1. Introduction:

In 1991, when Guido Van Rossum made python as a side project, he did not realize that one day it would be the world's fastest growing computer language in near future. Python turns out as a go to language for fast prototyping.

Currently, it is one of the most famous and best paid programming language world-wide.

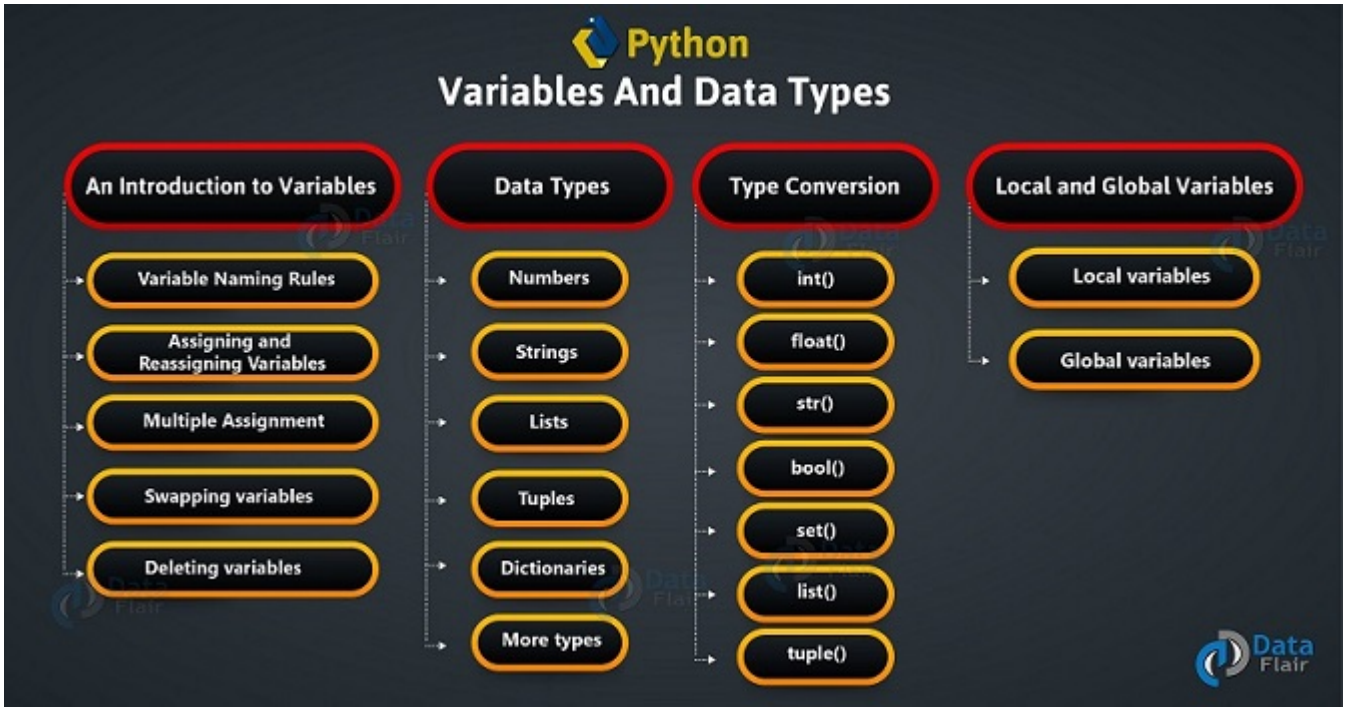
If we look at the philosophy of the Python language, we see:

- Less complexity.
- Can become common language for data science.
- Can become common language for production of web based analytics products.
- Easier to implement.
- No compilation required, execution can be done directly.
- Awesome online community.

2. Data Types in Python:

Every value in Python has a datatype. Since everything in Python is an object, data types are actually classes and variable are instances (objects) of these classes.

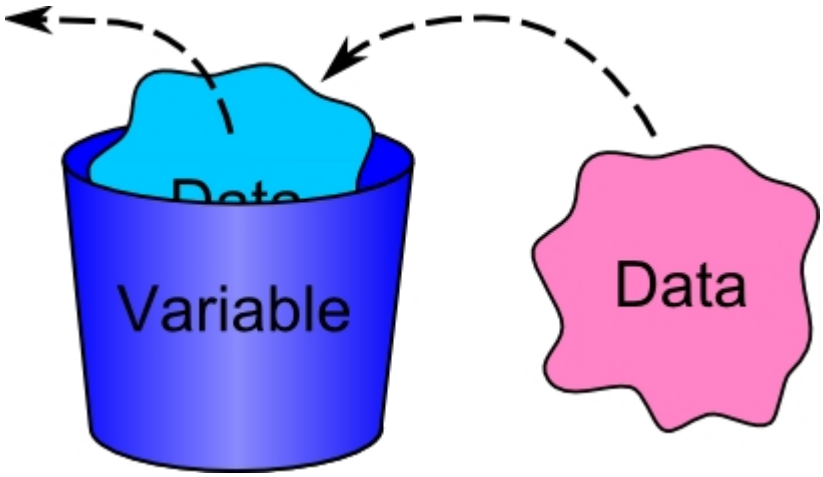
Following data types are used in Python to store and access the data. Those are explained below.



WHY, WHEN & WHERE

- **Why?** In Python, like in all programming languages, data types are used to classify one particular type of data.
- **When?** This is important because the specific data type you use will determine what values you can assign to it and what you can do to it.
- **Where?** Whenever we want to save the value in a variable.

2.1 Variables:



- Technically, variable is supposed to be a bag to store books in it and those books can be replaced any time.
- But in python unlike most of the programming languages values are not assigned to variables, whereas the python gives the reference of the object (value) to the variable.
- Thus in python, variables do not need explicit declaration to occupy memory space. The type of variable is decided after assigning the value to it.

Some examples are given as follows:

In [1]:
```
#float type variable
Height = 5.8
Length = 3.2
Width = 10.2

#print all values
print (Height)
print (Length)
print (Width)
```

```
5.8
3.2
10.2
```

In [2]:
```
#integer type variable
Age = 23
Number = 12345
Miles = 5

#print all values
print (Age)
print (Number)
print (Miles)
```

```
23
12345
5
```

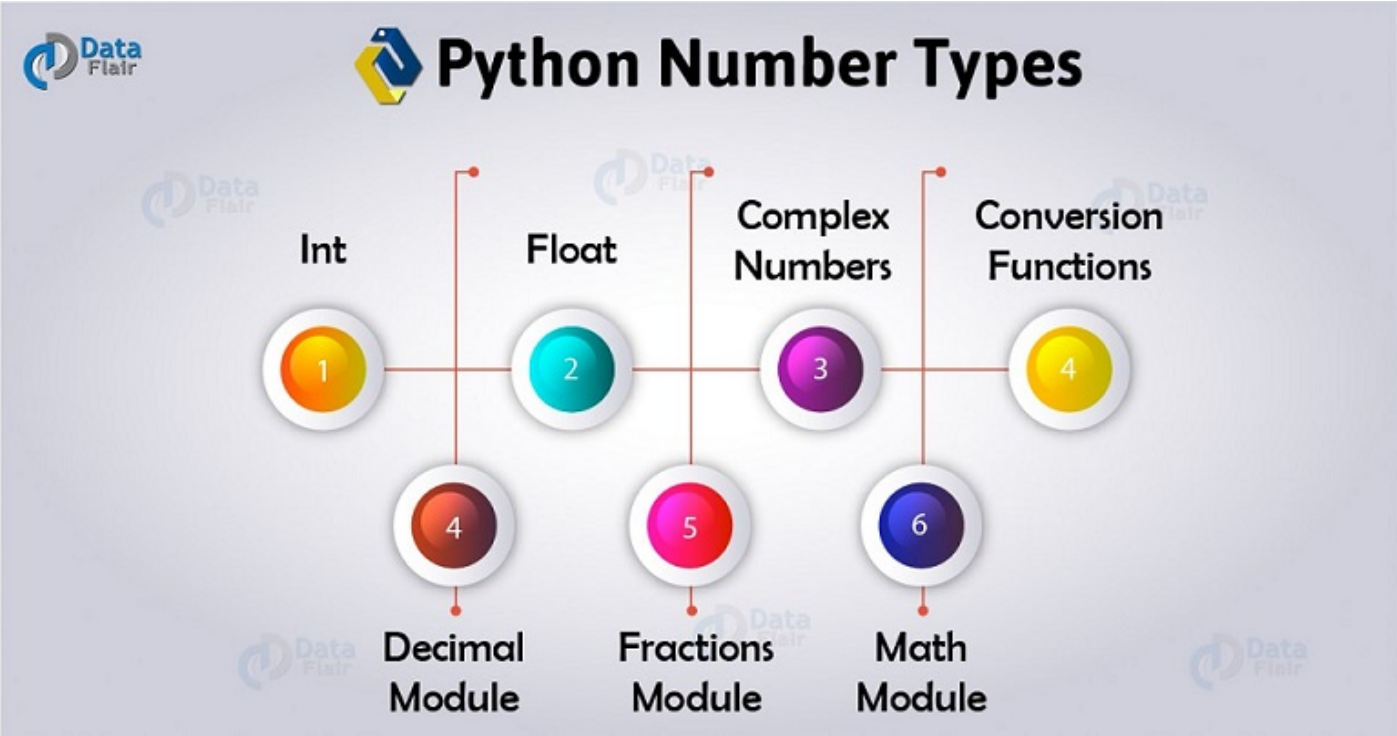
In [7]:
```
#String type variable
FName = "Omar"
LName = "Ali"
Department = "Chemistry"

#print all values
print (FName)
print (LName)
print (Department)
```

```
Omar
Ali
Chemistry
```

WHY, WHEN & WHERE

- **Why?** It is done in order to allocate a space in your disk to store values for retrieval of values when needed.
- **When?** When it is desired to store value for retrieval afterwards.
- **Where?** Variables are used to wherever in our program we want to save the values, they allocate memory for that data.

2.2 Numbers



In Python, numbers data type store numerical values.

They are immutable data types (Means changing the value of Number data type will results in a newly allocated object).

There are four different numerical types in Python:

- int -- plain integers are just positive or negative whole numbers.
- long -- long integers are integers of infinite size and expressed by letter "L" (ex: 140L).
- float -- floats represent real numbers, but are written with decimal points.
- complex -- represented by the formula a + bJ, where a and b are floats, and J is the square root of -1.

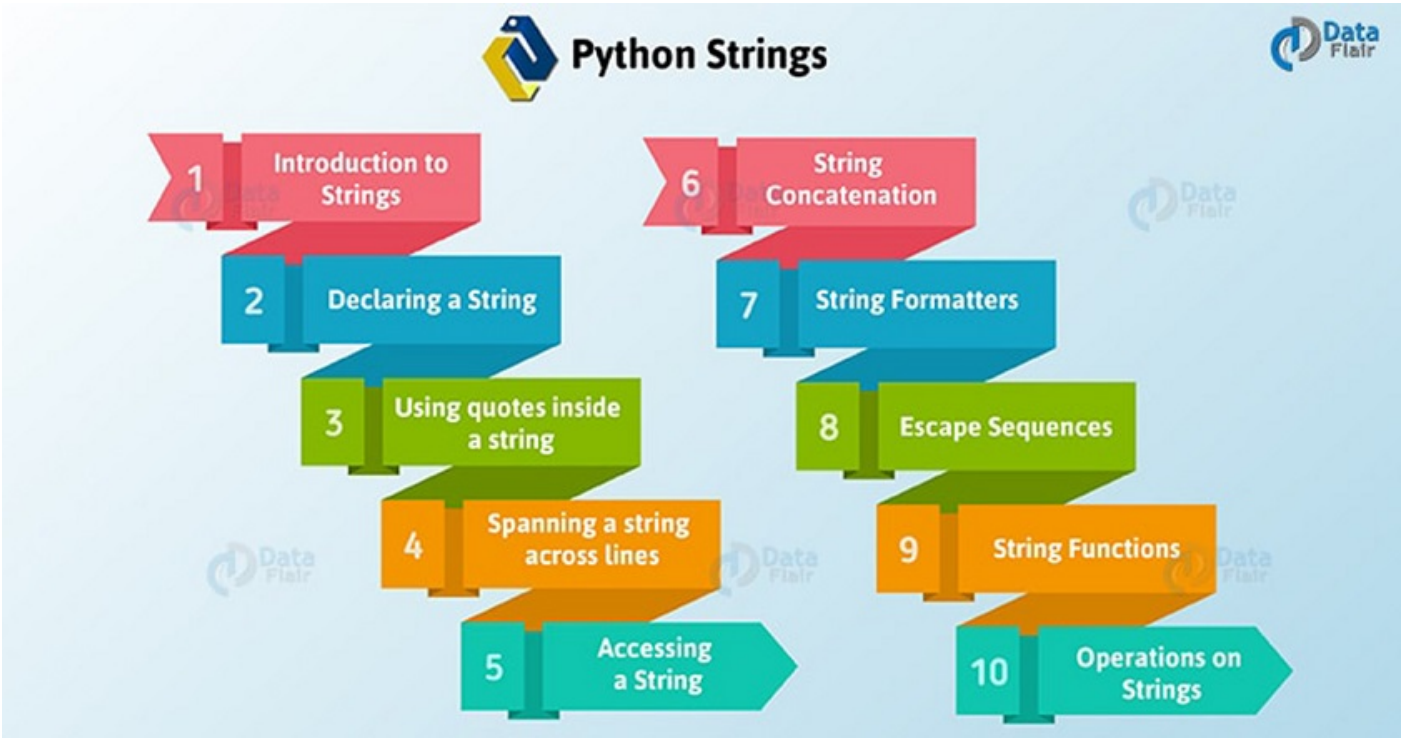
In [1]:
```
#data type numbers declared
a = 50
b = 2.0
c = 1+2j

# Output shown with datatype
print(a, "is of type", type(a))
print(b, "is of type", type(b))
print(c, "is complex number?", isinstance(1+2j,complex))
```

```
50 is of type <class 'int'>
2.0 is of type <class 'float'>
(1+2j) is complex number? True
```

WHY, WHEN & WHERE

- **Why?** A mathematical object used to count, measure, and perform functions on those numerical values.
- **When?** When calcualtions are needed to be done in our program.
- **Where?** Where numbers are involved. Used in databases, for ordering etc.

2.3 Strings



Strings are sequences of character data used to store words or combination of words having letters, numbers, special characters etc.

String literals may be delimited using either single or double quotes.

In [21]:
```
#String type variable
FName = "Omar"
LName = "Ali"
Department = "Chemistry"

#print all values
print (FName)
print (LName)
print (Department)
```

```
Omar
Ali
Chemistry
```

In [ ]:
```

```

Acess values in String

Python does not support a character type; they are treated as strings of length one, thus also considered as substring.

In order to access substrings, square brackets are used for slicing string along with the index or indices to obtain the required substring.

There are few examples below which demonstrate the results as follows:

In [25]:
```
#String type variable
FName = "Omar"
LName = "Ali"
Department = "Chemistry"

#Output is shown in the form of Substrings
print ("FName: ",FName[0])
print ("LName: ",LName[2])
print ("Department: ",Department[1:5])
```

```
FName:  O
LName:  i
Department:  hemi
```

WHY, WHEN & WHERE

- **Why?** A string is a data type used in python used to represent text.
- **When?** When text is involved.
- **Where?** Where the actions are needed to be taken on alphanumeric characters.

Updating String

In Python, value of string can be updated by assigning a new value to it which will replace the older one.

New value must be of same data type as previous one.

'+' operator is used for this purpose as follows:

In [30]:
```
String1 = "Hello Older String"
print("Initial String: ")
print(String1)

# Updating a String
String1 = "Welcome New String"
print("\nUpdated String: ")
print(String1)
```

```
Initial String:
Hello Older String

Updated String:
Welcome New String
```

In [32]:
```
myString = "Hello World"
myString = myString[:6] + "Python"
print("Updated string: ",myString)
```

```
Updated string:  Hello Python
```

In [35]:
```
myString2= 'Java rocks'
myString2= "Python" + myString2[4:]
print ("Updated String: ",myString2)
```

```
Updated String:  Python rocks
```

Delete String

Deletion of entire String is possible with the use of del keyword.

Further, if we try to print string, this will produce an error because we deleted its object (i.e. reference variable).

Here are few examples given below as:

In [189]:
```
myStr = 'Hello World'
print (myStr)

# Deleting a String
# with the use of del
```

```
Hello World
```

Operators are special symbols in Python that carry out arithmetic or logical computation.

The value that operator operates on is called Operand.

'+' Operator

- '+' Operator is used for concatenation.

In [44]:
```
var= 'Python'
print("Hello " + var)

var2= ' to the '
print (var + var2)

var3= 'Future'

print (var+var2+var3)
```

```
Hello Python
Python to the
Python to the Future
```

'*' Operator

- '*' Operator is used for Repetition

In [50]:
```
var = 'Hello '
print (var * 3)

var2 ='echo '
print ((var + var2) *2)

var3 = (var + var2) *2
print ((var3 * 2) + ' Adionic' *2 )
```

```
Hello Hello Hello
Hello echo Hello echo
Hello echo Hello echo Hello echo Hello echo  Adionic Adionic
```

'[]' Operator

- '[]' Operator is used to give character index in String.

In [61]:
```
var= 'Hello'
print(var[0])

var2 = 'Hello India'
print(var2[6])

var3= 'Boy'
print ("Hello " + var3[0] + var3[1] + var3[2])
```

```
H
I
Hello Boy
```

'[:]' Operator

- '[:]' Operator is used to give a range of characters from String.

In [63]:
```
var= 'Hello Pakistan'
print (var[:5])

var2 = 'Hello World'
print(var2[4:6])

var3= 'World'
print ("Hello " + var3[0:])
```

```
Hello
o
Hello World
```

'in' Operator

- 'in' Operator returns true if given character exists in String, otherwise false.

In [67]:
```
var = "Hello"
print('H' in var)

var2 = 'World'
print ('H' in var2)

var3 = "Hello World"
print ('H' in var , 'W' in var2)
```

```
True
False
True True
```

'in' Operator

- 'in' Operator returns true if given character does not exists in String, otherwise false.

In [69]:
```
var = "Hello"
print('H' not in var)

var2 = 'World'
print ('W' in var2)

var3 = "Hello World"
print ('H' in var , 'H' in var2)
```

```
False
True
True False
```

String Formatting Operator

- Python uses C-style string formatting to create new, formatted strings.
- The "%" operator is used to format a set of variables enclosed in a "tuple" (a fixed size list).

  See following examples:

In [72]:
```
print ('Python will overtake all programming languages in %s' %("Future"))

print ('1 mile is %s' %('1.6 km'))

print ('World is %s by %d' %("Shocked", 911))
```

```
Python will overtake all programming languages in Future
1 mile is 1.6 km
World is Shocked by 911
```

2.4 Lists

- Python offers a range of compound datatypes often referred to as sequences.
- List is one of the most frequently used and very versatile datatype used in Python.
- In Python, list is created by placing all the items inside a square bracket '[ ]', separated by commas.
- Python can have any number of items and they may be of different types (integer, float, string etc.).

In [3]:
```
list1 = ['Mehvish','1880']
print(list1)

list2 = ['1','2','3','4','5']
print(list2)

list3 = ['India','Pakistan','War']
print(list3)
```

```
['Mehvish', '1880']
['1', '2', '3', '4', '5']
['India', 'Pakistan', 'War']
```

WHY, WHEN & WHERE

- **Why?** Python offers a range of compound datatypes often referred to as sequences. List is one of the most frequently used and very versatile datatype used in Python..
- **When?** When any arbitrary objects are needeed to be saved.
- **Where?** When information is to be maintained for that list's lifetime.

Accessing values in List

There are numerous ways through which values are access from lists.

- We can use index operator '[]' to access an item in list.
- Index starts from 0, range can be given for indexing.
- Index must be an Integer value.
- Float value or any other type will result in *TypeError*

In [6]:
```
my_list = ['p','r','o','b','e']
print (my_list[0])

list1 = ['physics', 'chemistry', 1997, 2000];
print ("list1[0]: ", list1[0])

list2 = [1, 2, 3, 4, 5, 6, 7 ];
print ("list2[1:5]: ", list2[1:5])
```

```
p
list1[0]:  physics
list2[1:5]:  [2, 3, 4, 5]
```

Updating Lists

- We can update single or mupltiple elements of a list by giving slice on left hand of the assignment operator.
- *append()* add a new element at the end of the list.

In [10]:
```
my_list = ['p','r','o','b','e']
my_list[1]='R'
print (my_list)

my_list.append('s')
print (my_list)

list1=['Computer','Laptop']
list1[0]=list1[1]
list1[1]='Computer'
print(list1)
```

```
['p', 'R', 'o', 'b', 'e']
['p', 'R', 'o', 'b', 'e', 's']
['Laptop', 'Computer']
```

Delete List Element

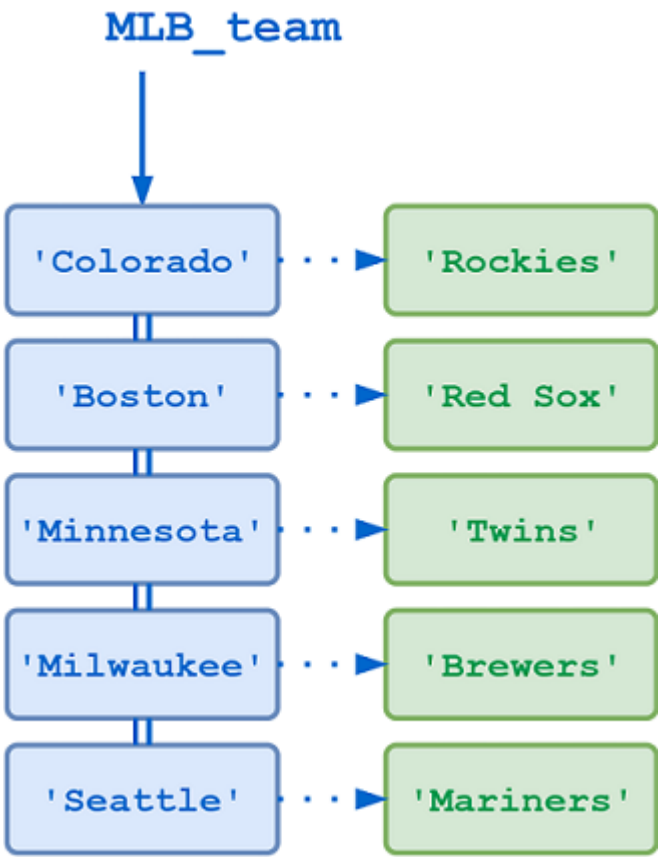
There are several ways to delete elements from list

- If you know the index of element you want to delete, you can use *pop* it will provide you with the removed value.
- If you have no clue about index of element it will remove the last element present in list.
- If you do not want the removed value, you can use *del* operator.
- If you know the element to remove but not the index you can use *remove()*.

In [17]:
```
list = ['a', 'b', 'c']
x= list.pop(1)
print(list)

print(x)

del list[1]
print(list)

list.remove('a')
print (list)
```

```
['a', 'c']
b
['a']
[]
```

2.5 Dictionaries



- Python dictionary is an unordered collection of items.
- Other compound data types have only value as an element, a dictionary has a key: value pair and each element seprated by comma ','.
- When the values are known, dictonaries are optimized in the way that the values are retreived easily.
- It is same as array of objects having key and values in PHP.
- Unique keys are used in dictonaries but values are not.
- Dictionary can have any data type value.
- Keys must be of immutable datatype like strings,numbers or tuple.

WHY, WHEN & WHERE

- **Why?** Values in a dictionary can be of any datatype and can be duplicated, whereas keys can't be repeated and must be immutable.
- **When?** When key must be unique and of immutable data type such as Strings, Integers and tuples, but the key-values can be repeated and be of any type
- **Where?** When we want to access data through keys because we don't have numerical index.

Accessing values in Dictionaries

- Unlike other container types which use indexing, *keys* are used in dictonaries.
- Key can be used either inside square brackets or with *get()* method.

In [22]:
```
dict = {'name':'Adeel', 'age': 26}
print (dict['name'])

released = {"iphone 4S" : 2011, "iphone 5" : 2012}
print(released)

print(dict.get('age'))
```

```
Adeel
{'iphone 4S': 2011, 'iphone 5': 2012}
26
```

Updating Dictionary

- Dictionaries are mutable, means we can add new items or change the existing items using '=' operator.

In [26]:
```
dict = {'name':'Adeel', 'age': 26}
dict['age']=27
print(dict)

dict['Department']='CS'
print (dict)

dict['Department']='EE'
print(dict)
```

```
{'name': 'Adeel', 'age': 27}
{'name': 'Adeel', 'age': 27, 'Department': 'CS'}
{'name': 'Adeel', 'age': 27, 'Department': 'EE'}
```

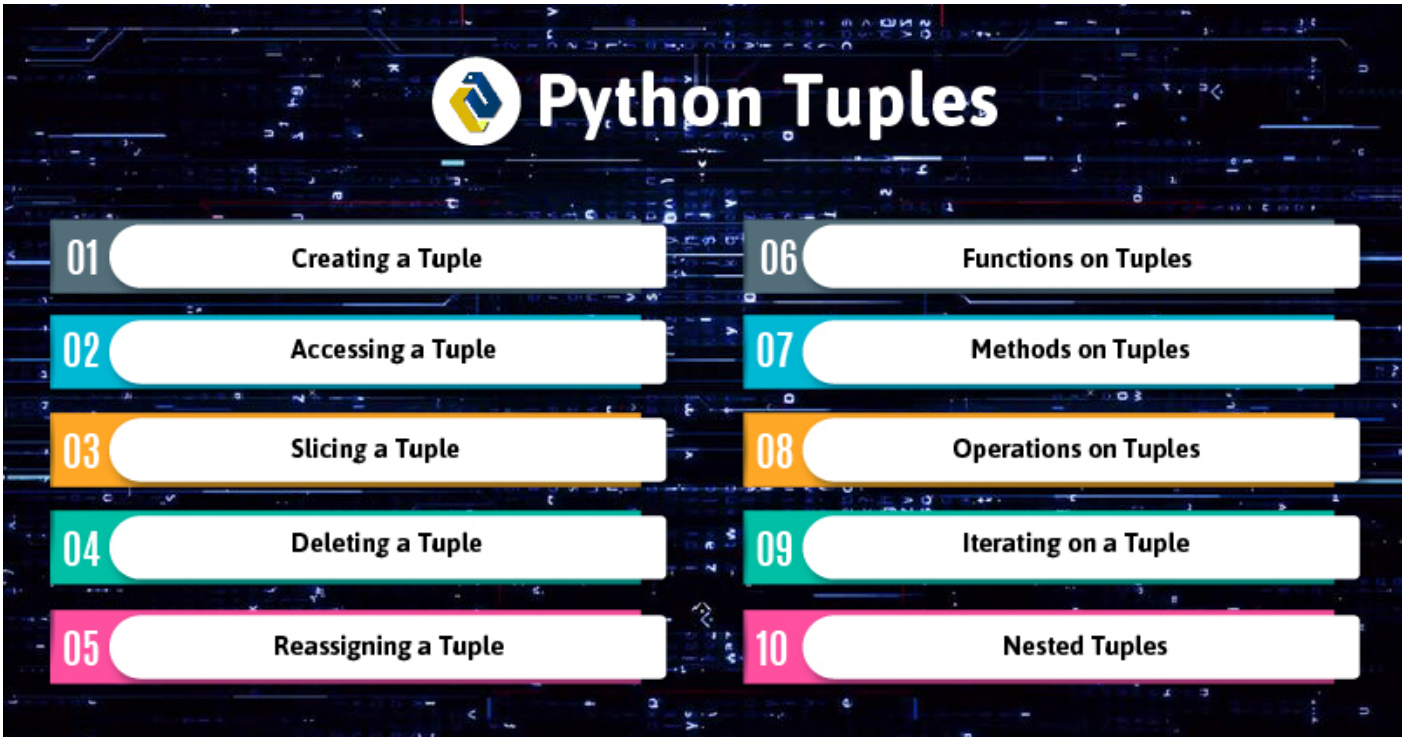
Delete Dictionary Element

- In order to remove particular element of dictionary *pop()* is used which also returns the removed value.
- *popItem()* can be used to remove arbitrary key and respective value.
- *del* keyword remove individual element or entire dictionary itself.
- *clear()* removes the entire dictionary.

In [29]:
```
# create a dictionary
squares = {1:1, 2:4, 3:9, 4:16, 5:25}

# remove a particular item
print(squares.pop(4))
print(squares)

print(squares.popitem())
print(squares)

squares.clear()
print(squares)

del squares
print(squares)
```

```
16
{1: 1, 2: 4, 3: 9, 5: 25}
(5, 25)
{1: 1, 2: 4, 3: 9}
{}

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-19810bcea6c7> in <module>
     13
     14 del squares
---> 15 print(squares)

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

2.6 Tuples



In Python, tuple is similar to *list*. The difference between two is that we cannot change the elements of tuple once assigned.

- Elements are seprated with comma(,).
- Empty tuple is shown as ().
- Even in order to write a tuple with single value, comma(,) must be included though.
- Tuples can have mutliple data type values.
- Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.
- We generally use tuple for heterogeneous (different) datatypes and list for homogeneous (similar) datatypes.
- Since tuple are immutable, iterating through tuple is faster than with list. So there is a slight performance boost.
- Tuples that contain immutable elements can be used as key for a dictionary. With list, this is not possible.

WHY, WHEN & WHERE

- **Why?** Tuple is similar to a list in terms of indexing, nested objects and repetition but a tuple is immutable unlike lists which are mutable.
- **When?** Tuple can simulate dictionary without keys. Tuple data is constant and must not be changed.
- **Where?** Tuple can be used as the key inside dictionary due to its immutable nature.

Accessing values in Tuples

- Square brackets are used along with index to obtain the required value.

In [33]:
```
my_tuple = ('p','e','r','m','i','t')
print(my_tuple[0])

print(my_tuple[1:4])

my_tuple = ('p','e','r','m','i','t')

print(my_tuple[-1])
```

```
p
('e', 'r', 'm')
t
```

Tuple = (0, 1, 2, 3, 4, 5)

0	1	2	3	4	5

Tuple[0] = 0

Tuple[1] = 1

Tuple[2] = 2

Tuple[3] = 3

Tuple[4] = 4

Tuple[5] = 5

Tuple[0:] = (0, 1, 2, 3, 4, 5)

Tuple[:] = (0, 1, 2, 3, 4, 5)

Tuple[2:4] = (2, 3)

Tuple[1:3] = (1, 2)

Tuple[:4] = (0, 1, 2, 3)

Updating Tuples

- Being immutable means we cannot change tuples.
- We are able to take portions of tuples to make a new tuple.

In [36]:
```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');
tup3 = tup1 + tup2;
print (tup3)

tup2= tup1+tup3
print (tup2)

tup1= tup1+tup1
print (tup1)
```

```
(12, 34.56, 'abc', 'xyz')
(12, 34.56, 12, 34.56, 'abc', 'xyz')
(12, 34.56, 12, 34.56)
```

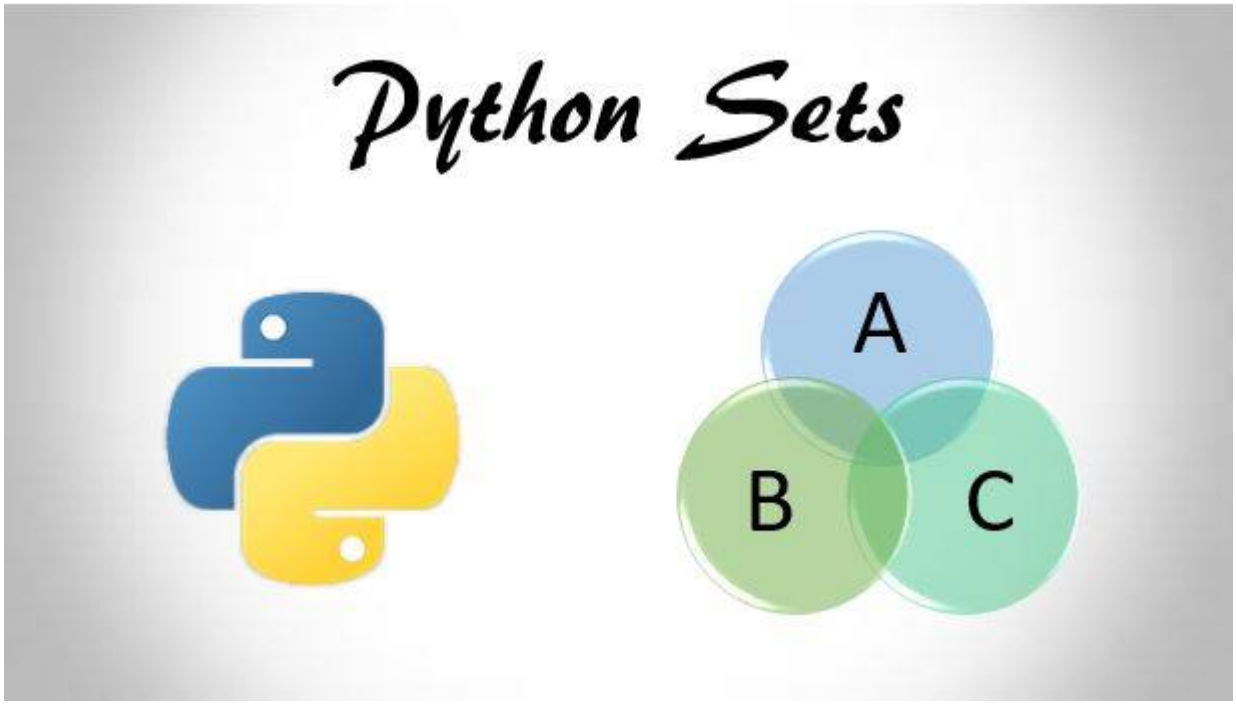
Deleting Tuples

- As discussed, tuples are immutable means their value cannot be change nor they can be deleted.
- *del* statement is used to remove entire tuple.

In [48]:
```
tup = ('physics', 'chemistry', 1997, 2000);
print (tup);
del tup;
print (tup);

my_tuple = ('p','r','o','g','r','a','m','i','z')
del my_tuple;
print(my_tuple);

tup_more=('1','2','3');
del tup_more;
print (tup_more);
```

```
('physics', 'chemistry', 1997, 2000)

---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-48-8e955533a791> in <module>
      4 del tup;
      5
----> 6 print (tup);
      7
      8 my_tuple = ('p','r','o','g','r','a','m','i','z')

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

2.7 Sets



Python’s built-in set type has the following characteristics:

- Sets are unordered.
- Set elements are unique.
- Duplicate elements are not allowed.
- A set itself may be modified, but the elements contained in the set must be of an immutable type.
- Sets can be used to perform mathematical set operations like union, intersection, symmetric difference etc.
- Empty set will be written as {}.
- Each item in set will be comma (,) seperated.
- We can make a set from a list using set() function.
- Data type can be found using type() function.
- add() is used to add single value, update() is used for adding multiple values.
- update() function can take tupe, strings, list or other set as argument.
- In all cases, duplicates will be avoided.
- discard() and remove() are used to to delete particular item from set.
- discard() will not raise an error if item doesn't exists in set.
- remove() will raise an error if item doesn't exists in set.

WHY, WHEN & WHERE

- **Why?** A Set is an unordered collection data type that is iterable, mutable, and has no duplicate elements.
- **When?** The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.
- **Where?** When various standard operations (union, intersection, difference) can be performed on sets.

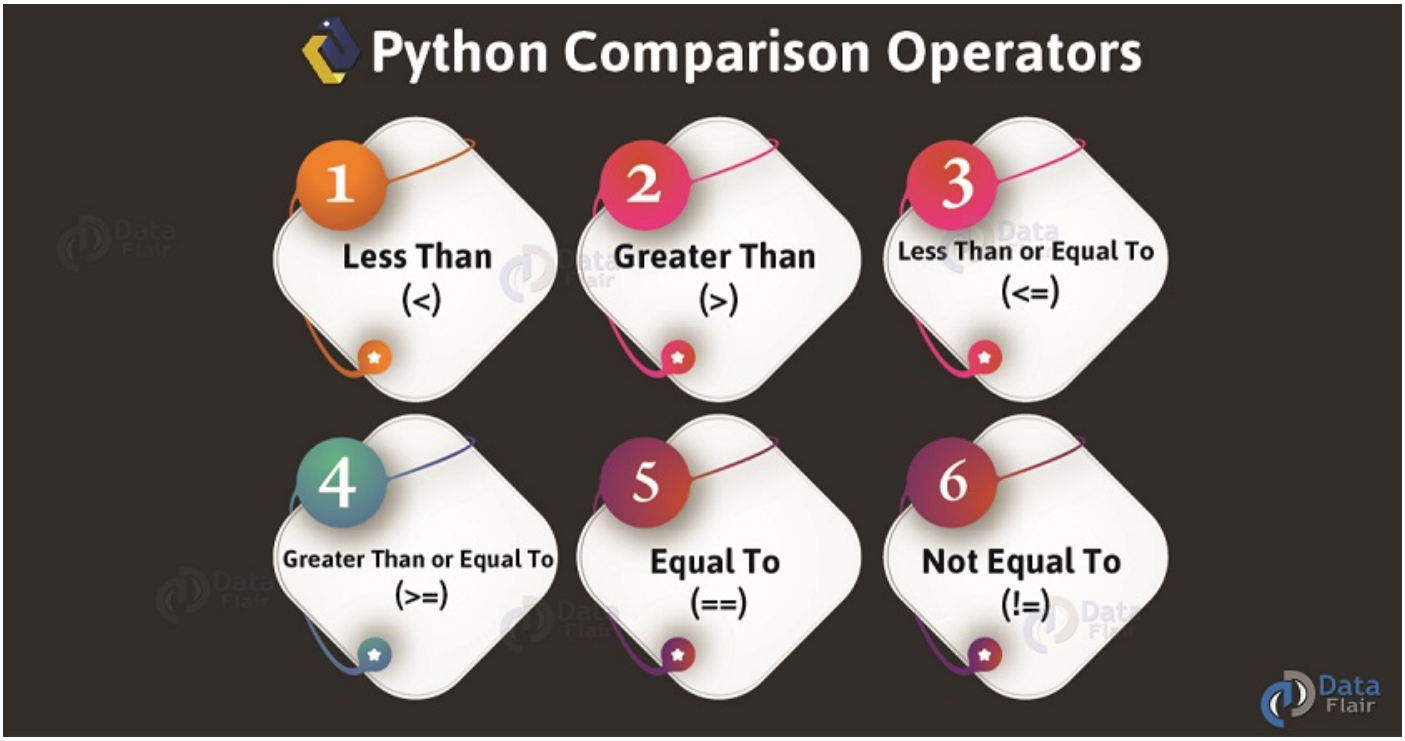
In [55]:
```
list1 = [1,2,3,4,5]
print (type(list1))
my_set = set(list1)
print(my_set)

my_set = {1,2,3}
print(my_set)

my_set = {1,"Hello", 1.2,'C'}
my_set.add('D')

print (my_set)

my_set.update(list1)
print (my_set)
my_set.discard('G')
my_set.remove('G')
```

```
<class 'list'>
{1, 2, 3, 4, 5}
{1, 2, 3}
{1, 1.2, 'C', 'D', 'Hello'}
{1, 1.2, 2, 3, 4, 'C', 5, 'D', 'Hello'}

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-55-108ac7d62236> in <module>
     15 print (my_set)
     16 my_set.discard('G')
---> 17 my_set.remove('G')

KeyError: 'G'
```

In [61]:
```
s = 'quux'
print (set(s))

my_set = {1.0, "Hello", (1, 2, 3)}
print(my_set)

my_set.add(2)
print(my_set)

my_set.update([2,3,4])
print(my_set)
```

```
{'u', 'x', 'q'}
{1.0, 'Hello', (1, 2, 3)}
{1.0, 'Hello', 2, (1, 2, 3)}
{1.0, 2, 3, 4, (1, 2, 3), 'Hello'}
```

3. Comparison Operators



- These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

Operator	Description	Example
==	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
<>	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.
>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
>=	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
<=	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

In [103]:
```
1<5
```

Out[103]: True

In [67]:
```
5<1
```

Out[67]: False

In [68]:
```
2<2
```

Out[68]: False

In [69]:
```
1<=5
```

Out[69]: True

In [70]:
```
5<=1
```

Out[70]: False

In [71]:
```
2<=2
```

Out[71]: True

In [72]:
```
'Mehvish' == "Zeenat"
```

Out[72]: False

In [73]:
```
'Mehvish' == "Omar"
```

Out[73]: False

In [74]:
```
'omar'=='omar'
```

Out[74]: True

In [76]:
```
'Mehvish' != "Zeenat"
```

Out[76]: True

In [75]:
```
'Mehvish' != "Omar"
```

Out[75]: True

In [77]:
```
'omar'!='omar'
```

Out[77]: False

In [78]:
```
(1==1) or (5 > 2)
```

Out[78]: True

In [79]:
```
(3==3) or (4>4)
```

Out[79]: True

In [82]:
```
(1!=1) or (2 > 3)
```

Out[82]: False

```
In [83]: (1 < 2) and (2 < 1)
Out[83]: False

In [84]: (4 < 2) and (3 < 11)
Out[84]: False

In [85]: (2 < 2) and (2 < 2)
Out[85]: False
```

4. If-Else Statement

- Decision making is required when we want to execute a code only if a certain condition is satisfied.
- The if…elif…else statement is used in Python for decision making.

Python if Statement Syntax

```
if test expression:
     statement(s)
```

The program evaluates the test expression and will execute statement(s) only if the text expression is **True**.

If the text expression is **False**, the statement(s) is not executed.

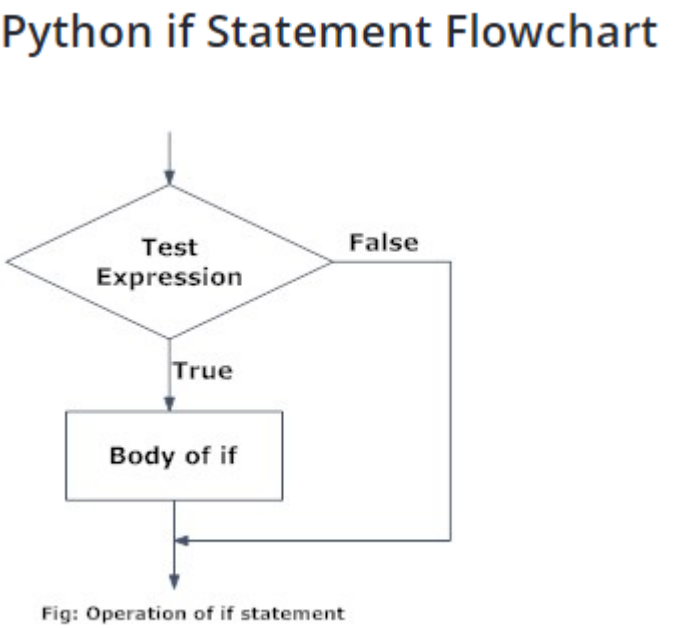


Fig: Operation of if statement

In [108]:
```
num = 3
if num > 0:
    print(num, "is a positive number.")
print("This is always printed.")
```

```
3 is a positive number.
This is always printed.
```

In [109]:
```
num = -1
if num > 0:
    print(num, "is a positive number.")
print("This is also always printed.")
```

```
This is also always printed.
```

In [112]:
```
num = 2
if num != 0:
    print(num, "is not a zero.")
print("This is also always printed.")
```

```
2 is not a zero.
This is also always printed.
```

Python if...else Statement

```
if test expression:
    Body of if
else:
    Body of else
```

The if..else statement evaluates test expression and will execute body of if only when test condition is **True**.

If the condition is **False**, body of else is executed. Indentation is used to separate the blocks.

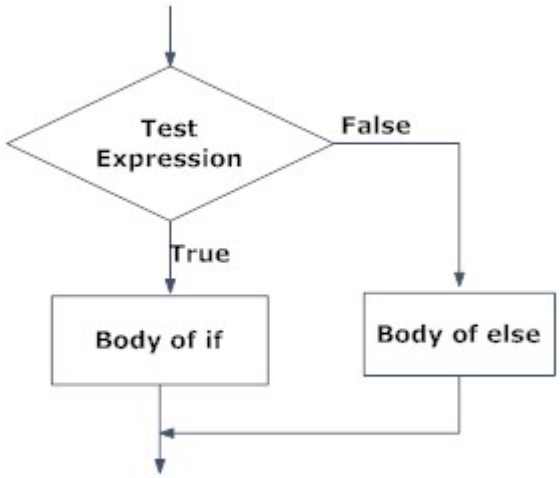


Fig: Operation of if...else statement

In [88]:
```
if 25 % 2:
    print('Even')
else:
    print('Odd')
```

```
Even
```

In [89]:
```
num = 3

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

```
Positive or Zero
```

In [113]:
```
num = -3

if num > 0:
    print("Positive number")
else:
    print("Negative number")
```

```
Negative number
```

Python if...elif...else Statement

```
if test expression:
    Body of if
elif test expression:
    Body of elif
else:
    Body of else
```

If the condition for if is **False**, it checks the condition of the next elif block and so on.

If all the conditions are **False**, body of else is executed.

Flowchart of if...elif...else

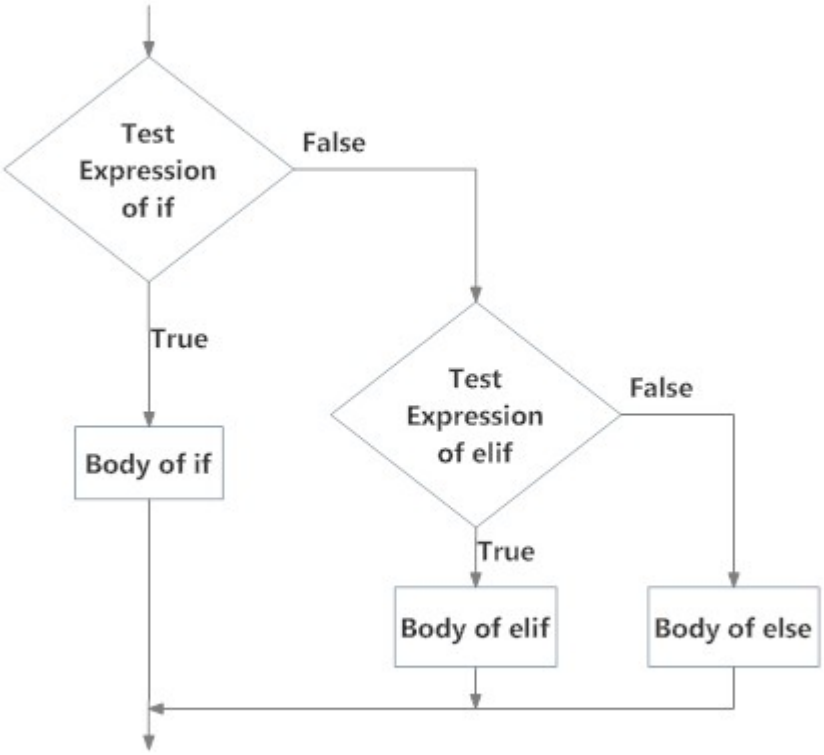


Fig: Operation of if...elif...else statement

In [114]:
```
num = 3.4

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

```
Positive number
```

In [115]:
```
num = -3.4

if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

```
Negative number
```

In [116]:
```
num = 0


if num > 0:
    print("Positive number")
elif num == 0:
    print("Zero")
else:
    print("Negative number")
```

```
Zero
```

4. Python for Loop

- The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

WHY, WHEN & WHERE

- Why? When sometimes in code we need to check the same condition over and over until it mets our requirement we use for or while loop to carry on executions sequentialy.
- When? When some statements are needed to execute until the required condition is met.
- Where? When to get required condition after a series of sequential executions.

Syntax of for loop

```
for val in sequence:
   Body of for
```

Here, **val** is the variable that takes the value of the item inside the sequence on each iteration.

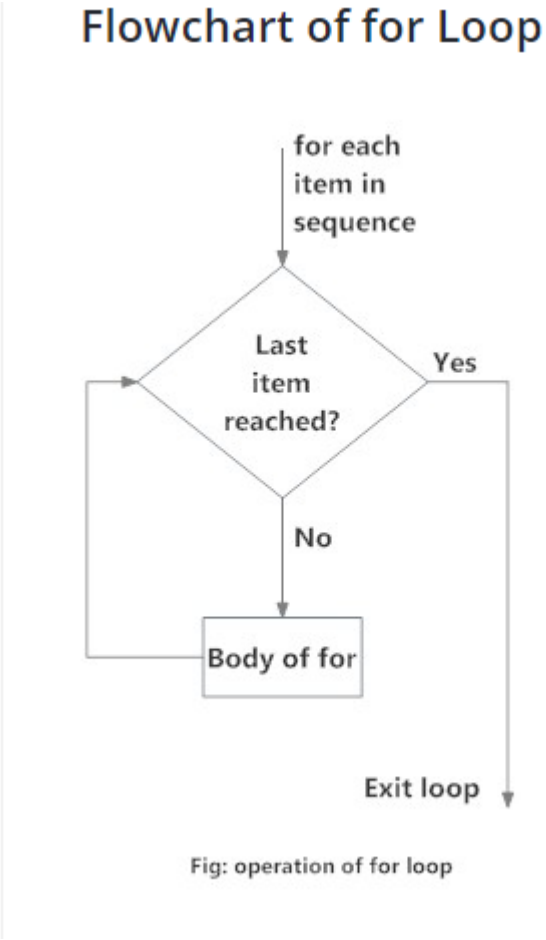


Fig: operation of for loop

In [92]:
```
fact = 1
N = 5
for i in range (1,N+1):
    fact*=i
    print (fact)
```

```
1
2
6
24
120
```

In [93]:
```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

sum = 0

for val in numbers:
    sum = sum+val

print("The sum is", sum)
```

```
The sum is 48
```

In [94]:
```
genre = ['pop', 'rock', 'jazz']

for i in range(len(genre)):
    print("I like", genre[i])
```

```
I like pop
I like rock
I like jazz
```

Python while Loop

- The while loop in Python is used to iterate over a block of code as long as the test expression (condition) is true.

Syntax of while loop

```
while test_expression:
    Body of while
```

The body of the loop is entered only if the test_expression evaluates to **True**. After one iteration, the test expression is checked again. This process continues until the test_expression evaluates to **False**.

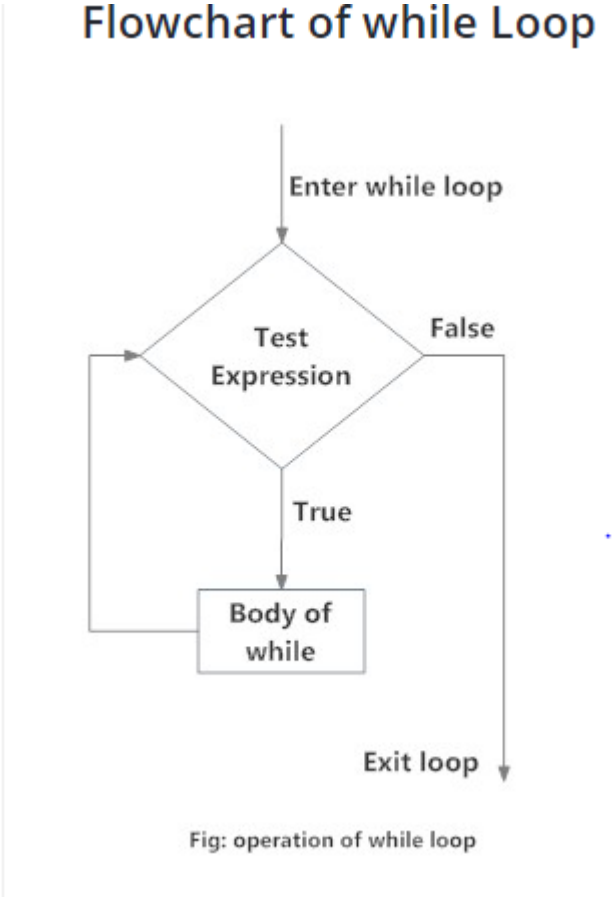


Fig: operation of while loop

In [95]:
```
a = 0
while a < 10:
    a = a+1
    print(a)
```

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

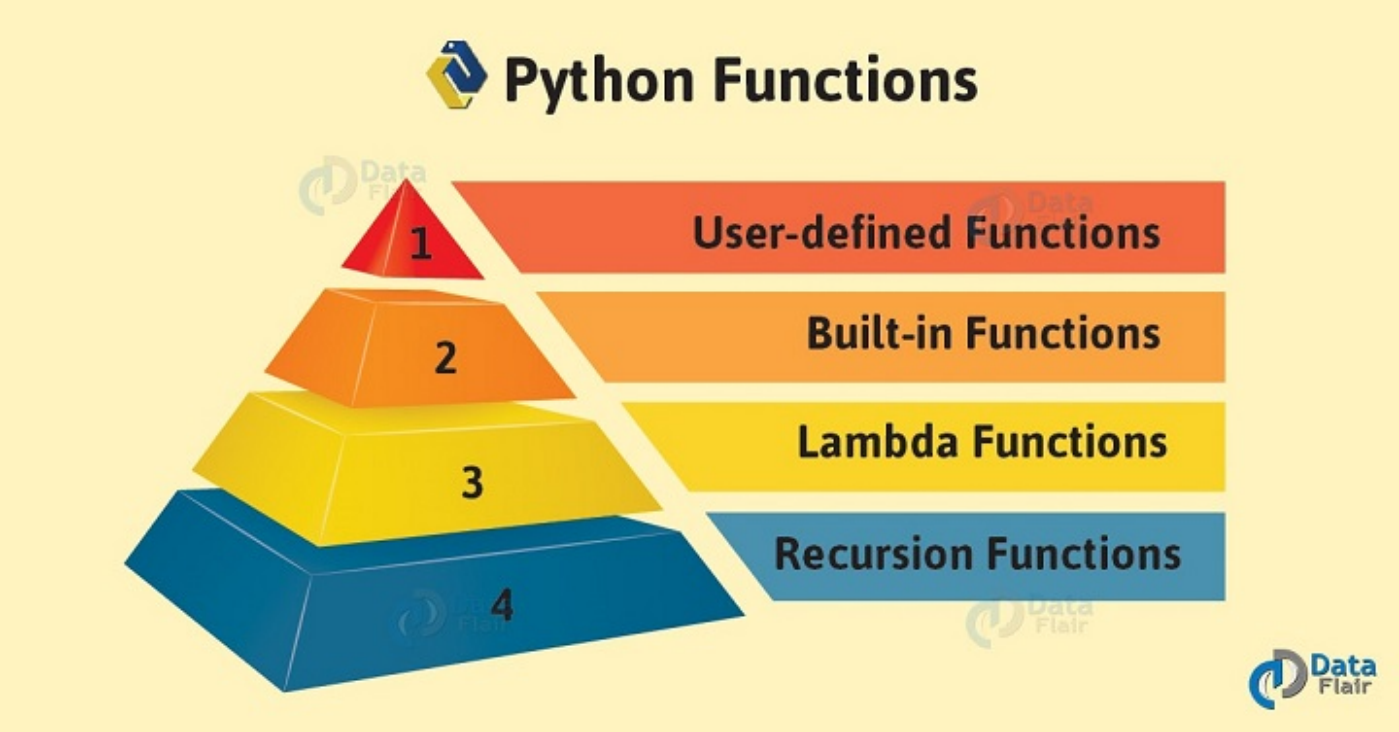
In [96]:
```
n = 10

sum = 0
i = 1

while i <= n:
    sum = sum + i
    i = i+1

print("The sum is", sum)
```

```
The sum is 55
```

In [97]:
```
counter = 0

while counter < 3:
    print("Inside loop")
    counter = counter + 1
else:
    print("Inside else")
```

```
Inside loop
Inside loop
Inside loop
Inside else
```

6. Functions

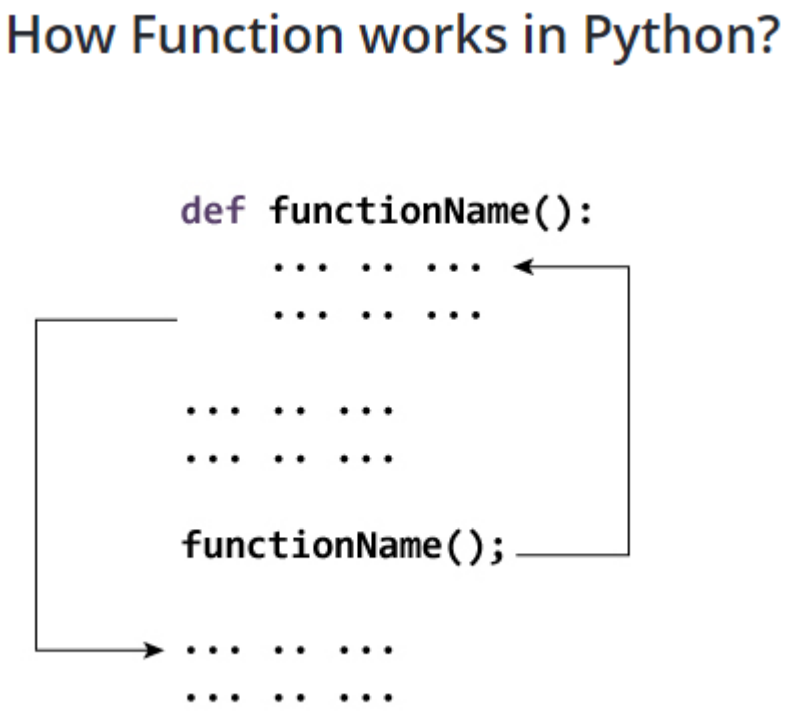


- Function in Python is a group of related statements that perform a specific task.
- Functions help break our program into smaller and modular chunks.
- Functions makes code more organized and manageable.
- Functions avoids repetition and makes code reusable.
- Functions is used to perform a single, related action.
- Functions provides high modularity for your application.
- Functions has a hight degree of code reusing.

Syntax of Function

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

- Keyword def marks the start of function header.
- Parameters (arguments) through which we pass values to a function.
- A colon (:) to mark the end of function header.
- An optional return statement to return a value from the function.



In [146]:
```
def my_function(fname):
  print(fname + " Refsnes")

my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

In [148]:
```
def absolute_value(num):
    """This function returns the absolute
    value of the entered number"""

    if num >= 0:
        return num
    else:
        return -num

print(absolute_value(-4))
```

```
4
```

In [147]:
```
def my_function():
  print("Hello from a function")

my_function()
```

```
Hello from a function
```

7. Lambda Functions

- A lambda function is a small anonymous function created at runtime, using construct called **lambda**.
- A lambda function can take any number of arguments, but can only have one expression.
- Lambda function doesn't include return statement, it always contains an expression which is returned.

Syntax of Lambda Function in python

```
lambda arguments: expression
```

- You need to keep in your knowledge that lambda functions are syntactically restricted to a single expression.

WHY, WHEN & WHERE

- Why? When arbitrary function is required at runtime.
- When? They can be used whenever a function objects are required, they are ordinary functions with def keyword.
- Where? When one time function is required to process data.

In [149]:
```
double = lambda x: x * 2

print(double(5))
```

```
10
```

In [150]:
```
x = lambda a : a + 10
print(x(5))
```

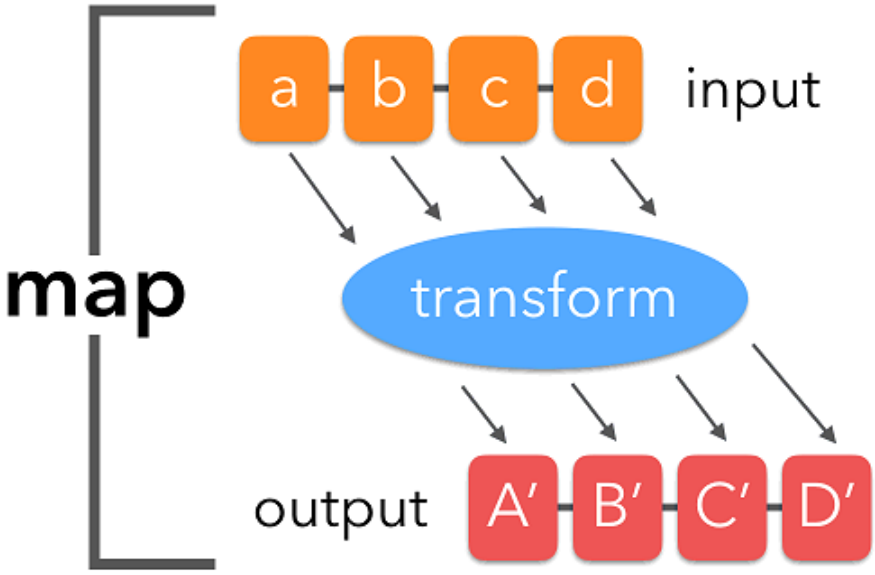
```
15
```

In [151]:
```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

```
13
```

7.1 Map()

- The map() function in Python takes in a function and a list.
- map() function returns a list of the results after applying the given function to each item of a given iterable (list, tuple etc.)
- Python map object is an iterator, so we can iterate over its elements



Syntax of Map()

```
map(function,iteratable,...)
```

- We can pass multiple iterable arguments to map() function, in that case, the specified function must have that many arguments.

In [199]:
```
sentence = 'It is raining cats and dogs'
words = sentence.split()
print (words)
lengths = map(lambda word: len(word), words)
tuple(lengths)
```

```
['It', 'is', 'raining', 'cats', 'and', 'dogs']
```

Out[199]:
```
(2, 2, 7, 4, 3, 4)
```

In [201]:
```
def calculateSquare(n):
  return n*n

numbers = (1, 2, 3, 4)
result = map(calculateSquare, numbers)

numbersSquare = set(result)
print(numbersSquare)
```

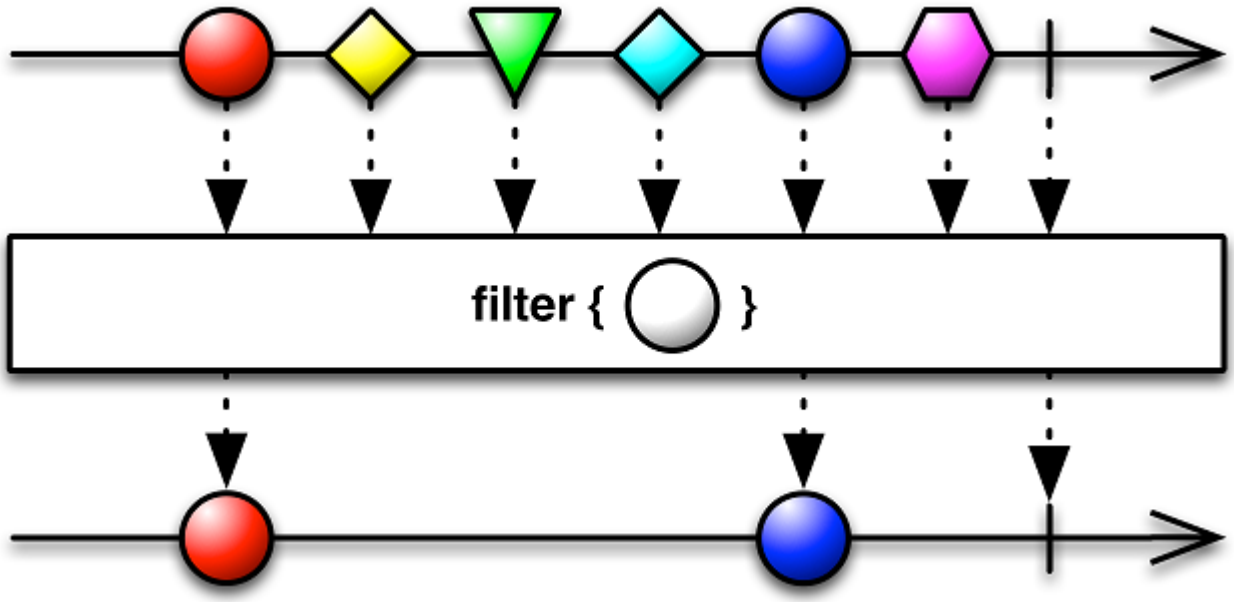
```
{16, 1, 4, 9}
```

In [203]:
```
numbers = (1, 2, 3, 4)
result = map(lambda x: x*x, numbers)
print(result)

numbersSquare = set(result)
print(numbersSquare)
```

```
<map object at 0x000001BA79E6A080>
{16, 1, 4, 9}
```

7.2 Filter() in Python

- The filter() method constructs an iterator from elements of an iterable for which a function returns true.
- The function filter(function, list) offers an elegant way to filter out all the elements of a list.



Filter() Parameters

The filter method takes two parameters:

- **function** - function that tests if elements of an iterable returns true or false. The function will be applied to every element of list **l**.
- **iterable** - iterable which is to be filtered, could be sets, lists, tuples, or containers of any iterators

In [178]:
```
alphabets = ['a', 'b', 'd', 'e', 'i', 'j', 'o']

def filterVowels(alphabet):
    vowels = ['a', 'e', 'i', 'o', 'u']

    if(alphabet in vowels):
        return True
    else:
        return False

filteredVowels = filter(filterVowels, alphabets)

print('The filtered vowels are:')
for vowel in filteredVowels:
    print(vowel)
```

```
The filtered vowels are:
a
e
i
o
```

In [179]:
```
randomList = [1, 'a', 0, False, True, '0']

filteredList = filter(None, randomList)

print('The filtered elements are:')
for element in filteredList:
    print(element)
```

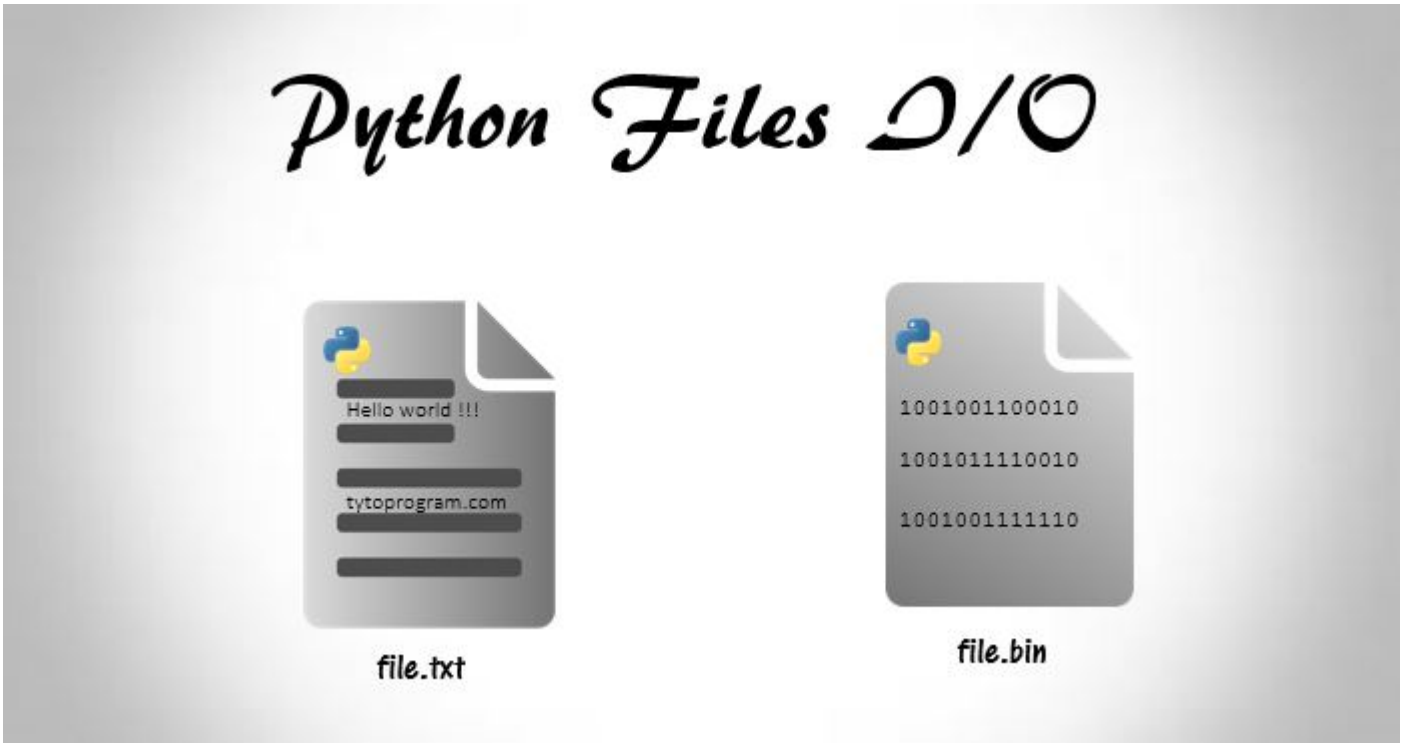
```
The filtered elements are:
1
a
True
0
```

In [196]:
```
seq = [0, 1, 2, 3, 5, 8, 13]
result = filter(lambda x: x % 2, seq)
print(tuple(result))

result = tuple(filter(lambda x: x % 2 == 0, seq))
print(result)
```

```
(1, 3, 5, 13)
(0, 2, 8)
```

8. File I/O

- In this section we will learn numerous built-in functions that are readily available to us by **Python**.



Taking Input in Python from Keyboard

There are two inbuilt functions in **Python** to take input from keyboard.

- raw_input(prompt)
- input (prompt)

raw_input(): In older version of python like **Python 2.x** this function exists. This function takes what exactly type with keyboard, convert it into string and then return it to variable wherever we want to store it.

input(): This function first takes the input from the user and then evaluates the expression.

- Python automatically identifies whether user entered a string or a number or list.
- If the input provided is not correct then either syntax error or exception is raised by python.

In [207]:
```
from six.moves import input
string = input("Enter your name: ");
print(string)
```

```
Enter your name: Ahmad
Ahmad
```

In [1]:
```
from six.moves import input
num = input("Enter a number: ");
print(num)
```

```
Enter a number: 2
2
```

In [2]:
```
num = input ("Enter number :")
print(num)
name1 = input("Enter name : ")
print(name1)

print ("type of number", type(num))
print ("type of name", type(name1))
```

```
Enter number :2
2
Enter name : Ahmad
Ahmad
type of number <class 'str'>
type of name <class 'str'>
```

I/O from or to Text File

Whenever it is needed to read or write something from file in **Python** it is open first. When the required processing is done on file it is needed to be closed so that the resources are tied with the file when freed.

File Opeartion in Python

- Open a file
- Read or write (perform operation)
- Close the file

Python File Modes

In this scenario, we'll read and write to a text file.

- **r** opens a file in read only mode.
- **r+** opens a file read and write mode.
- **w** opens a file in write mode only.
- **a** opens a file in append mode.
- **a+** opens a file in append and read mode.

In [9]:
```
 # Open a file to append
fileOpen = open("file.txt", "a+")
fileOpen.write("COMSATS University Islamabad, Lahore Campus");
fileOpen.close()

# Open a file to read
fileOpen = open("file.txt", "r+")
string = fileOpen.read();

print (string)
# Close opend file
fileOpen.close()
```

```
COMSATS University Islamabad, Lahore CampusCOMSATS University Islamabad, Laho
re Campus
```

In [15]:
```
fileOpen = open("new 1.txt", "w+")
fileOpen.write("Words travel fast");
fileOpen.close()


# Open a file to read
fileOpen = open("new 1.txt", "r+")
str = fileOpen.read();

print (str) # Close opend file

fileOpen.close()
```

```
Words travel fast
```

In [ ]:
```
fileOpen = open("new 1.txt", "w+")
fileOpen.write("Words travel fast");
fileOpen.close()

try:
   f = open("test.txt",encoding = 'utf-8')
finally:
   f.close()
```

File Position

- The method **tell()** returns the current position of the file read/write pointer within the file.

Syntax for tell()

```
fileObject.tell()
```

- The method **seek()** changes the current position of the file location.

Syntax for seek()

```
fileObject.seek()
```

In [ ]:
```
# Open a file
open_file = open("file.txt", "r+")
str = open_file.read(10);
print ("Read String is : \n", str)

# Check current position
position = open_file.tell();
print ("Current file position : \n", position)

# Reposition pointer at the beginning
position = open_file.seek(0, 0);
str = open_file.read(10);
print ("Again read String is : \n", str)

# Close opend file
open_file.close()
```

In [ ]:
```
import os
# rename a file
os.rename("file.txt", "newfile.txt")
```

In [ ]:
```
#remove file
os.remove("newfile.txt")
```

In [ ]:
```
# Open a file
fo = open("foo.txt", "rw+")
print "Name of the file: ", fo.name

line = fo.readline()
print "Read Line: %s" % (line)

# Get the current position of the file...
pos = fo.tell()
print "Current Position: %d" % (pos)

# Close opend file
fo.close()
```

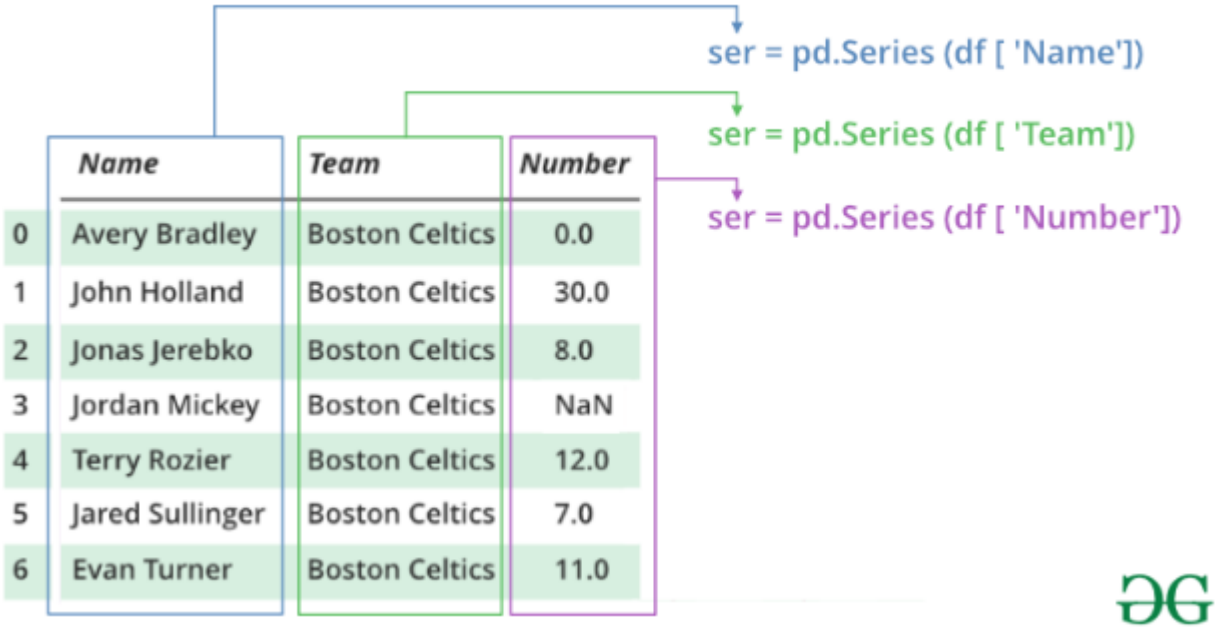
9. Pandas Introduction

- Pandas is an open-source Python Library providing high performance data maipulation and analysis tool using its powerful data structures.
- The name Pandas is derived from the word **Panel Data** – an Econometrics from Multidimensional data.
- In 2008, developer Wes McKinney started developing pandas when in need of high performance, flexible tool for analysis of data.
- Prior to Pandas, Python was majorly used for data munging and preparation.

  ### Key features of Panda
  - It allows for fast analysis and data cleaning and preparation.
  - It excels in performance and productivity.
  - It also has built-in visualization features.
  - It can work with data from variety of sources.
  - Fast and efficient DataFrame object with default and customized indexing.
  - Tools for loading data into in-memory data objects from different file formats.
  - Label-based slicing, indexing and subsetting of large data sets.
  - Group by data for aggregation and transformations.
  - High performance merging and joining of data.
  - Time Series functionality.

10. Series

- A series is very similar to NumPy array.
- Pandas Series is nothing but a column in an excel sheet.
- Series in one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects etc).
- The difference between the NumPy array from a Series, is that a Series can have axis labels, meaning it can be indexed by a label, instead of just a number location.



	Name	Team	Number
0	Avery Bradley	Boston Celtics	0.0
1	John Holland	Boston Celtics	30.0
2	Jonas Jerebko	Boston Celtics	8.0
3	Jordan Mickey	Boston Celtics	NaN
4	Terry Rozier	Boston Celtics	12.0
5	Jared Sullinger	Boston Celtics	7.0
6	Evan Turner	Boston Celtics	11.0

Syntax of Series

Following function is used to create a series:

```
s=pd.Series(data,index=index)
```

- In above function, data can be many different things:
  - A python dict
  - An ndarray
  - A scalar value
- The passed index is a list of axis label. So, this seprates into a few cases depending on what data is:

10.1 From ndarray

- If data is an ndarray, index must be the same length as data.
- If no index is passed, one will be created having values [0, ..., len(data) - 1

In [ ]:
```
import pandas as pd import numpy as np import matplotlib.pyplot as plt
"""following a fucntion is called from panas to create a series.   data would
 be 5 random values and indexes are assigned a-e"""
s = pd.Series(np.random.randn(5), index=['a', 'b', 'c', 'd', 'e'])
s
```

In [ ]:
```
 """Following function will print the index and its datatype"""
    s.index
```

In [ ]:
```
"""If we don't assign the index then it will of length having values [0.....l
en(data-1)]"""
pd.Series(np.random.randn(5))
```

In [ ]:
```
# create a series from scalar

import pandas as pd
import numpy as np
s = pd.Series(7, index=[0, 1, 2, 3])
print s
```

In [ ]:
```
import pandas as pd
import numpy as np
data = {'a' : 0., 'b' : 1., 'c' : 2.}
s = pd.Series(data,index=['b','c','d','a'])
print s
```

10.2 From Dict

- If data is a dict, if index is passed the values in data corresponding to the labels in the index will be pulled out.
- If index is not passed then it will be constructed from the sorted keys of the dict, if possible.

In [ ]:
```
""" In following example, indexes are not given to it is constructed from the
 sorted keys of the dict"""
d = {'a' : 0., 'b' : 1., 'c' : 2.}
# a python dict
pd.Series(d)
```

In [ ]:
```
pd.Series(d, index=['b', 'c', 'd', 'a'])
```

In [ ]:
```
import pandas as pd

dictionary = {'A' : 10, 'B' : 20, 'C' : 30}
series = pd.Series(dictionary)

print(series)
```

In [ ]:
```
import pandas as pd

dictionary = {'D' : 10, 'B' : 20, 'C' : 30}
series = pd.Series(dictionary)
print(series)
```

In [ ]:
```
import pandas as pd

dictionary = {'A' : 50, 'B' : 10, 'C' : 80}
series = pd.Series(dictionary, index =['B', 'C', 'A'])
print(series)
```

10.3 From a scalar value

- If data is a scalar value, an index must be provided. The value will be repeated to match the length of index

In [ ]:
```
"""In following example, a scalar value is given as data so it will be repeate
d to match the length of inde x"""
pd.Series(5., index=['a', 'b', 'c', 'd', 'e'])
```

In [ ]:
```
import pandas as pd
import numpy as np
s = pd.Series(5, index=[0, 1, 2, 3])
print s
```

In [ ]:
```
import pandas as pd
import numpy as np
s = pd.Series(5, index=['a','b','c','d','e'])
print s
```

10.4 Series is ndarray-like

- It acts very similarly to a ndarray.
- It is a valid argument to most Numpy functions. However, things like slicing also slice the index:

In [196]:
```
#we can access a value just like ndarray
#access single value
s[0]
```

Out[196]: 0.5822176222520606

In [195]:
```
 #access range of values
s[:5]
```

Out[195]:
```
0    0.582218
1   -0.391192
2   -0.464336
3    0.820533
4    0.051932
Name: something, dtype: float64
```

In [194]:
```
 #access range of values
s[2:4]
```

Out[194]:
```
2   -0.464336
3    0.820533
Name: something, dtype: float64
```

In [191]:
```
""" Following example will return a range of values in series whose value is g
reater than the median of serie s"""
s[s > s.median()]
```

Out[191]:
```
0    0.582218
3    0.820533
Name: something, dtype: float64
```

In [192]:
```
"""Following example is return the values in series with indexes. 4,3,1 are th
e positions of the indexs
For example: the index at 4,3,1 are e,d,b respectively"""
s[[4, 3, 1]]
```

Out[192]:
```
4    0.051932
3    0.820533
1   -0.391192
Name: something, dtype: float64
```

In [193]:
```
s[[2, 2, 4]]
```

Out[193]:
```
2   -0.464336
2   -0.464336
4    0.051932
Name: something, dtype: float64
```

In [197]:
```
""" Following example returns the exponent values. just like e^a (here a is in
dex and its respective data is placed here)"""
np.exp(s)
```

Out[197]:
```
0    1.790004
1    0.676251
2    0.628552
3    2.271710
4    1.053304
Name: something, dtype: float64
```

In [ ]:
```
 """Following example will get the data of given index"""
s['b','c']
```

In [199]:
```
s['e'] = 12.
s
```

Out[199]:
```
0     0.582218
1    -0.391192
2    -0.464336
3     0.820533
4     0.051932
e    12.000000
Name: something, dtype: float64
```

In [200]:
```
 """Using the get method, a missing label will return None or specified defaul
t"""
s.get('f') #it will return none
s.get('f', np.nan)
```

Out[200]:
```
nan
```

10.6 Vectorized operations and label alignment with Series

- When doing data analysis, as with raw NumPy arrays looping through Series value-by-value is usually not necessary.
- Series can also be passed into most NumPy methods expecting an ndarray.

In [201]:
```
import pandas as pd
import numpy as np
"""following will add the data of respective values of indexes.
For example, in given output, it is calculate d as:
a = s['a'] + s['a']
b = s['b'] + s['b']
c = s['c'] + s['c']
d = s['d'] + s['d']
e = s['e'] + s['e']
"""
s + s
```

Out[201]:
```
0     1.164435
1    -0.782383
2    -0.928672
3     1.641065
4     0.103863
e    24.000000
Name: something, dtype: float64
```

In [202]:
```
 """following will multiply the data of each values of indexes, with 2.
 For example, in given output, it is ca lculated as:
 a = s['a'] *2
 b = s['b'] *2
 c = s['c'] *2
 d = s['d'] *2
 e = s['e'] *2"""
s * 2
```

Out[202]:
```
0     1.164435
1    -0.782383
2    -0.928672
3     1.641065
4     0.103863
e    24.000000
Name: something, dtype: float64
```

In [122]:
```
s = pd.Series(np.random.randn(5), name='something')
s
```

Out[122]:
```
0    0.582218
1   -0.391192
2   -0.464336
3    0.820533
4    0.051932
Name: something, dtype: float64
```

In [123]:
```
s.name
```

Out[123]:
```
'something'
```

In [124]:
```
s2 = s.rename("different")
s2.name
```

Out[124]:
```
'different'
```

11. Data Frames

- DataFrames are the workhorse of pandas and are directly inspired by the R programming language.
- Like Series, DataFrame accepts many different kinds of input:
  - Dict of 1D ndarrays, lists, dicts, or Series
  - 2-D numpy.ndarray
  - Structured or record ndarray
  - A Series
  - Another DataFrame
- Along with the data, you can optionally pass index (row labels) and columns (column labels) arguments.
- If you pass an index and / or columns, you are guaranteeing the index and / or columns of the resulting DataFrame.
- Thus, a dict of Series plus a specific index will discard all data not matching up to the passed index.
- If axis labels are not passed, they will be constructed from the input data based on common sense rules

11.1 From dict of Series or dicts

- The result index will be the union of the indexes of the various Series.
- If there are any nested dicts, these will be first converted to Series.
- If no columns are passed, the columns will be the sorted list of dict keys.

In [129]:
```
import pandas as pd
import numpy as np
""" A dict is created """
d = {
 'one' : pd.Series([1., 2., 3.], index=['a', 'b', 'c']),
 'two' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd'])
  }

"""create a dataframe. row label will be the indes of a series."""
df = pd.DataFrame(d)
df
```

Out[129]:

	one	two
a	1.0	1.0
b	2.0	2.0
c	3.0	3.0
d	NaN	4.0

In [131]:
```
import pandas as pd
import numpy as np
d2 = {
 'first' : pd.Series([1., 2., 3.], index=['x', 'y', 'z']),
 'second' : pd.Series([1., 2., 3., 4.], index=['a', 'b', 'c', 'd']),
 'third' : pd.Series([1., 2., 3., 4., 5., 6., 7.], index=['a', 'b', 'c', 'd',
'e','f','g']),
  }

df = pd.DataFrame(d2)
df
```

Out[131]:

	first	second	third
a	NaN	1.0	1.0
b	NaN	2.0	2.0
c	NaN	3.0	3.0
d	NaN	4.0	4.0
e	NaN	NaN	5.0
f	NaN	NaN	6.0
g	NaN	NaN	7.0
x	1.0	NaN	NaN
y	2.0	NaN	NaN
z	3.0	NaN	NaN

In [4]:
```
import pandas as pd
import numpy as np
""" a data frame will be constructed for given row labels"""
pd.DataFrame(d, index=['d', 'b', 'a'])
```

Out[4]:

	one	two
d	NaN	4.0
b	2.0	2.0
a	1.0	1.0

In [132]:
```
pd.DataFrame(d, index=['z', 'y', 'a'])
```

Out[132]:

	one	two
z	NaN	NaN
y	NaN	NaN
a	1.0	1.0

In [5]:
```
import pandas as pd
import numpy as np
"""following example shows a data frame when we give coloumn labels"""
pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
```

Out[5]:

	two	three
d	4.0	NaN
b	2.0	NaN
a	1.0	NaN

In [134]:
```
pd.DataFrame(d, index=['a', 'b', 'a','b'], columns=['two', 'three','four'])
```

Out[134]:

	two	three	four
a	1.0	NaN	NaN
b	2.0	NaN	NaN
a	1.0	NaN	NaN
b	2.0	NaN	NaN

In [135]:
```
import pandas as pd
import numpy as np
df.columns
```

Out[135]:
```
Index(['first', 'second', 'third'], dtype='object')
```

11.2 From dict of ndarrays / lists

- The ndarrays must all be the same length.
- If an index is passed, it must clearly also be the same length as the arrays.
- If no index is passed, the result will be range(n), where n is the array length

In [7]:
```
import pandas as pd
import numpy as np
"""followind examples shows that ndarray has same length"""
d = {
 'one' : [1., 2., 3., 4.],
 'two' : [4., 3., 2., 1.]
 }
"""column labels are not given so the result will be range(n), where n is the
 array length"""
pd.DataFrame(d)
```

Out[7]:

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

In [136]:
```
d = {
 'one' : [1., 2., 3., 4.],
 'two' : [4., 3., 2., 1.],
 'three':[1., 2., 2., 1.],
 'four' :[5., 6., 7., 8.]
}
pd.DataFrame(d)
```

Out[136]:

	one	two	three	four
0	1.0	4.0	1.0	5.0
1	2.0	3.0	2.0	6.0
2	3.0	2.0	2.0	7.0
3	4.0	1.0	1.0	8.0

In [8]:
```
import pandas as pd
import numpy as np

"""If indexs are given then it would be same length as arrays"""
pd.DataFrame(d, index=['a', 'b', 'c', 'd'])
```

Out[8]:

	one	two
a	1.0	4.0
b	2.0	3.0
c	3.0	2.0
d	4.0	1.0

In [137]:
```
pd.DataFrame(d, index=['aa', 'ab', 'ac', 'ad'])
```

Out[137]:

	one	two	three	four
aa	1.0	4.0	1.0	5.0
ab	2.0	3.0	2.0	6.0
ac	3.0	2.0	2.0	7.0
ad	4.0	1.0	1.0	8.0

In [9]:
```
import pandas as pd
import numpy as np
"""constructing data frame from a list of dicts"""
data2 = [{'a': 1, 'b': 2}, {'a': 5, 'b': 10, 'c': 20}]
pd.DataFrame(data2)
```

Out[9]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

In [139]:
```
data3 = [{'x1': 10, 'x2': 20}, {'y1': 55, 'y2': 130, 'y3': 230}]
pd.DataFrame(data3)
```

Out[139]:

	x1	x2	y1	y2	y3
0	10.0	20.0	NaN	NaN	NaN
1	NaN	NaN	55.0	130.0	230.0

In [11]:
```
import pandas as pd
import numpy as np
"""passing list of dicts as data and indexes (row labels)"""
pd.DataFrame(data2, index=['first', 'second'])
```

Out[11]:

	a	b	c
first	1	2	NaN
second	5	10	20.0

In [141]:
```
pd.DataFrame(data3, index=['first', 'second'])
```

Out[141]:

	x1	x2	y1	y2	y3
first	10.0	20.0	NaN	NaN	NaN
second	NaN	NaN	55.0	130.0	230.0

In [12]:
```
import pandas as pd
import numpy as np
"""passing list of dicts as data and columns (columns labels)"""
pd.DataFrame(data2, columns=['a', 'b'])
```

Out[12]:

	a	b
0	1	2
1	5	10

In [144]:
```
pd.DataFrame(data3, columns=['x1', 'x2','y1','y2','y3'])
```

Out[144]:

	x1	x2	y1	y2	y3
0	10.0	20.0	NaN	NaN	NaN
1	NaN	NaN	55.0	130.0	230.0

11.4 From a dict of tuples

You can automatically create a multi-indexed frame by passing a tuples dictionary

In [145]:
```
pd.DataFrame({('a', 'b'): {('A', 'B'): 1, ('A', 'C'): 2},
 ('a', 'a'): {('A', 'C'): 3, ('A', 'B'): 4},
 ('a', 'c'): {('A', 'B'): 5, ('A', 'C'): 6},
 ('b', 'a'): {('A', 'C'): 7, ('A', 'B'): 8},
 ('b', 'b'): {('A', 'D'): 9, ('A', 'B'): 10}})
```

Out[145]:

		a			b	
		b	a	c	a	b
A	**B**	1.0	4.0	5.0	8.0	10.0
	C	2.0	3.0	6.0	7.0	NaN
	D	NaN	NaN	NaN	NaN	9.0

In [147]:
```
pd.DataFrame({('x', 'y'): {('X', 'Y'): 1, ('X', 'Z'): 2},
 ('x1', 'y1'): {('X', 'Z'): 3, ('X', 'Y'): 4},
 ('x2', 'y2'): {('X', 'W'): 9, ('X', 'Y'): 10}})
```

Out[147]:

		x	x1	x2
		y	y1	y2
X	**W**	NaN	NaN	9.0
	Y	1.0	4.0	10.0
	Z	2.0	3.0	NaN

11.5 Alternate Constructors

DataFrame.from_dict

- DataFrame.from_dict takes a dict of dicts or a dict of array-like sequences and returns a DataFrame.
- It operates like the DataFrame constructor except for the orient parameter which is 'columns' by default, but which can be set to 'index' in order to use the dict keys as row labels.

DataFrame.from_records

- DataFrame.from_records takes a list of tuples or an ndarray with structured dtype.
- Works analogously to the normal DataFrame constructor, except that index maybe be a specific field of the structured dtype to use as the index.

For example:

In [14]:
```
import pandas as pd
import numpy as np
data = np.zeros((2,), dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')])
data
```

Out[14]:
```
array([(0, 0., b''), (0, 0., b'')],
      dtype=[('A', '<i4'), ('B', '<f4'), ('C', 'S10')])
```

In [16]:
```
import pandas as pd
import numpy as np
pd.DataFrame.from_records(data, index='C')
```

Out[16]:

C	A	B
b''	0	0.0
b''	0	0.0

In [153]:
```
pd.DataFrame.from_records(data, index='A')
```

Out[153]:

A	B	C
0	0.0	b''
0	0.0	b''

In [154]:
```
pd.DataFrame.from_records(data, index='B')
```

Out[154]:

B	A	C
0.0	0	b"
0.0	0	b"

DataFrame.from_items

- DataFrame.from_items works analogously to the form of the dict constructor that takes a sequence of (key, value) pairs, where the keys are column (or row, in the case of orient='index') names, and the value are the column values (or row values).
- This can be useful for constructing a DataFrame with the columns in a particular order without having to pass an explicit list of columns

In [157]:
```
pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])])
```

```
C:\Users\HUNNY\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: FutureWar
ning: from_items is deprecated. Please use DataFrame.from_dict(dict(items),
...) instead. DataFrame.from_dict(OrderedDict(items)) may be used to preserve
the key order.
  """Entry point for launching an IPython kernel.
```

Out[157]:

	A	B
0	1	4
1	2	5
2	3	6

In [158]:
```
pd.DataFrame.from_items([('X', [11, 12, 13]), ('Y', [41, 51, 61])])
```

```
C:\Users\HUNNY\Anaconda3\lib\site-packages\ipykernel_launcher.py:1: FutureWar
ning: from_items is deprecated. Please use DataFrame.from_dict(dict(items),
...) instead. DataFrame.from_dict(OrderedDict(items)) may be used to preserve
the key order.
  """Entry point for launching an IPython kernel.
```

Out[158]:

	X	Y
0	11	41
1	12	51
2	13	61

In [20]:
```
import pandas as pd
import numpy as np
pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
 orient='index', columns=['one', 'two', 'three'])
```

```
C:\Users\HUNNY\Anaconda3\lib\site-packages\ipykernel_launcher.py:4: FutureWar
ning: from_items is deprecated. Please use DataFrame.from_dict(dict(items),
...) instead. DataFrame.from_dict(OrderedDict(items)) may be used to preserve
the key order.
  after removing the cwd from sys.path.
```

Out[20]:

	one	two	three
A	1	2	3
B	4	5	6

In [159]:
```
pd.DataFrame.from_items([('X', [11, 12, 13]), ('Y', [41, 51, 61])],
 orient='index', columns=['one', 'two', 'three'])
```

```
C:\Users\HUNNY\Anaconda3\lib\site-packages\ipykernel_launcher.py:2: FutureWar
ning: from_items is deprecated. Please use DataFrame.from_dict(dict(items),
...) instead. DataFrame.from_dict(OrderedDict(items)) may be used to preserve
the key order.
```

Out[159]:

	one	two	three
X	11	12	13
Y	41	51	61

11.6 Column selection, addition, deletion

- DataFrame can be treated semantically like a dict of like-indexed Series objects. Getting, setting, and deleting columns works with the same syntax as the analogous dict operations.

In [31]:
```
import pandas as pd
import numpy as np
df['one']
```

Out[31]:
```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

In [32]:
```
import pandas as pd
import numpy as np
df['three'] = df['one'] * df['two']
```

In [33]:
```
import pandas as pd
import numpy as np
df['flag'] = df['one'] > 2
```

In [34]:
```
import pandas as pd
import numpy as np
df
```

Out[34]:

	one	two	three	flag
a	1.0	1.0	1.0	False
b	2.0	2.0	4.0	False
c	3.0	3.0	9.0	True
d	NaN	4.0	NaN	False

Columns can be deleted or popped like with a dict:

In [35]:
```
del df['two']
```

In [36]:
```
three = df.pop('three')
```

In [37]:
```
df
```

Out[37]:

	one	flag
a	1.0	False
b	2.0	False
c	3.0	True
d	NaN	False

When inserting a scalar value, it will naturally be propagated to fill the column:

In [38]:
```
df['foo'] = 'bar'
```

In [39]:
```
df
```

Out[39]:

	one	flag	foo
a	1.0	False	bar
b	2.0	False	bar
c	3.0	True	bar
d	NaN	False	bar

When inserting a Series that does not have the same index as the DataFrame, it will be conformed to the DataFrame's index:

In [43]:
```
df['one_trunc'] = df['one'][:2]
```

In [42]:
```
df
```

Out[42]:

	one	flag	foo	one_trunc
a	1.0	False	bar	1.0
b	2.0	False	bar	2.0
c	3.0	True	bar	NaN
d	NaN	False	bar	NaN

By default, columns get inserted at the end. The insert function is available to insert at a particular location in the columns:

In [44]:
```
df.insert(1, 'bar2', df['one'])
```

In [45]:
```
df
```

Out[45]:

	one	bar2	flag	foo	one_trunc
a	1.0	1.0	False	bar	1.0
b	2.0	2.0	False	bar	2.0
c	3.0	3.0	True	bar	NaN
d	NaN	NaN	False	bar	NaN

11.7 Indexing / Selection

- Row selection, for example, returns a Series whose index is the columns of the DataFrame:

In [46]:
```
df.loc['b']
```

Out[46]:
```
one              2
bar2             2
flag         False
foo            bar
one_trunc        2
Name: b, dtype: object
```

In [164]:
```
df.loc['a']
```

Out[164]:
```
first     NaN
second    1.0
third     1.0
Name: a, dtype: float64
```

In [47]:
```
 df.iloc[2]
```

Out[47]:
```
one             3
bar2            3
flag         True
foo           bar
one_trunc     NaN
Name: c, dtype: object
```

In [165]:
```
df.iloc[3]
```

Out[165]:
```
first     NaN
second    4.0
third     4.0
Name: d, dtype: float64
```

11.8 Data alignment and arithmetic

- Data alignment between DataFrame objects automatically align on both the columns and the index.
- Again, the resulting object will have the union of the column and row labels.

In [48]:
```
df = pd.DataFrame(np.random.randn(10, 4), columns=['A', 'B', 'C', 'D'])
```

In [49]:
```
df2 = pd.DataFrame(np.random.randn(7, 3), columns=['A', 'B', 'C'])
```

In [50]:
```
df + df2
```

Out[50]:

	A	B	C	D
0	-0.447279	-1.568538	0.509418	NaN
1	0.424074	2.890490	1.282371	NaN
2	-1.645430	1.330161	1.539589	NaN
3	0.331215	-1.075427	1.909875	NaN
4	-3.133160	-2.698966	1.191488	NaN
5	0.891790	0.221977	-0.275837	NaN
6	2.422088	0.670811	-0.971842	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	NaN	NaN	NaN	NaN

In [166]:
```
df - df2
```

Out[166]:

	A	B	C	D	E	first	second	third
2013-01-01 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2013-01-02 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2013-01-03 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2013-01-04 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2013-01-05 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2013-01-06 00:00:00	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
a	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
b	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
c	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
d	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
e	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
f	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
g	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
x	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
y	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
z	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN

When doing an operation between DataFrame and Series, the default behavior is to align the Series index on the DataFrame columns. For example:

In [54]:
```
df - df.iloc[0]
```

Out[54]:

	A	B	C	D
0	0.000000	0.000000	0.000000	0.000000
1	0.063963	1.852888	-0.316088	-0.741770
2	0.216007	2.474731	1.036199	-0.835506
3	-1.046098	0.466852	1.003393	0.842332
4	-2.180281	-0.095977	-0.145080	0.585060
5	0.759285	0.693863	-0.632371	-0.032616
6	-0.492457	-0.336885	-0.611249	1.775446
7	-0.640803	0.196185	-1.993447	2.862560
8	0.003786	0.714664	-2.301199	-0.415774
9	0.184706	0.723159	-1.576223	-1.799558

In [167]:
```
df + df.iloc[0]
```

Out[167]:

	first	second	third
a	NaN	2.0	2.0
b	NaN	3.0	3.0
c	NaN	4.0	4.0
d	NaN	5.0	5.0
e	NaN	NaN	6.0
f	NaN	NaN	7.0
g	NaN	NaN	8.0
x	NaN	NaN	NaN
y	NaN	NaN	NaN
z	NaN	NaN	NaN

In [55]:
```
df * 5 + 2
```

Out[55]:

	A	B	C	D
0	3.721446	-2.216312	4.263979	3.083901
1	4.041260	7.048130	2.683538	-0.624952
2	4.801483	10.157342	9.444975	-1.093629
3	-1.509044	0.117949	9.280943	7.295561
4	-7.179959	-2.696199	3.538577	6.009203
5	7.517871	1.253005	1.102125	2.920819
6	1.259160	-3.900739	1.207734	11.961132
7	0.517430	-1.235385	-5.703256	17.396699
8	3.740378	1.357010	-7.242016	1.005031
9	4.644978	1.399481	-3.617137	-5.913892

In [168]:
```
df * 3 * 2 -1
```

Out[168]:

	first	second	third
a	NaN	5.0	5.0
b	NaN	11.0	11.0
c	NaN	17.0	17.0
d	NaN	23.0	23.0
e	NaN	NaN	29.0
f	NaN	NaN	35.0
g	NaN	NaN	41.0
x	5.0	NaN	NaN
y	11.0	NaN	NaN
z	17.0	NaN	NaN

In [56]:
```
1 / df
```

Out[56]:

	A	B	C	D
0	2.904535	-1.185871	2.208501	4.612969
1	2.449468	0.990466	7.314887	-1.904797
2	1.784769	0.612945	0.671594	-1.616225
3	-1.424890	-2.656676	0.686724	0.944187
4	-0.544665	-1.064691	3.249757	1.247131
5	0.906147	-6.693486	-5.568706	5.429951
6	-6.749095	-0.847352	-6.311013	0.501951
7	-3.372523	-1.545411	-0.649076	0.324745
8	2.872939	-7.776171	-0.541008	-5.025282
9	1.890375	-8.326134	-0.890133	-0.631800

In [169]:
```
2 / df
```

Out[169]:

	first	second	third
a	NaN	2.000000	2.000000
b	NaN	1.000000	1.000000
c	NaN	0.666667	0.666667
d	NaN	0.500000	0.500000
e	NaN	NaN	0.400000
f	NaN	NaN	0.333333
g	NaN	NaN	0.285714
x	2.000000	NaN	NaN
y	1.000000	NaN	NaN
z	0.666667	NaN	NaN

In [57]:
```
df ** 4
```

Out[57]:

	A	B	C	D
0	0.014051	0.505651	0.042035	0.002208
1	0.027779	1.039064	0.000349	0.075964
2	0.098553	7.084590	4.915559	0.146552
3	0.242591	0.020075	4.496458	1.258252
4	11.362736	0.778226	0.008966	0.413382
5	1.483222	0.000498	0.001040	0.001150
6	0.000482	1.939749	0.000630	15.752691
7	0.007730	0.175317	5.634007	89.914645
8	0.014679	0.000273	11.673117	0.001568
9	0.078308	0.000208	1.592869	6.275963

In [170]:
```
df ** 5
```

Out[170]:

	first	second	third
a	NaN	1.0	1.0
b	NaN	32.0	32.0
c	NaN	243.0	243.0
d	NaN	1024.0	1024.0
e	NaN	NaN	3125.0
f	NaN	NaN	7776.0
g	NaN	NaN	16807.0
x	1.0	NaN	NaN
y	32.0	NaN	NaN
z	243.0	NaN	NaN

In [58]:
```
df1 = pd.DataFrame({'a' : [1, 0, 1], 'b' : [0, 1, 1] }, dtype=bool)
```

In [59]:
```
df2 = pd.DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)
```

In [60]:
```
pd.DataFrame({'a' : [0, 1, 1], 'b' : [1, 1, 0] }, dtype=bool)
```

Out[60]:

	a	b
0	False	True
1	True	True
2	True	False

In [171]:
```
df11 = pd.DataFrame({'x' : [1, 0, 1], 'y' : [0, 1, 1] }, dtype=bool)
df22 = pd.DataFrame({'x' : [0, 1, 0], 'y' : [1, 1, 1] }, dtype=bool)
pd.DataFrame({'x' : [0, 1, 1], 'y' : [1, 1, 0] }, dtype=bool)
```

Out[171]:

	x	y
0	False	True
1	True	True
2	True	False

In [61]:
```
df1 & df2
```

Out[61]:

	a	b
0	False	False
1	False	True
2	True	False

In [172]:
```
df11 & df22
```

Out[172]:

	x	y
0	False	False
1	False	True
2	False	True

In [63]:
```
df1 | df2
```

Out[63]:

	a	b
0	True	True
1	True	True
2	True	True

In [173]:
```
df11 | df22
```

Out[173]:

	x	y
0	True	True
1	True	True
2	True	True

In [64]:
```
-df1
```

Out[64]:

	a	b
0	False	True
1	True	False
2	False	False

In [174]:
```
+df22
```

Out[174]:

	x	y
0	False	True
1	True	True
2	False	True

11.9 Transposing

- To transpose, access the T attribute (also the transpose function), similar to an ndarray

In [65]:
```
df[:5].T
```

Out[65]:

	0	1	2	3	4
A	0.344289	0.408252	0.560297	-0.701809	-1.835992
B	-0.843262	1.009626	1.631468	-0.376410	-0.939240
C	0.452796	0.136708	1.488995	1.456189	0.307715
D	0.216780	-0.524990	-0.618726	1.059112	0.801841

In [176]:
```
df[2:4].T
```

Out[176]:

	c	d
first	NaN	NaN
second	3.0	4.0
third	3.0	4.0

Creating a DataFrame by passing a numpy array, with a datetime index and labeled columns:

In [66]:
```
dates = pd.date_range('20130101', periods=6)
```

In [73]:
```
dates
```

Out[73]:
```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [184]:
```
dates2 = pd.date_range('20130101', periods=11)
dates2
```

Out[184]:
```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06', '2013-01-07', '2013-01-08',
               '2013-01-09', '2013-01-10', '2013-01-11'],
              dtype='datetime64[ns]', freq='D')
```

In [68]:
```
df = pd.DataFrame(np.random.randn(6,4), index=dates, columns=list('ABCD'))
```

In [69]:
```
df
```

Out[69]:

	A	B	C	D
2013-01-01	-0.767112	0.659955	1.535587	-0.531216
2013-01-02	1.404921	-2.025853	-1.246922	-0.698910
2013-01-03	-0.744812	-0.716729	0.202237	-0.189283
2013-01-04	1.010707	-0.426484	0.968721	1.346020
2013-01-05	2.386646	0.106212	0.271260	0.171864
2013-01-06	0.262806	-0.469035	0.009304	1.007151

Creating a DataFrame by passing a dict of objects that can be converted to series-like.

In [74]:
```
df2 = pd.DataFrame({ 'A' : 1.,
 'B' : pd.Timestamp('20130102'),
 'C' : pd.Series(1,index=list(range(4)),dtype='float32'),
 'D' : np.array([3] * 4,dtype='int32'),
 'E' : pd.Categorical(["test","train","test","train"]),
 'F' : 'foo' })
```

In [75]:
```
df2
```

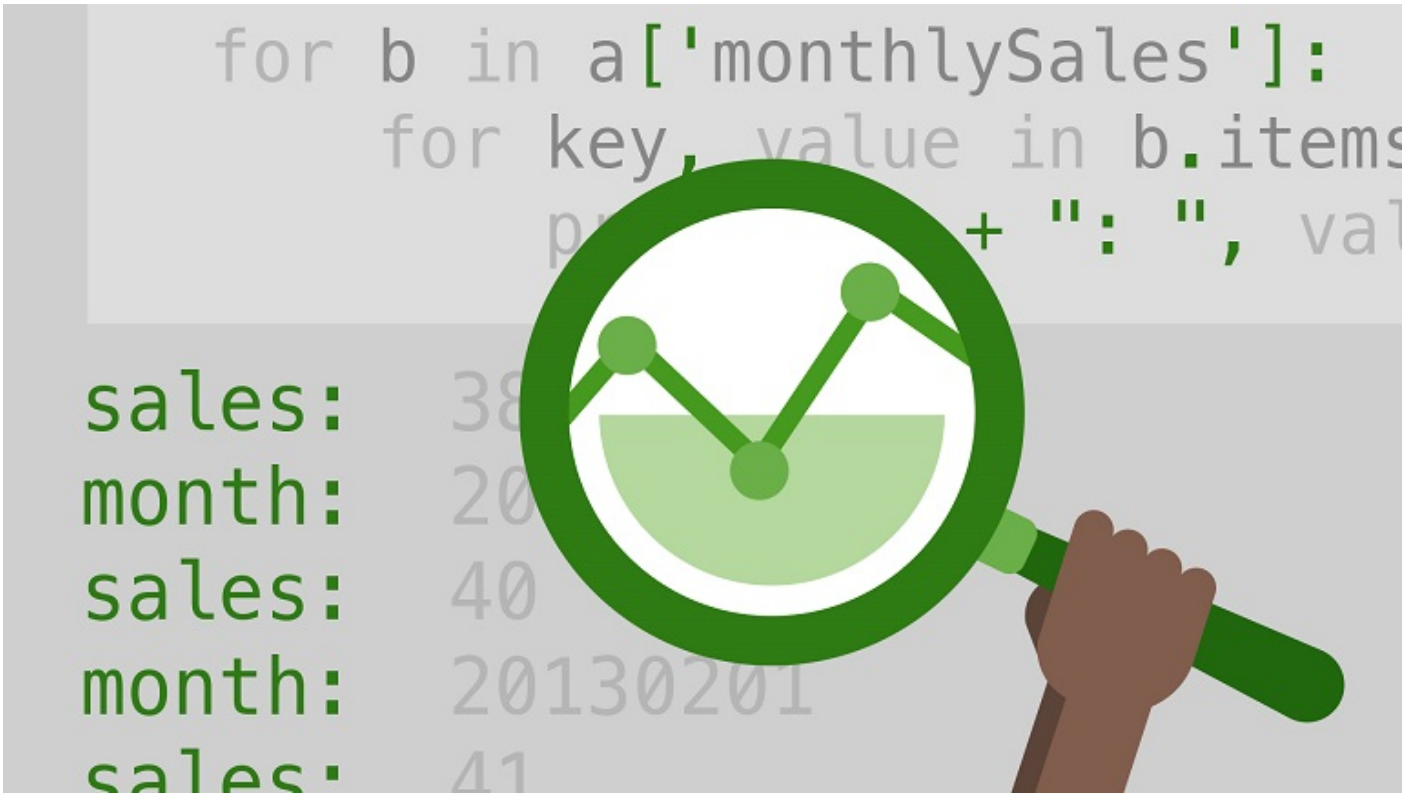
Out[75]:

	A	B	C	D	E	F
0	1.0	2013-01-02	1.0	3	test	foo
1	1.0	2013-01-02	1.0	3	train	foo
2	1.0	2013-01-02	1.0	3	test	foo
3	1.0	2013-01-02	1.0	3	train	foo

In [72]:
```
df2.dtypes
```

Out[72]:
```
A           float64
B    datetime64[ns]
C           float32
D             int32
E          category
F            object
dtype: object
```

12. Viewing Data



- We can view data / display data in different ways:
- See the top & bottom rows of the frame
- Selecting a single column
- Selecting via [], which slices the rows
- For getting a cross section using a label
- Selecting on a multi-axis by label
- Showing label slicing, both endpoints are included
- Reduction in the dimensions of the returned object
- For getting a scalar value
- For getting fast access to a scalar
- Select via the position of the passed integers
- By integer slices, acting similar to numpy/python
- By lists of integer position locations, similar to the numpy/python style
- For slicing rows explicitly
- For slicing columns explicitly
- For getting a value explicitly
- For getting fast access to a scalar
- Using a single column’s values to select data.
- Selecting values from a DataFrame where a boolean condition is met.
- Using the isin() method for filtering

In [79]:
```
df.head()
```

Out[79]:

	A	B	C	D
2013-01-01	-0.767112	0.659955	1.535587	-0.531216
2013-01-02	1.404921	-2.025853	-1.246922	-0.698910
2013-01-03	-0.744812	-0.716729	0.202237	-0.189283
2013-01-04	1.010707	-0.426484	0.968721	1.346020
2013-01-05	2.386646	0.106212	0.271260	0.171864

In [80]:
```
df.tail(3)
```

Out[80]:

	A	B	C	D
2013-01-04	1.010707	-0.426484	0.968721	1.346020
2013-01-05	2.386646	0.106212	0.271260	0.171864
2013-01-06	0.262806	-0.469035	0.009304	1.007151

In [82]:
```
df.index
```

Out[82]:
```
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

WHY, WHEN & WHERE

- Why? When statistical graphics needs to be visualized.
- When? When we need to represent data in graphical and informative manner.
- Where? When data is to be represented in frames for more informative view.

In [83]:
```
df.columns
```

Out[83]:
```
Index(['A', 'B', 'C', 'D'], dtype='object')
```

In [84]:
```
df.values
```

Out[84]:
```
array([[-0.76711192,  0.65995498,  1.53558728, -0.53121555],
       [ 1.40492068, -2.02585292, -1.24692163, -0.69890994],
       [-0.74481245, -0.71672901,  0.20223742, -0.18928336],
       [ 1.01070661, -0.42648438,  0.96872106,  1.34601994],
       [ 2.38664607,  0.10621228,  0.27126044,  0.17186384],
       [ 0.26280602, -0.46903543,  0.00930355,  1.00715077]])
```

In [85]:
```
df.T
```

Out[85]:

	2013-01-01 00:00:00	2013-01-02 00:00:00	2013-01-03 00:00:00	2013-01-04 00:00:00	2013-01-05 00:00:00	2013-01-06 00:00:00
A	-0.767112	1.404921	-0.744812	1.010707	2.386646	0.262806
B	0.659955	-2.025853	-0.716729	-0.426484	0.106212	-0.469035
C	1.535587	-1.246922	0.202237	0.968721	0.271260	0.009304
D	-0.531216	-0.698910	-0.189283	1.346020	0.171864	1.007151

In [86]:
```
df.sort_index(axis=1, ascending=False)
```

Out[86]:

	D	C	B	A
2013-01-01	-0.531216	1.535587	0.659955	-0.767112
2013-01-02	-0.698910	-1.246922	-2.025853	1.404921
2013-01-03	-0.189283	0.202237	-0.716729	-0.744812
2013-01-04	1.346020	0.968721	-0.426484	1.010707
2013-01-05	0.171864	0.271260	0.106212	2.386646
2013-01-06	1.007151	0.009304	-0.469035	0.262806

In [87]:
```
df.sort_values(by='B')
```

Out[87]:

	A	B	C	D
2013-01-02	1.404921	-2.025853	-1.246922	-0.698910
2013-01-03	-0.744812	-0.716729	0.202237	-0.189283
2013-01-06	0.262806	-0.469035	0.009304	1.007151
2013-01-04	1.010707	-0.426484	0.968721	1.346020
2013-01-05	2.386646	0.106212	0.271260	0.171864
2013-01-01	-0.767112	0.659955	1.535587	-0.531216

In [88]:
```
df.describe()
```

Out[88]:

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	0.592193	-0.478656	0.290031	0.184271
std	1.248962	0.904634	0.944576	0.831675
min	-0.767112	-2.025853	-1.246922	-0.698910
25%	-0.492908	-0.654806	0.057537	-0.445733
50%	0.636756	-0.447760	0.236749	-0.008710
75%	1.306367	-0.026962	0.794356	0.798329
max	2.386646	0.659955	1.535587	1.346020

In [89]:
```
df['A']
```

Out[89]:
```
2013-01-01   -0.767112
2013-01-02    1.404921
2013-01-03   -0.744812
2013-01-04    1.010707
2013-01-05    2.386646
2013-01-06    0.262806
Freq: D, Name: A, dtype: float64
```

In [90]:
```
df[0:3]
```

Out[90]:

	A	B	C	D
2013-01-01	-0.767112	0.659955	1.535587	-0.531216
2013-01-02	1.404921	-2.025853	-1.246922	-0.698910
2013-01-03	-0.744812	-0.716729	0.202237	-0.189283

In [91]:
```
df['20130102':'20130104']
```

Out[91]:

	A	B	C	D
2013-01-02	1.404921	-2.025853	-1.246922	-0.698910
2013-01-03	-0.744812	-0.716729	0.202237	-0.189283
2013-01-04	1.010707	-0.426484	0.968721	1.346020

In [93]:
```
df.loc[:,['A','B']]
```

Out[93]:

	A	B
2013-01-01	-0.767112	0.659955
2013-01-02	1.404921	-2.025853
2013-01-03	-0.744812	-0.716729
2013-01-04	1.010707	-0.426484
2013-01-05	2.386646	0.106212
2013-01-06	0.262806	-0.469035

In [94]:
```
df.loc['20130102':'20130104',['A','B']]
```

Out[94]:

	A	B
2013-01-02	1.404921	-2.025853
2013-01-03	-0.744812	-0.716729
2013-01-04	1.010707	-0.426484

In [95]:
```
df.loc['20130102',['A','B']]
```

Out[95]:
```
A    1.404921
B   -2.025853
Name: 2013-01-02 00:00:00, dtype: float64
```

In [96]:
```
df.loc[dates[0],'A']
```

Out[96]:
```
-0.767111915128349
```

In [97]:
```
df.at[dates[0],'A']
```

Out[97]:
```
-0.767111915128349
```

In [98]:
```
df.iloc[3]
```

Out[98]:
```
A    1.010707
B   -0.426484
C    0.968721
D    1.346020
Name: 2013-01-04 00:00:00, dtype: float64
```

In [100]:
```
df.iloc[3:5,0:2]
```

Out[100]:

	A	B
2013-01-04	1.010707	-0.426484
2013-01-05	2.386646	0.106212

In [101]:
```
df.iloc[[1,2,4],[0,2]]
```

Out[101]:

	A	C
2013-01-02	1.404921	-1.246922
2013-01-03	-0.744812	0.202237
2013-01-05	2.386646	0.271260

In [102]:
```
df.iloc[:,1:3]
```

Out[102]:

	B	C
2013-01-01	0.659955	1.535587
2013-01-02	-2.025853	-1.246922
2013-01-03	-0.716729	0.202237
2013-01-04	-0.426484	0.968721
2013-01-05	0.106212	0.271260
2013-01-06	-0.469035	0.009304

In [103]:
```
df.iloc[1,1]
```

Out[103]: -2.0258529161619285

In [104]:
```
df[df.A > 0]
```

Out[104]:

	A	B	C	D
2013-01-02	1.404921	-2.025853	-1.246922	-0.698910
2013-01-04	1.010707	-0.426484	0.968721	1.346020
2013-01-05	2.386646	0.106212	0.271260	0.171864
2013-01-06	0.262806	-0.469035	0.009304	1.007151

In [105]:
```
df[df > 0]
```

Out[105]:

	A	B	C	D
2013-01-01	NaN	0.659955	1.535587	NaN
2013-01-02	1.404921	NaN	NaN	NaN
2013-01-03	NaN	NaN	0.202237	NaN
2013-01-04	1.010707	NaN	0.968721	1.346020
2013-01-05	2.386646	0.106212	0.271260	0.171864
2013-01-06	0.262806	NaN	0.009304	1.007151

In [106]:
```
df2 = df.copy()
```

In [108]:
```
df2['E'] = ['one', 'one','two','three','four','three']
```

In [110]:
```
df2
```

Out[110]:

	A	B	C	D	E
2013-01-01	-0.767112	0.659955	1.535587	-0.531216	one
2013-01-02	1.404921	-2.025853	-1.246922	-0.698910	one
2013-01-03	-0.744812	-0.716729	0.202237	-0.189283	two
2013-01-04	1.010707	-0.426484	0.968721	1.346020	three
2013-01-05	2.386646	0.106212	0.271260	0.171864	four
2013-01-06	0.262806	-0.469035	0.009304	1.007151	three