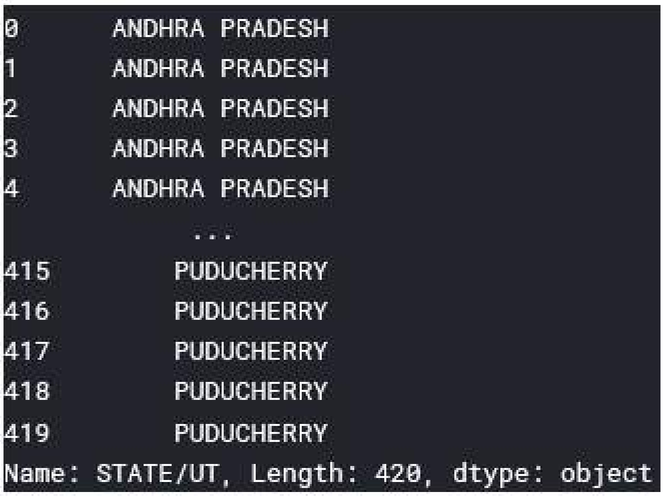
```
>>> dataset.iloc[0]
```



```
STATE/UT                                ANDHRA PRADESH
Year                                              2001
Murder                                              45
Assault on women                                    69
Kidnapping and Abduction                            22
Dacoity                                              3
Robbery                                              2
Arson                                                6
Hurt                                               518
Prevention of atrocities (POA) Act                 950
Protection of Civil Rights (PCR) Act               312
Other Crimes Against SCs                          1006
Name: 0, dtype: object
```

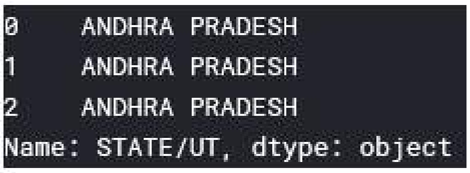
Ont the contrary to ow index is done in python, both of the Pandas indexing methods(loc and iloc) are row-first and column-second. Making it slightly easier to access rows and slightly easier to access columns. To access first column using iloc we use the following method.

```
>>> dataset.iloc[ : , 0] # 0 for first column, 1 for second and so on
```


```
0      ANDHRA PRADESH
1      ANDHRA PRADESH
2      ANDHRA PRADESH
3      ANDHRA PRADESH
4      ANDHRA PRADESH
            ...
415        PUDUCHERRY
416        PUDUCHERRY
417        PUDUCHERRY
418        PUDUCHERRY
419        PUDUCHERRY
Name: STATE/UT, Length: 420, dtype: object
```

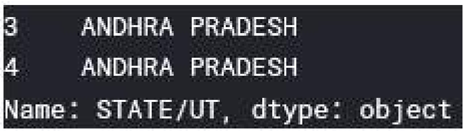
Here the ‘:’ operator just like in native python is being used to refer to a range of values if the both the boundaries are not provided it is synonymous to “everything”. Lets use this operator to return only the first three rows this time.

```
>>> dataset.iloc[ : 3, 0] # stop copying form index 3
```


```
0    ANDHRA PRADESH
1    ANDHRA PRADESH
2    ANDHRA PRADESH
Name: STATE/UT, dtype: object
```

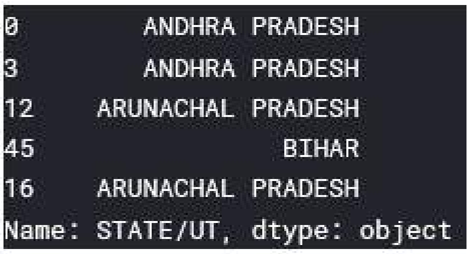
For selecting only the fourth and fifth entries we use,

```
>>> dataset.iloc[3 : 5, 0] # start from index three and stop from index 5
```


```
3    ANDHRA PRADESH
4    ANDHRA PRADESH
Name: STATE/UT, dtype: object
```

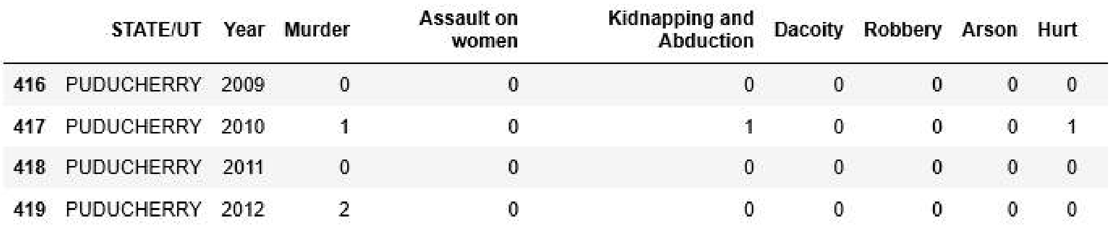
It’s also allowed to pass a list of indexes as follows.

```
>>> dataset.iloc[[0, 3, 12, 45, 16], 0]
```


```
0         ANDHRA PRADESH
3         ANDHRA PRADESH
12    ARUNACHAL PRADESH
45                 BIHAR
16    ARUNACHAL PRADESH
Name: STATE/UT, dtype: object
```

Just like in native python negative numbers can be used. An example is shown below where the selection starts from four rows from the end.

```
>>> dataset.iloc[-4 : ]
```

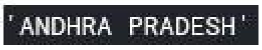


	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
416	PUDUCHERRY	2009	0	0	0	0	0	0	0
417	PUDUCHERRY	2010	1	0	1	0	0	0	1
418	PUDUCHERRY	2011	0	0	0	0	0	0	0
419	PUDUCHERRY	2012	2	0	0	0	0	0	0

5.2.3 Label based selection

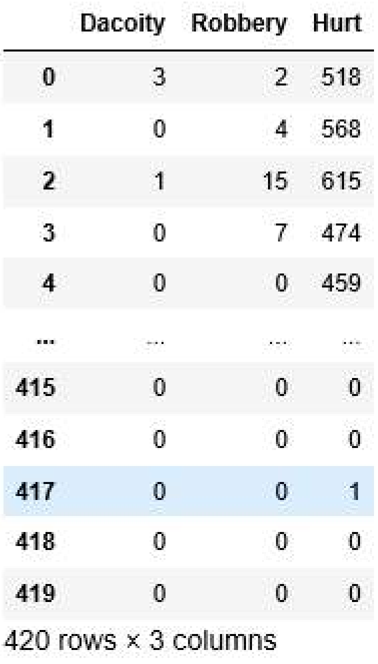
The second selection paradigm that pandas provides is implemented using the loc operator. This operator uses the value of the data instead of its index to perform the indexing operation. Take an example of retrieving the first entry of our DataFrame using the loc operator. This can be done as follows.

```
>>> dataset.loc[0, "STATE/UT"]
```



```
'ANDHRA PRADESH'
```

Conceptually speaking iloc is more simpler than loc as it ignores the information in the dataset's indices and considers the dataset as one big matrix. On the contrary to iloc, the loc operator uses the information in the dataset's indices for indexing. As dataset's usually include include indices it is easier to use loc instead for indexing. Here is an example that shows how indexing using loc is easier with dataset's that include indices.



	Dacoity	Robbery	Hurt
0	3	2	518
1	0	4	568
2	1	15	615
3	0	7	474
4	0	0	459
...	...	...	...
415	0	0	0
416	0	0	0
417	0	0	1
418	0	0	0
419	0	0	0

420 rows × 3 columns

5.2.4 How to choose between loc and iloc

There are some points to keep in mind when choosing between the loc and iloc indexing operators as these operators have slightly different methods of indexing. The iloc operator starts including from the first element of the provided range and starts excluding from the last element as iloc uses the indexing scheme in Python's stdlib where the first element is included and the last last element is excluded. For example [3 : 7] selects the entries from 3 to 6. On the contrary loc indexes inclusively meaning it includes the last index in the selection, so [3 : 7] selects the entries from 3 to 7.

This change is made for the loc operator as the loc operator can index any stdlib type like strings which makes this change necessary for cases like the example shown below.

Say for a DataFrame "df" with index values "Monday", "Tuesday", … we require to select the days of the week between "Thursday" and "Sunday". Indexing them is conveniently made possible using,

```
>>> df.loc["Thursday", "Sunday"]
```

Choosing between these two operators especially gets confusing when dealing with a numerical list like 0…100 in which case,

```
>>> df.loc[0 : 100]
```

Will return a hundred entries from 0 to 99, where as when we provide the same range to the loc operator as below.

```
>>> df.loc[0 : 100]
```

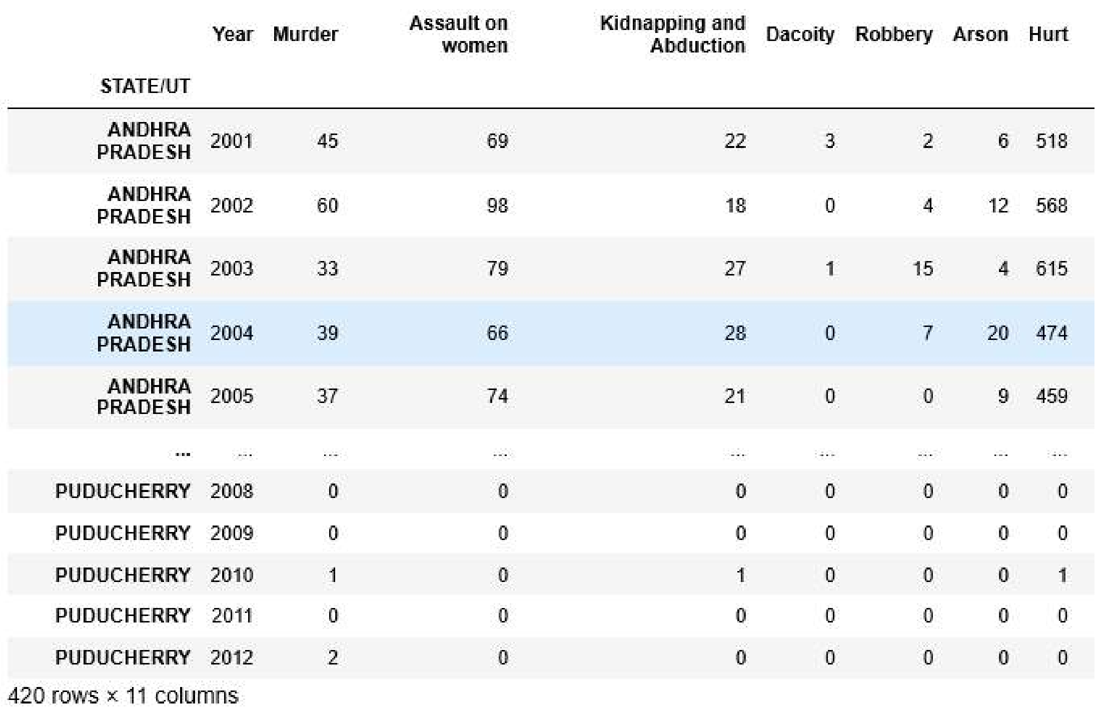
This returns 101 entries from 0 to 100. In all other aspects he semantics of using the iloc operator are the same as for the loc operator.

5.2.5 Index manipulation

Label based selection using loc operator is especially useful when the DataFrame being dealt with has labels for its indices. The indices are not immutable so we can change is as needed. The DataFrame class of Pandas provides a method that can help with this task.

Now lets use this function inn our crime data set. Although it may not be a suitable index for a data set the instruction below sets the "STATE/UT" field as the index of the DataFrame.

```
>>> dataset.set_index("STATE/UT")
```



	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
STATE/UT								
ANDHRA PRADESH	2001	45	69	22	3	2	6	518
ANDHRA PRADESH	2002	60	98	18	0	4	12	568
ANDHRA PRADESH	2003	33	79	27	1	15	4	615
ANDHRA PRADESH	2004	39	66	28	0	7	20	474
ANDHRA PRADESH	2005	37	74	21	0	0	9	459
...	...	...	...	...	...	...	...	...
PUDUCHERRY	2008	0	0	0	0	0	0	0
PUDUCHERRY	2009	0	0	0	0	0	0	0
PUDUCHERRY	2010	1	0	1	0	0	0	1
PUDUCHERRY	2011	0	0	0	0	0	0	0
PUDUCHERRY	2012	2	0	0	0	0	0	0

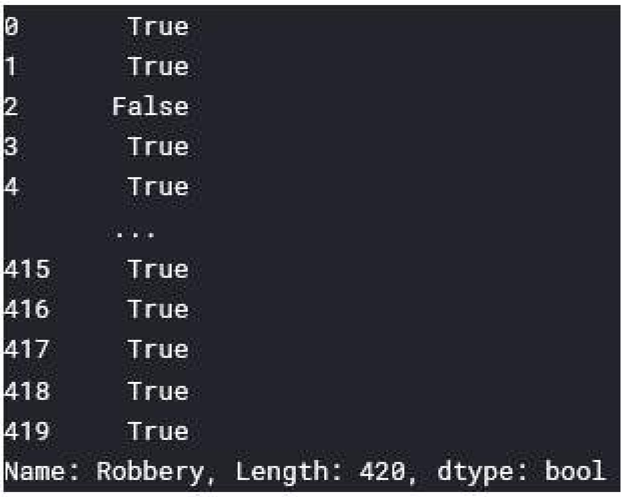
420 rows × 11 columns

This function is would be useful if the dataset being used had a field that could be used as an index.

5.2.6 Conditional selection

How do we select only the entries that satisfy a provided condition. For example selecting all the states of the country that had less than 10 robbery cases in a year. For this we can start by checking each state, if had less than 10 robbery cases. This can be done as shown below.

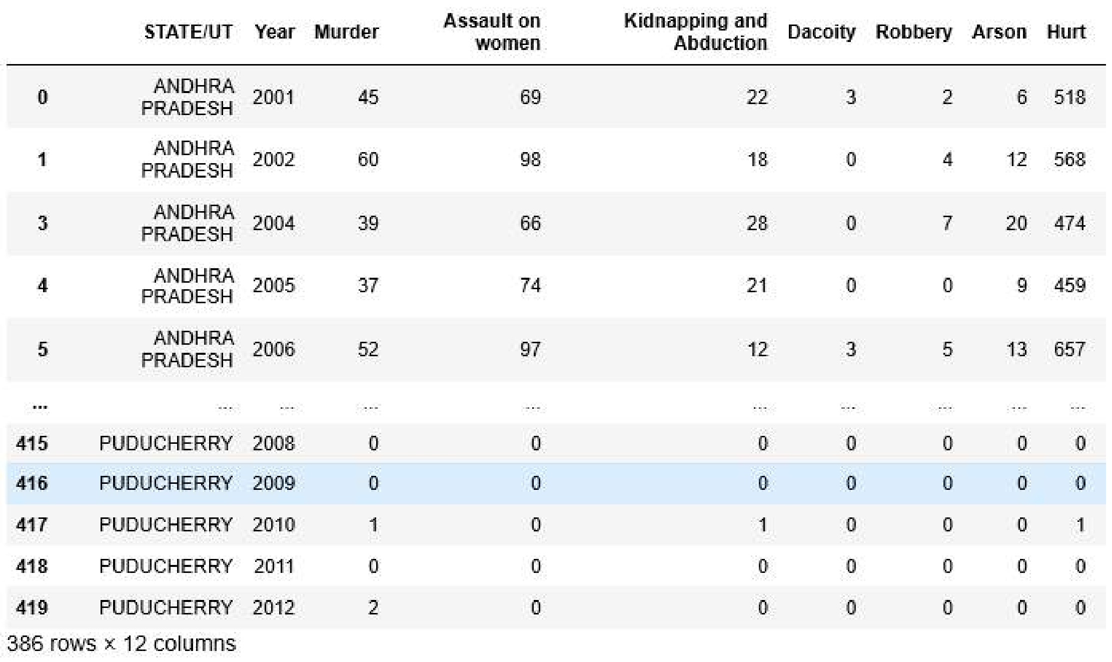
```
>>> dataset.Robbery < 10
```



```
0       True
1       True
2      False
3       True
4       True
       ...
415     True
416     True
417     True
418     True
419     True
Name: Robbery, Length: 420, dtype: bool
```

Observe that what this operation returned is a Series object consisting of boolean values each indicating whether the appropriate entry satisfies the provided condition or not. This Series object can now be used with the loc operator to select data that is relevant as follows.

```
>>> dataset.loc[dataset.Robbery < 10]
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
1	ANDHRA PRADESH	2002	60	98	18	0	4	12	568
3	ANDHRA PRADESH	2004	39	66	28	0	7	20	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	9	459
5	ANDHRA PRADESH	2006	52	97	12	3	5	13	657
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	0	0
416	PUDUCHERRY	2009	0	0	0	0	0	0	0
417	PUDUCHERRY	2010	1	0	1	0	0	0	1
418	PUDUCHERRY	2011	0	0	0	0	0	0	0
419	PUDUCHERRY	2012	2	0	0	0	0	0	0

386 rows × 12 columns

This DataFrame has 386 rows which means that very few state and year combinations have seen 10 or more robberies a year. Pandas also supports multiple conditions which can be joined together using the & (and) or the |(or) operators. Lets select all the states that had 10 or more robberies in the year 2012.

```
>>> dataset.loc[(dataset.Robbery > 10) & (dataset.Year == 2012)]
```

	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
83	GUJARAT	2012	23	45	32	5	16	3	236
179	MAHARASHTRA	2012	36	97	23	16	13	9	81

In case we want all the states that had either more than 20 cases of Arson or 15 cases of robbery in the year 2010 then we could use the instruction shown below.

```
>>> dataset.loc[((dataset.Arson > 20) | (dataset.Robbery > 15)) & (dataset.Year ==
    2010)]
```

	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
81	GUJARAT	2010	15	34	28	6	20	4	291
165	MADHYA PRADESH	2010	102	316	69	0	3	24	877
177	MAHARASHTRA	2010	24	89	25	22	20	12	77
261	RAJASTHAN	2010	56	200	51	0	0	31	564
309	UTTAR PRADESH	2010	229	311	248	3	6	29	323

Pandas provides a set of built in conditional selectors. Here we'll go through two of them which are,

1. `isin()`
2. `isnull()` and `notnull()`

The isin selector allows selecting data which is in a list of provided values. This is how we can select all records of 2001, 2007 and 20012.

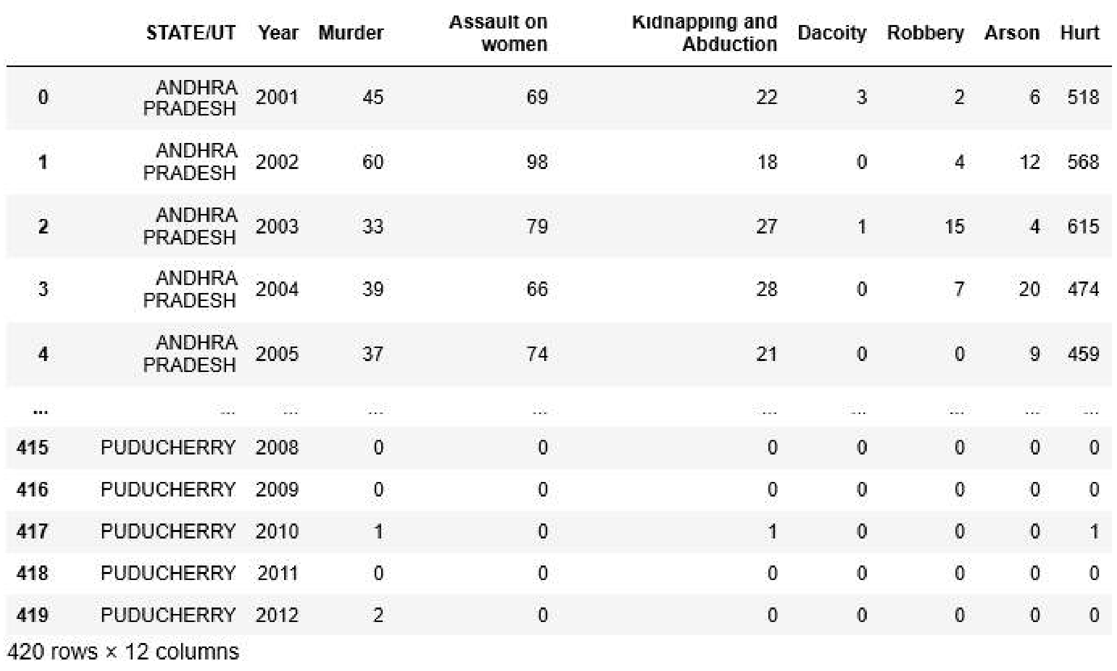
```
>>> dataset.loc[dataset.Year.isin([2001, 2007, 2012])]
```

	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
6	ANDHRA PRADESH	2007	46	105	25	0	0	17	541
11	ANDHRA PRADESH	2012	54	104	11	0	1	7	626
12	ARUNACHAL PRADESH	2001	0	0	0	0	0	0	0
18	ARUNACHAL PRADESH	2007	0	0	0	0	0	0	0
...	...	...	...	...	...	...	...	...	...
402	LAKSHADWEEP	2007	0	0	0	0	0	0	0
407	LAKSHADWEEP	2012	0	0	0	0	0	0	0
408	PUDUCHERRY	2001	0	0	0	0	0	0	0
414	PUDUCHERRY	2007	1	0	0	0	0	0	0
419	PUDUCHERRY	2012	2	0	0	0	0	0	0

105 rows × 12 columns

The second selector we'll discuss here is "isnull" and 'notnull". These selectors allow filtering out values that aren't/are null. Lets try filtering out all records that have a null value for robbery. We can accomplish this task with the following line of code.

```
>>> dataset.loc[dataset.Year.notnull()]
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
1	ANDHRA PRADESH	2002	60	98	18	0	4	12	568
2	ANDHRA PRADESH	2003	33	79	27	1	15	4	615
3	ANDHRA PRADESH	2004	39	66	28	0	7	20	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	9	459
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	0	0
416	PUDUCHERRY	2009	0	0	0	0	0	0	0
417	PUDUCHERRY	2010	1	0	1	0	0	0	1
418	PUDUCHERRY	2011	0	0	0	0	0	0	0
419	PUDUCHERRY	2012	2	0	0	0	0	0	0

420 rows × 12 columns

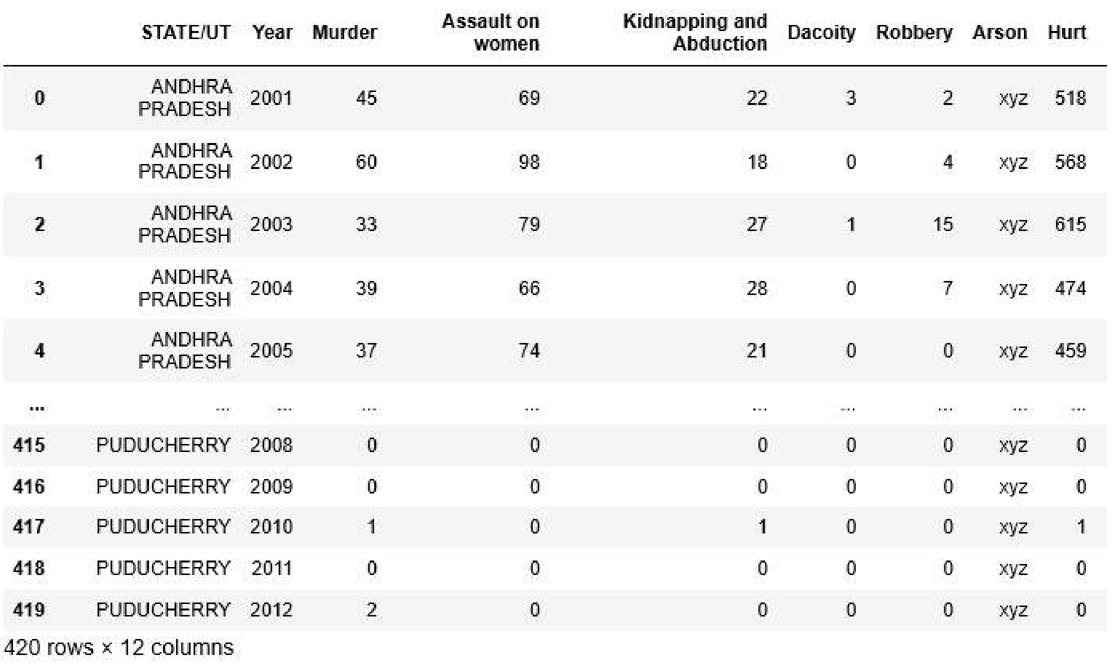
Observe that all the rows were returned and none of the rows were filtered out as all the rows had a value for the “Year” column.

5.2.7 Assigning values

Values can be assigned to the entries of a DataFrame. Values assigned can be a constant value or an iterable of values. An example of assigning constant values is shown below.

```
>>> dataset["Arson"] = "xyz"
```

```
>>> dataset
```



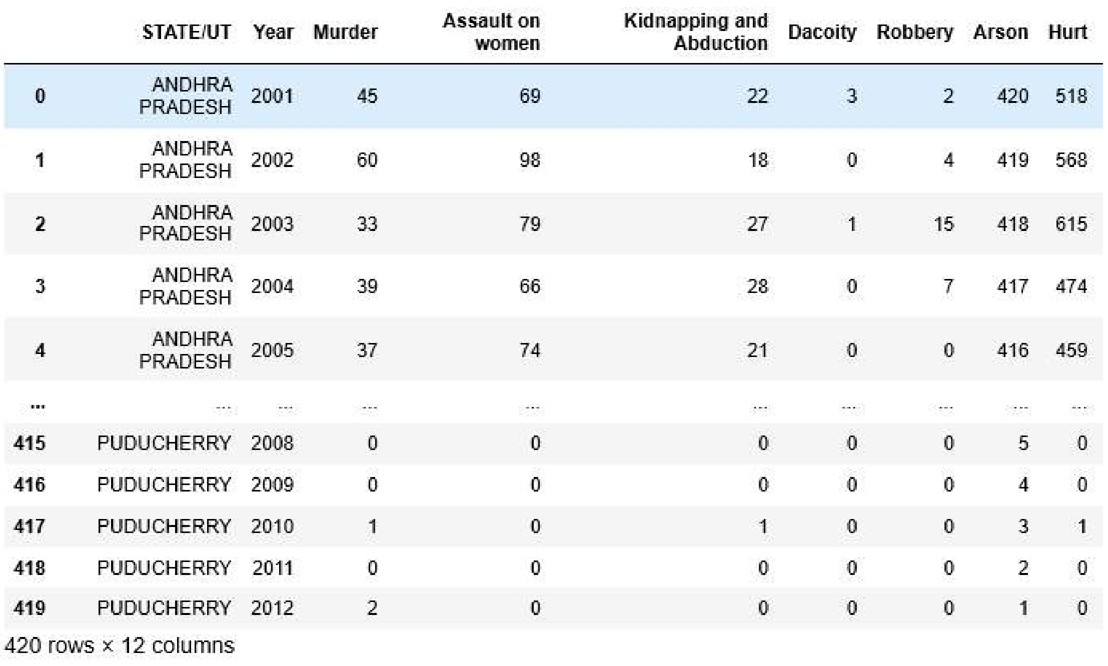
	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	xyz	518
1	ANDHRA PRADESH	2002	60	98	18	0	4	xyz	568
2	ANDHRA PRADESH	2003	33	79	27	1	15	xyz	615
3	ANDHRA PRADESH	2004	39	66	28	0	7	xyz	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	xyz	459
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	xyz	0
416	PUDUCHERRY	2009	0	0	0	0	0	xyz	0
417	PUDUCHERRY	2010	1	0	1	0	0	xyz	1
418	PUDUCHERRY	2011	0	0	0	0	0	xyz	0
419	PUDUCHERRY	2012	2	0	0	0	0	xyz	0

420 rows × 12 columns

An example of assigning an iterable of values is given below.

```
>>> dataset["Arson"] = range(len(dataset), 0, -1)
```

```
>>> dataset
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	420	518
1	ANDHRA PRADESH	2002	60	98	18	0	4	419	568
2	ANDHRA PRADESH	2003	33	79	27	1	15	418	615
3	ANDHRA PRADESH	2004	39	66	28	0	7	417	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	416	459
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	5	0
416	PUDUCHERRY	2009	0	0	0	0	0	4	0
417	PUDUCHERRY	2010	1	0	1	0	0	3	1
418	PUDUCHERRY	2011	0	0	0	0	0	2	0
419	PUDUCHERRY	2012	2	0	0	0	0	1	0

420 rows × 12 columns

5.3 Summarizing functions and maps

In this previous section we learned how to select the appropriate data that was required for the task at hand. Although selecting the data that we require is a crucial task, it’s important to note that the data that we have at hand is not always in the format that is required. In which case we need to reformat the data in a way that is appropriate for the task at hand.

We will continue using the crime data set of the previous section in this this section as well.

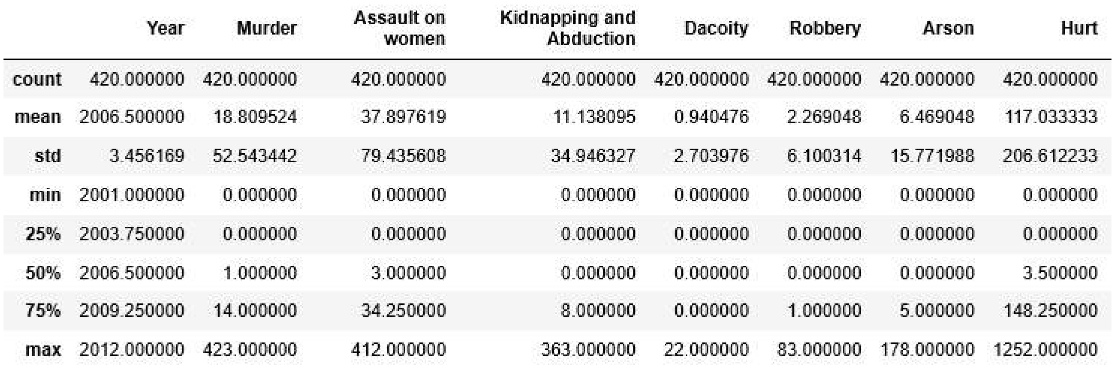
5.3.1 Summary Functions

Pandas provides several functions to summarize data, meaning that functions can be used for restructuring the data in a useful manner.

`describe( )`

We’ll go through some important functions here,starting with the describe( ) function. It is available for DataFrames as well as Series objects. The instruction below calls the describe( ) of the DataFrame named “dataset”.

```
>>> dataset.describe()
```

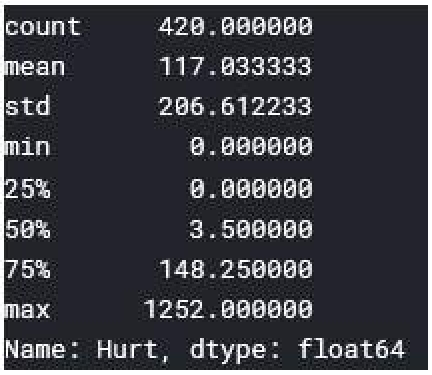


	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
count	420.000000	420.000000	420.000000	420.000000	420.000000	420.000000	420.000000	420.000000
mean	2006.500000	18.809524	37.897619	11.138095	0.940476	2.269048	6.469048	117.033333
std	3.456169	52.543442	79.435608	34.946327	2.703976	6.100314	15.771988	206.612233
min	2001.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	2003.750000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	2006.500000	1.000000	3.000000	0.000000	0.000000	0.000000	0.000000	3.500000
75%	2009.250000	14.000000	34.250000	8.000000	0.000000	1.000000	5.000000	148.250000
max	2012.000000	423.000000	412.000000	363.000000	22.000000	83.000000	178.000000	1252.000000

When the describe( ) method is called for a DataFrame object it Returns a DataFrame with the count, mean standard deviation, minimum value, lower quartile, middle quartile(median), upper quartile and maximum value of all numerical and floating point columns of the DataFrame.

The describe( ) method is also made available for Series objects. The describe methods returns different results based on the data type of the entries that the Series object contains. In the instruction below we call the describe function for the column “Hurt” which is a column of type integer.

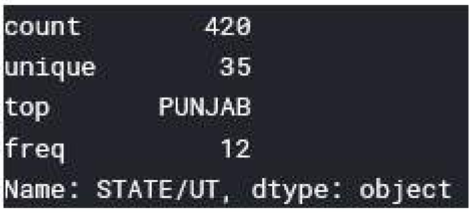
```
>>> dataset["Hurt"].describe()
```



```
count      420.000000
mean       117.033333
std        206.612233
min          0.000000
25%          0.000000
50%          3.500000
75%        148.250000
max       1252.000000
Name: Hurt, dtype: float64
```

Now lets call the describe method for a column with string entries like “STATE/UT”.

```
>>> dataset["STATE/UT"].describe()
```

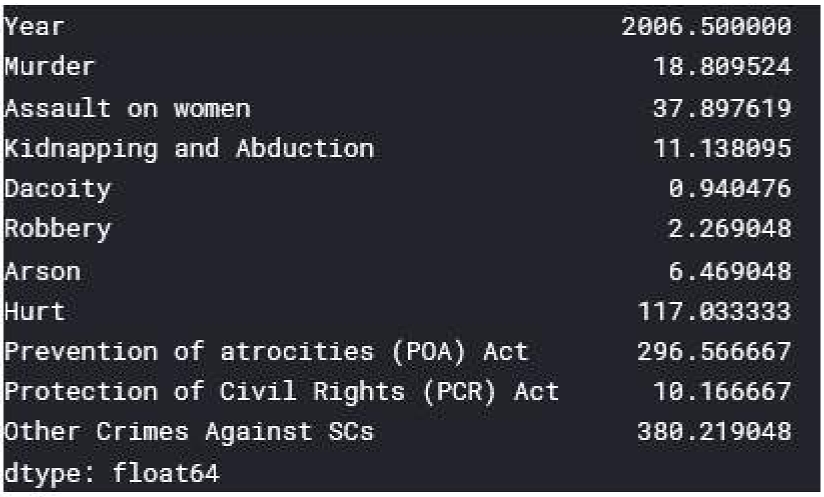


```
count          420
unique          35
top         PUNJAB
freq            12
Name: STATE/UT, dtype: object
```

mean( )

Pandas provides several functions that help acquire simple summary statistics about a Series/column or a DataFrame. Lets take a look at a few starting with the mean method which returns the average of all entries. Here is an example of using the using the mean method of a DataFrame object.

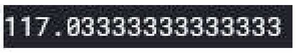
```
>>> dataset.mean()
```



```
Year                                     2006.500000
Murder                                     18.809524
Assault on women                           37.897619
Kidnapping and Abduction                   11.138095
Dacoity                                     0.940476
Robbery                                     2.269048
Arson                                       6.469048
Hurt                                      117.033333
Prevention of atrocities (POA) Act        296.566667
Protection of Civil Rights (PCR) Act       10.166667
Other Crimes Against SCs                  380.219048
dtype: float64
```

Here is an example of using the mean method of a Series object.

```
>>> dataset["Hurt"].mean()
```

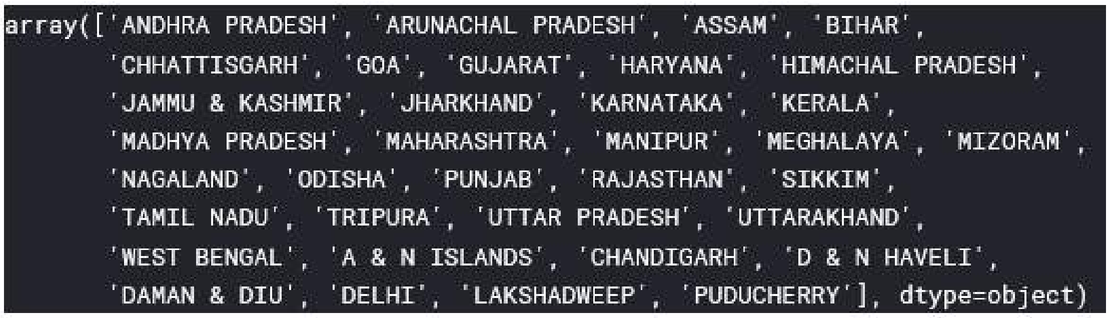


```
117.03333333333333
```

unique( )

This method returns a list of all the unique values of the Series object on which it was called upon.

```
>>> dataset["STATE/UT"].unique()
```

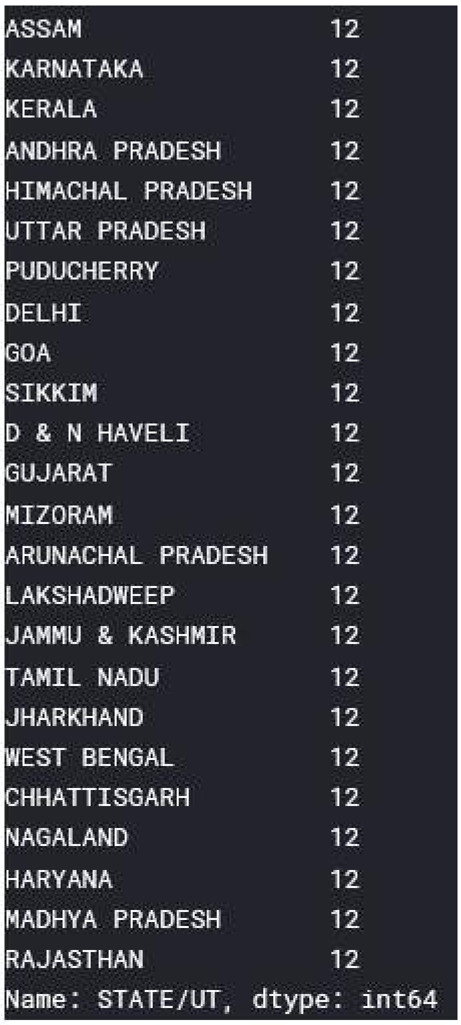


```
array(['ANDHRA PRADESH', 'ARUNACHAL PRADESH', 'ASSAM', 'BIHAR',
       'CHHATTISGARH', 'GOA', 'GUJARAT', 'HARYANA', 'HIMACHAL PRADESH',
       'JAMMU & KASHMIR', 'JHARKHAND', 'KARNATAKA', 'KERALA',
       'MADHYA PRADESH', 'MAHARASHTRA', 'MANIPUR', 'MEGHALAYA', 'MIZORAM',
       'NAGALAND', 'ODISHA', 'PUNJAB', 'RAJASTHAN', 'SIKKIM',
       'TAMIL NADU', 'TRIPURA', 'UTTAR PRADESH', 'UTTARAKHAND',
       'WEST BENGAL', 'A & N ISLANDS', 'CHANDIGARH', 'D & N HAVELI',
       'DAMAN & DIU', 'DELHI', 'LAKSHADWEEP', 'PUDUCHERRY'], dtype=object)
```

value_counts( )

This method can e used to see the unique values of a Series object along with the number of occurrences of that value in the Series.

```
>>> dataset["STATE/UT"].value_counts()
```

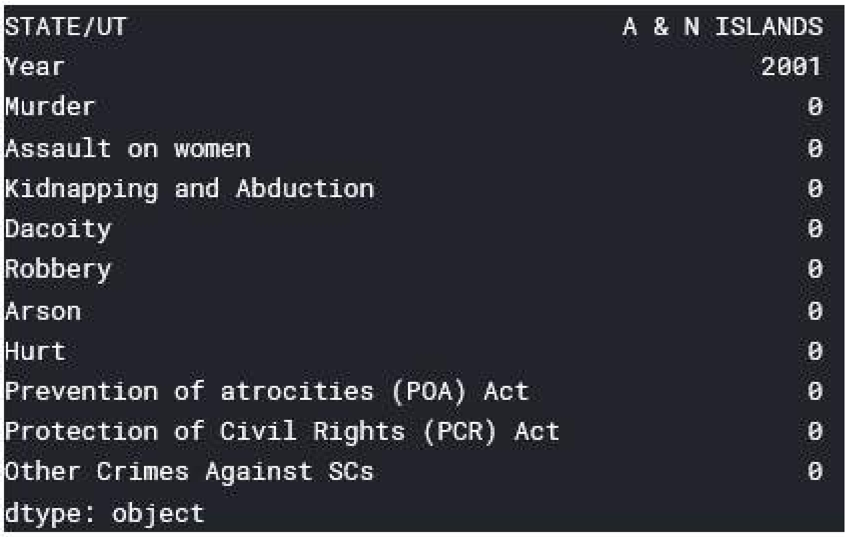


```
ASSAM                 12
KARNATAKA             12
KERALA                12
ANDHRA PRADESH        12
HIMACHAL PRADESH      12
UTTAR PRADESH         12
PUDUCHERRY            12
DELHI                 12
GOA                   12
SIKKIM                12
D & N HAVELI          12
GUJARAT               12
MIZORAM               12
ARUNACHAL PRADESH     12
LAKSHADWEEP           12
JAMMU & KASHMIR       12
TAMIL NADU            12
JHARKHAND             12
WEST BENGAL           12
CHHATTISGARH          12
NAGALAND              12
HARYANA               12
MADHYA PRADESH        12
RAJASTHAN             12
Name: STATE/UT, dtype: int64
```

min( )

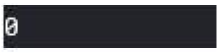
The DataFrame.min( ) method returns a Series consisting of the minimum values of all the columns of the DataFrame.

```
>>> dataset.min()
```


```
STATE/UT                                        A & N ISLANDS
Year                                                     2001
Murder                                                      0
Assault on women                                            0
Kidnapping and Abduction                                    0
Dacoity                                                     0
Robbery                                                     0
Arson                                                       0
Hurt                                                        0
Prevention of atrocities (POA) Act                          0
Protection of Civil Rights (PCR) Act                        0
Other Crimes Against SCs                                    0
dtype: object
```

The Series.min( ) method returns the smallest value in the Series object.

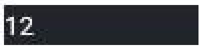
```
>>> dataset["Hurt"].min()
```


```
0
```

idxmin( )

The Series.idxmin( ) method return the index of the smallest value in this Series object and in case there more than one, it returns the index of the first occurrence of this value.

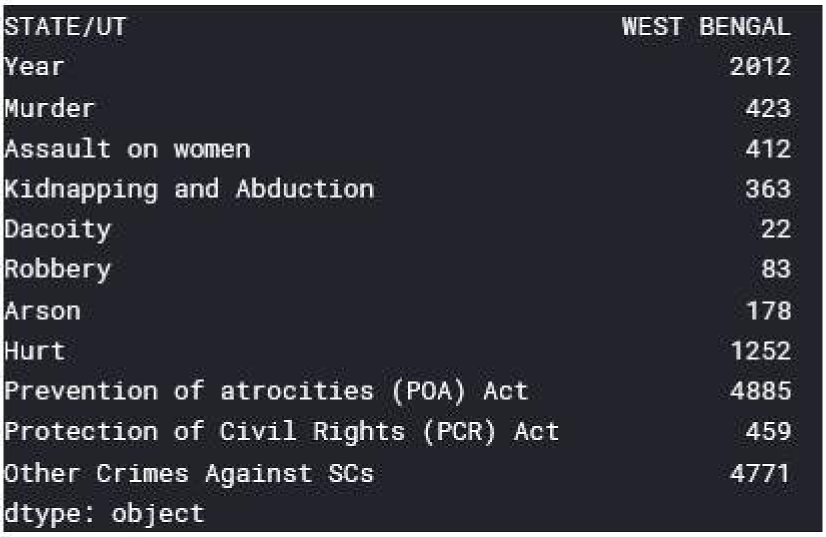
```
>>> dataset.Hurt.idxmin()
```


```
12
```

max( )

The DataFrame.max( ) method returns a Series consisting of the maximum values of all the columns of the DataFrame.

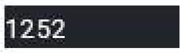
```
>>> dataset.max()
```



```
STATE/UT                                   WEST BENGAL
Year                                              2012
Murder                                             423
Assault on women                                   412
Kidnapping and Abduction                           363
Dacoity                                             22
Robbery                                             83
Arson                                              178
Hurt                                              1252
Prevention of atrocities (POA) Act                4885
Protection of Civil Rights (PCR) Act               459
Other Crimes Against SCs                          4771
dtype: object
```

The Series.max( ) method returns the largest value in the Series object.

```
>>> dataset["Hurt"].max()
```



```
1252
```

idxmax( )

The Series.idxmax( ) method return the index of the largest value in this Series object and in case there more than one, it returns the index of the first occurrence of this value.

```
>>> dataset.Hurt.idxmax()
```

```
157
```

5.3.2 Maps

This is a word taken from mathematics which refers to taking one set of values and mapping them to another set of values by applying certain operations on them. When working with data you will very often have the need to create new representations of this data or transform it into other formats.

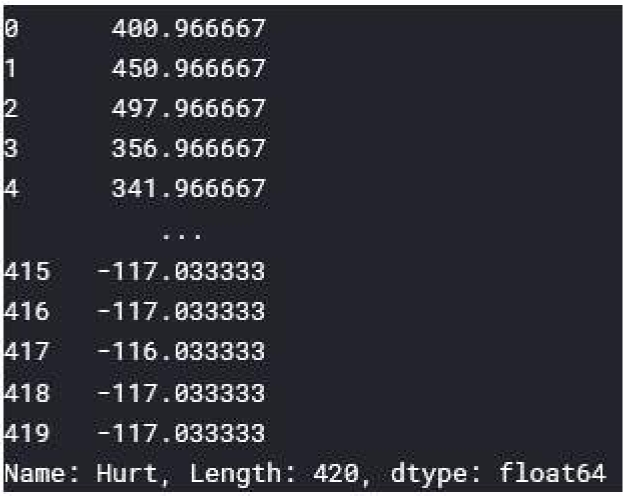
Here we will discuss two mapping methods that are used very often by data scientists and they are,

1. map( )
2. apply( )

map( )

The map( ) method is the more simpler one and it can be called for a Series object or a DataFrame column. This method takes a function as an argument which in turn takes a single value as an argument and returns the transformed value back to the caller. The map( ) method returns a Series object where all entries where transformed by the provided transformation function. Now to re-mean the entries of the column “Hurt” to 0, we do,

```
>>> dataset_Hurt_mean = dataset.Hurt.mean()

>>> dataset.Hurt.map(lambda x: x - dataset_Hurt_mean)
```

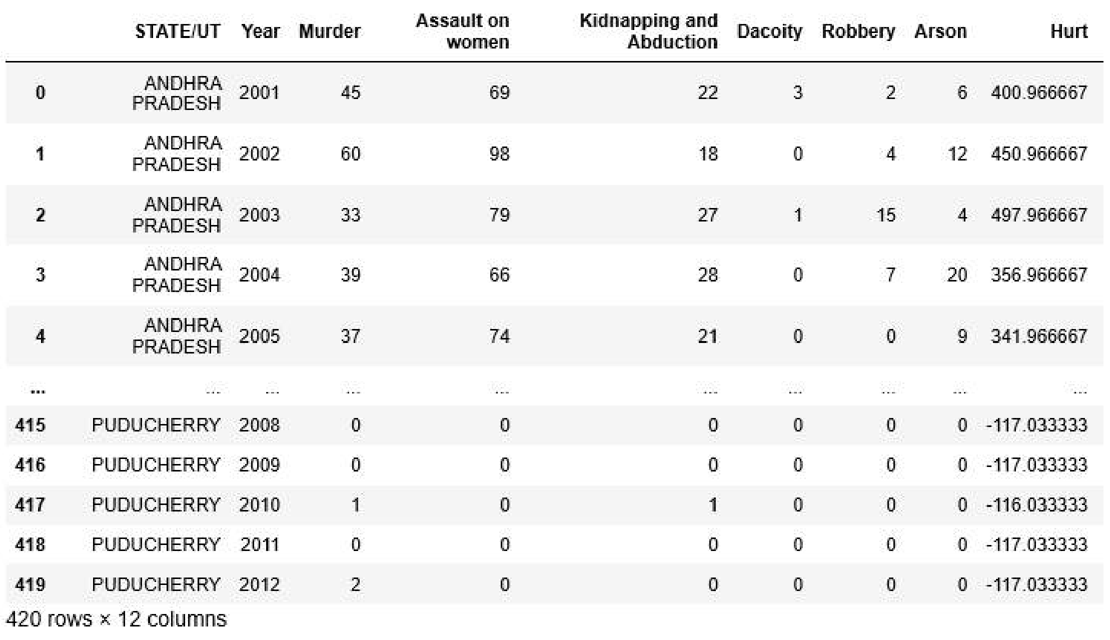


```
0      400.966667
1      450.966667
2      497.966667
3      356.966667
4      341.966667
          ...
415   -117.033333
416   -117.033333
417   -116.033333
418   -117.033333
419   -117.033333
Name: Hurt, Length: 420, dtype: float64
```

apply( )

The apply( ) method performs the same functionality as the map( ) method but it is meant for DataFrame objects instead. The code below shows the same re-mean operation performed on every row of the "dataset" DataFrame.

```
dataset_Hurt_mean = dataset.Hurt.mean()     # calculate mean of Hurt column

def Hurt_mean(row):
    row.Hurt = row.Hurt - dataset_Hurt_mean # function to transform a row
    return row

dataset.apply(Hurt_mean, axis = "columns")   # apply  transformation function
                                            # on every row
```

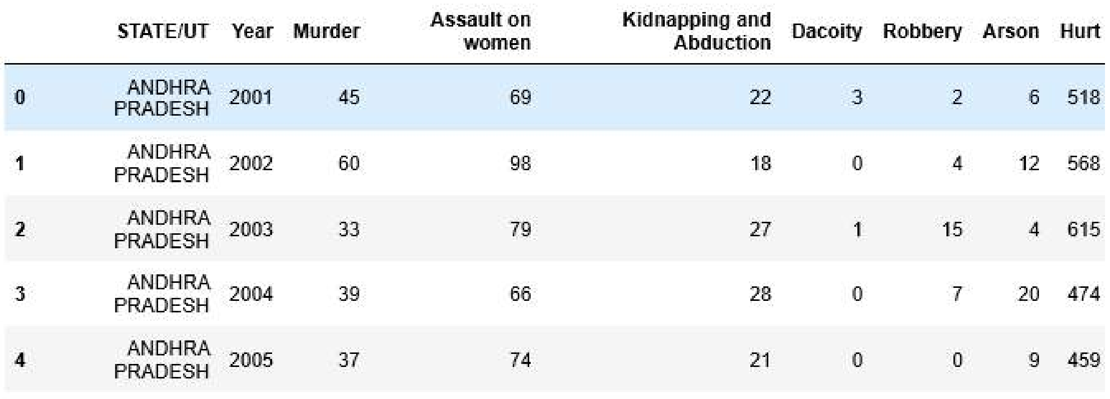


	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	6	400.966667
1	ANDHRA PRADESH	2002	60	98	18	0	4	12	450.966667
2	ANDHRA PRADESH	2003	33	79	27	1	15	4	497.966667
3	ANDHRA PRADESH	2004	39	66	28	0	7	20	356.966667
4	ANDHRA PRADESH	2005	37	74	21	0	0	9	341.966667
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	0	-117.033333
416	PUDUCHERRY	2009	0	0	0	0	0	0	-117.033333
417	PUDUCHERRY	2010	1	0	1	0	0	0	-116.033333
418	PUDUCHERRY	2011	0	0	0	0	0	0	-117.033333
419	PUDUCHERRY	2012	2	0	0	0	0	0	-117.033333

420 rows × 12 columns

If the `dataset.apply()` method was called with `axis = ‘index’` then instead of passing a function that transforms each row, we need to pass it a function that transforms each column. The map and apply methods return new Series and DataFrame objects without modifying the original objects that they were called upon. To verify this take a look at the first 5 rows of the dataset object and note that the Hurt column remains unchanged.

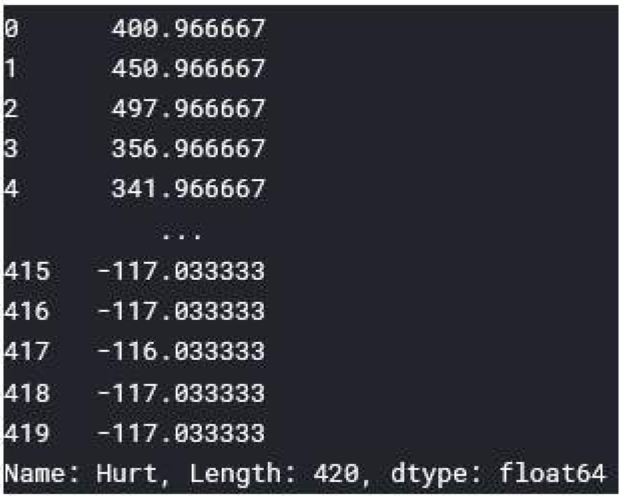
```
>>> dataset.head(5)
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
1	ANDHRA PRADESH	2002	60	98	18	0	4	12	568
2	ANDHRA PRADESH	2003	33	79	27	1	15	4	615
3	ANDHRA PRADESH	2004	39	66	28	0	7	20	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	9	459

Pandas provides several common mapping operations that are built so lets use one of those operation by writing a simpler way to re-mean the entries of a column.

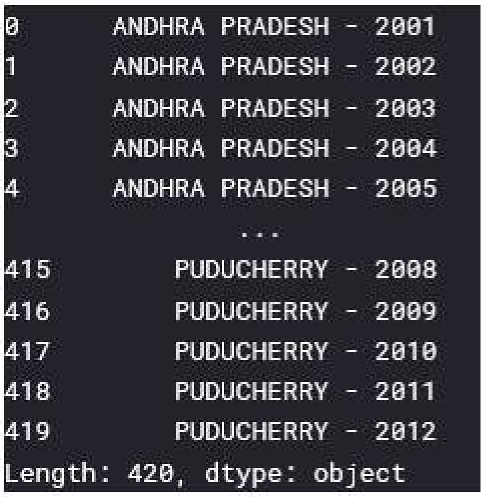
```
>>> dataset_Hurt_mean = dataset.Hurt.mean()
>>> dataset.Hurt - dataset_Hurt_mean
```



```
0      400.966667
1      450.966667
2      497.966667
3      356.966667
4      341.966667
          ...
415   -117.033333
416   -117.033333
417   -116.033333
418   -117.033333
419   -117.033333
Name: Hurt, Length: 420, dtype: float64
```

Observe that here we are subtracting an integer object from a series object with several entries. Pandas will take the integer object on the right side of the operator and subtract it from all the entries of the Series object on the left sided of the operator. This will also work in this case where Two equal sized Series objects are being used as shown below. Take for example an easy way to connect the "STATE/UT" and "Year" columns is shown below.

```
>>> Year_str = dataset.Year.map(lambda x: str(x))

>>> dataset["STATE/UT"] + " - " + Year_str
```



```
0      ANDHRA PRADESH - 2001
1      ANDHRA PRADESH - 2002
2      ANDHRA PRADESH - 2003
3      ANDHRA PRADESH - 2004
4      ANDHRA PRADESH - 2005
                ...
415        PUDUCHERRY - 2008
416        PUDUCHERRY - 2009
417        PUDUCHERRY - 2010
418        PUDUCHERRY - 2011
419        PUDUCHERRY - 2012
Length: 420, dtype: object
```

These operators are faster than the map( ) and the apply( ) methods mentioned earlier as they use speed up mechanisms that pandas has built in. All the standard operators like >, ==, <, and others work the same way but they are not as flexible map( ) and apply methods as they have scope to involve more complex instructions like conditional logic and other instructions which can not be performed simply by using the addition and subtraction operators

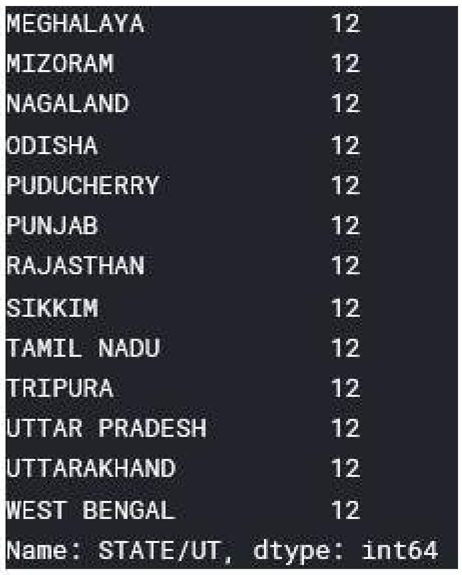
5.4 Grouping and sorting

The concept of maps that we've learned in the last section is useful for transforming data of a DataFrame or a Series object.Often data scientists encounter situations where they need to group data into distinct parts and perform certain operations on these parts. This task can be achieved using the groupby( ) method.

5.4.1 Grouping

We have already seen the use of the value_counts( ) method in the previous sections. Lets use the groupby( ) method to perform the same operation as the value_counts( ) method.

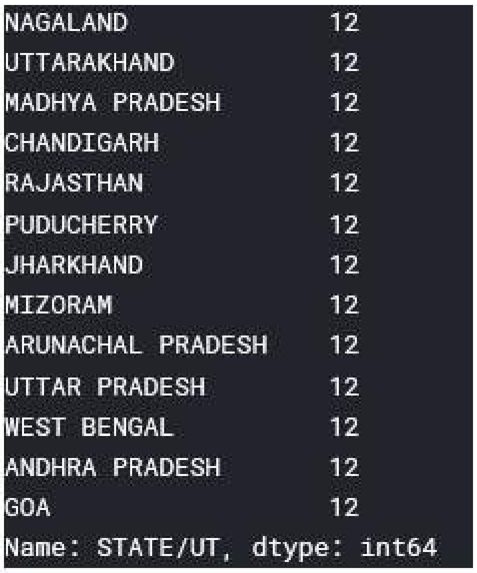
```
>>> dataset.groupby("STATE/UT")["STATE/UT"].count()
```



```
MEGHALAYA           12
MIZORAM             12
NAGALAND            12
ODISHA              12
PUDUCHERRY          12
PUNJAB              12
RAJASTHAN           12
SIKKIM              12
TAMIL NADU          12
TRIPURA             12
UTTAR PRADESH       12
UTTARAKHAND         12
WEST BENGAL         12
Name: STATE/UT, dtype: int64
```

The instruction shown above is performs the same operation as

```
>>> dataset["STATE/UT"].value_counts()
```

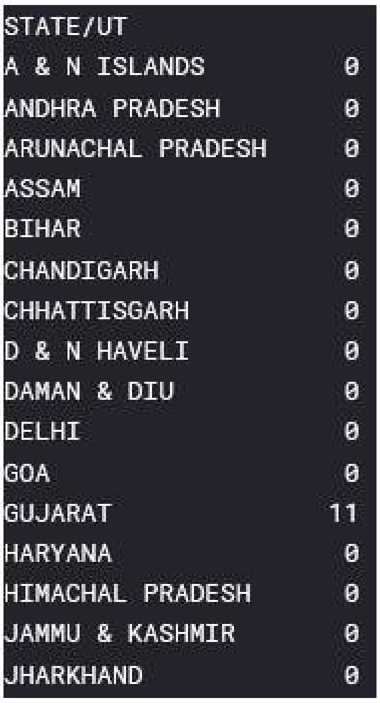


```
NAGALAND             12
UTTARAKHAND          12
MADHYA PRADESH       12
CHANDIGARH           12
RAJASTHAN            12
PUDUCHERRY           12
JHARKHAND            12
MIZORAM              12
ARUNACHAL PRADESH    12
UTTAR PRADESH        12
WEST BENGAL          12
ANDHRA PRADESH       12
GOA                  12
Name: STATE/UT, dtype: int64
```

The groupby function used in the above instruction groups the records with the same value for the column "STATE/UT", then for each group, counts the number of appearances of values in the "STATE/UT" column.

It is possible to use any of Pandas summary functions that we discussed before with the groupby( ) method. For example the code below uses the min( ) summary function. Here it returns the least number of Robberies that each state encountered in any of the years that the data is made available for.

```
>>> dataset.groupby("STATE/UT").Robbery.min()
```

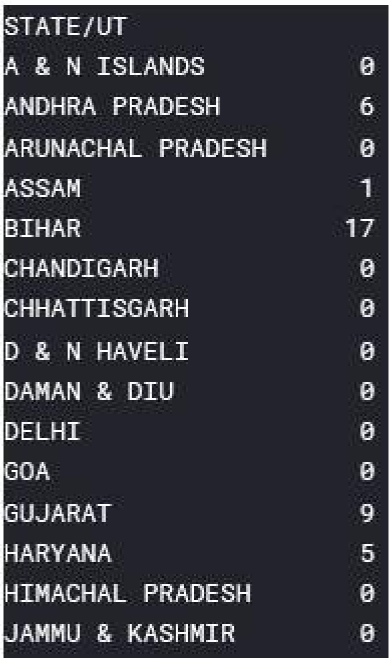


```
STATE/UT
A & N ISLANDS         0
ANDHRA PRADESH        0
ARUNACHAL PRADESH     0
ASSAM                 0
BIHAR                 0
CHANDIGARH            0
CHHATTISGARH          0
D & N HAVELI          0
DAMAN & DIU           0
DELHI                 0
GOA                   0
GUJARAT              11
HARYANA               0
HIMACHAL PRADESH      0
JAMMU & KASHMIR       0
JHARKHAND             0
```

Using `apply( )` with `groupby( )`

Each group can be considered as a subset of the original data set with only the matching values. These groups can be accessed using the apply( ) method that we have learned about earlier. The apply method can be used to manipulate the data as required. The instruction below is of example of using apply( ) method of the DataFrameGroupby object which is returned by calling the groupby method of a DataFrame object. This instruction calls the creates groups of data using the using the “STATE/UT” column then selects the number of Arson crimes from the first record of every group.

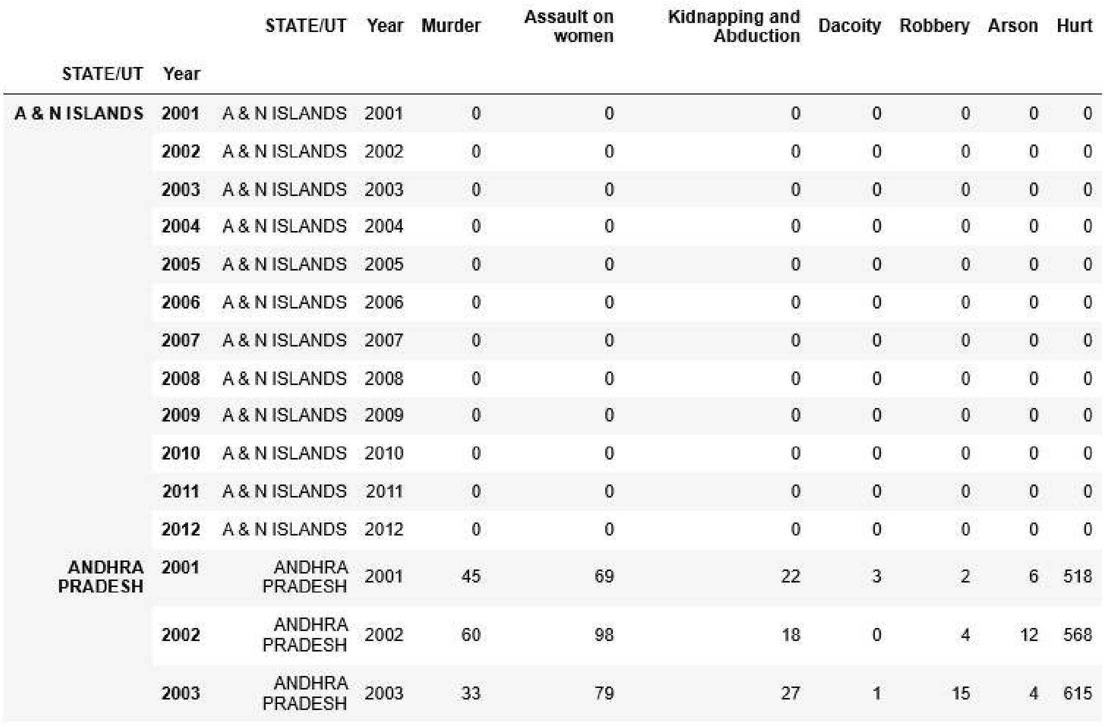
```
>>> dataset.groupby("STATE/UT").apply(lambda row: row["Arson"].iloc[0])
```


```
STATE/UT
A & N ISLANDS          0
ANDHRA PRADESH         6
ARUNACHAL PRADESH      0
ASSAM                  1
BIHAR                 17
CHANDIGARH             0
CHHATTISGARH           0
D & N HAVELI           0
DAMAN & DIU            0
DELHI                  0
GOA                    0
GUJARAT                9
HARYANA                5
HIMACHAL PRADESH       0
JAMMU & KASHMIR        0
```

groupby( ) on multiple columns

Another useful way the groupby method of DataFrames can be used is by supplying more than one column to group the DataFrame with. This give much more control than using a single column. The instruction below selects records with highest number of arson's by State and Year. This example although not practically useful as there is only one record for each state and year combination, it is provided to understand grouping with multiple columns.

```
>>> dataset.groupby(["STATE/UT", "Year"]).apply(lambda record :
    record.loc[record["Arson"].idxmax()]).head(15)
```

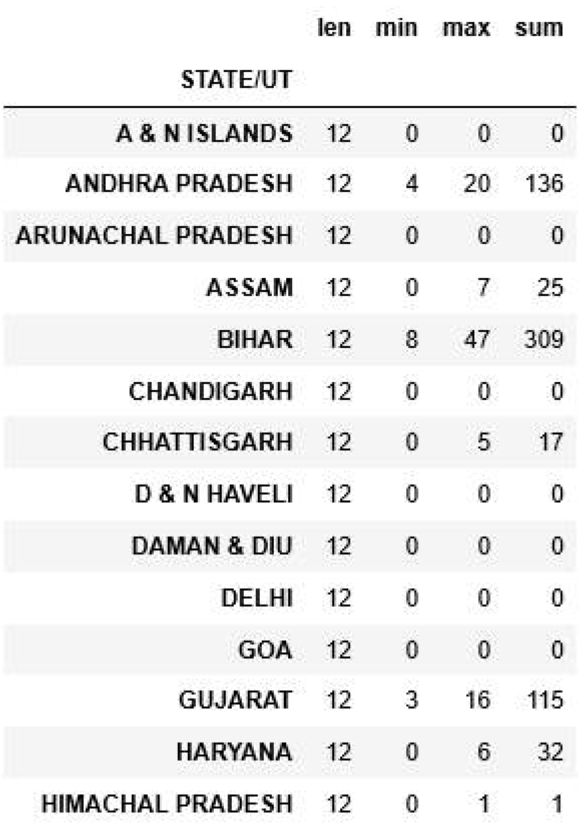


		STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
STATE/UT	**Year**									
A & N ISLANDS	**2001**	A & N ISLANDS	2001	0	0	0	0	0	0	0
	2002	A & N ISLANDS	2002	0	0	0	0	0	0	0
	2003	A & N ISLANDS	2003	0	0	0	0	0	0	0
	2004	A & N ISLANDS	2004	0	0	0	0	0	0	0
	2005	A & N ISLANDS	2005	0	0	0	0	0	0	0
	2006	A & N ISLANDS	2006	0	0	0	0	0	0	0
	2007	A & N ISLANDS	2007	0	0	0	0	0	0	0
	2008	A & N ISLANDS	2008	0	0	0	0	0	0	0
	2009	A & N ISLANDS	2009	0	0	0	0	0	0	0
	2010	A & N ISLANDS	2010	0	0	0	0	0	0	0
	2011	A & N ISLANDS	2011	0	0	0	0	0	0	0
	2012	A & N ISLANDS	2012	0	0	0	0	0	0	0
ANDHRA PRADESH	**2001**	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
	2002	ANDHRA PRADESH	2002	60	98	18	0	4	12	568
	2003	ANDHRA PRADESH	2003	33	79	27	1	15	4	615

agg( )

Another useful method is the agg( ) method which can be applied to DataFrameGroupby objects. This method can be used to run several different functions on a column of a DataFrame object simultaneously making it useful to generate statistical summaries of a data set. An example is shown below for the Arson column of our DataFrame object.

```
>>> dataset.groupby("STATE/UT").Arson.agg([len, min, max, sum])
```

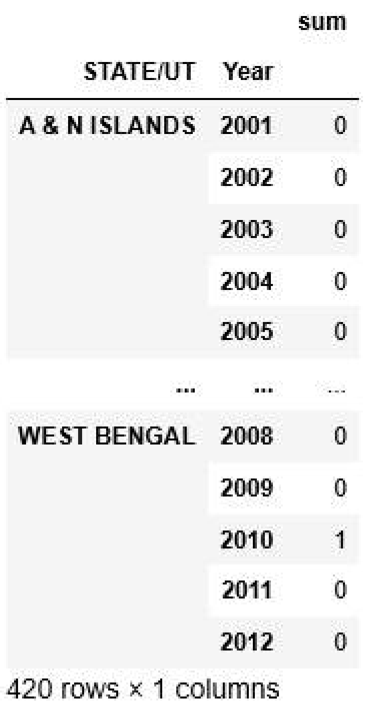


STATE/UT	len	min	max	sum
A & N ISLANDS	12	0	0	0
ANDHRA PRADESH	12	4	20	136
ARUNACHAL PRADESH	12	0	0	0
ASSAM	12	0	7	25
BIHAR	12	8	47	309
CHANDIGARH	12	0	0	0
CHHATTISGARH	12	0	5	17
D & N HAVELI	12	0	0	0
DAMAN & DIU	12	0	0	0
DELHI	12	0	0	0
GOA	12	0	0	0
GUJARAT	12	3	16	115
HARYANA	12	0	6	32
HIMACHAL PRADESH	12	0	1	1

5.4.2 Multiple indexes

The DataFrame and Series objects of most of the examples that we have seen so far had a single label index. The groupby( ) method sometimes results in more thasn one index based on the number of columns provided to the groupby( ) method during the call. In such cases the groupby( ) method returns objects with a multi-index. The difference between a regular index and a multi-index is that a regular index has a single index whereas a multi-index has multiple levels of indexes. Take a look at the example below.

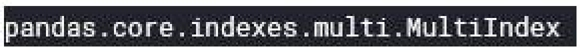
```
>>> Arson_sum = dataset.groupby(["STATE/UT", "Year"]).Arson.agg([sum])

>>> Arson_sum
```



		sum
STATE/UT	**Year**	
A & N ISLANDS	**2001**	0
	2002	0
	2003	0
	2004	0
	2005	0
...	**...**	...
WEST BENGAL	**2008**	0
	2009	0
	2010	1
	2011	0
	2012	0

420 rows × 1 columns

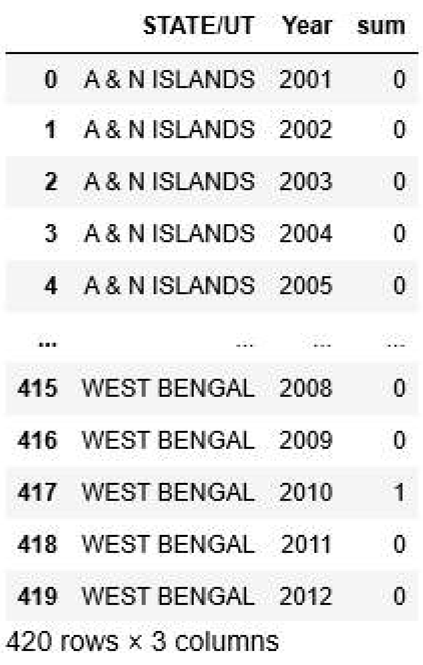
```
>>> Arson_sum_index = Arson_sum.index

>>> type(Arson_sum_index)
```



```
pandas.core.indexes.multi.MultiIndex
```

Pandas provides several functions for handling the tiered structure of objects with a multi-index, but the one that you may often use is the reset_index( ) method which is used to convert a DataFrame with a multi-index back to it's version with a regular index. An example of it in use is shown below.

```
>>> Arson_sum.reset_index()
```



	STATE/UT	Year	sum
0	A & N ISLANDS	2001	0
1	A & N ISLANDS	2002	0
2	A & N ISLANDS	2003	0
3	A & N ISLANDS	2004	0
4	A & N ISLANDS	2005	0
...	...	...	...
415	WEST BENGAL	2008	0
416	WEST BENGAL	2009	0
417	WEST BENGAL	2010	1
418	WEST BENGAL	2011	0
419	WEST BENGAL	2012	0

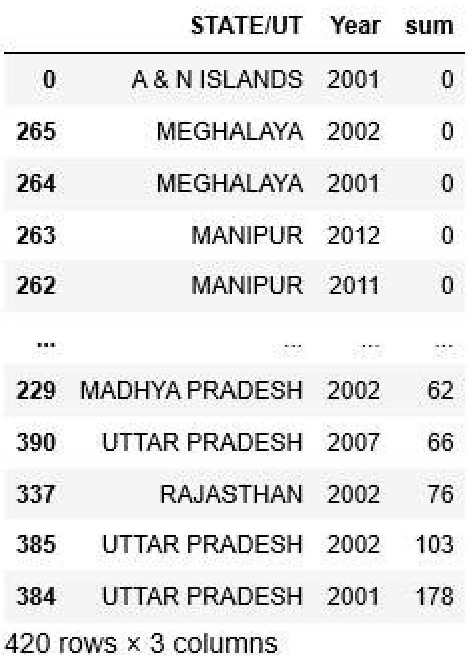
420 rows × 3 columns

5.4.3 Sorting

The ordering of rows in the Arson_sum variable after resting the index is in index order instead of value order. That is because, for the output of the groupby operation the order of the output depends on the index values instead of the data values. Pandas provides methods to sort the data in the way that we want. The sort_values( ) can be used to sort the data in the required manner.

Lets sort the rows of the Arson_sum DataFrame in ascending order of the column "sum".

```
>>> Arson_sum = Arson_sum.reset_index()
>>> Arson_sum.sort_values(by = 'sum')
```

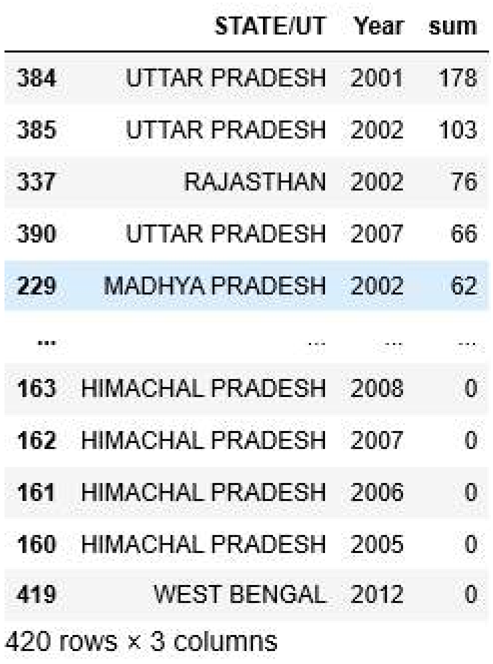


	STATE/UT	Year	sum
0	A & N ISLANDS	2001	0
265	MEGHALAYA	2002	0
264	MEGHALAYA	2001	0
263	MANIPUR	2012	0
262	MANIPUR	2011	0
...	...	...	...
229	MADHYA PRADESH	2002	62
390	UTTAR PRADESH	2007	66
337	RAJASTHAN	2002	76
385	UTTAR PRADESH	2002	103
384	UTTAR PRADESH	2001	178

420 rows × 3 columns

The default order for the sort_values( ) method is ascending, but the often case is that we will need our data in descending order where the largest values come first this can be done as follows.

```
>>> Arson_sum.sort_values(by = 'sum', ascending = False)
```



	STATE/UT	Year	sum
384	UTTAR PRADESH	2001	178
385	UTTAR PRADESH	2002	103
337	RAJASTHAN	2002	76
390	UTTAR PRADESH	2007	66
229	MADHYA PRADESH	2002	62
...	...	...	...
163	HIMACHAL PRADESH	2008	0
162	HIMACHAL PRADESH	2007	0
161	HIMACHAL PRADESH	2006	0
160	HIMACHAL PRADESH	2005	0
419	WEST BENGAL	2012	0

420 rows × 3 columns

To sort the rows by the index values, you can use the sort_index( ) which has the same default order and arguments.

```
>> Arson_sum.sort_index()
```

	STATE/UT	Year	sum
0	A & N ISLANDS	2001	0
1	A & N ISLANDS	2002	0
2	A & N ISLANDS	2003	0
3	A & N ISLANDS	2004	0
4	A & N ISLANDS	2005	0
...	...	...	...
415	WEST BENGAL	2008	0
416	WEST BENGAL	2009	0
417	WEST BENGAL	2010	1
418	WEST BENGAL	2011	0
419	WEST BENGAL	2012	0

420 rows × 3 columns

To sort the data by more than one column the following approach can be used.

```
>>> dataset.sort_values(by = ['Arson', 'Robbery'], ascending = False)
```

	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
300	UTTAR PRADESH	2001	423	412	219	16	83	178	821
301	UTTAR PRADESH	2002	371	305	130	4	37	103	567
253	RAJASTHAN	2002	44	123	27	1	5	76	361
306	UTTAR PRADESH	2007	310	318	153	4	14	66	394
157	MADHYA PRADESH	2002	79	412	49	0	17	62	1252
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	0	0
416	PUDUCHERRY	2009	0	0	0	0	0	0	0
417	PUDUCHERRY	2010	1	0	1	0	0	0	1
418	PUDUCHERRY	2011	0	0	0	0	0	0	0
419	PUDUCHERRY	2012	2	0	0	0	0	0	0

420 rows × 12 columns

5.5 Data types and missing values

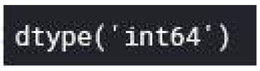
In this section we will focus on Identifying data types of Series objects and columns of DataFrame objects. We will also have an introduction to working with missing values in a data set.

5.5.1 Data types

The data type of a DataFrame object's columns or a series object can be accessed using the dtype attribute. This attribute is useful to investigate the type of data that such objects hold.

The statement below accesses the dtype attribute of Series object. This returns the type of values that the series object holds.

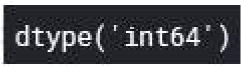
```
>>> Hurt = pd.Series(dataset.Hurt)  # converts column to Series type

>>> Hurt.dtype
```



```
dtype('int64')
```

The statement below accesses the dtype attribute of the column of a DataFrame object.This returns the type of values that this DataFrame column holds

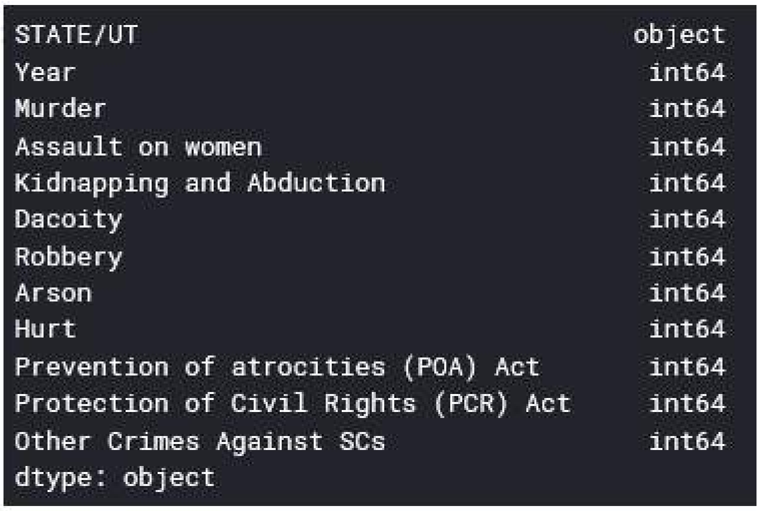
```
>>> dataset.Hurt.dtype
```



```
dtype('int64')
```

The statement below accesses the dtype attribute of DataFrame object. This returns the type of values that each that each column of the DataFrame holds.

```
>>> dataset.dtype
```

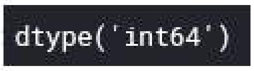


```
STATE/UT                                   object
Year                                        int64
Murder                                      int64
Assault on women                            int64
Kidnapping and Abduction                    int64
Dacoity                                     int64
Robbery                                     int64
Arson                                       int64
Hurt                                        int64
Prevention of atrocities (POA) Act          int64
Protection of Civil Rights (PCR) Act        int64
Other Crimes Against SCs                    int64
dtype: object
```

Changing data types

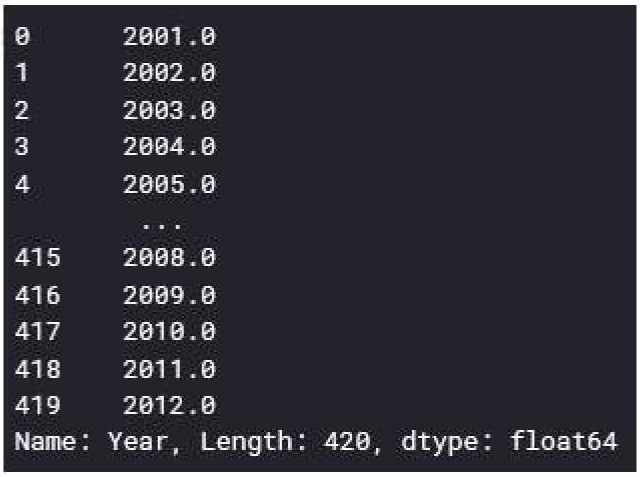
The astype() method of series objects can be used to change the data type that a series object holds. This method takes a string argument indicating the target data type to convert all entries into. In the example below the data type of the Year column of our DataFrame is converted from integer type to floating point type.

```
>>> dataset.Year.dtype
```

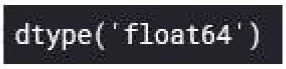


```
dtype('int64')
```

```
>>> Year_col = dataset.Year.astype('float64')

>>> Year_col
```

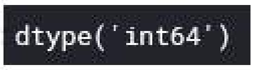

```
0      2001.0
1      2002.0
2      2003.0
3      2004.0
4      2005.0
         ...
415    2008.0
416    2009.0
417    2010.0
418    2011.0
419    2012.0
Name: Year, Length: 420, dtype: float64
```

```
>>> Year_col.dtype
```

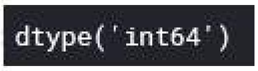

```
dtype('float64')
```

The index of a DataFrame object or a Series object has a data type as well. This can be seen from the examples shown below.

```
>>> pd.Series(Year_col).index.dtype
```


```
dtype('int64')
```

```
>>> dataset.index.dtype
```

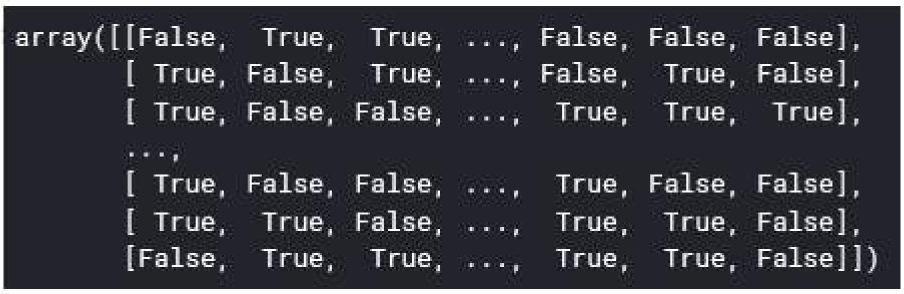

```
dtype('int64')
```

5.5.2 Dealing with missing values

There are several data sets which have missing values. This missing data is usually a result of difficulties in obtaining the values to those entries. Pandas fills missing values with “NaN” which is short for “Not a Number”.Pandas treats NaN values as floats.

In this section we will go through how to identify and replace missing values in a dataset. For that we will randomly introduce missing values to our dataset. The code below copies the the first two columns(STATE?UT and Year) from the DataFrame, stores them as series objects and drops these columns from the DataFrame(This is because we don't want to introduce missing values to the first two columns). Line-4 creates a matrix of boolean values where 40% of the entries are true.

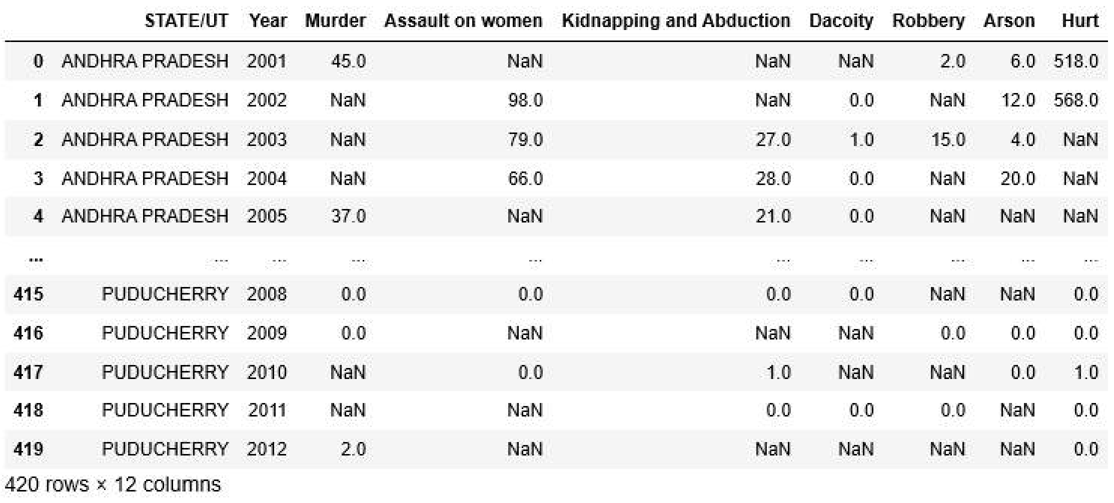
```python
1  States = pd.Series(dataset["STATE/UT"])
2  Years = pd.Series(dataset["Year"])
3  dataset_no_keys = dataset.drop(["STATE/UT", "Year"], axis = 1)
4  nan_matrix = np.random.random(dataset_no_keys.shape) < 0.4 # 40% of entries made
                                                               # NaN
5
6  nan_matrix
```



```
array([[False,  True,  True, ..., False, False, False],
       [ True, False,  True, ..., False,  True, False],
       [ True, False, False, ...,  True,  True,  True],
       ...,
       [ True, False, False, ...,  True, False, False],
       [ True,  True, False, ...,  True,  True, False],
       [False,  True,  True, ...,  True,  True, False]])
```

The mask method of a DataFrame object, for all entries in the passed array that is "True", turns those entries in the caller DataFrame to NaN. Now to this new DataFrame with missing values add the two columns that were removed earlier. This is our DataFrame with missing values.

```python
dataset_nan = dataset_no_keys.mask(nan_matrix)
dataset_nan.insert(0, "Year", Years, True)
dataset_nan.insert(0, "STATE/UT", States, True)
dataset_nan
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45.0	NaN	NaN	NaN	2.0	6.0	518.0
1	ANDHRA PRADESH	2002	NaN	98.0	NaN	0.0	NaN	12.0	568.0
2	ANDHRA PRADESH	2003	NaN	79.0	27.0	1.0	15.0	4.0	NaN
3	ANDHRA PRADESH	2004	NaN	66.0	28.0	0.0	NaN	20.0	NaN
4	ANDHRA PRADESH	2005	37.0	NaN	21.0	0.0	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0.0	0.0	0.0	0.0	NaN	NaN	0.0
416	PUDUCHERRY	2009	0.0	NaN	NaN	NaN	0.0	0.0	0.0
417	PUDUCHERRY	2010	NaN	0.0	1.0	NaN	NaN	0.0	1.0
418	PUDUCHERRY	2011	NaN	NaN	0.0	0.0	0.0	NaN	0.0
419	PUDUCHERRY	2012	2.0	NaN	NaN	NaN	NaN	NaN	0.0

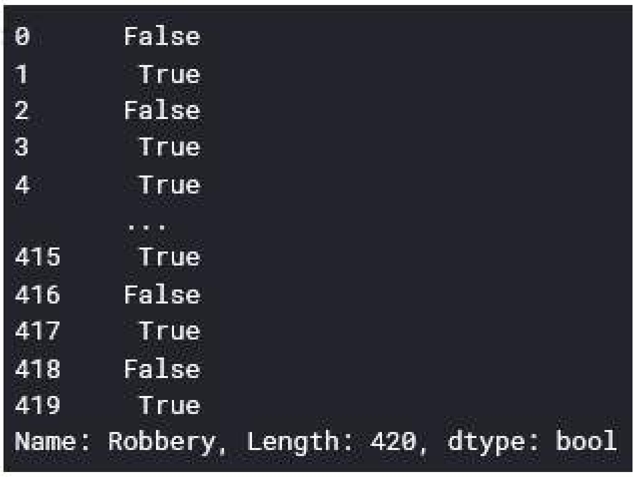
420 rows × 12 columns

Pandas has several methods that help work with missing values. The methods that can be used to select entries with missing values or entries without missing values are,

1. pd.isnull( )
2. pd.notnull( )

They can be used as shown below. When the isnull or notnull method is called with a Series object returns the a series object of type bool indicating which entries are missing in case of isnull( ) and indicates which entries are not missing in case of notnull( ).

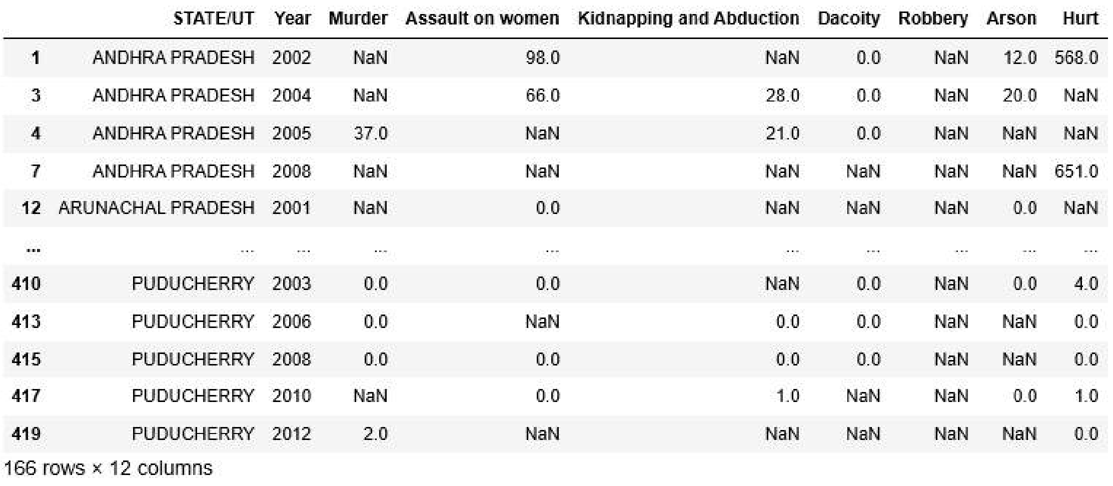
```
>>> Robbery_missing = pd.isnull(dataset_nan.Robbery) # if NaN then True else False

>>> Robbery_missing
```



```
0      False
1       True
2      False
3       True
4       True
        ...
415     True
416    False
417     True
418    False
419     True
Name: Robbery, Length: 420, dtype: bool
```

The statement shown below returns all rows where the entry for robbery was missing. This same approach can be used to obtain all the rows were entries of a column are not missing using Pandas notnull method.

```
>>> dataset_nan[pd.isnull(dataset_nan.Robbery)]
```

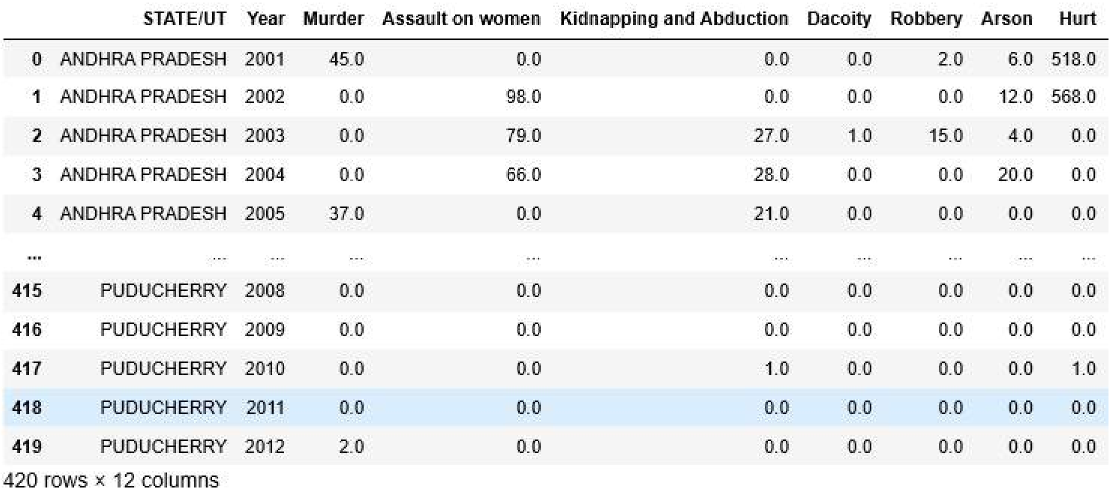


	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
1	ANDHRA PRADESH	2002	NaN	98.0	NaN	0.0	NaN	12.0	568.0
3	ANDHRA PRADESH	2004	NaN	66.0	28.0	0.0	NaN	20.0	NaN
4	ANDHRA PRADESH	2005	37.0	NaN	21.0	0.0	NaN	NaN	NaN
7	ANDHRA PRADESH	2008	NaN	NaN	NaN	NaN	NaN	NaN	651.0
12	ARUNACHAL PRADESH	2001	NaN	0.0	NaN	NaN	NaN	0.0	NaN
...	...	...	...	...	...	...	...	...	...
410	PUDUCHERRY	2003	0.0	0.0	NaN	0.0	NaN	0.0	4.0
413	PUDUCHERRY	2006	0.0	NaN	0.0	0.0	NaN	NaN	0.0
415	PUDUCHERRY	2008	0.0	0.0	0.0	0.0	NaN	NaN	0.0
417	PUDUCHERRY	2010	NaN	0.0	1.0	NaN	NaN	0.0	1.0
419	PUDUCHERRY	2012	2.0	NaN	NaN	NaN	NaN	NaN	0.0

166 rows × 12 columns

When a data set that is being worked on has missing values, a common values that is performed is replacing the NaN value to another suitable value. Pandas provides a method for this named fillna( ) which can can be called on Series and DataFrame objects and takes the value to replace the missing value with. An example for this is replacing the NaN value to 0 for an entire DataFrame.

```
>>> dataset_nan.fillna(0)
```

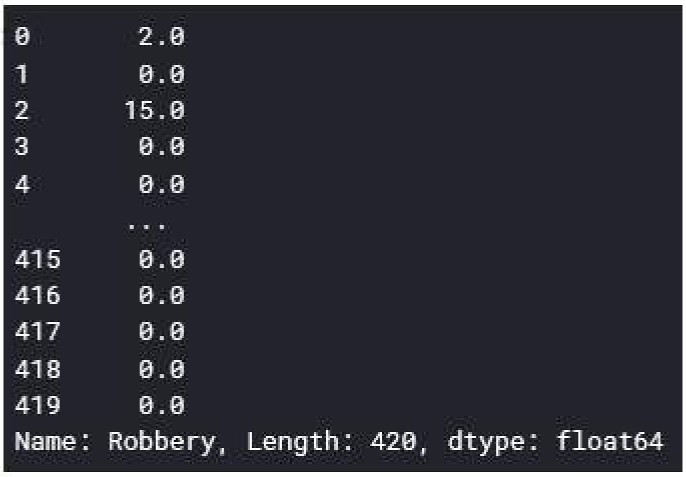


	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45.0	0.0	0.0	0.0	2.0	6.0	518.0
1	ANDHRA PRADESH	2002	0.0	98.0	0.0	0.0	0.0	12.0	568.0
2	ANDHRA PRADESH	2003	0.0	79.0	27.0	1.0	15.0	4.0	0.0
3	ANDHRA PRADESH	2004	0.0	66.0	28.0	0.0	0.0	20.0	0.0
4	ANDHRA PRADESH	2005	37.0	0.0	21.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0.0	0.0	0.0	0.0	0.0	0.0	0.0
416	PUDUCHERRY	2009	0.0	0.0	0.0	0.0	0.0	0.0	0.0
417	PUDUCHERRY	2010	0.0	0.0	1.0	0.0	0.0	0.0	1.0
418	PUDUCHERRY	2011	0.0	0.0	0.0	0.0	0.0	0.0	0.0
419	PUDUCHERRY	2012	2.0	0.0	0.0	0.0	0.0	0.0	0.0

420 rows × 12 columns

The statement below return the Robbery column of our DataFrame with all missing values replaced with 0.

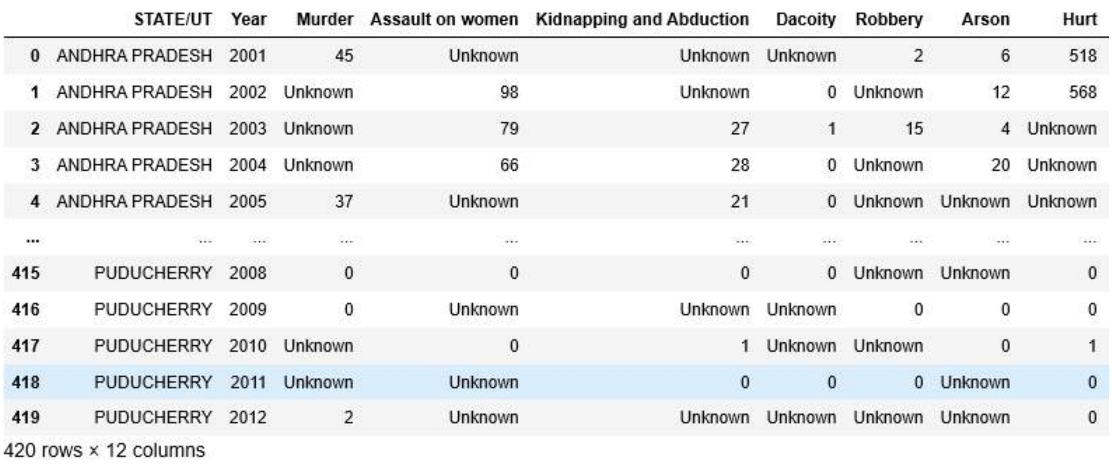
```
>>> dataset_nan.Robbery.fillna(0)
```



```
0       2.0
1       0.0
2      15.0
3       0.0
4       0.0
       ...
415     0.0
416     0.0
417     0.0
418     0.0
419     0.0
Name: Robbery, Length: 420, dtype: float64
```

Incase the dataset that is being worked on has missing values already filled in by some value, for example unknown as shown below. Pandas provides a method replace that can be used to replace a specified value with another specified value. Lets fill all missing values in our dataset with the string “Unknown”.

```
>>> dataset_nan = dataset_nan.fillna("Unknown")

>>> dataset_nan
```

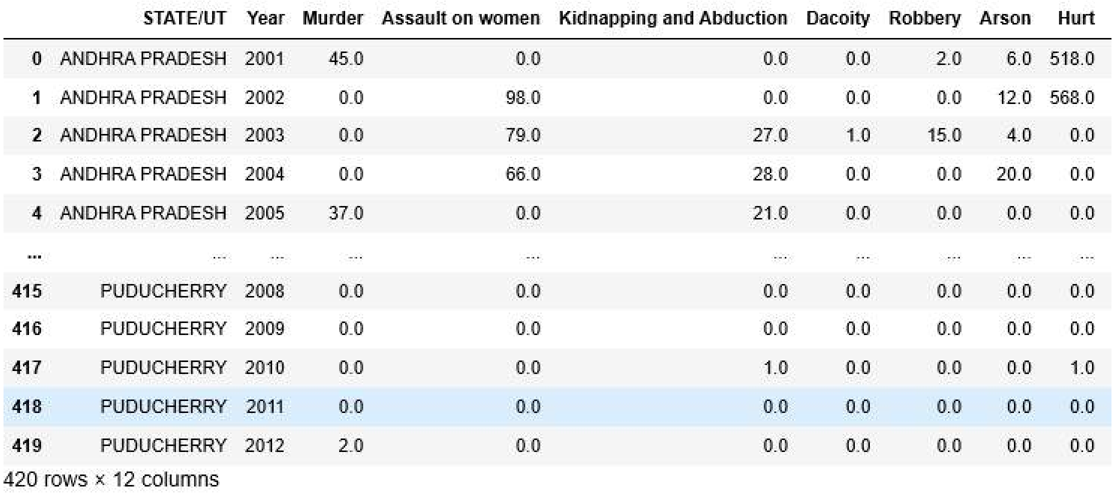


	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	Unknown	Unknown	Unknown	2	6	518
1	ANDHRA PRADESH	2002	Unknown	98	Unknown	0	Unknown	12	568
2	ANDHRA PRADESH	2003	Unknown	79	27	1	15	4	Unknown
3	ANDHRA PRADESH	2004	Unknown	66	28	0	Unknown	20	Unknown
4	ANDHRA PRADESH	2005	37	Unknown	21	0	Unknown	Unknown	Unknown
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	Unknown	Unknown	0
416	PUDUCHERRY	2009	0	Unknown	Unknown	Unknown	0	0	0
417	PUDUCHERRY	2010	Unknown	0	1	Unknown	Unknown	0	1
418	PUDUCHERRY	2011	Unknown	Unknown	0	0	0	Unknown	0
419	PUDUCHERRY	2012	2	Unknown	Unknown	Unknown	Unknown	Unknown	0

420 rows × 12 columns

We can use the replace method to replace all occurrences of "Unknown" to a value that is more preferred in our case, like the integer 0.

```
>>> dataset_nan.replace("Unknown", 0)
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45.0	0.0	0.0	0.0	2.0	6.0	518.0
1	ANDHRA PRADESH	2002	0.0	98.0	0.0	0.0	0.0	12.0	568.0
2	ANDHRA PRADESH	2003	0.0	79.0	27.0	1.0	15.0	4.0	0.0
3	ANDHRA PRADESH	2004	0.0	66.0	28.0	0.0	0.0	20.0	0.0
4	ANDHRA PRADESH	2005	37.0	0.0	21.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0.0	0.0	0.0	0.0	0.0	0.0	0.0
416	PUDUCHERRY	2009	0.0	0.0	0.0	0.0	0.0	0.0	0.0
417	PUDUCHERRY	2010	0.0	0.0	1.0	0.0	0.0	0.0	1.0
418	PUDUCHERRY	2011	0.0	0.0	0.0	0.0	0.0	0.0	0.0
419	PUDUCHERRY	2012	2.0	0.0	0.0	0.0	0.0	0.0	0.0

420 rows × 12 columns

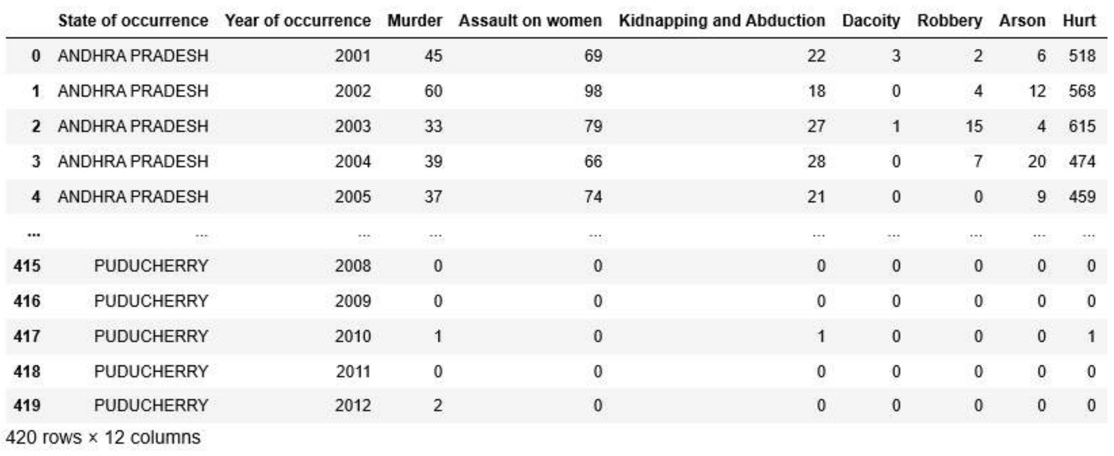
5.6 Renaming and combining

The dataset at hand very often comes with unsatisfactory column names, index names and naming conventions, in which case we will want to change the column and index names. There is also very often a need to combine multiple different DataFrame and Series objects into one object.

5.6.1 Renaming

Pandas provides a method rename( ) that can be used to change a DataFrames column names and/or index names. For example to change the "STATE/UT" column to "State of occurrence"and "Year" column to "Year of occurrence" we could use the statement shown below.

```
>>> dataset.rename(columns = {"STATE/UT" : "State of occurrence", "Year" : "Year
    of occurrence"})
```

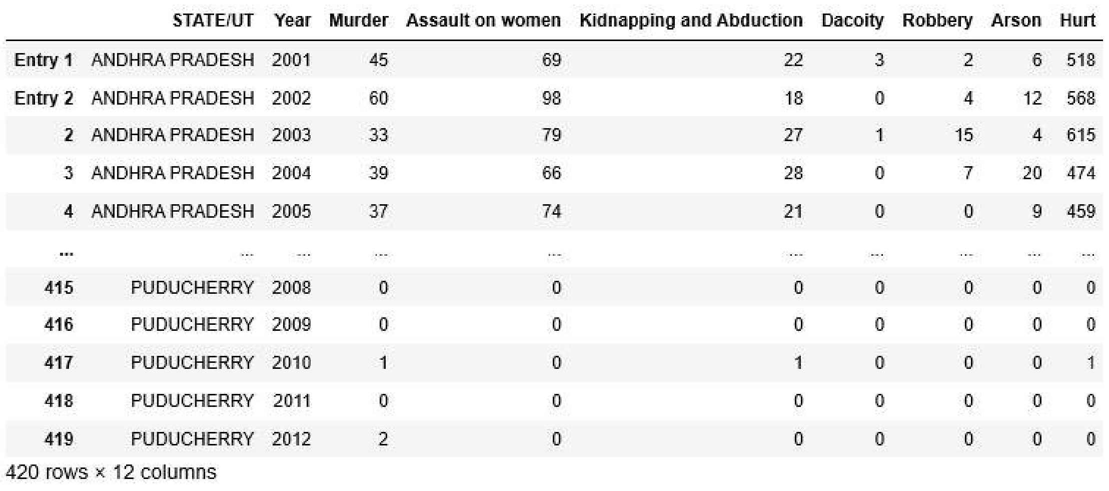


	State of occurrence	Year of occurrence	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
0	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
1	ANDHRA PRADESH	2002	60	98	18	0	4	12	568
2	ANDHRA PRADESH	2003	33	79	27	1	15	4	615
3	ANDHRA PRADESH	2004	39	66	28	0	7	20	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	9	459
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	0	0
416	PUDUCHERRY	2009	0	0	0	0	0	0	0
417	PUDUCHERRY	2010	1	0	1	0	0	0	1
418	PUDUCHERRY	2011	0	0	0	0	0	0	0
419	PUDUCHERRY	2012	2	0	0	0	0	0	0

420 rows × 12 columns

Rename can be performed on either or both index or column by specifying the index or column keyword parameter. Although this method supports several input parameters, dictionaries are the usually the most convenient. The statement below renames the first and second row indexes.

```
>>> dataset.rename(index = {0 : "Entry 1", 1 : "Entry 2"})
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
Entry 1	ANDHRA PRADESH	2001	45	69	22	3	2	6	518
Entry 2	ANDHRA PRADESH	2002	60	98	18	0	4	12	568
2	ANDHRA PRADESH	2003	33	79	27	1	15	4	615
3	ANDHRA PRADESH	2004	39	66	28	0	7	20	474
4	ANDHRA PRADESH	2005	37	74	21	0	0	9	459
...	...	...	...	...	...	...	...	...	...
415	PUDUCHERRY	2008	0	0	0	0	0	0	0
416	PUDUCHERRY	2009	0	0	0	0	0	0	0
417	PUDUCHERRY	2010	1	0	1	0	0	0	1
418	PUDUCHERRY	2011	0	0	0	0	0	0	0
419	PUDUCHERRY	2012	2	0	0	0	0	0	0

420 rows × 12 columns

Pandas also provides a method that can be used to rename the axis names. To make it easier to understand lets set both “STATE/UT” and “Year” as the index columns for our DataFrame. This can be done with the set_index( ) method of DataFrames that we have learned in the previous sections.

```
>>> dataset.set_index(["STATE/UT", "Year"])
```

STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
ANDHRA PRADESH	2001	45	69	22	3	2	6	518
	2002	60	98	18	0	4	12	568
	2003	33	79	27	1	15	4	615
	2004	39	66	28	0	7	20	474
	2005	37	74	21	0	0	9	459
...	...	...	...	...	...	...	...	...
PUDUCHERRY	2008	0	0	0	0	0	0	0
	2009	0	0	0	0	0	0	0
	2010	1	0	1	0	0	0	1
	2011	0	0	0	0	0	0	0
	2012	2	0	0	0	0	0	0

420 rows × 10 columns

The code snippet below is used to combine both index column “STATE/UT” and “Year” into a single index column.

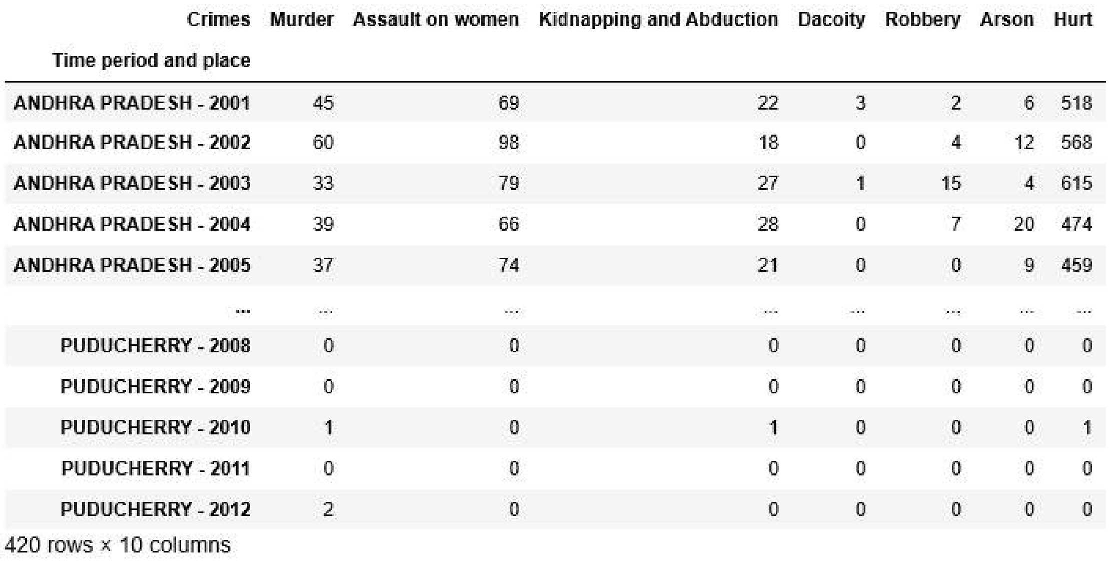
```python
Year_str = dataset.Year.map(lambda x: str(x))
place_and_year = pd.Series(dataset["STATE/UT"] + " - " + Year_str)
dataset_copy = dataset.drop(["STATE/UT", "Year"], axis = 1)
dataset_copy.insert(0, "STATE/UT - Year", place_and_year, True)
dataset_copy = dataset_copy.set_index("STATE/UT - Year")
dataset_copy
```

STATE/UT - Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
ANDHRA PRADESH - 2001	45	69	22	3	2	6	518
ANDHRA PRADESH - 2002	60	98	18	0	4	12	568
ANDHRA PRADESH - 2003	33	79	27	1	15	4	615
ANDHRA PRADESH - 2004	39	66	28	0	7	20	474
ANDHRA PRADESH - 2005	37	74	21	0	0	9	459
...	...	...	...	...	...	...	...
PUDUCHERRY - 2008	0	0	0	0	0	0	0
PUDUCHERRY - 2009	0	0	0	0	0	0	0
PUDUCHERRY - 2010	1	0	1	0	0	0	1
PUDUCHERRY - 2011	0	0	0	0	0	0	0
PUDUCHERRY - 2012	2	0	0	0	0	0	0

420 rows × 10 columns

The rename_axis( ) method can be used to rename the axis for both rows or columns. The statement below renames the axis for the rows from “STATE/UT - Year” to “Time period and place”, then renames the axis for columns to “Crimes”.

```
>>> dataset_copy.rename_axis("Time period and place", axis =
    "rows").rename_axis("Crimes", axis = "columns")
```



```
               Crimes  Murder  Assault on women  Kidnapping and Abduction  Dacoity  Robbery  Arson  Hurt
Time period and place
ANDHRA PRADESH - 2001      45                69                        22        3        2      6   518
ANDHRA PRADESH - 2002      60                98                        18        0        4     12   568
ANDHRA PRADESH - 2003      33                79                        27        1       15      4   615
ANDHRA PRADESH - 2004      39                66                        28        0        7     20   474
ANDHRA PRADESH - 2005      37                74                        21        0        0      9   459
                  ...     ...               ...                       ...      ...      ...    ...   ...
    PUDUCHERRY - 2008       0                 0                         0        0        0      0     0
    PUDUCHERRY - 2009       0                 0                         0        0        0      0     0
    PUDUCHERRY - 2010       1                 0                         1        0        0      0     1
    PUDUCHERRY - 2011       0                 0                         0        0        0      0     0
    PUDUCHERRY - 2012       2                 0                         0        0        0      0     0
420 rows × 10 columns
```

5.6.2 Combining

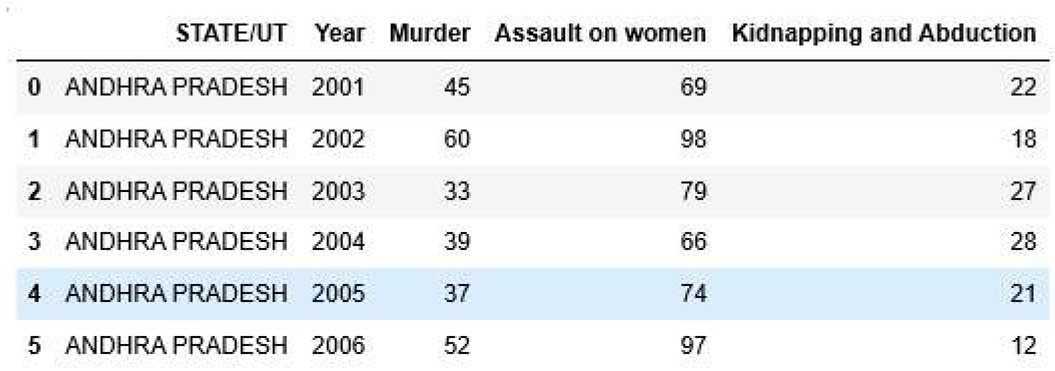
There are often cases where there is a need to combine several DataFrames and/or Series together. Pandas provides three methods to achieve this a nd they are listed below in the order of increasing complexity.

1. `concat( )`
2. `join( )`
3. `merge( )`

In this section we will learn to use the first two as most of what can be achieved using merge can also be done using the join( ) method.

To demonstrate combining DataFrames and Series we will use subsets of our dataset. The statement below creates a DataFrame consists of data entries for the first six years and 3 crime features for the first state in the dataset.

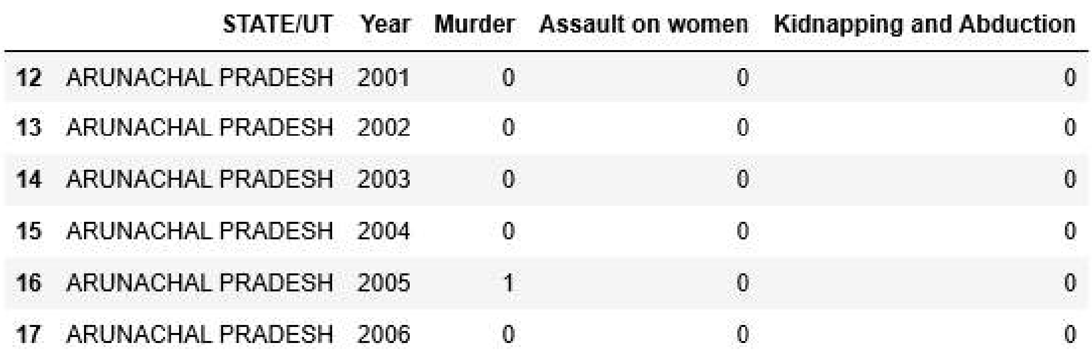
```
>>> State_1 = dataset.iloc[ : 6, : 5]

>>> State_1
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction
0	ANDHRA PRADESH	2001	45	69	22
1	ANDHRA PRADESH	2002	60	98	18
2	ANDHRA PRADESH	2003	33	79	27
3	ANDHRA PRADESH	2004	39	66	28
4	ANDHRA PRADESH	2005	37	74	21
5	ANDHRA PRADESH	2006	52	97	12

Select similar entries for the second state as shown in the statement below.

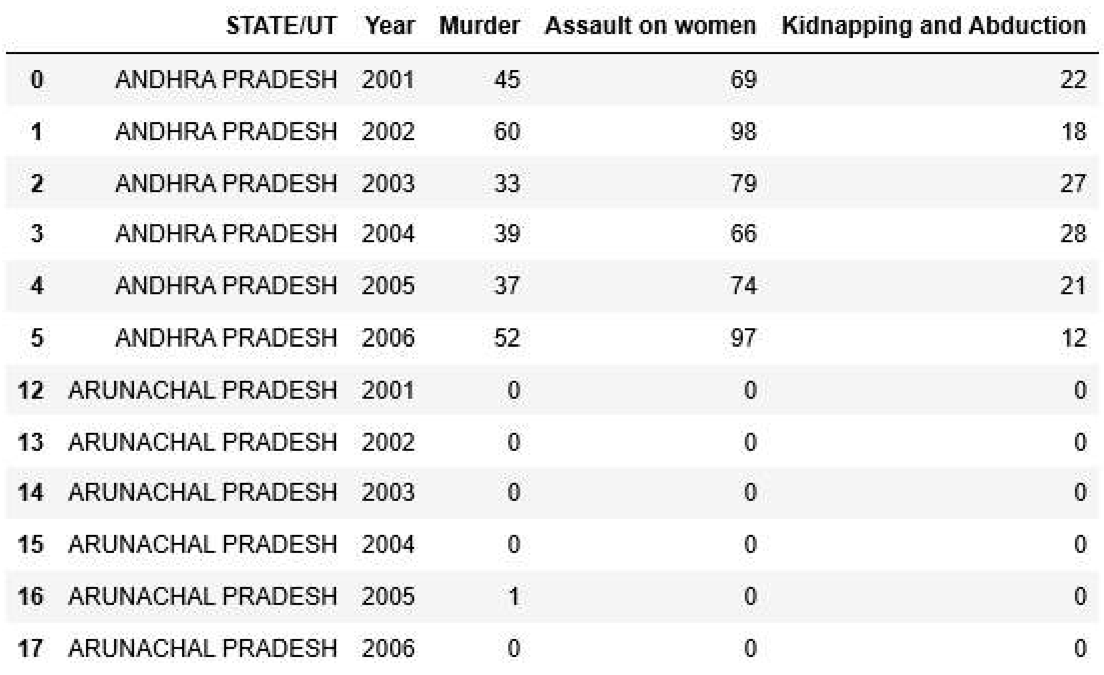
```
>>> State_2 = dataset.iloc[12 : 18, : 5]

>>> State_2
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction
12	ARUNACHAL PRADESH	2001	0	0	0
13	ARUNACHAL PRADESH	2002	0	0	0
14	ARUNACHAL PRADESH	2003	0	0	0
15	ARUNACHAL PRADESH	2004	0	0	0
16	ARUNACHAL PRADESH	2005	1	0	0
17	ARUNACHAL PRADESH	2006	0	0	0

The concat( ) method is used to combine a set of elements together along an axis as shown below.

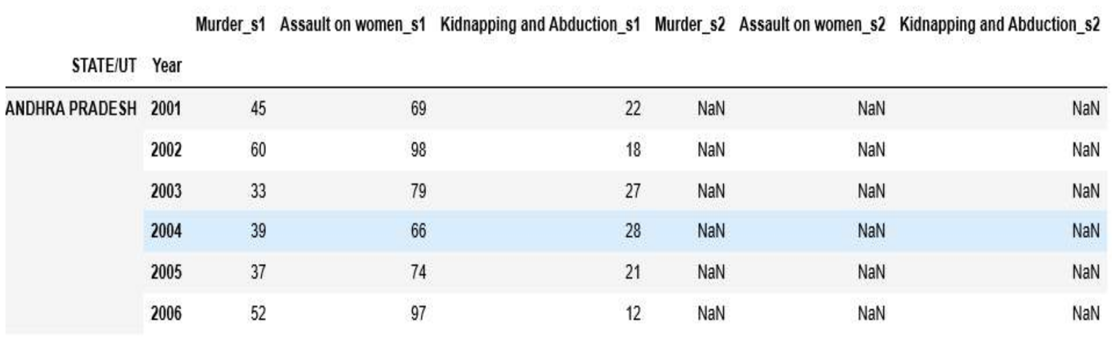
```
>>> pd.concat([State_1, State_2])
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction
0	ANDHRA PRADESH	2001	45	69	22
1	ANDHRA PRADESH	2002	60	98	18
2	ANDHRA PRADESH	2003	33	79	27
3	ANDHRA PRADESH	2004	39	66	28
4	ANDHRA PRADESH	2005	37	74	21
5	ANDHRA PRADESH	2006	52	97	12
12	ARUNACHAL PRADESH	2001	0	0	0
13	ARUNACHAL PRADESH	2002	0	0	0
14	ARUNACHAL PRADESH	2003	0	0	0
15	ARUNACHAL PRADESH	2004	0	0	0
16	ARUNACHAL PRADESH	2005	1	0	0
17	ARUNACHAL PRADESH	2006	0	0	0

The second method that Pandas provides is the join( ) method which allows combining DataFrames that have an index in common an example is shown below. In this example we create two DataFrames called “left” and “right” with the index for both being the “STATE/UT” and “Year” columns. Then the join method of the DataFrame “left” is called by passing it the DataFrame “right” as an argument and set the suffix for columns of from these two DataFrames to help us differentiate.

```
>>> left = State_1.set_index(["STATE/UT", "Year"])

>>> right = State_2.set_index(["STATE/UT", "Year"])

>>> left.join(right, lsuffix = "_s1", rsuffix = "_s2")
```



		Murder_s1	Assault on women_s1	Kidnapping and Abduction_s1	Murder_s2	Assault on women_s2	Kidnapping and Abduction_s2
STATE/UT	Year						
ANDHRA PRADESH	2001	45	69	22	NaN	NaN	NaN
	2002	60	98	18	NaN	NaN	NaN
	2003	33	79	27	NaN	NaN	NaN
	2004	39	66	28	NaN	NaN	NaN
	2005	37	74	21	NaN	NaN	NaN
	2006	52	97	12	NaN	NaN	NaN

6. Data visualization with seaborn

This chapter goes through the basics of data visualization and will help the reader get started on making their own visualizations of the data they have. We will deal with several of the most common and generally important plot types, most of which can be created using a single line of code. Before trying any of the examples of the later sections make sure you import the data, parse the dates and set Year as the index column, then import the required python libraries including matplotlib and seaborn(Which is what we will use to create visualizations).

```
>>> dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv",
     index_col= "Year", parse_dates = True)
```

```
>>> pd.plotting.register_matplotlib_converters()
```

```
>>> import matplotlib.pyplot as plt
```

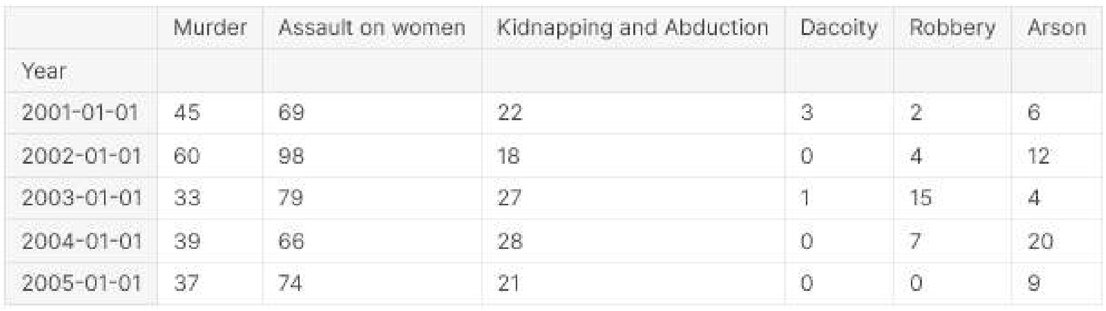
```
>>> import seaborn as sns
```

6.1 Line charts

I this section we will discuss when and how to use lineplots. A “trend” can be defined as a pattern by which something changes. Lineplots are best used to show trends over a time period. Multiple plot can be used to show how different variables change over the same time period.

From our dataset select a subset of the crimes for the first state in the dataset and display the first five rows of our selection.

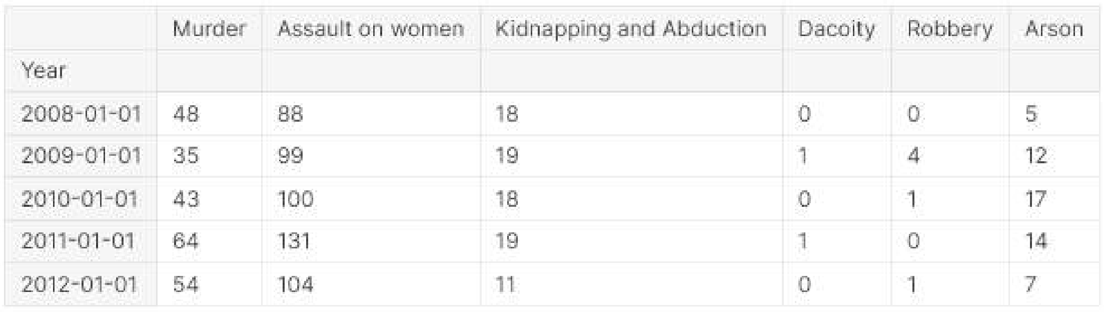
```
>>> state_1 = dataset.iloc[ : 12, 1 : 7]a
>>> state_1.head(5)
```



	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson
Year						
2001-01-01	45	69	22	3	2	6
2002-01-01	60	98	18	0	4	12
2003-01-01	33	79	27	1	15	4
2004-01-01	39	66	28	0	7	20
2005-01-01	37	74	21	0	0	9

Display the last five rows of our selection as shown below.

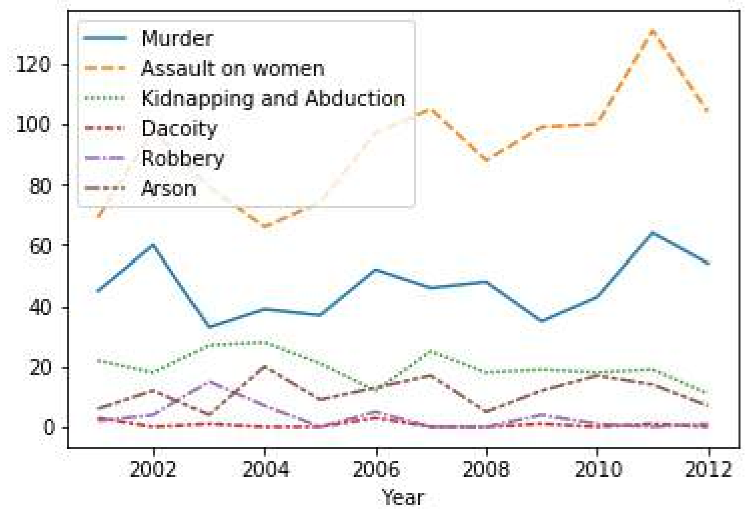
```
>>> state_1.tail(5)
```



	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson
Year						
2008-01-01	48	88	18	0	0	5
2009-01-01	35	99	19	1	4	12
2010-01-01	43	100	18	0	1	17
2011-01-01	64	131	19	1	0	14
2012-01-01	54	104	11	0	1	7

Seaborn library provides the lineplot( ) function which creates a line plot when it is passed data that it can used to plot. Line plots are generally used to show how variables change over time. The line plot for our DataFrame "state_1" is shown below.

```
>>> sns.lineplot(data = state_1)
```



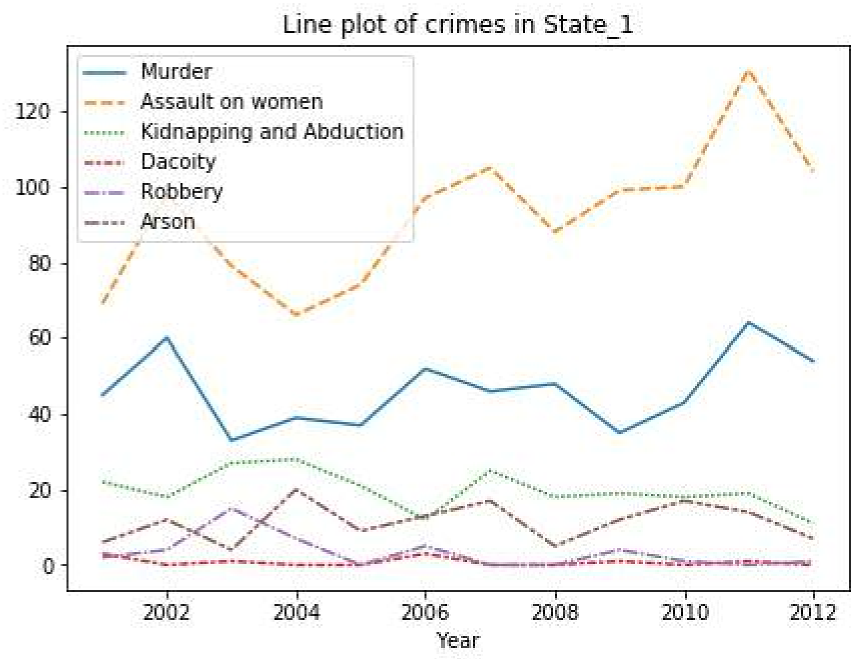
Almost every command that will be used to create a visualization in this chapter will start with "sns." which indicates that all the visualization functions being used are made available by the seaborn library. The above line of code has two parts

- sns.lineplot( ) - which indicates that a lineplot is to be created.
- data = state_1 - which indicates what data is to be used to create the plot

You may also want to change certain aspects of the visualization like the figure size and the title, that can be achieved as shown below.

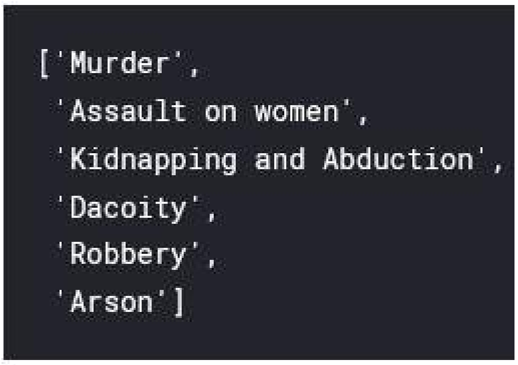
```
>>> plt.figure(figsize = (7, 5)) # Set width to 7 inches and height to 5 inches

>>> plt.title("Line plot of crimes in State_1") # Set title

>>> sns.lineplot(data = state_1)
```

Here the first line of code set the figure size to 7 inches in width and 5 inches in height. The second line set the Title of the plot to the string that is passed as an argument.



We've learned how to create a lineplot for every column of DataFrame, but what if we want to create a lineplot for a subset of the columns. Lets start by printing the columns of DataFrame "state_1".

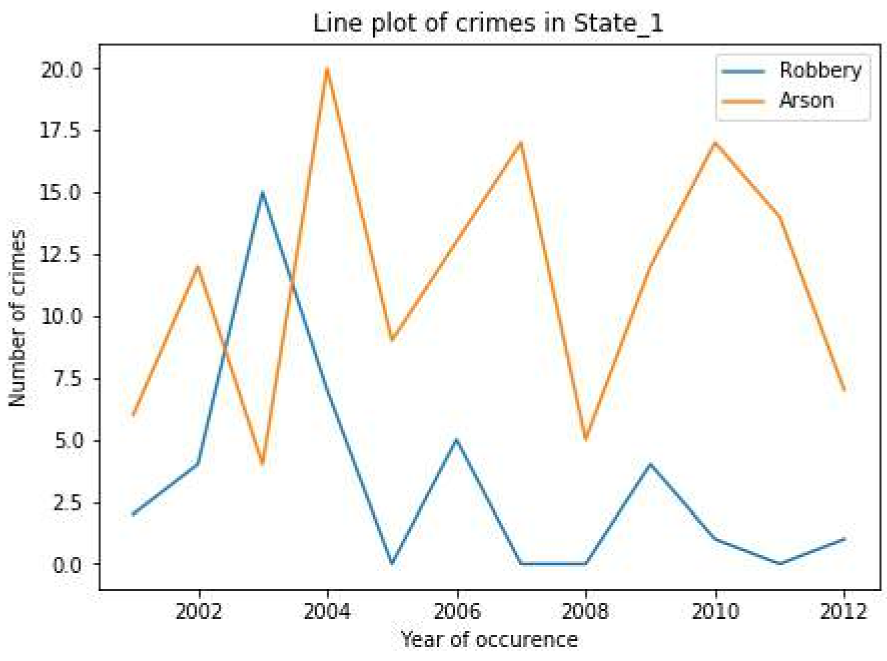
```
>>> list(state_1.columns) # list of all columns in the dataframe
```



```
['Murder',
 'Assault on women',
 'Kidnapping and Abduction',
 'Dacoity',
 'Robbery',
 'Arson']
```

Lets create a lineplot which include the columns Robbery and Arson, in which case we call the lineplot function for each column by passing the column as the value for the "data" parameter of the function call. The label parameter is used to set a user specified label for the plot of that column. The "xlabel" and "ylabel" attributes can be used to set the labels of the x an y axes of the plot.

```python
plt.figure(figsize = (7, 5))
plt.title("Line plot of crimes in State_1")

sns.lineplot(data = state_1["Robbery"], label = "Robbery")
sns.lineplot(data = state_1.Arson, label = "Arson")

plt.xlabel("Year of occurence")
plt.ylabel("Number of crimes")
```

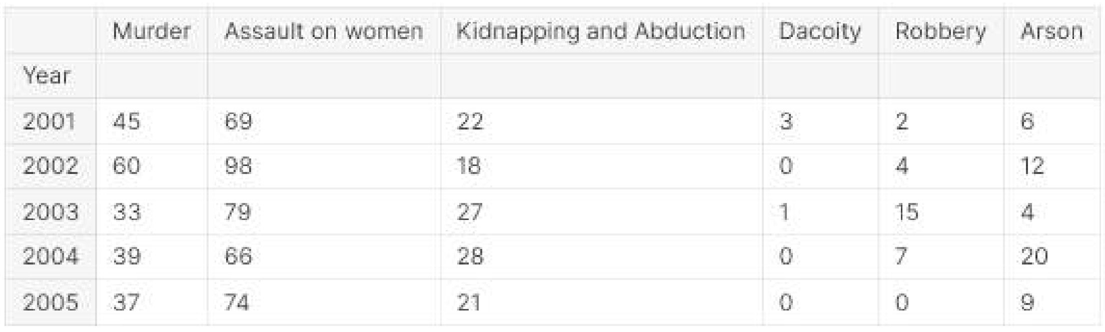


6.2 Bar charts and heatmaps

In this section we will learn of when and how to use bar charts and heat maps. We will use the same selection that was used in the last section.

```
>>> dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv",
    index_col= "Year")
```

```
>>> state_1 = dataset.iloc[ : 12, 1 : 7]
>>> state_1.head(5)
```



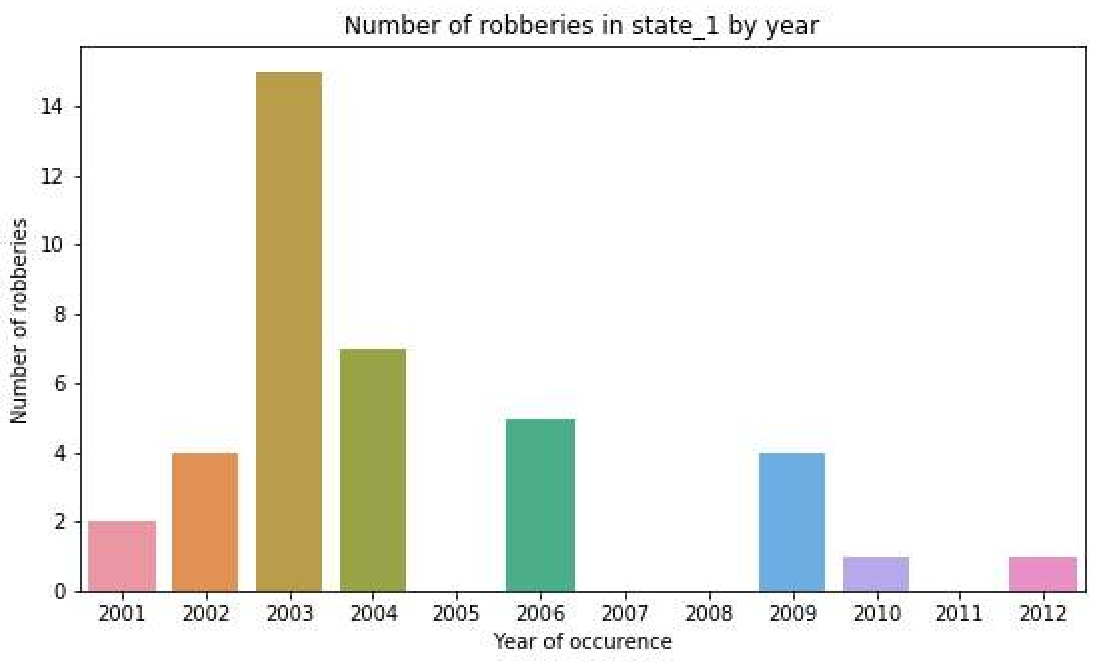
	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson
Year						
2001	45	69	22	3	2	6
2002	60	98	18	0	4	12
2003	33	79	27	1	15	4
2004	39	66	28	0	7	20
2005	37	74	21	0	0	9

6.2.1 Bar charts

Bar charts can be used to understand the relationship between different variables in your data. They help in the comparison of quantities that belong to different groups.

The code below creates a bar chart that shows the number of robberies in the first state of our dataset by year.

```python
plt.figure(figsize = (9, 5))
plt.title("Number of robberies in state_1 by year")

sns.barplot(x = state_1.index, y = state_1.Robbery)

plt.xlabel("Year of occurence")
plt.ylabel("Number of robberies")

```



Here the way to set figure size, title, x axis label and y axis label are the same as how it was done in the previous section. Line - 4 consists of three components

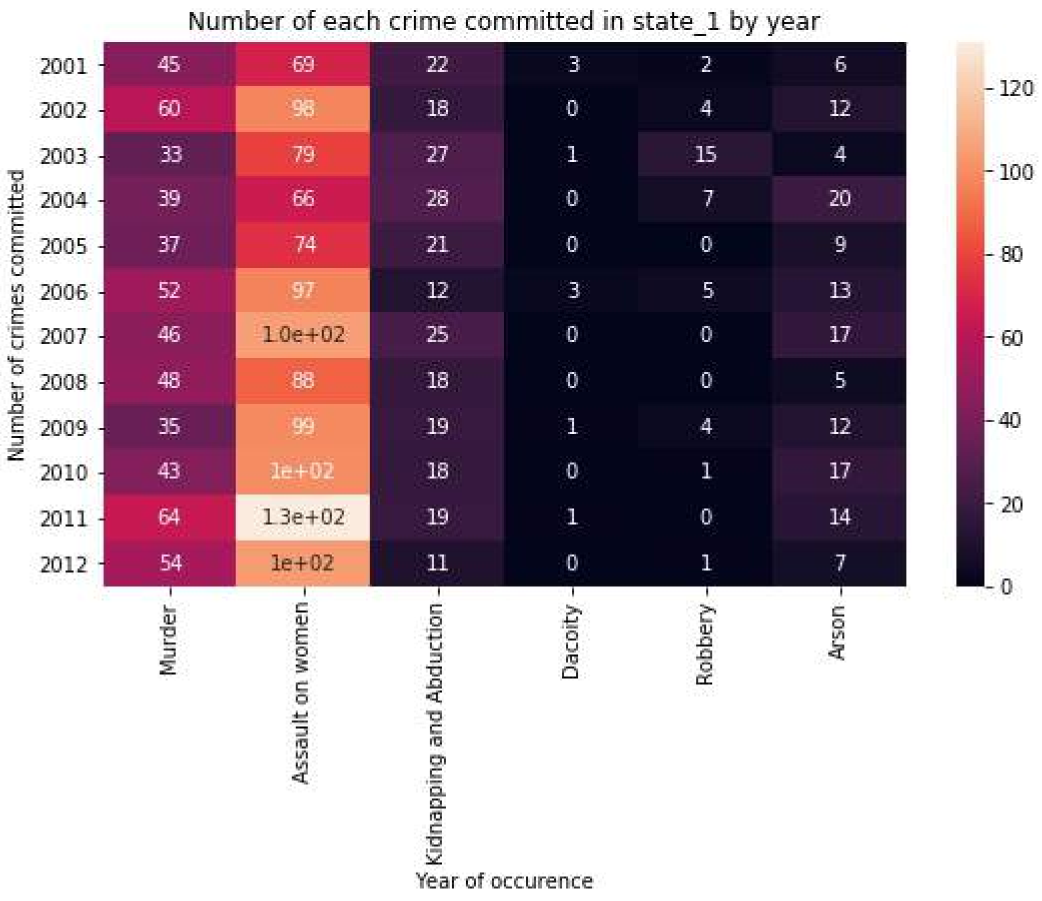
- Sns.barplot - indicates that the visualization to be created is a barplot.
- X = state_1.index - indicates that the index is used for the horizontal axis
- Y = state_1.Robbery - Setting the column that determines each bar's height

6.2.2 Heatmaps

Heatmaps can be used to understand the relationships between variables/features in data. Heatmaps provide color coded patterns in a table of integers and floating point values. Heatmaps can take a matrix of numerical values and create a chart that uses color contrast to show the relationship between variables. Here the way to set figure size, title, x axis label and y axis label are the same as how it was done in the previous section. The code below creates a histogram for the data in "state_1". The histogram fills every cell with a color based on it's numerical value. This helps us to visualize patterns in the dataset.

```
plt.figure(figsize = (9, 5))
plt.title("Number of each crime committed in state_1 by year")

sns.heatmap(data = state_1, annot = True)
```

```
5
6  plt.xlabel("Year of occurence")
7  plt.ylabel("Number of crimes committed")
8
```



Line - 4 has three components and they are,

- Sns.heatmap - indicates that the chart to draw is a heatmap.
- Data = state_1 - what data is used to draw the chart
- Annot = True - makes sure that the numerical value of each cell appear on the cell

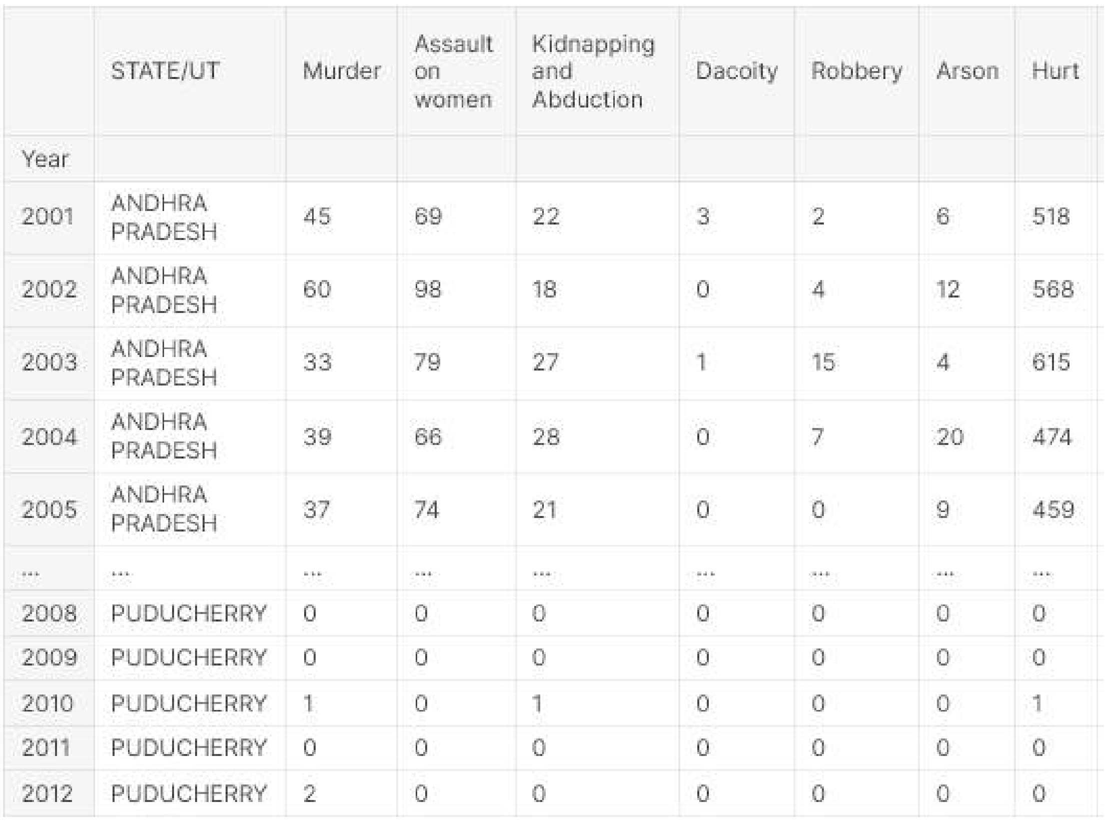
6.3 Scatter plots

Scatter plot is another plot type that helps understand the relationship between the data better. They are generally used to to visualize the relationship between two continuous variables. This can be further extended to show the relationship with another categorical variable by using color codes.

In this section we will work with the same crime data that we have been using so far.

```
>>> dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv",
    index_col= "Year")
```

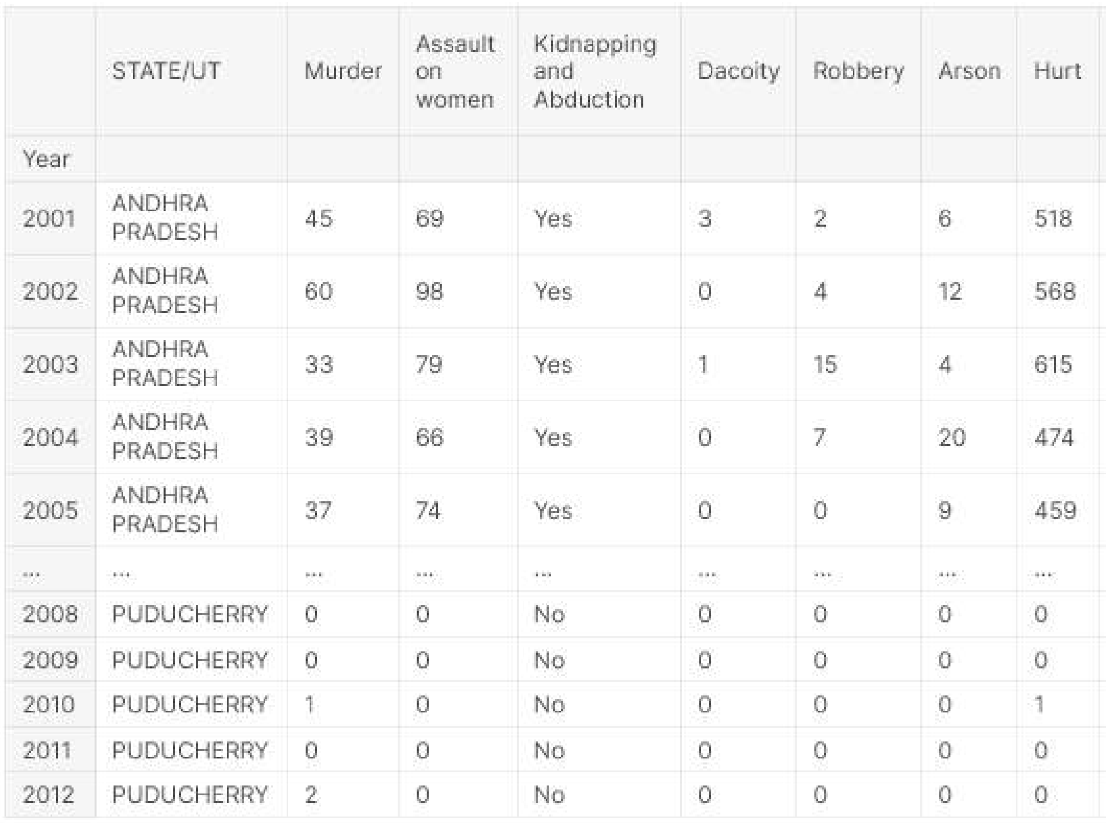
```
>>> dataset
```



	STATE/UT	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
Year								
2001	ANDHRA PRADESH	45	69	22	3	2	6	518
2002	ANDHRA PRADESH	60	98	18	0	4	12	568
2003	ANDHRA PRADESH	33	79	27	1	15	4	615
2004	ANDHRA PRADESH	39	66	28	0	7	20	474
2005	ANDHRA PRADESH	37	74	21	0	0	9	459
...	...	...	...	...	...	...	...	...
2008	PUDUCHERRY	0	0	0	0	0	0	0
2009	PUDUCHERRY	0	0	0	0	0	0	0
2010	PUDUCHERRY	1	0	1	0	0	0	1
2011	PUDUCHERRY	0	0	0	0	0	0	0
2012	PUDUCHERRY	2	0	0	0	0	0	0

For the explanation of certain concepts in this section we will use the code below to change the entry in the “Kidnapping and Abduction” column to “Yes” if the value of the entry was greater than or equal to 10, otherwise the value is changed to “No”. We define a function “is_high” for this and using the apply method of DataFrame objects we change the entries of the “Kidnapping and Abduction” column.

```
def is_high(row):
    if row["Kidnapping and Abduction"] >= 10:
        row["Kidnapping and Abduction"] = "Yes"
    else:
        row["Kidnapping and Abduction"] = "No"
    return row

dataset_copy = dataset.apply(is_high, axis = "columns")
dataset_copy
```

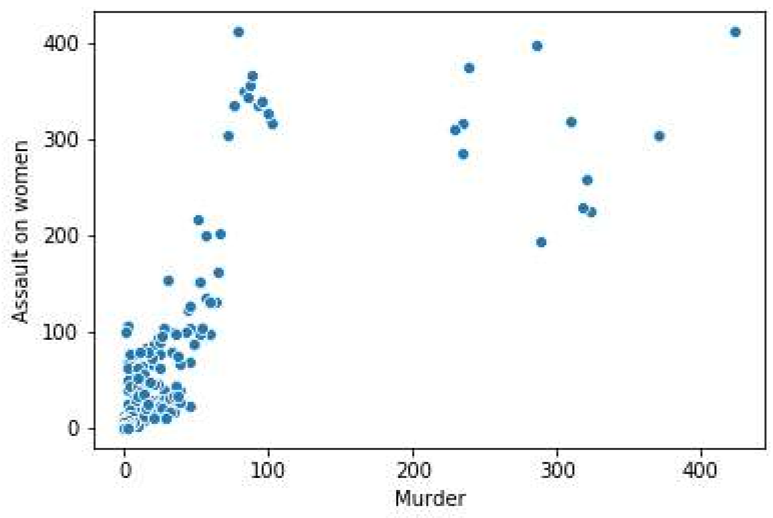


	STATE/UT	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
Year								
2001	ANDHRA PRADESH	45	69	Yes	3	2	6	518
2002	ANDHRA PRADESH	60	98	Yes	0	4	12	568
2003	ANDHRA PRADESH	33	79	Yes	1	15	4	615
2004	ANDHRA PRADESH	39	66	Yes	0	7	20	474
2005	ANDHRA PRADESH	37	74	Yes	0	0	9	459
...	...	...	...	...	...	...	...	...
2008	PUDUCHERRY	0	0	No	0	0	0	0
2009	PUDUCHERRY	0	0	No	0	0	0	0
2010	PUDUCHERRY	1	0	No	0	0	0	1
2011	PUDUCHERRY	0	0	No	0	0	0	0
2012	PUDUCHERRY	2	0	No	0	0	0	0

For creating a simple scatter plot we can use the sns.scatterplot( ) function as shown below. Here the two parameters passed are,

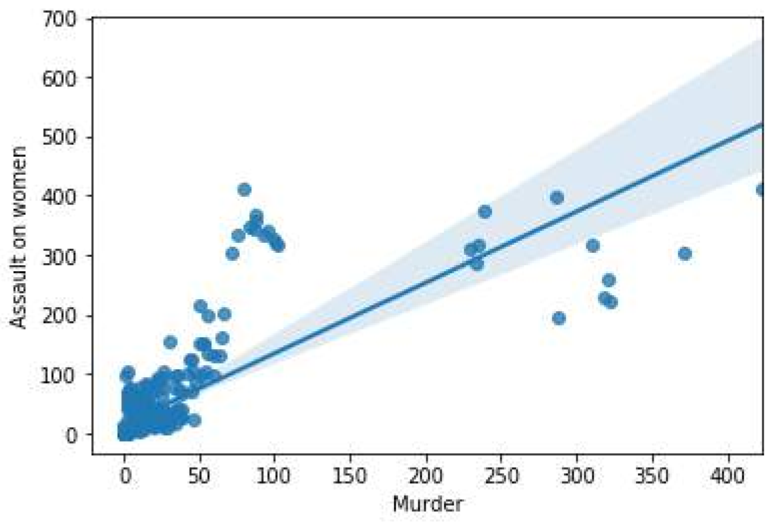
- X - variable on the horizontal axis (In this case column “Murder”)
- Y - variable on the vertical axis (In this case column “Assault on women”)

```
>>> sns.scatterplot(x = dataset_copy["Murder"], y = dataset_copy["Assault on
    women"])
```



The above scatter plot shows that both these variables are positively correlated(upward slope). This shoes that states with higher number Murders also has a high number of assaults on women. Although this relationship can be easily seen we may want to verify the strength of this relationship that we just found by adding a regression line that represents the line that best fits the data. This can simply be done by using the function sns.regplot( ) instead with the same parameters as shown below.

```
>>> sns.regplot(x = dataset_copy["Murder"], y = dataset_copy["Assault on women"])
```

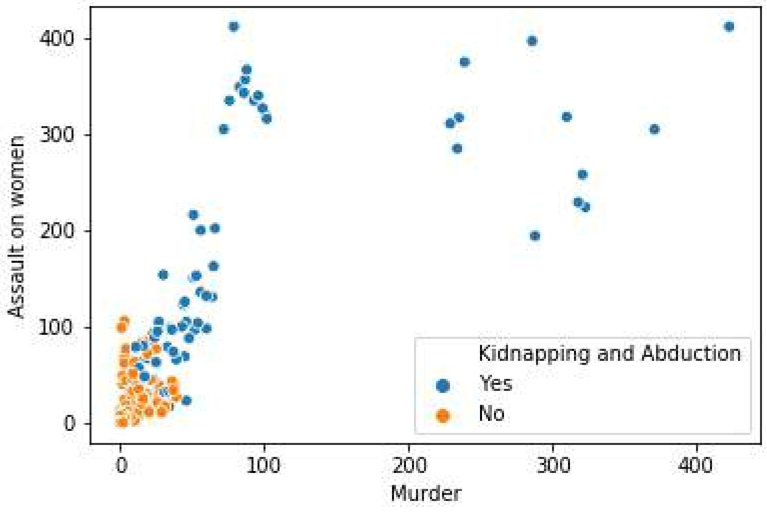


Color coded scatter plots

Scatter plots can also be used to show the relationship between three variables not necessarily using another axis for the third variable but by using color coding points instead.

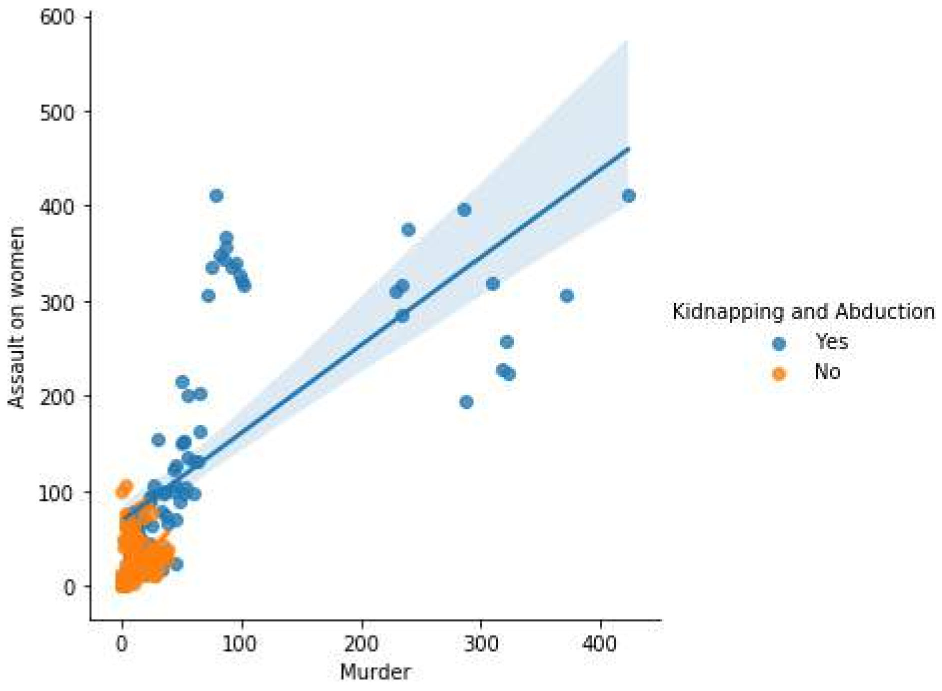
For example to see if the variable "Kidnapping and Abduction" has any relationship with these two variables that we already have a scatter plot for, we can call the scatter plot function with an added hue argument as shown below.

```
>>> sns.scatterplot(x = dataset_copy["Murder"], y = dataset_copy["Assault on
    women"], hue = dataset_copy["Kidnapping and Abduction"])
```



This scatter plot shows that states with higher number of kidnapping and abduction cases tend to have higher number of murders and assaults on women. It also shows that in states with lesser kidnapping cases the assault on women cases inrease very rapidly compared to the case where there are more number of kidnapping cases, in which the increase is slightly lower. To verify and emphasize this correctness of our observation we can use the sns.lmplot( ) function which shows lines that represent the data for every category of the categorical variable provided(Kidnapping and Abduction in this case).

```
>>> sns.lmplot(x = "Murder", y = "Assault on women", hue = "Kidnapping and Abduction",
    data = dataset_copy)
```

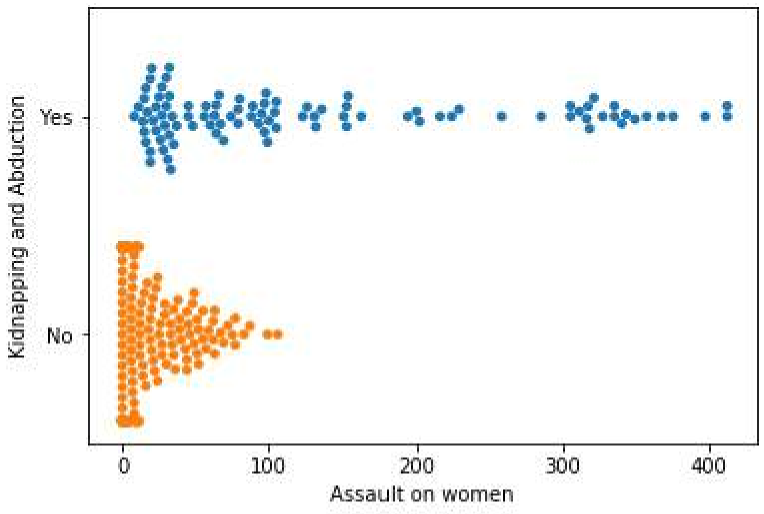


Observe that the sns.lmplot( ) differs from the scatter plot function regarding the parameters it takes as well.

- ‘x’, ‘y’ and “hue” are passed the column names instead of the entire column
- The DataFrame object to search for the column names is also passed

There is another plot type that looks different from the scatter plot that we have dealt with so far. Scatter plots we have seen so far had at least two continuous variables. Now he plot we have below shows what is called a categorical scatter plot which involves a categorical variable and a continuous variable. This type of plot can be used with the sns.swarmplot( ) function. This type of function is shown below using the variables “Kidnapping and Abduction” and “Assault on women”.

```
>>> sns.swarmplot(x = dataset_copy["Assault on women"], y =
    dataset_copy["Kidnapping and Abduction"])
```



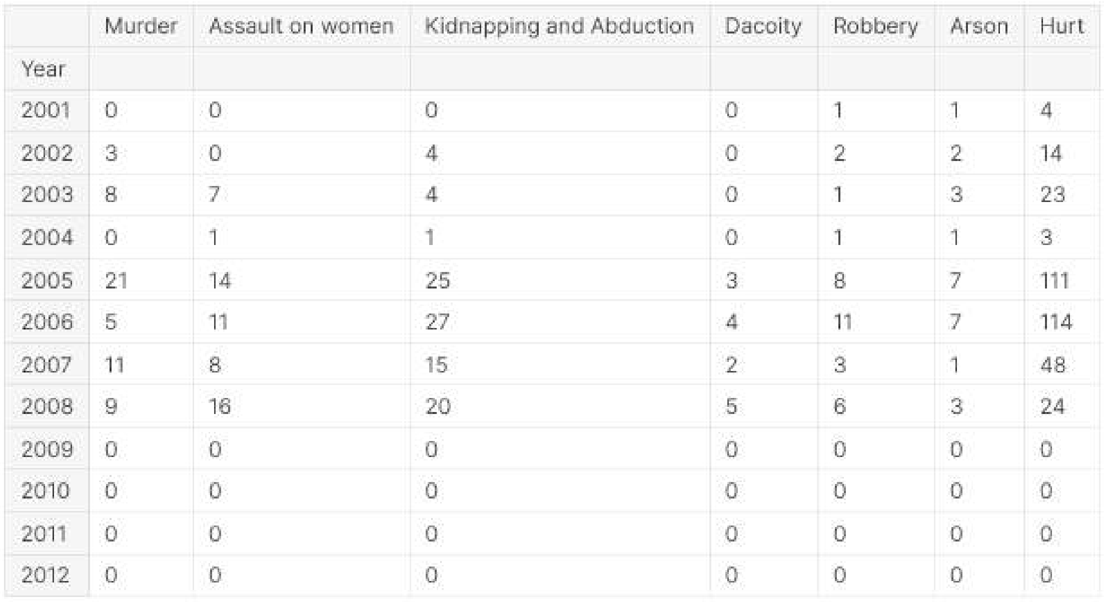
6.4 Distributions

Visualizing distributions of data help understand what the possible values that we can expect are, and how likely each value is. In this section we will cover four kinds of distribution plots and they are,

- Histograms: Can show the distribution of one numerical variable
- KDE plots: Shows a smooth estimated distribution of a single numerical variable
- 2KDE plots: Shows a smooth estimated distribution of a two numerical variables
- Joint plot: Shows 2KDE plots along with the KDE plots of each of the variables individually.

The code below loads our crime dataset, selects data for the first three states storing them in separate DataFrames and displays the data for the third state

```
dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv",
index_col= "Year")

state_1 = dataset.iloc[ : 12, 1 : 8]

state_2 = dataset.iloc[12 : 24, 1 : 8]

state_3 = dataset.iloc[24 : 36, 1 : 8]

state_3
```



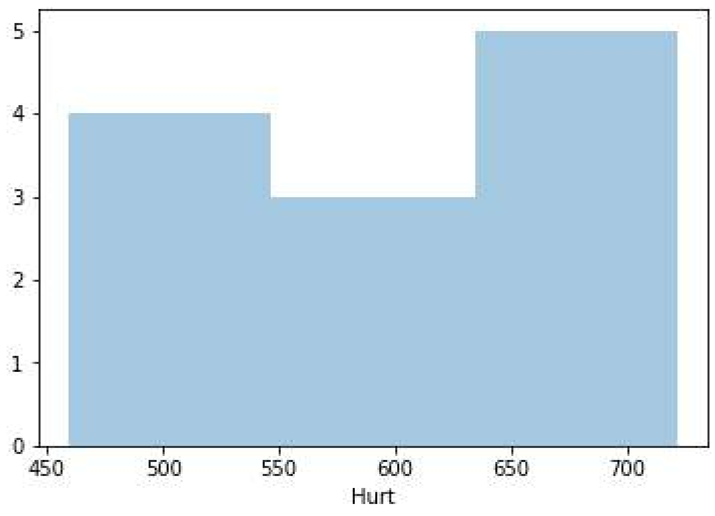
	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
Year							
2001	0	0	0	0	1	1	4
2002	3	0	4	0	2	2	14
2003	8	7	4	0	1	3	23
2004	0	1	1	0	1	1	3
2005	21	14	25	3	8	7	111
2006	5	11	27	4	11	7	114
2007	11	8	15	2	3	1	48
2008	9	16	20	5	6	3	24
2009	0	0	0	0	0	0	0
2010	0	0	0	0	0	0	0
2011	0	0	0	0	0	0	0
2012	0	0	0	0	0	0	0

Histograms

The function sns.distplot( ) can be used to create a histogram. To create a histogram to visualize the distribution of the "Hurt" column of the first state we can use the function call shown below. Here the arguments are,

- a: the column to plot
- kde: False is a value to provide here. if it is not provided we may get a different result.

```
>>> sns.distplot(a = state_1["Hurt"], kde = False)
```



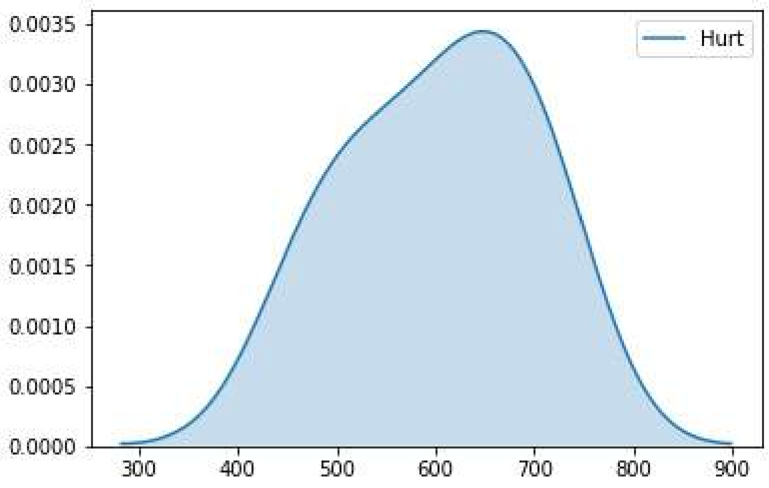
Density Plots

Now lets discuss the “kernel density estimate plot”(kdeplot). This type of plot can be easily understood as a smoothed histogram. To make a kdeplot we can use the sns.kdeplot command.

We set the following arguments to the sns.kdeplot command:

- data: It’s purpose is similar to that of the purpose of the data argument in the sns.distplot command
- shade: Setting it’s value to true colors the area under the curve

```
>>> sns.kdeplot(data = state_1["Hurt"], shade = True)
```



2D kde plots

We're not limited to simply a single dimension for kde plots. To create kde plots with 2 dimensions/columns we can use the sns.jointplot command.

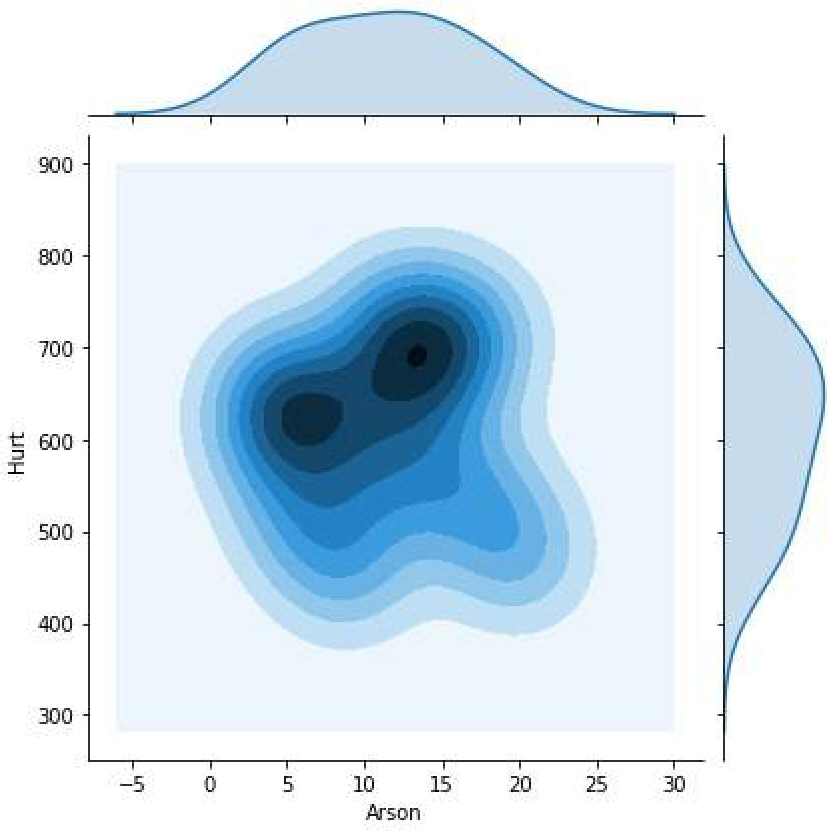
We set the following arguments to the sns.jointplot command:

- x: used to identify the first column to plot
- y: used to identify the second column to plot
- kind: this parameter is set to kde indicating that we are plotting a kde plot.

The color coding tin the plot below is meant to help us understand how likely each part of the plot is. Where the relatively darker regions are highly likely and the relatively lighter regions are less likely to occur.

Also in the plot below observe that in addition the 2D KDE plot that is shown in the centre of the plot, we also have the kde plot of each of the variables made available on the top and right side of plot. Here the kde plot on the top is for the data on the x-axis, the kde plot on the right is for the data on the y-axis.

```
>>> sns.jointplot(x = state_1["Arson"], y = state_1["Hurt"], kind = "kde")
```

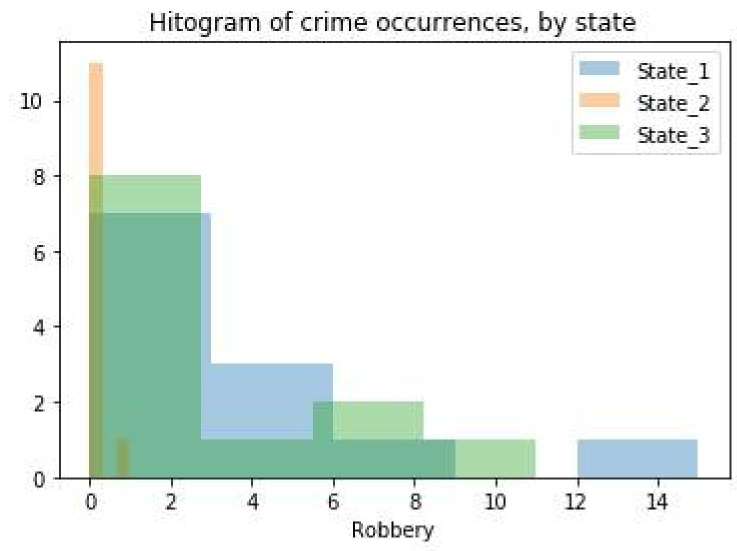


Color coded plots

Now we’ll cover plots that can be used to understand the difference between individual states. We’ll see how state 1, state 2 and state 3 differ when it comes to the crime “Robbery”. For plotting color coded plots we’ll be using the sns.displot command for the “Robbery” column of each state that we are looking for.

In the code cell below we create histograms for “Robbery” column of state 1, state 2 and state 3 using the sns.distplot command. Observe that we assign a label to each plot we create. Here the legend does not automatically appear on the plot, so to force it to appear we use the plt.legend( ) function

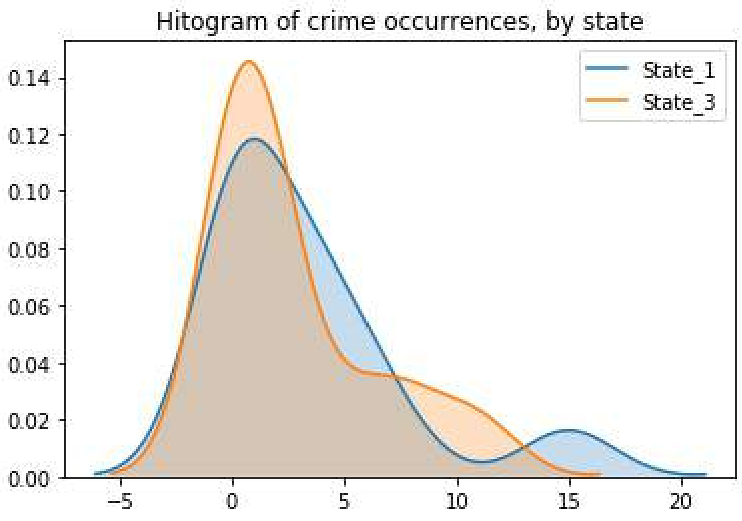
```python
sns.distplot(a = state_1["Robbery"],label = "State_1",  kde = False)
sns.distplot(a = state_2["Robbery"],label = "State_2",  kde = False)
sns.distplot(a = state_3["Robbery"],label = "State_3",  kde = False)

plt.title("Hitogram of crime occurrences, by state")
plt.legend()
```



Observe that all of these three states have most of the values of the Robbery column close to 0 and that state 1 has once had around 14 cases.

We can also create a KDE plot for every state using the sns.kdeplot command. Just as in the previous example make sure you assign the labels for the column values when calling the sns.kdeplot command

```
sns.kdeplot(data = state_1["Robbery"],label = "State_1",  shade = True)
sns.kdeplot(data = state_3["Robbery"],label = "State_3",  shade = True)

plt.title("Hitogram of crime occurrences, by state")
```



Observe that for both these states majority of values lies close to 0.

In the next chapters we'll got through creating your first machine learning models and the entire machine learning work flow.

7. Introduction to machine learning

Before we begin creating powerful machine learning models, we'll first go through what machine learning models are, how they work and how they can be used. Readers who are already familiar with statistical modelling and machine learning before can skip ahead to sections that are suitable to their level.

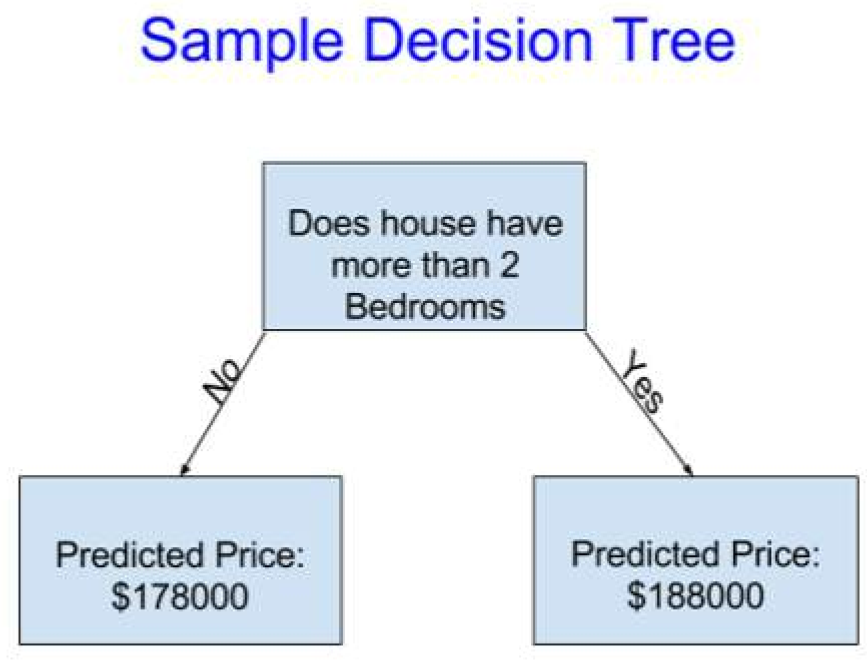
7.1 Introduction

In this and the next chapter you will create machine learning models to predict the number of murders that occurred in the state by using only the number of each the other crimes that occurred in the same state in that year. We will start by creating basic models with lower accuracy and will gradually build more complex models that have better prediction accuracy.

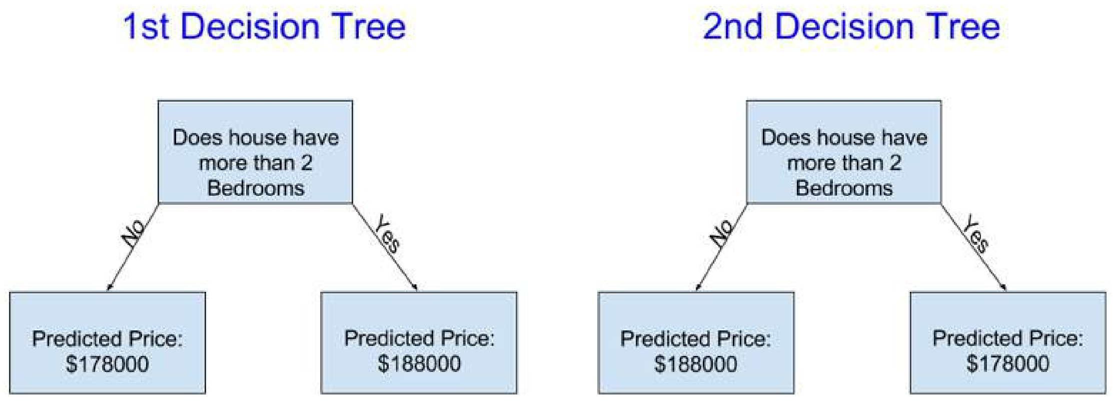
To understand how machine learning works let's begin with a prediction model called "decision tree". Although decision trees are quite simple compared to the other more complex models out there we'll be using decision tree's to understand how machine learning models work because they are quite easy to comprehend and also that they are also foundations to other powerful machine learning models.

To explain the working of decision tree's we'll be using the following scenario. Your friend has made a lot of money simply by real estate speculation. He wants to become a business partner with you because he's found out that you're a data science genius. The deal was that you should create models that can predict how much each of the houses are worth and he'll bring in the money. To get some insight into how your friend use to predict the prices in the past, you ask him how he makes his prediction, to which he responds by saying that he studied price patterns over a period of time and he was able to use this knowledge to predict trends in future prices. This is exactly what is done in machine learning. We use algorithms that learn from historic data to predict future trends and values.

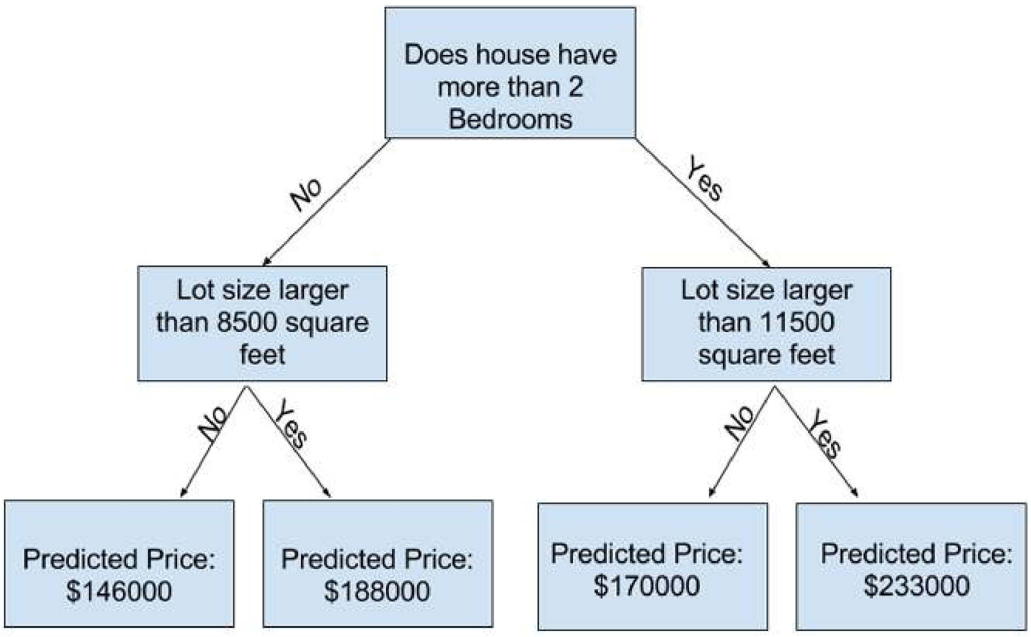
Given below is a simple decision tree that that was trained to predict the house price to be one of two values based on the historical average price of the house that belong to the same category. The decision tree algorithm uses the data to decide how to split the houses into different groups and also uses this data to decide what price to assign to each leaf node of the tree. This part where the machine learning algorithm learns from the data is called fitting or training. The data that the algorithm uses for training is called the training data. After the data has been got you can uses this trained model to predict new or future data



Take a look at the two decision trees down below. Realistically speaking the decision tree on the right is more likely to be the one that was trained with proper data as it makes more sense that houses with more bedrooms cost more. The biggest shortcoming of this division tree is that it does not use other important factors that affect the price of houses like the size of the lot, location, number of bathrooms, etc.



More factors can be included in the decision making process by using what is called “splits”. More splits can be used in decision trees to create deeper trees. The tree shown below includes lot size in its decision making process, obviously making prediction of the price that are more accurate than the previous decision tree.


Does house have more than 2 Bedrooms
No
Yes
Lot size larger than 8500 square feet
Lot size larger than 11500 square feet
No
Yes
No
Yes
Predicted Price: $146000
Predicted Price: $188000
Predicted Price: $170000
Predicted Price: $233000

When it comes to decision trees the training phase involves learning from the data to create an optimal decision tree and the prediction is done by traversing from the top of the tree to one of the leaves which indicates the prediction of the tree.

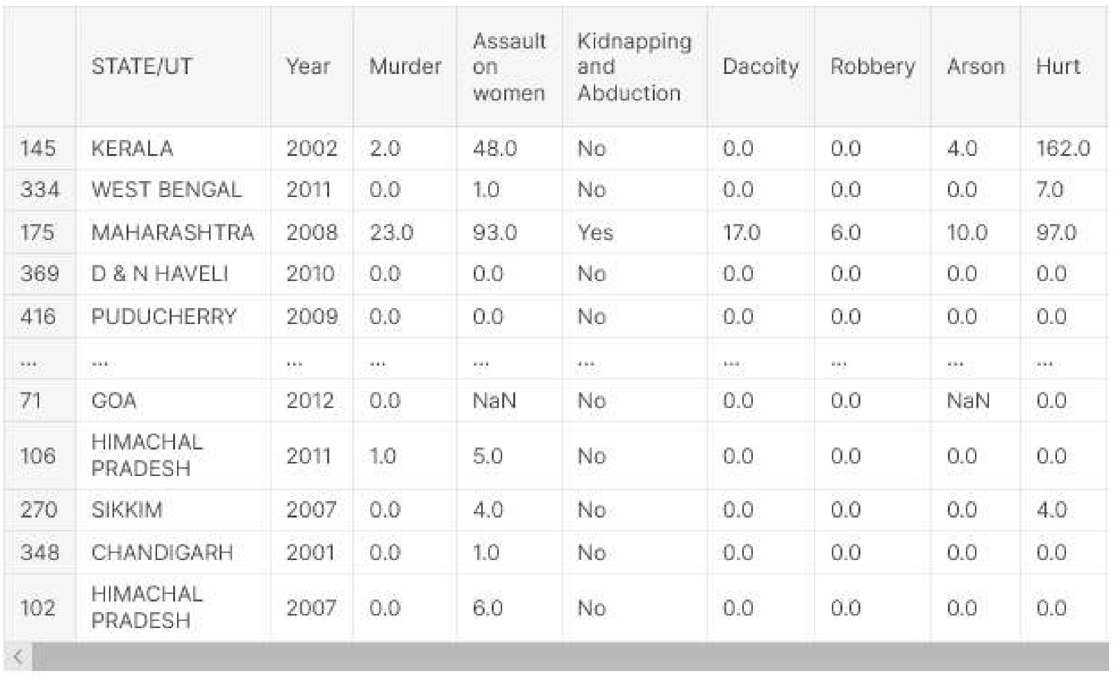
7.2 Creating your first AI model

In this section we'll be creating your first machine learning model that can predict the number of murders that occurred in a specific state and year by looking at the number of each of the other crimes that occurred in the same state and year.

Before we get started with creating our machine learning model let’s start by preparing our data that we will be using to create our prediction model. In the code listing below we perform two operations to our dataframe that we have been using all along, “dataset”, and they are,

1. Replace all entries of kidnapping column to “yes” if the value of entry is more than 5, otherwise replace it with “no”. This is to indicate whether the number of kidnappings is high or low.
2. Randomly replace 2% of all entries to nan. This is for helping us learn to deal with null values

```python
import pandas as pd

dataset = pd.read_csv("../input/crimeanalysis/crime_by_state_rt.csv")
dataset = dataset.sample(frac = 1, random_state = 42)

States = pd.Series(dataset["STATE/UT"])
Years = pd.Series(dataset["Year"])
dataset = dataset.drop(["STATE/UT", "Year"], axis = 1)

def is_high(row):
    if row["Kidnapping and Abduction"] >= 5:
        row["Kidnapping and Abduction"] = "Yes"
    else:
        row["Kidnapping and Abduction"] = "No"
    return row

dataset = dataset.apply(is_high, axis = "columns")

np.random.seed(42)
nan_matrix = np.random.random(dataset.shape) < 0.02 # 2% of entries made NaN
dataset = dataset.mask(nan_matrix)

dataset.insert(0, "Year", Years, True)
dataset.insert(0, "STATE/UT", States, True)

dataset
```

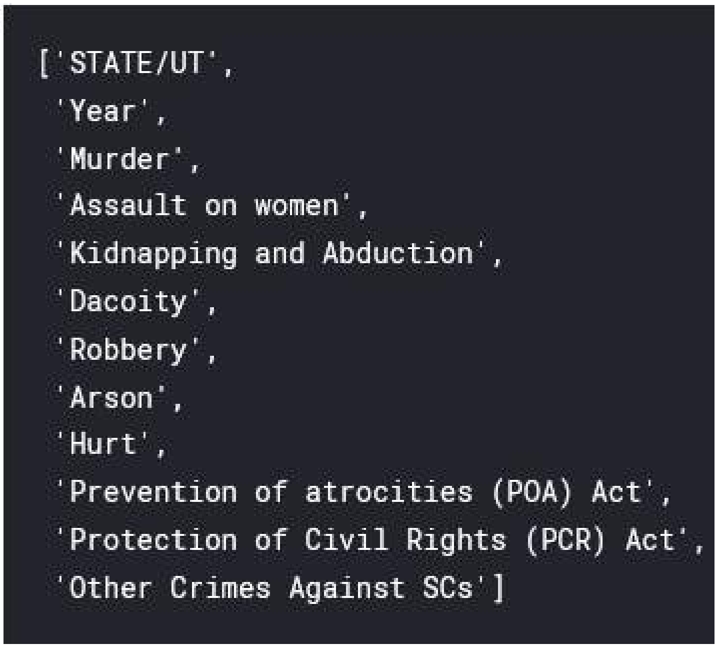


	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
145	KERALA	2002	2.0	48.0	No	0.0	0.0	4.0	162.0
334	WEST BENGAL	2011	0.0	1.0	No	0.0	0.0	0.0	7.0
175	MAHARASHTRA	2008	23.0	93.0	Yes	17.0	6.0	10.0	97.0
369	D & N HAVELI	2010	0.0	0.0	No	0.0	0.0	0.0	0.0
416	PUDUCHERRY	2009	0.0	0.0	No	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...
71	GOA	2012	0.0	NaN	No	0.0	0.0	NaN	0.0
106	HIMACHAL PRADESH	2011	1.0	5.0	No	0.0	0.0	0.0	0.0
270	SIKKIM	2007	0.0	4.0	No	0.0	0.0	0.0	4.0
348	CHANDIGARH	2001	0.0	1.0	No	0.0	0.0	0.0	0.0
102	HIMACHAL PRADESH	2007	0.0	6.0	No	0.0	0.0	0.0	0.0

420 rows × 12 columns

You will often find it difficult to decide what features you are going to feed to your model so it can learn patterns from it to predict the target variable. To make things easy it’s best to print out all the available columns. This can be done by accessing the columns data attribute of a dataframe as shown below

```
>>> list(dataset.columns)
```



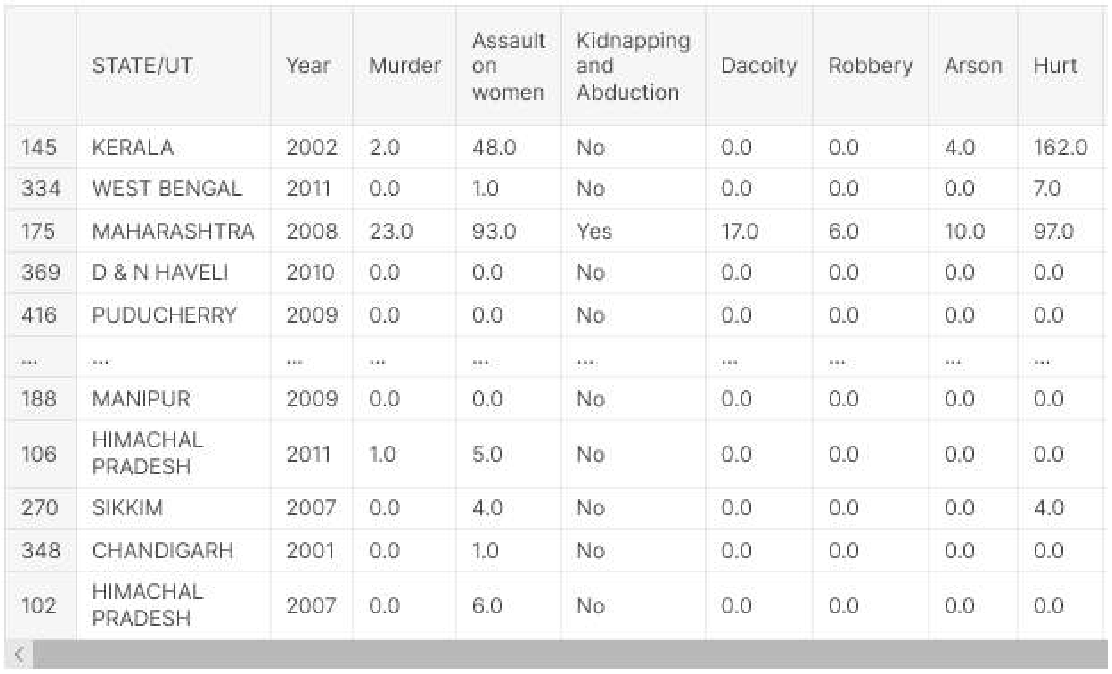
```
['STATE/UT',
 'Year',
 'Murder',
 'Assault on women',
 'Kidnapping and Abduction',
 'Dacoity',
 'Robbery',
 'Arson',
 'Hurt',
 'Prevention of atrocities (POA) Act',
 'Protection of Civil Rights (PCR) Act',
 'Other Crimes Against SCs']
```

For now let's drop all rows that have any null values by using the dropna method of data frames and setting the axis parameter to 0.

```
>>> dataset = dataset.dropna(axis = 0)
```

Now let's see how the dataframe that we'll be using to train our first machine learning model looks like.

```
>>> dataset
```



	STATE/UT	Year	Murder	Assault on women	Kidnapping and Abduction	Dacoity	Robbery	Arson	Hurt
145	KERALA	2002	2.0	48.0	No	0.0	0.0	4.0	162.0
334	WEST BENGAL	2011	0.0	1.0	No	0.0	0.0	0.0	7.0
175	MAHARASHTRA	2008	23.0	93.0	Yes	17.0	6.0	10.0	97.0
369	D & N HAVELI	2010	0.0	0.0	No	0.0	0.0	0.0	0.0
416	PUDUCHERRY	2009	0.0	0.0	No	0.0	0.0	0.0	0.0
...	...	...	...	...	...	...	...	...	...
188	MANIPUR	2009	0.0	0.0	No	0.0	0.0	0.0	0.0
106	HIMACHAL PRADESH	2011	1.0	5.0	No	0.0	0.0	0.0	0.0
270	SIKKIM	2007	0.0	4.0	No	0.0	0.0	0.0	4.0
348	CHANDIGARH	2001	0.0	1.0	No	0.0	0.0	0.0	0.0
102	HIMACHAL PRADESH	2007	0.0	6.0	No	0.0	0.0	0.0	0.0

341 rows × 12 columns

The variable that we want our model to predict is the number of murders, so let's assign that column of the dataframe to a variable "y" so it can be referenced for later use.

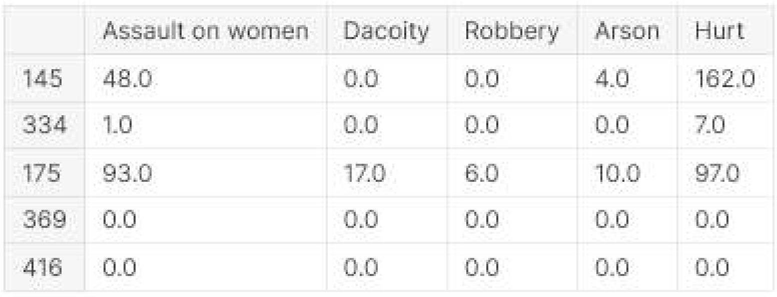
```
>>> y = dataset.Murder
```

We also want to be able to access the data from the columns that we will use to train our model, so the code below creates a list of features that we selected to train our model and we create the variable ”x” that can access all these columns of our data frame

```
>>> selected_features = ['Assault on women','Dacoity','Robbery','Arson','Hurt']
>>> X = dataset[selected_features]
```

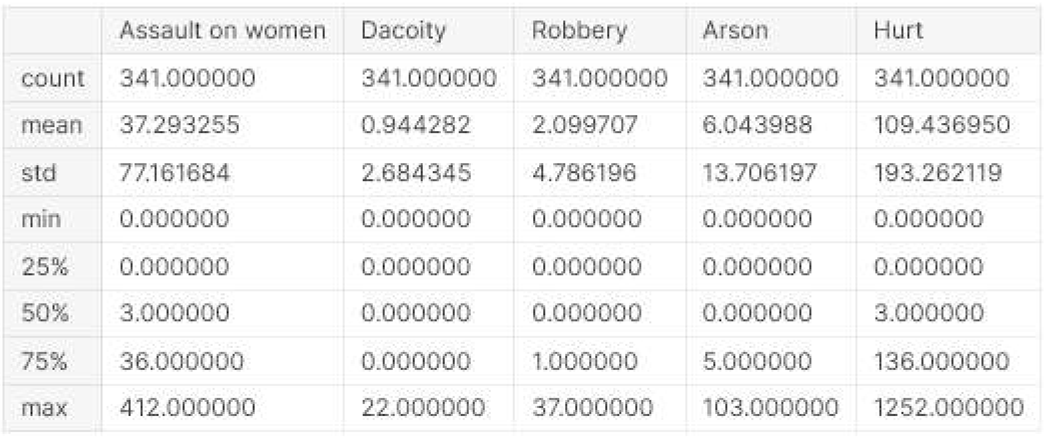
Let's have a closer look at the data that we will be using to train or model with the head and describe functions.

```
>>> X.head(5)
```



	Assault on women	Dacoity	Robbery	Arson	Hurt
145	48.0	0.0	0.0	4.0	162.0
334	1.0	0.0	0.0	0.0	7.0
175	93.0	17.0	6.0	10.0	97.0
369	0.0	0.0	0.0	0.0	0.0
416	0.0	0.0	0.0	0.0	0.0

```
>>> X.describe()
```



	Assault on women	Dacoity	Robbery	Arson	Hurt
count	341.000000	341.000000	341.000000	341.000000	341.000000
mean	37.293255	0.944282	2.099707	6.043988	109.436950
std	77.161684	2.684345	4.786196	13.706197	193.262119
min	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.000000	0.000000	0.000000	0.000000	0.000000
50%	3.000000	0.000000	0.000000	0.000000	3.000000
75%	36.000000	0.000000	1.000000	5.000000	136.000000
max	412.000000	22.000000	37.000000	103.000000	1252.000000

In this and the following sections we will be using the sci-kit learn(sklearn when your coding and importing) library to build your machine learning models. Why we are using sci-kit learn is because it is one of the most easy to use and one of the most popular machine learning libraries that is open source and can be used to model daat that is stored in data frames.

There are 4 steps when it comes to building and using a model and they are,

1. Define - Defining what type of model it is and what are its parameters
2. Fit - Provide it data to learn from and extract patterns from
3. Evaluate - Determine if the accuracy of prediction is satisfactory
4. Predict - Use the learnt patterns to predict for future cases

To create the decision tree regression models that we are looking for, we should first import the decisiontreeregressor model from the sklearn.tree module. Make sure you install scikit learn before you try importing it in your code.

```
>>> from sklearn.tree import DecisionTreeRegressor
```

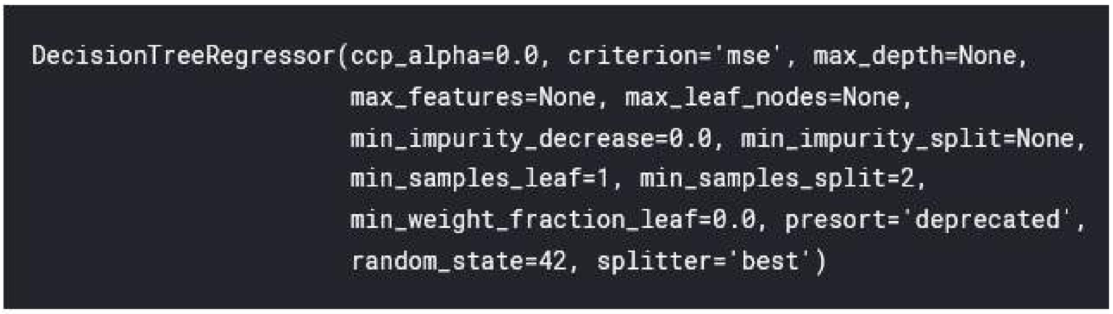
Now that you have the decisiontreeregressor class in your environment, create a new instance of a decision tree regressor by calling it’s constructor with the argument random_state set to any integer. Setting random_state to a specific value is important because by not setting this value, we get a slightly different model every time we run the script as the decisiontreeregressor has some instance variables that are randomly initialised.

```
>>> crime_predictor = DecisionTreeRegressor(random_state = 42)
```

```
Use the fit method of the decisiontreeregressor instance to train the decision tree
with the data that we have prepared before hand. Here the parameter X corresponds
to the training data and y to the variable to predict
```

```
>>> crime_predictor.fit(X, y)
```

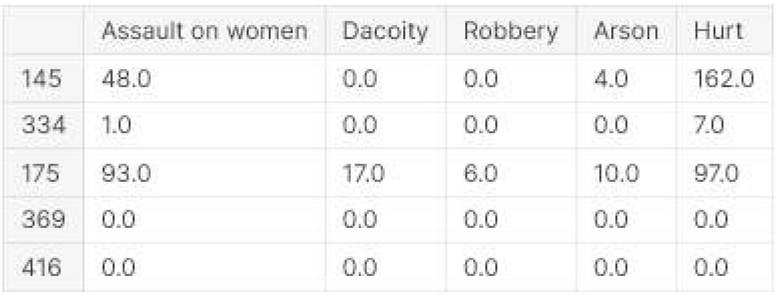
After the fitting process is over the string format of the decisiontreeregressor is printed to the console along with the values of every parameter of the trained model as show below



```
DecisionTreeRegressor(ccp_alpha=0.0, criterion='mse', max_depth=None,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_decrease=0.0, min_impurity_split=None,
                      min_samples_leaf=1, min_samples_split=2,
                      min_weight_fraction_leaf=0.0, presort='deprecated',
                      random_state=42, splitter='best')
```

You now have created your very first machine learning model. How do we use it? Now you can use your trained model to predict the number of murders simply by looking at the number of each of the other crimes that occurred in the same place and time. For now to verify that our model learnt from the data we trained it with let's check what it officers for the first 5 rows of our training data. Although this is not the right approach to teaching a machine learning model, this is what we will be doing for now.

```
>>> X.head(5)
```

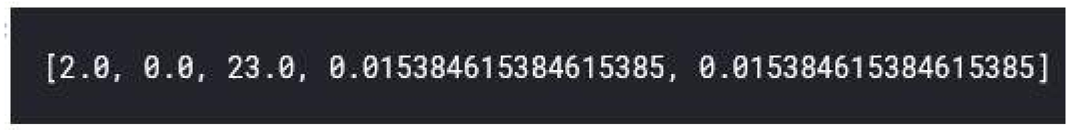


	Assault on women	Dacoity	Robbery	Arson	Hurt
145	48.0	0.0	0.0	4.0	162.0
334	1.0	0.0	0.0	0.0	7.0
175	93.0	17.0	6.0	10.0	97.0
369	0.0	0.0	0.0	0.0	0.0
416	0.0	0.0	0.0	0.0	0.0

Now to get our models predictions call the predict method of the trained model and pass the data that you want to get the predictions for as an argument as shown below

```
>>> predictions = crime_predictor.predict(X.head(5))
```

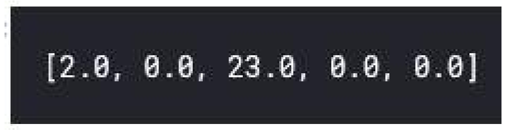
```
>>> list(predictions)
```



```
[2.0, 0.0, 23.0, 0.015384615384615385, 0.015384615384615385]
```

The list that is shown above is the predictions that our model returned as predictions. Let's compare it's predictions to the actual number of murders that have occurred. This information is available in our dataset

```
>>> list(dataset["Murder"].head(5))
```



```
[2.0, 0.0, 23.0, 0.0, 0.0]
```

Observe that the origins it model made regarding the number of murders that occured almost exactly match with with the actual number of murders that occurred in that place and time.

7.3 Validating your first AI model

In this section we will be discussing why your models need to be validated before use and how you can validate them.

You always want to validate your models(majority of cases). In most of the situations that you will encounter the measure of model quality is measured as predictive accuracy, which indicates how accurate the models predictions are.

One of the most common mistakes beginners do when it comes evaluating model quality is that they measure the accuracy of the model by checking how the model makes prediction of the data that it was trained with. This is exactly the method that we used to have a feel for the quality of the first machine learning model we created in the previous section. The reason this approach is a bad idea is because we want to make sure our model is able to accurately make predictions regarding data that it has never encountered before. By evaluating our model with the data that it was trained with we are ensuring that our model is making quality predictions regarding data that it was trained with, which isn't what we are creating the model for. To ensure that our model has an acceptable predictive accuracy, we need to validate our model on data that it hasn't seen before.

When it comes to data science there are many metrics to measure model quality but what we will be discussing here is the metric known as “mean absolute error”. Before we get into mean absolute error let's understand what “error” means on the context of predictive accuracy of machine learning models.

When it comes to Machine learning models, the error of a models prediction is calculated as shown below

Error = actual value -predicted value

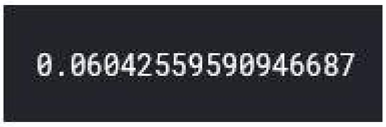
For example the prediction that our model made was 50 ocuurences but the actual number of occurrences was 60 then the error of prediction is

Error = 60 – 50 =10

The mean absolute error method of measuring predictive quality calculates the absolute value of the error of each of the predictions, adds them and divides them by the number of test cases provided.

To measure the mean absolute error of our model we first want to import the mean_absolute_error function from the sklearn module. After importing it, call the function and pass on the arguments as the actual target value of the test samples passed followed by predictions made by our model. Take a look at the code shown below.

```
>>> from sklearn.metrics import mean_absolute_error

>>> predictions = crime_predictor.predict(X)

>>> mean_absolute_error(y , predictions)
```



```
0.06042559590946687
```

We can see that the accuracy is almost perfect when we evaluate our model using the data that it was trained with. To perform a more reliable evaluation with the data at hand the method we use is to divide our training data into parts

1. Training data - To train the model with
2. Validation data - To evaluate the model with
3. Test data - To test the model with unseen data

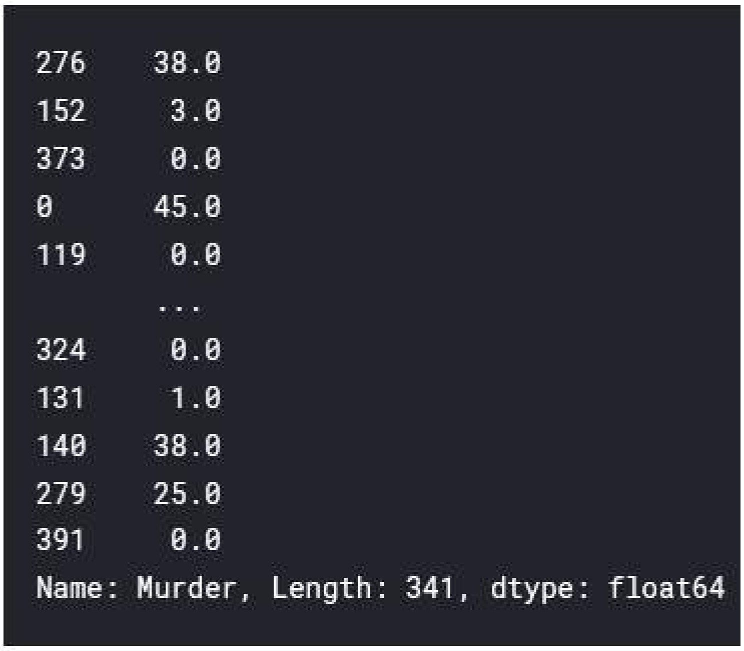
But in our examples we will only be splitting our data into training and test data. The process of splitting the data involves two steps which are,

1. Randomly shuffling the data records - in case there is any sequential relationship we want to eliminate

2. Splitting the dataframe into multiple dataframes

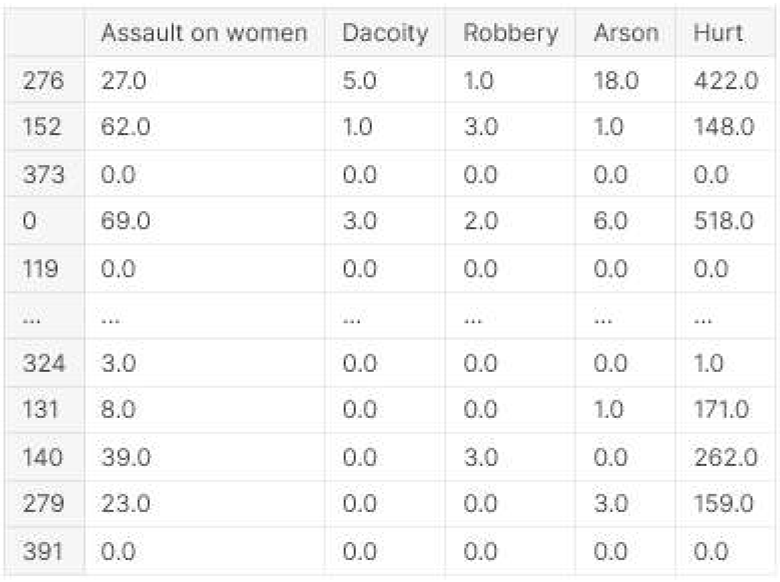
This process can be done manually or we can use pre-existing functions from libraries like scikitlearn. In case you are trying to manually split your data ,you will first need to shuffle it. This ca be done as shown below by using the sample method of dataframe objects and by assigning the value of the frac parameter as 1

```
>>> y = dataset.sample(frac = 1, random_state = 42).Murder

>>> y
```



```
276    38.0
152     3.0
373     0.0
0      45.0
119     0.0
       ...
324     0.0
131     1.0
140    38.0
279    25.0
391     0.0
Name: Murder, Length: 341, dtype: float64
```

```
>>> X = dataset.sample(frac = 1, random_state = 42)[selected_features]

>>> X
```



	Assault on women	Dacoity	Robbery	Arson	Hurt
276	27.0	5.0	1.0	18.0	422.0
152	62.0	1.0	3.0	1.0	148.0
373	0.0	0.0	0.0	0.0	0.0
0	69.0	3.0	2.0	6.0	518.0
119	0.0	0.0	0.0	0.0	0.0
...	...	...	...	...	...
324	3.0	0.0	0.0	0.0	1.0
131	8.0	0.0	0.0	1.0	171.0
140	39.0	0.0	3.0	0.0	262.0
279	23.0	0.0	0.0	3.0	159.0
391	0.0	0.0	0.0	0.0	0.0

341 rows × 5 columns

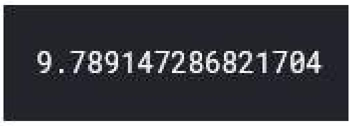
After shuffling the loc or iloc operations can be used to split the data into different parts. The more simpler way to split data into training and test sets is to use the train_test_split( ) method of the sklearn library. Call the function train_test_split by passing the parameters as

1. Training variables for the model
2. Target variable for the model
3. Random_state - To get the same split every time the program is executed

The train_test_split function when called with these arguments returns 4 dataframes and they are,

1. X, y for training
2. X, y for validation

```python
from sklearn.model_selection import train_test_split

train_X,  val_X, train_y, val_y = train_test_split(X, y, random_state = 42)

crime_predictor = DecisionTreeRegressor(random_state = 42)
crime_predictor.fit(train_X, train_y)

val_predictions = crime_predictor.predict(val_X)
print(mean_absolute_error(val_y, val_predictions))
```

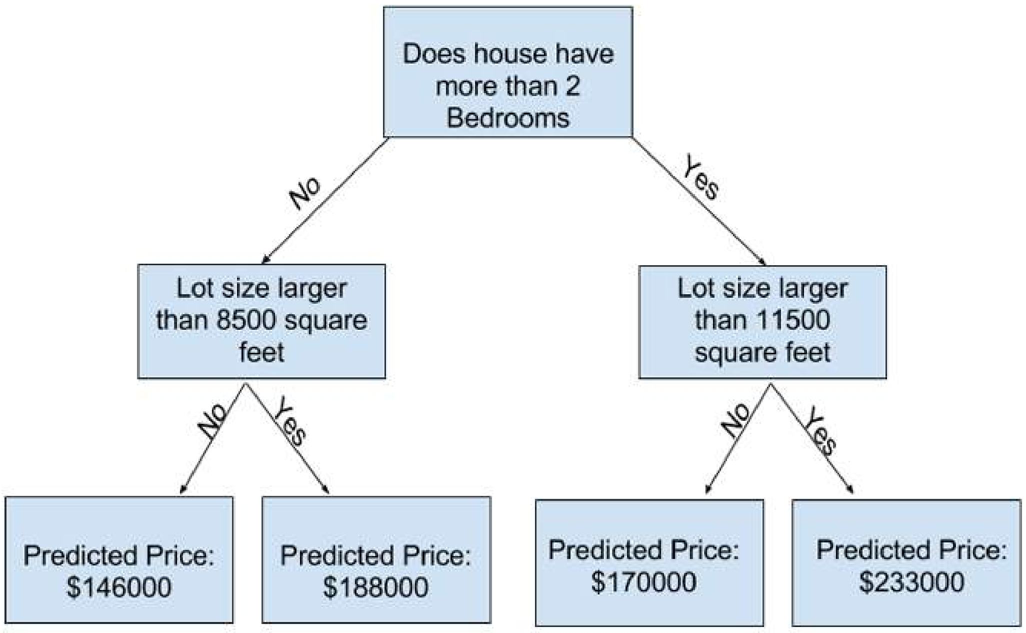


```
9.789147286821704
```

7.4 Underfitting and overfitting

Now we know how to measure accuracy. We can try different machine learning models and different model parameters with different accuracy levels and stop when we arrive at a model with satisfactory accuracy levels that we are looking for.

Sci-ki learn offers several machine learning models and each one having several parameters that can be tuned. In our case one of the most important parameters is the tree depth. A trees depth is the number of splits that the decisoin tree makes before arriving at a prediction. The tree in our earlier example was a shallow tree with only two splits before reaching a prediction.

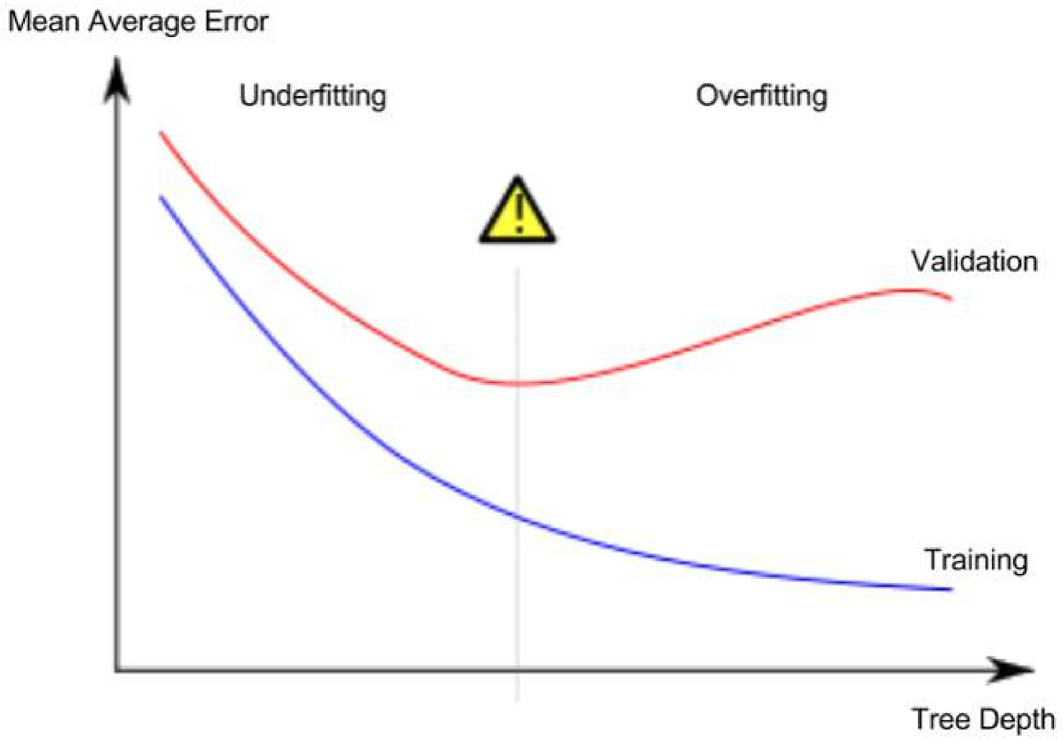

Does house have more than 2 Bedrooms
No
Yes
Lot size larger than 8500 square feet
Lot size larger than 11500 square feet
No
Yes
No
Yes
Predicted Price: $146000
Predicted Price: $188000
Predicted Price: $170000
Predicted Price: $233000

Practically speaking it is very rare to encounter models with only two splits or even ten splits. During a decision trees training process as the tree splits up each time going deeper and deeper the data set is also split up upon every split of the tree. For a tree with only one split the dataset is divided once and there are two leaves of the tree each of which contains a group of the dataset that was divided based on the condition of the split.

If the tree is very deep the leaves of that deep tree will only have a few data points at almost each of its leaves which means the decision making process is based on only a few data points. When the model is trained in such a manner the machine learning model is said to be overfit. This means that the model is very well fit to the data at hand making it very effective at predicting for the data that it was trained with but performs poorly when it comes to data that it has not seen before. So in conclusion we don't want our model to be overfit. Over fitting of machine learning models is caused by over training the models on the training data making them learn generic patterns that the we want the model to learn as well as specific patterns that only apply to the training data but not any unseen cases. An example of an overfit model is a decision tree with a100 splits

If the tree is very shallow the most leaves of the tree will have many days points which means that the decision making process is based on to many points, making the decision based on very general factors on missing very important factors for the decision making process. When a machine learning model is trained in such a manner the model is said to be underfit. This means that the model is not properly for to the data at hand planning behind a lot of scope for improvement. So again in conclusion we don't want our model to be underfit. Under fitting of machine learning models is caused by under training the models on the training data, making them miss out on several key patterns that are important for decision making. An example of an underfit model is a decision tree with only one split dividing the data set into two groups.

The curves shown below genetically represent how the predictive accuracy of a model changes with the change in the amount of patterns it picks up from the data it was trained with. The blue curve is the model being evaluated on the training data and the red curve is the model being evaluated on the test data.



In the code listing below we write a function get_score_mae that take in the following parameters

1. Max_leaf_nodes – the maximum number of leaf nodes to allow
2. Train_x
3. Train_y
4. Val_x
5. Val_y

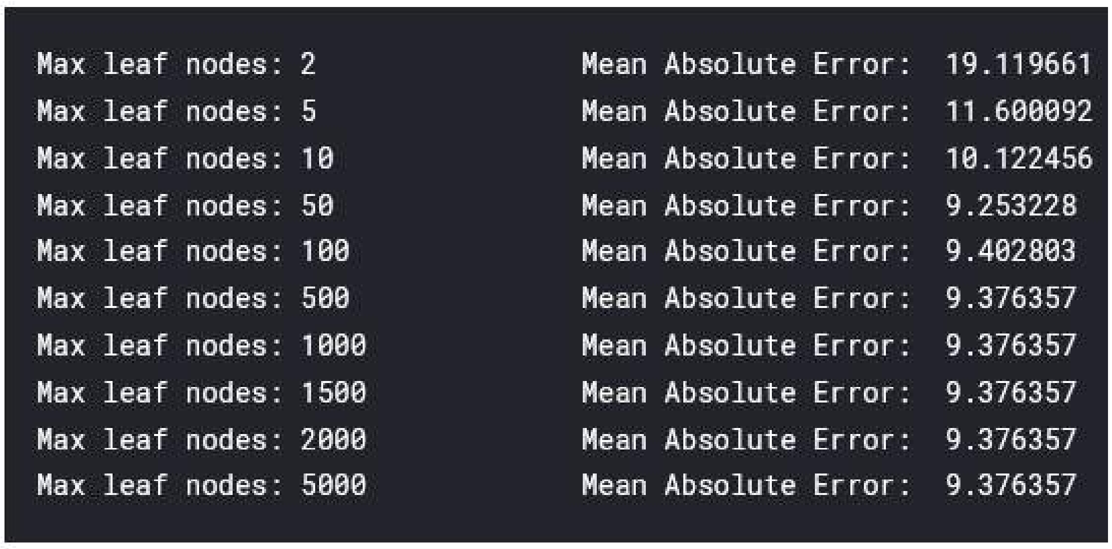
This function takes these arguments passed by the caller, creates a decision tree with the specified parameters, trains the tree with the training data provided, calculates the mean absolute error with the validation data provided and returns this mean absolute error.

```
def get_score_mae(max_leaf_nodes, train_X, val_X, train_y, val_y):
    model = DecisionTreeRegressor(max_leaf_nodes=max_leaf_nodes, random_state
    = 42)
    model.fit(train_X, train_y)
    val_predictions = model.predict(val_X)
    mae = mean_absolute_error(val_y, val_predictions)
```

```python
    return(mae)
```

Now that we have our evaluation function defined we can create a list of parameters to performs those operations on.the code below calls our function get_score_mae on a list of parameter values for the decsion tree parameter named “max_leaf_node”

```python
for max_leaf_nodes in [2, 5, 10, 50, 100, 500, 1000, 1500, 2000, 5000]:
    score_mae = get_score_mae(max_leaf_nodes, train_X, val_X, train_y, val_y)
    print("Max leaf nodes: %d  \t\t Mean Absolute Error:  %f" %(max_leaf_nodes, score_mae))

```



```
Max leaf nodes: 2              Mean Absolute Error:  19.119661
Max leaf nodes: 5              Mean Absolute Error:  11.600092
Max leaf nodes: 10             Mean Absolute Error:  10.122456
Max leaf nodes: 50             Mean Absolute Error:  9.253228
Max leaf nodes: 100            Mean Absolute Error:  9.402803
Max leaf nodes: 500            Mean Absolute Error:  9.376357
Max leaf nodes: 1000           Mean Absolute Error:  9.376357
Max leaf nodes: 1500           Mean Absolute Error:  9.376357
Max leaf nodes: 2000           Mean Absolute Error:  9.376357
Max leaf nodes: 5000           Mean Absolute Error:  9.376357
```

,

From the output above, we can see that as we increase the number of leaf nodes of the tree the predictive accuracy of the tree increases until the number of leaf nodes reaches 500 after which the predictive accuracy does not show any change.

7.5 The random forest model

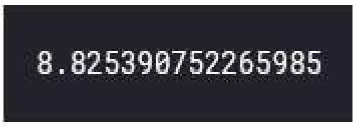
Decision trees leave your stranded with a tough design process with hard consequences on the predictive accuracy of the model. A model with too many leaves makes the model overfit as every prediction is based on only a few records of the dataset available on the leaf. On the other hand a model with very less leaves becomes underfit as the model has not yet picked up on key patterns that are required for proper prediction. This problem exists in for even the very sophisticated modelling techniques, but there are some algorithms that use ver y clever techniques to handle this problem, like the random forest algorithm.

The random forest algorithm uses several decision trees in its learning and prediction process. Prediction is performed by averaging out the predictions made by each prediction from the individual decision tree component of the random forest model. This approach handles the problem of underfitting and overfitting and drastically increases predictive accuracy.

To use the random forest model imoprt it from the sklearn module

```
>>> from sklearn.ensemble import RandomForestRegressor
```

```
crime_predictor = RandomForestRegressor(random_state = 42)
crime_predictor.fit(train_X, train_y)
val_predictions = crime_predictor.predict(val_X)
print(mean_absolute_error(val_y, val_predictions))
```



```
8.825390752265985
```

[illegible]