Table of Contents

8. File I/O

9. Pandas

10. Series

11. Data Frames

12. Viewing Data

1. Introduction

- Python is a language that has become very popular in very little time.
- It is open Source and Free to install.
- It is very easy to learn and implement.
- Python is preffered for handling data analysis.
- Web development can also be done by python.
- Execution can be done directly as compilation for this language is not required.

Language Introduction

- Python is a dynamic, interpreted (bytecode-compiled) language.
- There are no type declarations of variables, parameters, functions, or methods in source code.
- This makes the code short and flexible, and you lose the compile-time type checking of the source code.
- Python tracks the types of all values at runtime and flags code that does not make sense as it runs.

Python source code

- Python source files use the ".py" extension and are called "modules." With a Python module hello.py, the easiest way to run it is with the shell command "python hello.py Alice" which calls the Python interpreter to execute the code in hello.py, passing it the command line argument "Alice".

- Here's a very simple hello.py program (notice that blocks of code are delimited strictly using indentation rather than curly braces):

In [219]:

```
#!/usr/bin/env python

# import modules used here -- sys is a very standard one
import sys

# Gather our code in a main() function
def main():
    print ('Hello there'), sys.argv[1]
    # Command line args are in sys.argv[1], sys.argv[2] ...
    # sys.argv[0] is the script name itself and can be ignored

# Standard boilerplate to call the main() function to begin
# the program.
if   name   == '  main  ':
    main()
```

```
Hello there
```

Running this program from the command line looks like:

- python hello.py Ali
- Hello there Ali
- ./hello.py Alice ## without needing 'python' first (Unix)
- Hello there Alice

Indentation

- One unusual Python feature is that the whitespace indentation of a piece of code affects its meaning.
- A logical block of statements such as the ones that make up a function should all have the same indentation, set in from the indentation of their parent function or "if" or whatever.
- If one of the lines in a group has a different indentation, it is flagged as a syntax error.

- Python's use of whitespace feels a little strange at first, but it's logical and I found I got used to it very quickly.
- Avoid using TABs as they greatly complicate the indentation scheme (not to mention TABs may mean different things on different platforms). Set your editor to insert spaces instead of TABs for Python code.

In [226]:

```
my_age=22

if(my_age<20):
    print(my_age) #this will not print as the if condition is false
print(my_age) #this will print as it is outside if block
```

```
22
```

Comments

- There are two ways to comment in python

  - With a hash '#' for single line commenting
  - With triple quotes for multiple line commenting

In [256]:

```
# This is a single line comment
"""This is a multiline
comment."""

print('Comments  Example')
```

```
Comments  Example
```

2. Data Types

- Python provides several built-in data types. Some of which are explained below:

2.1 Variables

- Unlike other programming languages, Python has no command for declaring a variable.
- A variable is created the moment you first assign a value to it.
- The type of the variable will be decided after assigning the value to it.

Some examples are given below:

In [229]:

```
name = "Ahmed" #string type variable
counter = 100.00 #float type variable
miles = 100 #integer type variable

#print all values
print (name)
print (counter)
print (miles)
```

```
Ahmed
100.0
100
```

To confirm the type of the variable, we can ask python to tell us the variable types:

In [230]:

```
type(counter)
```

Out[230]:

```
float
```

In [231]:

```
type(miles)
```

Out[231]:

```
int
```

In [232]:

```
type(name)
```

Out[232]:

```
str
```

2.2 Numbers

- It is a data type that stores numeric values.
- These are immutable data types which means changing the value of 'Number data type' will result in a newly allocated object.

Some examples are given below

In [320]:

```
2 #integer
```

Out[320]:

2

In [321]:

```
2+3 #integer
```

Out[321]:

5

In [322]:

```
2.3+5.5 #float
```

Out[322]:

7.8

In [323]:

```
var1 = 1
var2 = 10

print(var1, var2)
```

1 10

You can delete a single object or multiple objects by using the del statement:

In [324]:

```
del var1, var2
```

In [325]:

```
2**2 #exponent
```

Out[325]:

4

In [326]:

```
5/2 #divison
```

Out[326]:

```
2.5
```

In [327]:

```
5//2 #rounded down to next integer
```

Out[327]:

```
2
```

In [328]:

```
print(float(10)) #coverting any number to float
print(int(3.33)) #converting any number to integer
print(int('807') + 1) #converting a string into an integer
```

```
10.0
3
808
```

2.3 Strings

- Strings are amongst the most popular types in Python.
- We can create them simply by enclosing characters in quotes.
- Python treats single quotes the same as double quotes.

Creating strings is as simple as assigning a value to a variable:

In [233]:

```
name = 'Ali'
university = "COMSATS"

print(name,university)
```

```
Ali COMSATS
```

2.3.1 Access values in String

- Python doesn't support Character data type. These are treated as Strings of length one.
- Square brackets are used to access substrings.
- We can access a specific character or range of characters from string.

It would be cleared by using following examples.

In [42]:

```
stringVariable = 'Hello World'

print("stringVariable[0]:",  stringVariable[0])
print("stringVariable[1:5]:",  stringVariable[1:5])
```

```
stringVariable[0]:  H
stringVariable[1:5]:  ello
```

2.3.2 Updating String

- You can "update" an existing string by (re)assigning a variable to another string or concatenating another string with it.
- The new value can be related to its previous value or to a completely different string altogether.
- But the new value must have the same data type that means it should also be a string.

In [237]:

```
randomString = "old string"
print(randomString)

randomString = "updated string"
print(randomString)
```

```
old string
updated string
```

In [239]:

```
# '+' operator will be used for updating with concatenation as follows:
stringVariable = "Hello World"
print(stringVariable)

stringVariable = stringVariable[:6] + "Python"
print("Updated string: ", stringVariable)
```

```
Hello World
Updated string:  Hello Python
```

2.3.3 Delete String

- To delete the value of the string, use the 'del' command and delete the variable containing the string.

For Example:

In [24]:

```
stringVariable = "Hello World"
print (stringVariable)
del stringVariable
print (stringVariable) #this will give an error as the variable is deleted above.
```

```
Hello World
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call la
st)
<ipython-input-24-b0e35d2fee0c> in <module>()
      2 print (stringVariable)
      3 del stringVariable
----> 4 print (stringVariable) #this will give an error as the variable
 is deleted above.

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

2.3.4 String special operators

Some special operators are used in Python for mulitple purposes. These are described below with examples:

- **'+' operator is used for concatenation.**

In [25]:

```
variable = "Hello"
print(variable+"Python")
```

```
HelloPython
```

- **'*' operator is used for Repetition.**

In [28]:

```
variable = "Hello "
print(variable*3)
```

```
Hello Hello Hello
```

- **'[]' gives the character of string at given index.**

In [29]:

```
variable = "Hello"
print(variable[1])
```

```
e
```

- **'[:]' is used to get a range of characters from string.**

In [43]:

```
variable = "Hello"
print(variable[1:3])
```

```
el
```

- **'in' returns true if given character exists in string, otherwise false.**

In [31]:

```
variable = "Hello"
print('H' in variable)
```

```
True
```

- **'in' returns true if given character doesn't exists in string, otherwise false.**

In [32]:

```
variable = "Hello"
print('G' not in variable)
```

```
True
```

2.3.5 String formatting operator

- One of the coolest feature is string formatting operator within print statement.

See following examples:

In [44]:

```
print ("My name is %s and age is %d kg!" %("Zara",21))
```

```
My name is Zara and age is 21 kg!
```

In [48]:

```
print ("My height is %f feet" %(5.11))
```

```
My height is 5.110000 feet
```

In [55]:

```
print ("%c is the first letter of my name" %("Z"))
```

```
Z is the first letter of my name
```

2.4 Lists

- Python has a great built-in list type named "list".
- List literals are written within square brackets [ ]. It is written as a list of comma-seperated values within square brackets.
- Lists work similarly to strings -- use the len() function and square brackets [ ] to access data, with the first element at index 0.
- Different types of data can be stored in a List.
- Elements of a list can be changed.

2.4.1 Why, When and Where to use Lists?

- A list can be used to store and access data efficently.
- Lists can be used when you just have a list of data without keys and might change data in the future as the elements are mutabale.
- A list keeps order, so when you care about order, you **MUST** use a list.
- Lists can be used as stacks, queues, matrix etc.

In [59]:

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptu
ne']
print(planets)
```

```
['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Ne
ptune']
```

In [58]:

```
primes = [2, 3, 5, 7]
print(primes)
```

```
[2, 3, 5, 7]
```

In [63]:

```
mixlist = ['Snakes', 5.5,'250']
print(mixlist)
```

```
['Snakes', 5.5, '250']
```

As a stack:

In [258]:

```
stack = [3, 4, 5]
stack.append(6)
stack.append(7)
stack
```

Out[258]:

```
[3, 4, 5, 6, 7]
```

In [259]:

```
stack.pop()
stack.pop()
stack
```

Out[259]:

```
[3, 4, 5]
```

As a Queue:

In [419]:

```
from collections import deque

queue = deque(["Eric", "John", "Michael"])
queue.append("Terry")           # Terry arrives
queue.append("Graham")          # Graham arrives
queue
```

Out[419]:

```
deque(['Eric', 'John', 'Michael', 'Terry', 'Graham'])
```

In [420]:

```
queue.popleft()                 # The first to arrive now leaves
queue.popleft()                 # The second to arrive now leaves
queue                           # Remaining queue in order of arrival
```

Out[420]:

```
deque(['Michael', 'Terry', 'Graham'])
```

As a Matrix:

In [421]:

```
multiple = [ ['1', 'A', 'B'],
         ['2', 'G', 'H'],
         ['3', 'Y', 'Z'],
         # (Comma after the last element is optional)
        ]
# (I could also have written this on one line, but it can get hard to read)
multiple = [['1', 'A', 'B'], ['2', 'G', 'H'], ['3', 'Y', 'Z']]
print(multiple)
```

```
[['1', 'A', 'B'], ['2', 'G', 'H'], ['3', 'Y', 'Z']]
```

2.4.2 Accessing values in Lists

- Square brackets are used to access the values of a list.
- We can access a specific values or range of values from list.

In [69]:

```
favPlanets = ['earth', 'jupiter','mars'];
print ("favPlanets[0]:", favPlanets[0]) #one value
```

```
favPlanets[0]: earth
```

In [71]:

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

```
randomNumbers[1:5]:  [2, 3, 4, 5]
```

In [425]:

```
random_mul = [ ['J', 'Q', 'K'],
         ['2', '2', '2'],
         ['6', 'A', 'K'],
        ]
print("random_mul[2]:",  multiple[2])
```

```
random_mul[2]: ['6', 'A', 'K']
```

In [426]:

```
print("random_mul[0][1]:",  multiple[0][1])
```

```
random_mul[0][1]: Q
```

2.4.3 Updating Lists

- We can update single or mupltiple elements of a list by giving slice on left hand of the assignment operator.
- It can also be updated using append() function.

In [83]:

```
favPlanets = ['earth', 'jupiter','mars'];
favPlanets[2] = "venus"
print(favPlanets)
```

```
['earth', 'jupiter', 'venus']
```

In [84]:

```
favPlanets.append('saturn')
print(favPlanets)
```

```
['earth', 'jupiter', 'venus', 'saturn']
```

In [85]:

```
favPlanets = ['earth', 'jupiter','mars'];
favPlanets[4] = "mercury" #this will give out of range error
print(favPlanets)
```

2.4.4 Delete List element

- del statement is used to remove when you know the position or index of element to be deleted.
- remove() function can be used when you don't know the position of element to be removed.

In [93]:

```
favPlanets = ['earth', 'jupiter','mars']
print(favPlanets)

del favPlanets[2]

print ("After deleting value at index 2 : ",favPlanets)
```

```
['earth', 'jupiter', 'mars']
After deleting value at index 2 :  ['earth', 'jupiter']
```

In [95]:

```
favPlanets = ['earth', 'jupiter','mars']
print(favPlanets)

favPlanets.remove('jupiter')

print ("After removing value at index 1 : ", favPlanets)
```

```
['earth', 'jupiter', 'mars']
After removing value at index 1 :  ['earth', 'mars']
```

In [97]:

```
multiple = [ ['J', 'Q', 'K'],
         ['2', '2', '2'],
         ['6', 'A', 'K'],
        ]
print(multiple)

del multiple[1]
print("After deleting the list at index 1: ", multiple)
```

```
[['J', 'Q', 'K'], ['2', '2', '2'], ['6', 'A', 'K']]
After deleting the list at index 1:  [['J', 'Q', 'K'], ['6', 'A', 'K']]
```

2.5 Dictionaries

- Another useful data type built into Python is the dictionary.
- Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”.
  Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys, which can be any immutable type; strings and numbers can always be keys.
  It is best to think of a dictionary as a set of key: value pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}.
  Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.
  The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten.
- It is an error to extract a value using a non-existent key.

2.5.1 Why, When and Where to use Dictionaries?

> - A dictionary can also be used to store and access data efficently but with keys.
> - Dictionaries can be used when you have data that have keys associated with it, for example list of students with their roll numbers.
> - Checking for membership of a value in a dict, for keys is blazingly fast as compared to list for example. So, if you care about fast searching, you **MUST** use dicts.

Here are some examples using a dictionary:

In [275]:

```
students = {'Hamza':100, "Ali":101, "Ahmed":102}
students
```

```
{'Hamza': 100, 'Ali': 101, 'Ahmed': 102}
{'joker': 4098, 'thor': 4139, 'batman': 4127}
4098
```

In [278]:

```
print('Hamza' in students)

print('Ali' not in students)
```

```
True
False
```

In [276]:

```
hobbies = {'Hamza':"gaming", "Ali":"playing football", "Ahmed":"reading books"}
hobbies
del tel['thor']
tel['flash'] = 4127
print(tel)

print(list(tel))
print(sorted(tel))
```

```
{'Hamza': 'gaming', 'Ali': 'playing football', 'Ahmed': 'reading books'}
{'joker': 4098, 'batman': 4127, 'flash': 4127}
['joker', 'batman', 'flash']
['batman', 'flash', 'joker']
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

In [290]:

```
squares = {x: x**2 for x in (2, 4, 6)}
squares
```

Out[290]:

```
{2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

In [316]:

```
from collections import defaultdict as dict

random = dict(sape=4139, guido=4127, jack=4098)
print(random)
print("jack's value:", random['jack'])
```

```
defaultdict(None, {'sape': 4139, 'guido': 4127, 'jack': 4098})
jack's value: 4098
```

2.5.2 Accessing value in Dictionary

- We can use the familiar square brackets along with the key to obtain its value.

A simple example:

In [281]:

```
dict = {'Name':"Ahmed", 'Department': "CS", 'Batch': 2018}
print (dict['Department'])
```

```
CS
```

In [282]:

```
print(dict['University']) #gives an error for the wrong key
```

```
---------------------------------------------------------------------------
---
KeyError                                  Traceback (most recent call la
st)
<ipython-input-282-5dcd1875bed4> in <module>()
----> 1 print(dict['University']) #gives an error for the wrong key

KeyError: 'University'
```

2.5.3 Updating Dictionary

- We can add a new entry in dictionary.
- We can also update already existing value in dictionary.

For example:

In [283]:

```
dict = {'Name':"Ahmed", 'Depart': "CS", 'Batch': 2014}
print (dict)
dict['Batch'] = "2018"
print (dict)

dict['University'] = "COMSATS"
print(dict)
```

```
{'Name': 'Ahmed', 'Depart': 'CS', 'Batch': 2014}
{'Name': 'Ahmed', 'Depart': 'CS', 'Batch': '2018'}
{'Name': 'Ahmed', 'Depart': 'CS', 'Batch': '2018', 'University': 'COMSAT
S'}
```

2.5.4 Delete Dictionary Element

- We can delete an element individually of a dicitionary and we can also delete all contents of the dictionary.
- del is used to remove one element and clear() function is used to remove an entire dictionary.

For Example:

In [284]:

```
dict = {'Name':"Ahmed", 'Depart': "CS", 'Batch': 2018}
print (dict)

del dict['Batch']
print (dict)
```

```
{'Name': 'Ahmed', 'Depart': 'CS', 'Batch': 2018}
{'Name': 'Ahmed', 'Depart': 'CS'}
```

In [285]:

```
dict = {'Name':"Ahmed", 'Depart': "CS", 'Batch': 2018}
dict.clear()
print (dict)
```

```
{}
```

2.5.5 Looping Techniques

- When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the items() method.

Some Examples are:

In [287]:

```
knights = {'gallahad': 'the pure', 'robin': 'the brave'}
for k, v in knights.items():
     print(k, v)
```

```
gallahad the pure
robin the brave
```

To loop over two or more sequences at the same time, the entries can be paired with the zip() function:

In [289]:

```
questions = ['name', 'quest', 'favorite color']
answers = ['lancelot', 'the holy grail', 'blue']
for q, a in zip(questions, answers):
     print('What is your {0}?  It is {1}.'.format(q, a))
```

```
What is your name?  It is lancelot.
What is your quest?  It is the holy grail.
What is your favorite color?  It is blue.
```

2.6 Tuples

- A tuple is a collection which is ordered and unchangeable.
- In Python tuples are written with round brackets '()'.
- Each of the item is seperated by a comma within the round brackets.
- Empty Tuple is shown as ().
- To write a tuple containing a single value you have to include a comma (,) even though there is only one value. For Exampe: tup = (30,).
- Tuples are heterogeneous data structures (i.e., their entries have different meanings). Multiple data type values.
- Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

2.6.1 Why, When and Where to use Tuples?

- A tuple is used to store and access data efficently.
- As tuples are immutable so they can also be used as 'keys' in dictionaries.
- If you want to maintain order with your data remaining constant, that means no changes can be made to your data throughout, then you **MUST** use tuples.

2.6.2 Accessing values in Tuples

- Square brackets along with the index can be used to obtain its value.
- Inside a bracket, 'index : length' can be used to get a range of values.

Following is a simple example:

In [105]:

```
tup = ('Math','98', 'C programming', '99')
print (tup)

print(tup[1])
print(tup[1:3])
```

```
('Math', '98', 'C programming', '99')
98
('98', 'C programming')
```

2.6.3 Updating Tuples

- Tuples are immutable that means we can't change their values. We cannot update them, once they are declared.
- It is read only.
- Although what we can do is take portions of tuples to make a new one.

In [108]:

```
tup1 = (12, 34.56);
tup2 = ('abc', 'xyz');

# Following action is not valid for tuples
# tup1[0] = 100;

# So let's create a new tuple as follows
tup3 = tup1 + tup2
print (tup3)
```

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

2.6.4 Delete Tuple elements

- Removing individual elements is also not pssible in a Tuple because they are immutable(cannot be changed).
- del statment is used to remove entire Tuple.

In [112]:

```
tup = ('physics', 'chemistry', 1997, 2000)
print (tup)
del tup
print ("After deleting tup : ", tup) #will give an error
```

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

---------------------------------------------------------------------------
---
NameError                                 Traceback (most recent call la
st)
<ipython-input-112-89acaf732cb0> in <module>()
      2 print (tup)
      3 del tup
----> 4 print ("After deleting tup : ", tup) #will give an error

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

2.7 Sets

- Python also includes a data type for sets.
- A set is an unordered collection.
- No duplicate elements are allowed, each element is unique.
- Elements in a set are immutable (they cannot be changed).
- But the set as a whole is mutable. Items can be added or removed from it.
- Each item is written in curly braces.
- Each item is separated by a comma '{,}'.
- Curly braces or the set() function can be used to create sets. Note: to create an empty set you have to use set(), not {}; the latter creates an empty dictionary.
- Data type can be found using type() function.
- A set can be made from a list using set() function.
- Basic uses include membership testing and eliminating duplicate entries.
- Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

2.7.1 Why, When and Where to use Sets?

- A Set is used to store and access data efficently.
- Set requires items to be hashable, so when you are using sets, make sure that the items are hashable.
- Checking for membership of a value in a set is very fast (taking about a constant, short time), while in a list it takes time proportional to the list's length in the average and worst cases.
- So, if you have hashable items, don't care either way about order or duplicates, and want speedy membership checking, **a Set should be used**.

In [330]:

```
#list
list1 = [1,2,3,4,5]
print(list1)
print (type(list1))

my_set = set(list1) #conversion of list to set
print(my_set)
print (type(my_set))
```

```
[1, 2, 3, 4, 5]
<class 'list'>
{1, 2, 3, 4, 5}
<class 'set'>
```

In [331]:

```
# set of integers
my_set = {1,2,3}
print (my_set)
```

```
{1, 2, 3}
```

2.7.2 Accessing Items in Sets

- You cannot access items in a set by referring to an index, since sets are unordered the items has no index.
- But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Examples:

In [333]:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
  print(x)
```

```
apple
banana
cherry
```

In [334]:

```
#Check if "banana" is present in the set:

thisset = {"apple", "banana", "cherry"}
print("banana" in thisset)
```

```
True
```

2.7.3 Updating/Adding values in Sets

- 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.

For Example:

In [332]:

```
# set of mixed data types
my_set = {1,"Hello", 1.2,'C'}

# adding a single value
my_set.add('D')

#adding multiple values
my_set.update(list1)
print (my_set)
```

```
{1, 1.2, 2, 3, 4, 5, 'Hello', 'D', 'C'}
```

2.7.4 Deleting values in Sets

- discard() and remove() are used to to delete a particular item from set.
- discard() will not raise an error if item does not exist in set.
- remove() will raise an error if item doesn't exist in set.

In [337]:

```
this_set = {"apple", "banana", "cherry"}

this_set.remove("banana")
print(this_set)

this_set.discard("banana") #won't raise an error
this_set.remove("banana") #will raise an error
```

```
{'apple', 'cherry'}

---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call la
st)
<ipython-input-337-5bac31cdbabf> in <module>()
      5
      6 this_set.discard("banana") #won't raise an error
----> 7 this_set.remove("banana") #will raise an error

KeyError: 'banana'
```

The del keyword will delete the set completely:

In [340]:

```
thisset = {"apple", "banana", "cherry"}

del thisset

print(thisset)
```

```
---------------------------------------------------------------------------
---
NameError                                 Traceback (most recent call la
st)
<ipython-input-340-b0ed94c1bb6f> in <module>()
      3 del thisset
      4
----> 5 print(thisset)

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

The clear() method empties the set:

In [341]:

```
thisset = {"apple", "banana", "cherry"}

thisset.clear()

print(thisset)
```

```
set()
```

3. Comparison Operators

- Comparison Operators are used to compare values (string, numbers etc.) and return true/false according to situation:
- Python supports the usual logical conditions from mathematics:

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

In [159]:

```
1<3
```

Out[159]:

True

In [160]:

```
15 <= 23
```

Out[160]:

True

In [161]:

```
'Mehvish' == "Zeenat"
```

Out[161]:

False

In [162]:

```
"Mehvish" != "Mehvish"
```

Out[162]:

False

In [163]:

```
(1==1) or (5 > 2)
```

Out[163]:

True

In [342]:

```
(1 < 2) and (2 < 1)
```

Out[342]:

False

In [344]:

```
list1 = [1,2,3]
list2 = [1,2,3]
list3 = ['1','2','3']

print(list1==list2)
print(list2==list3)
```

```
True
False
```

4. If-Else Statements

- If-Else statments are used to execute a block of code depending on conditions.
- 'If' block is executed if the 'if' condition is true otherwise 'else' block is executed.

Examples:

In [172]:

```
if 9 % 2:
    print('Odd')
else:
    print('Even')
```

```
Odd
```

4.1 Elif

- The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition".

In [8]:

```
a = 33
b = 33
if b > a:
  print("b is greater than a")
elif a == b:
  print("a and b are equal")
```

```
a and b are equal
```

In [9]:

```
#code to find the largest number
num1=5
num2=12
num3=6

if num1>num2 and num1>num3:
    print("num1 is the largest")

elif num2>num1 and num2>num3: #else if
    print("num2 is the largest")

else:
    print("num3 is the largest")
```

```
num2 is the largest
```

4.2 Short Hand

- If you have only one statement to execute, you can put it on the same line as the if statement.
- Also works for if ... Else, one for if, and one for else, you can put it all on the same line.

Example:

In [348]:

```
a=10
b=5

#one line if statement
if a > b: print("a is greater than b")

#one line if Else statement
print("A") if a > b else print("B")

#multiple conditions
print("A") if a > b else print("=") if a == b else print("B")
```

```
a is greater than b
A
A
```

---

5. For and While Loop

- Python has for and while loop for iteration.
- These are used when we want to perform a specific task repeatedly untill a specific condition is met.

5.1 For Loops

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- This is less like the for keyword in other programming language, and works more like an iterator method as found in other object-orientated programming languages.
- With the for loop we can execute a set of statements, once for each item in a list, tuple, set etc.

Examples:

In [352]:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

```
apple
banana
cherry
```

In [354]:

```
#iterating over a string
for x in "banana":
  print(x)
```

```
b
a
n
a
n
a
```

5.1.1 The break statement

- With the break statement we can stop the loop before it has looped through all the items:

In [355]:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
  if x == "banana":
    break
```

```
apple
banana
```

5.1.2 The continue statement

- With the continue statement we can stop the current iteration of the loop, and continue with the next:

In [356]:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  if x == "banana":
    continue
  print(x)
```

```
apple
cherry
```

5.1.3 The range() Function

- To loop through a set of code a specified number of times, we can use the range() function,
- The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

In [357]:

```
for x in range(6):
  print(x)
```

```
0
1
2
3
4
5
```

In [359]:

```
for x in range(2, 6): #using a start parameter
  print(x)
```

```
2
3
4
5
```

5.1.4 Else in For Loop

- The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

In [360]:

```
for x in range(6):
  print(x)
else:
  print("Finally  finished!")
```

```
0
1
2
3
4
5
Finally  finished!
```

5.1.5 Nested Loops

- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":

In [361]:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

5.2 The While Loop

- With the while loop we can execute a set of statements as long as a condition is true.

In [349]:

```
# Print i as long as i is less than 6:
i = 1
while i < 6:
  print(i)
  i += 1
```

```
1
2
3
4
5
```

5.2.1 The break statement

- With the break statement we can stop the loop even if the while condition is true:

In [350]:

```
i = 1
while i < 6:
  print(i)
  if i == 3:
    break
  i += 1
```

```
1
2
3
```

5.2.2 The continue statement

- With the continue statement we can stop the current iteration, and continue with the next:

In [351]:

```
i = 0
while i < 6:
  i += 1
  if i == 3:
    continue
  print(i)
```

```
1
2
4
5
6
```

6. Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.

The syntax is:

```
def functionName( parameters ):

    "function_docstring"

    function_suite

    return [expression]
```

6.1 Creating a Function

In Python a function is defined using the def keyword:

In [363]:

```
def my_function():
  print("Hello from a function")
```

6.2 Calling a Function

- To call a function, use the function name followed by parenthesis:

In [364]:

```
def my_function():
  print("Hello from a function")

my_function()
```

```
Hello from a function
```

6.3 Parameters

- Information can be passed to functions as parameter.
- Parameters are specified after the function name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.
- The following example has a function with one parameter (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name:

In [366]:

```
def my_function(fname):
  print(fname + " Khan")

my_function("Hamza")
my_function("Ali")
my_function("Abdullah")
```

```
Hamza Khan
Ali Khan
Abdullah Khan
```

7. Lambda Functions

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

The next examples show the difference between a normal function definition ("f") and a lambda function ("g"):

In [199]:

```
#Normal function
def f (x):
    return x**2
print (f(8))
```

64

In [200]:

```
#Lambda Function
#lambda expressions
times3 = lambda var: var*3
times3(10)
#lambda expressions: another way to write a function in line
```

Out[200]:

30

In [367]:

```
#A lambda function that multiplies argument a with argument b and print the result:

x = lambda a, b : a * b
print(x(5, 6))
```

30

7.1 Why Use Lambda Functions?

- The power of lambda is better shown when you use them as an anonymous function inside another function.
- Say you have a function definition that takes one argument, and that argument will be multiplied with an unknown number:

In [368]:

```
#Use this function definition to make a function that always doubles the number you s
end in:

def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)

print(mydoubler(11))
```

22

In [369]:

```
#Use this function definition to make a function that always doubles and triples the
 number you send in:

def myfunc(n):
  return lambda a : a * n

mydoubler = myfunc(2)
mytripler = myfunc(3)

print(mydoubler(11))
print(mytripler(11))
```

```
22
33
```

7.2 Map()

- Map() function is used with two arguments. Just like: r = map(func, seq)
- The first argument func is the name of a function and the second a sequence (e.g. a list).
- seq.map() applies the function func to all the elements of the sequence seq. It returns a new list with the elements changed by func.

In [214]:

```
sentence = 'It is raining cats and dogs'
words = sentence.split()
print(words)

lengths = map(lambda word: len(word), words)
list(lengths)
```

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

Out[214]:

```
[2, 2, 7, 4, 3, 4]
```

7.3 Filter()

- The function filter(function, list) offers an elegant way to filter out all the elements of a list.
- The function filter(f,l) needs a function f as its first argument. f returns a Boolean value, i.e. either True or False.
- This function will be applied to every element of the list l.
- Only if f returns True will the element of the list be included in the result list.

In [217]:

```
fib = [0,1,1,2,3,5,8,13,21,34,55]
result1 = filter(lambda x: x % 2, fib) #will return only oddd numbers
list(result1)
```

Out[217]:

```
[1, 1, 3, 5, 13, 21, 55]
```

In [218]:

```
fib = [0,1,1,2,3,5,8,13,21,34,55]
result2 = filter(lambda x: x % 2 == 0, fib) #will return only even numbers
list (result2)
```

Out[218]:

```
[0, 2, 8, 34]
```

8. File Input/Output

- In this section, we'll cover all basic Input/Output functions(methods).

8.1 Printing to the Screen

- The simplest way to produce output is using the print statement where you can pass zero or more expressions separated by commas.
- This function converts the expressions you pass into a string and writes the result to standard output as follows:

In [3]:

```
print ("Python is really a great language,", "isn't it?")
```

```
Python is really a great language, isn't it?
```

8.2 Reading input from Keyboard

- For reading input from keyboard, input() method is used.
- It reads only one line from standard input and returns it as a string.

In [4]:

```
print('Enter your name:')
x = input()
print('Hello, ' + x)
```

```
Enter your name:
xyz
Hello, xyz
```

In [5]:

```
x = input('Enter your name:')
print('Hi, ' + x)
```

```
Enter your name:Asif
Hi, Asif
```

8.3 I/O from or to Text File

- 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 [227]:

```
# Open a file to read
file = open("file.txt", "r+")
str = file.read() #to read specific content from start you can use read(12). It will
 read 12 characters from the start of file
print (str)
# Close the opened file
file.close()
```

```
This is the content that is written inside the file!
```

In [231]:

```
# Open a file to append and read
file = open("file.txt", "a+")
file.write(" This is text that is appended in the file")
file.close()

file = open("file.txt", "r+")
string = file.read()
print(string)
fileOpen.close()
```

```
This is the content that is written inside the file! This is text that i
s appended in the file
```

8.4 File Position

- tell() method tells the currenct position within the file.
- seek() method changes the current file location.

In [6]:

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

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

# Reposition pointer at the beginning once again
position = fo.seek(0, 0);
str = fo.read(10);
print ("Again read String is : \n", str)
# Close opend file
fo.close()
```

```
Read String is :
 This is th
Current file position :
 10
Again read String is :
 This is th
```

In [3]:

```
import os
# rename a file
os.rename("file.txt",  "newfile.txt")
```

In [5]:

```
#remove file
os.remove("newfile.txt")
```

9. Pandas

9.1 Introduction

- Pandas is an open-source Python Library providing high-performance data manipulation 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.
- It had very little contribution towards data analysis. Pandas solved this problem.
- Using Pandas, we can accomplish five typical steps in the processing and analysis of data, regardless of the origin of data — **load, prepare, manipulate, model, and analyze.**
- Python with Pandas is used in a wide range of fields including academic and commercial domains including finance, economics, Statistics, analytics, etc.

Key Features of Pandas

- Fast and efficient DataFrame object with default and customized indexing.
- Tools for loading data into in-memory data objects from different file formats.
- Data alignment and integrated handling of missing data.
- Reshaping and pivoting of date sets.
- Label-based slicing, indexing and subsetting of large data sets.
- Columns from a data structure can be deleted or inserted.
- Group by data for aggregation and transformations.
- High performance merging and joining of data.
- Time Series functionality.

9.2 Environment Setup

- Standard Python distribution doesn't come bundled with Pandas module. A lightweight alternative is to install NumPy using popular Python package installer, **pip**.
- *'pip install pandas'*

If you install Anaconda Python package, Pandas will be installed by default with the following:

Windows

- **Anaconda** (from https://www.continuum.io (https://www.continuum.io)) is a free Python distribution for SciPy stack. It is also available for Linux and Mac.
- **Canopy** (https://www.enthought.com/products/canopy/ (https://www.enthought.com/products/canopy/)) is available as free as well as commercial distribution with full SciPy stack for Windows, Linux and Mac.
- **Python** (x,y) is a free Python distribution with SciPy stack and Spyder IDE for Windows OS. (Downloadable from http://python-xy.github.io/ (http://python-xy.github.io/))

Linux

- Package managers of respective Linux distributions are used to install one or more packages in SciPy stack.

For Ubuntu Users

- sudo apt-get install python-numpy python-scipy python-matplotlibipythonipythonnotebook
- python-pandas python-sympy python-nose

For Fedora Users

- sudo yum install numpyscipy python-matplotlibipython python-pandas sympy
- python-nose atlas-devel

To use pandas we usually import it as:

In [378]:

```
import pandas as pd
```

9.3 Pandas Data Structures

- Pandas deals with the following three data structures:

  - Series
  - DataFrame
  - Panel

- These data structures are built on top of **Numpy array**, which means they are fast.

Now the question that arises here is, what is Numpy array?

9.3.1 Python Numpy Array

- NumPy is, just like SciPy, Scikit-Learn, Pandas, etc. one of the packages that you just can't miss when you're learning data science.
- Mainly because this library provides you with an array data structure that holds some benefits over Python lists, such as: being **more compact, faster access in reading and writing items, being more convenient and more efficient**.
- As the name kind of gives away, a NumPy array is a central data structure of the **numpy library**.
- The library's name is actually short for "Numeric Python" or "Numerical Python".

- This already gives an idea of what you're dealing with, right?
- In other words, NumPy is a Python library that is the core library for scientific computing in Python.
- It contains a collection of tools and techniques that can be used to solve on a computer mathematical models of problems in Science and Engineering.
- One of these tools is a high-performance multidimensional array object that is a powerful data structure for efficient computation of arrays and matrices.
- To work with these arrays, there's a huge amount of high-level mathematical functions operate on these matrices and arrays.

Why, When and Where to use Numpy Arrays?

- First consider dictionaries / lists. If these allow you to do all data operations that you need, then all is fine. If not, start considering numpy arrays.
- Some typical reasons for moving to numpy arrays are:
  - Your data is 2-dimensional (or higher). Although nested dictionaries/lists can be used to represent multi-dimensional data, in most situations numpy arrays will be more efficient.
  - You have to perform a bunch of numerical calculations. Numpy will give a significant speed-up in this case. Furthermore numpy arrays come bundled with a large amount of mathematical functions.

Making a numpy Array

- To make a numpy array, **import numpy as np** and just use the **np.array()** function.
- All you need to do is pass a list to it and optionally, you can also specify the data type of the data.

In [376]:

```
import numpy as np

# Make the array `my_array`
my_array = np.array([[1,2,3,4], [5,6,7,8]], dtype=np.int64) #dtype is explained below

# Print `my_array`
print(my_array)
```

```
[[1 2 3 4]
 [5 6 7 8]]
```

Data type objects (dtype)

- A data type object (an instance of **numpy.dtype** class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted. It describes the following aspects of the data:
- Type of the data (integer, float, Python object, etc.)
- Size of the data (how many bytes is in e.g. the integer)
- Byte order of the data (little-endian or big-endian)
- If the data type is structured, an aggregate of other data types, (e.g., describing an array item consisting of an integer and a float),
    1. what are the names of the “fields” of the structure, by which they can be accessed,
    2. what is the data-type of each field, and
    3. which part of the memory block each field takes.
- If the data type is a sub-array, what is its shape and data type.

Some more examples:

In [418]:

```
# Create an array of zeros
print(np.zeros((1,3,2),dtype=np.int16),"\n")

# Create an empty array
print(np.empty((3,2)),"\n")

# Create a full array
print(np.full((2,3),7),"\n")

# Create an array of evenly-spaced values
print(np.arange(10,25,5),"\n")
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]

[[0. 0.]
 [0. 0.]
 [0. 0.]]

[[7 7 7]
 [7 7 7]]

[10 15 20]
```

9.3.2 Dimension & Description

- The best way to think of these data structures is that the higher dimensional data structure is a container of its lower dimensional data structure.
- For example, DataFrame is a container of Series, Panel is a container of DataFrame.

Data Structure	Dimensions	Description
Series	1	1D labeled homogeneous array, size-immutable.
Data Frames	2	General 2D labeled, size-mutable tabular structure
		with potentially heterogeneously typed columns.
Panel	3	General 3D labeled, size-mutable array.

- Building and handling two or more dimensional arrays is a tedious task, burden is placed on the user to consider the orientation of the data set when writing functions.
- But using Pandas data structures, the mental effort of the user is reduced.
- For example, with tabular data (DataFrame) it is more semantically helpful to think of the index (the rows) and the columns rather than axis 0 and axis 1.

Mutability

- All Pandas data structures are value mutable (can be changed) and except Series all are size mutable. Series is size immutable.
- **Note** − DataFrame is widely used and one of the most important data structures. Panel is used much less.

9.2.1 Why, When and Where to use Series and Dataframes?

- Then there are also some reasons for going beyond numpy arrays and to the more-complex but also more-powerful pandas series/dataframes:
  - You have to merge multiple data sets with each other, or do reshaping/reordering of your data.
  - You have to import data from or export data to a specific file format like Excel, HDF5 or SQL.
  - Pandas comes with convenient import/export functions for that.

Before we go further to explain these data structures we need to explain one more concept that we is going to be used:

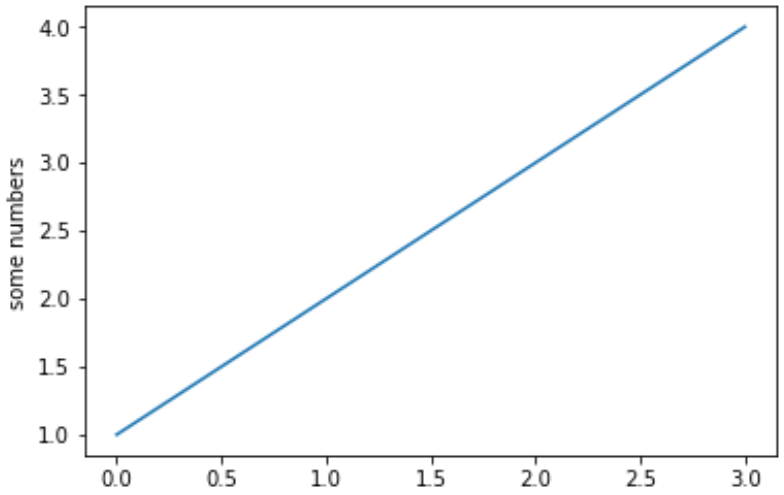
9.2.2 Pyplot

- **matplotlib.pyplot** is a collection of command style functions that make matplotlib work like MATLAB.
- Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.
- In matplotlib.pyplot various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area.
- The plotting functions are directed to the current axes (please note that “axes” here and in most places in the documentation refers to the axes part of a figure and not the strict mathematical term for more than one axis).

Example:

In [430]:

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('some  numbers')
plt.show()
```



You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are [0,1,2,3].

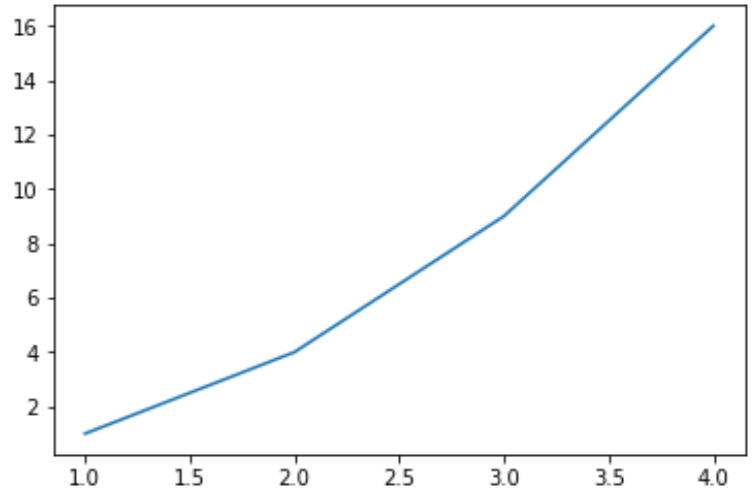
plot() is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

In [433]:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

Out[433]:

```
[<matplotlib.lines.Line2D at 0x2d9a47bf128>]
```

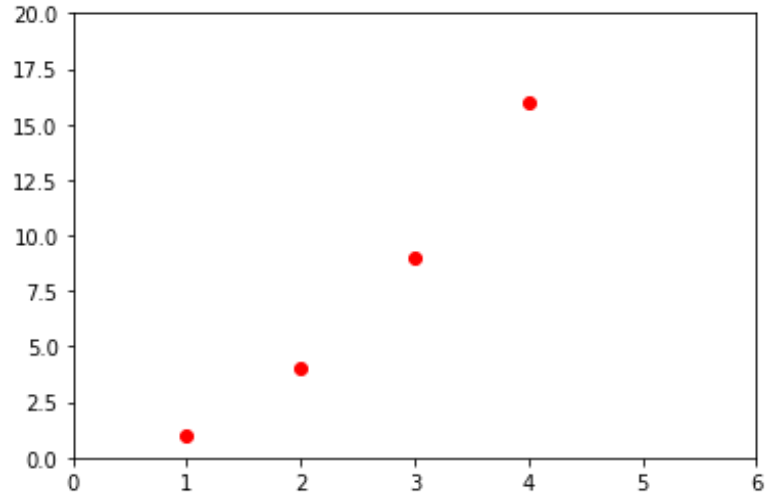


For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is ‘b-‘, which is a solid blue line.

For example, to plot the above with red circles, you would issue:

In [441]:

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```



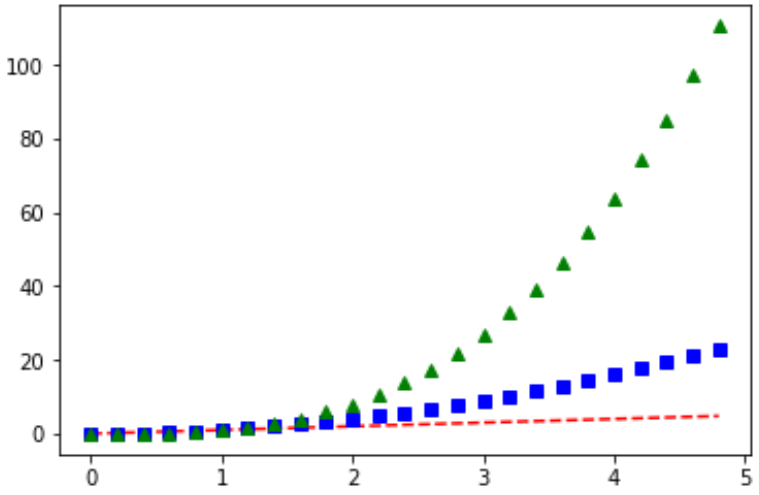
If matplotlib were limited to working with lists, it would be fairly useless for numeric processing. Generally, you will use **numpy arrays**. In fact, all sequences are converted to numpy arrays internally.

The example below illustrates plotting several lines with different format styles in one command using arrays:

In [444]:

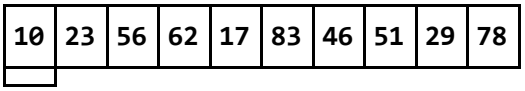
```
import numpy as np
import matplotlib.pyplot as plt

#evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)
print(t)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

```
[0.  0.2 0.4 0.6 0.8 1.  1.2 1.4 1.6 1.8 2.  2.2 2.4 2.6 2.8 3.  3.2 3.4
 3.6 3.8 4.  4.2 4.4 4.6 4.8]
```



10. Series

Series is a one-dimensional array like structure with homogeneous data. For example, the following series is a collection of integers 10, 23, 56, …



- A series is very similar to NumPy array.
- Series is 1-D array labeled array capable of holding any type of data (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
- The axis labels are collectively referred to as the index.
- Following function is used to create a series: **s = pd.Series(data,index = index)**

Simple Examples:

In [464]:

```
import pandas as pd
x = pd.Series([6,3,4,6])
x
```

Out[464]:

```
0    6
1    3
2    4
3    6
dtype: int64
```

In [467]:

```
y = pd.Series(['A','B','C','D'])
y
```

Out[467]:

```
0    A
1    B
2    C
3    D
dtype: object
```

In [468]:

```
z = pd.Series(['A',1,'B',2])
```

Out[468]:

```
0    A
1    1
2    B
3    2
dtype: object
```

- In above function, data can be many different things:
  - An ndarray
  - A scalar value (For exmple: 3)
  - A python dict
- The passed index is a list of axis labels. So, this seperates 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 [447]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

"""following fucntion is called from pandas 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
```

Out[447]:

```
a    0.313762
b   -0.233705
c   -0.936880
d   -0.814797
e   -0.748546
dtype: float64
```

In [448]:

```
"""Following function will print the index and its datatype"""
s.index
```

Out[448]:

```
Index(['a', 'b', 'c', 'd', 'e'], dtype='object')
```

In [449]:

```
"""If we don't assign the index then it will be of length having values [0.....len(d
ata-1)]"""
pd.Series(np.random.randn(5))
```

Out[449]:

```
0    0.451951
1    0.456103
2   -0.845164
3   -0.883012
4   -0.560818
dtype: float64
```

In [450]:

```
pd.Series([1,'abc',3,'def',5])
```

Out[450]:

```
0      1
1    abc
2      3
3    def
4      5
dtype: object
```

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 [451]:

```
""" 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)
```

Out[451]:

```
a    0.0
b    1.0
c    2.0
dtype: float64
```

In [452]:

```
""" In following example, index are given, so the values in data corresponding in the
 index will be pulled out"""
pd.Series(d, index=['b', 'c', 'd', 'a'])
```

Out[452]:

```
b    1.0
c    2.0
d    NaN
a    0.0
dtype: float64
```

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 [453]:

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

Out[453]:

```
a    5.0
b    5.0
c    5.0
d    5.0
e    5.0
dtype: float64
```

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 [454]:

```
#we can access a value just like ndarray

#access single value
s[0] #s(a series of 5 random numbers) is already created above
```

Out[454]:

```
0.31376182794771545
```

In [455]:

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

Out[455]:

```
a    0.313762
b   -0.233705
c   -0.936880
d   -0.814797
e   -0.748546
dtype: float64
```

In [456]:

```
""" Following example will return a range of values in series whose value is greater
 than the median of series 's'"""
s[s > s.median()]
```

Out[456]:

```
a    0.313762
b   -0.233705
dtype: float64
```

In [457]:

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

Out[457]:

```
e   -0.748546
d   -0.814797
b   -0.233705
dtype: float64
```

In [458]:

```
""" Following example returns the exponent values. just like e^a (here a is index and
 its respective data is placed here)"""
np.exp(s)
```

Out[458]:

```
a    1.368564
b    0.791595
c    0.391848
d    0.442729
e    0.473054
dtype: float64
```

In [459]:

```
"""Following example will get the data of given index"""
s['a']
```

Out[459]:

```
0.31376182794771545
```

In [460]:

```
"""Following example will update the data of the given index"""
s['e'] = 12.
s #before updating: -0.433903 and #after updating: 12.000000
```

Out[460]:

```
a     0.313762
b    -0.233705
c    -0.936880
d    -0.814797
e    12.000000
dtype: float64
```

In [461]:

```
""" Following will return true if 'e' is in the values of index otherwise false"""
'e' in s
```

Out[461]:

True

In [462]:

```
"""If a label is not contained and you are trying to access its data, an exception is
 raised: """
s['f'] # This will create error
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call la
st)
D:\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_value
(self, series, key)
   3108             try:
-> 3109                 return libindex.get_value_box(s, key)
   3110             except IndexError:

pandas\_libs\index.pyx in pandas._libs.index.get_value_box()

pandas\_libs\index.pyx in pandas._libs.index.get_value_box()

TypeError: 'str' object cannot be interpreted as an integer

During handling of the above exception, another exception occurred:

KeyError                                  Traceback (most recent call la
st)
<ipython-input-462-f57a1489b9f3> in <module>()
      1 """If a label is not contained and you are trying to access its
 data, an exception is raised: """
----> 2 s['f'] # This will create error

D:\Anaconda3\lib\site-packages\pandas\core\series.py in __getitem__(sel
f, key)
    764         key = com._apply_if_callable(key, self)
    765         try:
--> 766             result = self.index.get_value(self, key)
    767
    768             if not is_scalar(result):



D:\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_value
(self, series, key)
   3115                     raise InvalidIndexError(key)
   3116                 else:
-> 3117                     raise e1
   3118             except Exception:  # pragma: no cover
   3119                 raise e1
```

```
D:\Anaconda3\lib\site-packages\pandas\core\indexes\base.py in get_value
(self, series, key)
   3101         try:
   3102             return self._engine.get_value(s, k,
-> 3103                                           tz=getattr(series.dtyp
e, 'tz', None))
   3104         except KeyError as e1:
   3105             if len(self) > 0 and self.inferred_type in ['intege
r', 'boolean']:

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_value()

pandas\_libs\index.pyx in pandas._libs.index.IndexEngine.get_loc()

pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.PyObje
ctHashTable.get_item()


pandas\_libs\hashtable_class_helper.pxi in pandas._libs.hashtable.PyObje
ctHashTable.get_item()

KeyError: 'f'
```

In [463]:

```
"""Using the get method, a missing label will return None or specified default"""
s.get('f') #it will return none
s.get('f', np.nan) #it will return default value
```

Out[463]:

```
nan
```

10.5 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 [32]:

```
"""following will add the data of respective values of indexes. For example, in given
 output, it is calculated 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[32]:

```
a    -0.804187
b    -1.579531
c    -0.679385
d     3.870302
e    24.000000
dtype: float64
```

In [33]:

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

s * 2
```

Out[33]:

```
a    -0.804187
b    -1.579531
c    -0.679385
d     3.870302
e    24.000000
dtype: float64
```

In [34]:

```
s = pd.Series(np.random.randn(5), name='something')
s
```

Out[34]:

```
0   -0.940375
1    1.112877
2   -0.746168
3   -0.416899
4    1.321165
Name: something, dtype: float64
```

In [35]:

```
s.name #print the name attribute of series
```

Out[35]:

```
'something'
```

In [48]:

```
#rename the series name attribute and assign to s2 object. Note that s and s2 refer t
o different objects.

s2 = s.rename('different')
s2.name
```

Out[48]:

```
'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 [61]:

```
""" 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 indexes of a series. As coloum labels ar
e not given so it will be sorted list of dict keys"""

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

Out[61]:

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

In [62]:

```
""" a data frame will be constructed for given row labels"""
pd.DataFrame(d, index=['d', 'b', 'a'])
```

Out[62]:

	one	two
d	NaN	4.0
b	2.0	2.0
a	1.0	1.0

In [63]:

```
"""following example shows a data frame when we give coloumn labels"""
pd.DataFrame(d, index=['d', 'b', 'a'], columns=['two', 'three'])
```

Out[63]:

	two	three
d	4.0	NaN
b	2.0	NaN
a	1.0	NaN

In [64]:

```
df.columns
```

Out[64]:

```
Index(['one', 'two'], 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 [66]:

```
"""following example shows that ndarray has same length"""
d = {
'one' : [1., 2., 3., 4.],
'two' : [4., 3., 2., 1.]
}

"""row labels are not given so  the result will be range(n), where n is the array len
gth"""
pd.DataFrame(d)
```

Out[66]:

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

In [70]:

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

Out[70]:

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

11.3 From a list of dicts

We created a data frame using just one dict above and now we will create the data frame using multiple dicts.

In [71]:

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

Out[71]:

	a	b	c
0	1	2	NaN
1	5	10	20.0

In [72]:

```
"""passing list of dicts as data and indexes (row labels)"""
pd.DataFrame(data2, index=['first', 'second'])
```

Out[72]:

	a	b	c
first	1	2	NaN
second	5	10	20.0

In [73]:

```
"""passing list of dicts as data and columns (columns labels)"""
pd.DataFrame(data2, columns=['a', 'b'])
```

Out[73]:

	a	b
0	1	2
1	5	10

11.4 From a dict of tuples

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

In [74]:

```
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}})

#"NaN shows missing data
```

Out[74]:

		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

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 [75]:

```
data = np.zeros((2,), dtype=[('A', 'i4'),('B', 'f4'),('C', 'a10')])
data
```

Out[75]:

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

In [104]:

```
pd.DataFrame.from_records(data,  index='C')
```

Out[104]:

	A	B
C		
b"	0	0.0
b"	0	0.0

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 [110]:

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

Out[110]:

	A	B
0	1	4
1	2	5
2	3	6

If you pass orient='index', the keys will be the row labels. But in this case you must also pass the desired column names:

In [111]:

```
pd.DataFrame.from_items([('A', [1, 2, 3]), ('B', [4, 5, 6])],
orient='index', columns=['one', 'two', 'three'])
```

Out[111]:

	one	two	three
A	1	2	3
B	4	5	6

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 [113]:

```
df['one'] #it is displaying data under coloumn 'one'
```

Out[113]:

```
a    1.0
b    2.0
c    3.0
d    NaN
Name: one, dtype: float64
```

In [114]:

```
df['three'] = df['one'] * df['two'] #assigning values to a column named 'three' after
 calculation
```

In [115]:

```
df['flag'] = df['one'] > 2 #check if value at column 'one' is > 2 then assign True ot
herwise false
```

In [116]:

```
df #print a complete data frame
```

Out[116]:

	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 [117]:

```
del df['two'] #delete a coloumn 'two' from data frame
```

In [118]:

```
three = df.pop('three') #pop a complete coloumn 'three' from dataframe
```

In [119]:

```
df
```

Out[119]:

	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 populated to fill the column:

In [120]:

```
df['foo'] = 'bar' #a coloumn 'foo' will be populated with 'bar'
```

In [121]:

```
df
```

Out[121]:

	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 [122]:

```
"""following example will take values from coloumn one untill given range and will po
pulate the new coloumn"""
df['one_trunc'] = df['one'][:2]
```

In [123]:

```
df
```

Out[123]:

	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 the column at a particular location:

In [128]:

```
"""following function has three arguments.
First argument: index where new coloumn will be inserted. Second argument: label or t
itle of a new coloumn
Third argument: the name of the already present column that will populate this new co
lumn"""

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

In [129]:

```
df
```

Out[129]:

	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 [131]:

```
df.loc['b'] #it will return the coloumn labels and values on row label 'b'
```

Out[131]:

```
one              2
bar2             2
flag         False
foo            bar
one_trunc        2
Name: b, dtype: object
```

In [132]:

```
df.iloc[2] #it will return the values of those coloumns that is > than 2
```

Out[132]:

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

11.8 Data alignment and arithmetic

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

In [137]:

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

In [144]:

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

In [147]:

```
df + df2 # add values of respective coloumn labels
```

Out[147]:

	A	B	C	D
0	-1.974816	0.360188	-0.742612	NaN
1	0.541451	-1.771817	0.527759	NaN
2	1.859478	-0.316806	0.550463	NaN
3	3.190466	-2.057855	0.878869	NaN
4	1.093747	0.010837	-0.968320	NaN
5	0.341195	-1.139960	1.045478	NaN
6	1.262262	-0.456398	1.876012	NaN
7	NaN	NaN	NaN	NaN
8	NaN	NaN	NaN	NaN
9	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 [148]:

```
df - df.iloc[0]
```

Out[148]:

	A	B	C	D
0	0.000000	0.000000	0.000000	0.000000
1	0.668340	-0.717429	0.627596	0.531112
2	0.893743	0.847650	-0.051870	-1.609875
3	2.228690	-1.320612	2.128006	0.666029
4	1.276339	0.198371	-1.127995	-1.413660
5	0.129557	-1.905015	1.216310	-0.406491
6	1.111995	-0.738976	0.430870	-0.310515
7	0.635546	-0.300696	1.736101	-2.543695
8	1.215803	-1.803987	1.088200	-0.308149
9	0.392031	-0.956003	2.577666	-2.011896

In [149]:

```
df * 5 + 2
```

Out[149]:

	A	B	C	D
0	-0.472975	4.036144	-3.374018	6.662768
1	2.868723	0.448999	-0.236038	9.318328
2	3.995738	8.274394	-3.633366	-1.386605
3	10.670474	-2.566917	7.266010	9.992914
4	5.908720	5.028000	-9.013994	-0.405533
5	0.174812	-5.488930	2.707532	4.630313
6	5.086998	0.341265	-1.219667	5.110192
7	2.704756	2.532662	5.306486	-6.055707
8	5.606040	-4.983792	2.066984	5.122024
9	1.487179	-0.743870	9.514313	-3.396714

In [150]:

```
1 / df
```

Out[150]:

	A	B	C	D
0	-2.021856	2.455621	-0.930403	1.072324
1	5.755576	-3.223724	-2.236098	0.683216
2	2.505339	0.796890	-0.887569	-1.476405
3	0.576670	-1.094830	0.949485	0.625554
4	1.279191	1.651255	-0.453968	-2.078542
5	-2.739444	-0.667652	7.066816	1.900914
6	1.619696	-3.014345	-1.552956	1.607618
7	7.094657	9.386813	1.512179	-0.620678
8	1.386563	-0.715943	74.644424	1.601525
9	-9.749999	-1.822244	0.665397	-0.926490

In [151]:

```
df ** 4
```

Out[151]:

	A	B	C	D
0	0.059841	0.027501	1.334493e+00	0.756302
1	0.000911	0.009259	3.999784e-02	4.589523
2	0.025382	2.479746	1.611357e+00	0.210464
3	9.042558	0.696005	1.230401e+00	6.530411
4	0.373472	0.134507	2.354504e+01	0.053575
5	0.017756	5.032677	4.009636e-04	0.076586
6	0.145300	0.012112	1.719348e-01	0.149716
7	0.000395	0.000129	1.912435e-01	6.738057
8	0.270547	3.806143	3.221147e-08	0.152008
9	0.000111	0.090693	5.101256e+00	1.357180

In [152]:

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

In [153]:

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

In [154]:

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

Out[154]:

	a	b
0	False	True
1	True	True
2	True	False

In [155]:

```
df1 & df2 #and logical operator
```

Out[155]:

	a	b
0	False	False
1	False	True
2	True	False

In [156]:

```
df1 | df2 #or logical operator
```

Out[156]:

	a	b
0	True	True
1	True	True
2	True	True

In [157]:

```
-df1
```

Out[157]:

	a	b
0	False	True
1	True	False
2	False	False

11.9 Transposing

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

In [159]:

```
# only show the first 5 rows
df[:5].T
```

Out[159]:

	0	1	2	3	4
A	-0.494595	0.173745	0.399148	1.734095	0.781744
B	0.407229	-0.310200	1.254879	-0.913383	0.605600
C	-1.074804	-0.447208	-1.126673	1.053202	-2.202799
D	0.932554	1.463666	-0.677321	1.598583	-0.481107

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

In [169]:

```
dates = pd.date_range('20180101', periods=6)
dates
```

Out[169]:

```
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [170]:

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

In [171]:

```
df
```

Out[171]:

	A	B	C	D
2018-01-01	0.999403	0.579029	-1.429721	1.149797
2018-01-02	-0.815102	1.432211	-0.192513	-1.109863
2018-01-03	-0.646430	1.166314	-1.463591	0.081215
2018-01-04	0.039611	1.640956	1.564192	1.276074
2018-01-05	0.171305	1.961174	-1.255927	-0.044999
2018-01-06	-0.559830	0.686738	-0.330407	0.352707

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

In [174]:

```
df2 = pd.DataFrame({ 'A' : 1.,
'B' : pd.Timestamp('20180102'),
'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 [175]:

```
df2
```

Out[175]:

	A	B	C	D	E	F
0	1.0	2018-01-02	1.0	3	test	foo
1	1.0	2018-01-02	1.0	3	train	foo
2	1.0	2018-01-02	1.0	3	test	foo
3	1.0	2018-01-02	1.0	3	train	foo

In [178]:

```
#Having specific dtypes
df2.dtypes
```

Out[178]:

```
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 [179]:

```
df.head() #display first 5 records
```

Out[179]:

	A	B	C	D
2018-01-01	0.999403	0.579029	-1.429721	1.149797
2018-01-02	-0.815102	1.432211	-0.192513	-1.109863
2018-01-03	-0.646430	1.166314	-1.463591	0.081215
2018-01-04	0.039611	1.640956	1.564192	1.276074
2018-01-05	0.171305	1.961174	-1.255927	-0.044999

In [180]:

```
df.tail(3) #display last 3 records
```

Out[180]:

	A	B	C	D
2018-01-04	0.039611	1.640956	1.564192	1.276074
2018-01-05	0.171305	1.961174	-1.255927	-0.044999
2018-01-06	-0.559830	0.686738	-0.330407	0.352707

In [181]:

```
df.index #display indexes
```

Out[181]:

```
DatetimeIndex(['2018-01-01', '2018-01-02', '2018-01-03', '2018-01-04',
               '2018-01-05', '2018-01-06'],
              dtype='datetime64[ns]', freq='D')
```

In [182]:

```
df.columns #display coloumns
```

Out[182]:

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

In [183]:

```
df.values #print all values
```

Out[183]:

```
array([[ 0.99940281,  0.57902862, -1.42972089,  1.14979713],
       [-0.81510165,  1.43221094, -0.19251282, -1.10986301],
       [-0.64643015,  1.16631417, -1.46359089,  0.08121456],
       [ 0.03961081,  1.64095598,  1.5641923 ,  1.27607435],
       [ 0.17130472,  1.96117436, -1.25592727, -0.04499933],
       [-0.55982974,  0.6867384 , -0.33040705,  0.35270713]])
```

In [184]:

```
#Transposing your data
df.T
```

Out[184]:

	2018-01-01 00:00:00	2018-01-02 00:00:00	2018-01-03 00:00:00	2018-01-04 00:00:00	2018-01-05 00:00:00	2018-01-06 00:00:00
A	0.999403	-0.815102	-0.646430	0.039611	0.171305	-0.559830
B	0.579029	1.432211	1.166314	1.640956	1.961174	0.686738
C	-1.429721	-0.192513	-1.463591	1.564192	-1.255927	-0.330407
D	1.149797	-1.109863	0.081215	1.276074	-0.044999	0.352707

In [187]:

```
#Sorting by an axis
df.sort_index(axis=1,  ascending=False)
```

Out[187]:

	D	C	B	A
2018-01-01	1.149797	-1.429721	0.579029	0.999403
2018-01-02	-1.109863	-0.192513	1.432211	-0.815102
2018-01-03	0.081215	-1.463591	1.166314	-0.646430
2018-01-04	1.276074	1.564192	1.640956	0.039611
2018-01-05	-0.044999	-1.255927	1.961174	0.171305
2018-01-06	0.352707	-0.330407	0.686738	-0.559830

In [188]:

```
#Sorting by values
df.sort_values(by='B')
```

Out[188]:

	A	B	C	D
2018-01-01	0.999403	0.579029	-1.429721	1.149797
2018-01-06	-0.559830	0.686738	-0.330407	0.352707
2018-01-03	-0.646430	1.166314	-1.463591	0.081215
2018-01-02	-0.815102	1.432211	-0.192513	-1.109863
2018-01-04	0.039611	1.640956	1.564192	1.276074
2018-01-05	0.171305	1.961174	-1.255927	-0.044999

In [189]:

```
# Describe shows a quick statistic summary of your data
df.describe()
```

Out[189]:

	A	B	C	D
count	6.000000	6.000000	6.000000	6.000000
mean	-0.135174	1.244404	-0.517994	0.284155
std	0.680553	0.541498	1.161600	0.875300
min	-0.815102	0.579029	-1.463591	-1.109863
25%	-0.624780	0.806632	-1.386272	-0.013446
50%	-0.260109	1.299263	-0.793167	0.216961
75%	0.138381	1.588770	-0.226986	0.950525
max	0.999403	1.961174	1.564192	1.276074

In [190]:

```
#Selecting a single column, which yields a Series, equivalent to df.A
df['A']
```

Out[190]:

```
2018-01-01    0.999403
2018-01-02   -0.815102
2018-01-03   -0.646430
2018-01-04    0.039611
2018-01-05    0.171305
2018-01-06   -0.559830
Freq: D, Name: A, dtype: float64
```

In [191]:

```
#Selecting via [], which slices the rows.
df[0:3]
```

Out[191]:

	A	B	C	D
2018-01-01	0.999403	0.579029	-1.429721	1.149797
2018-01-02	-0.815102	1.432211	-0.192513	-1.109863
2018-01-03	-0.646430	1.166314	-1.463591	0.081215

In [193]:

```
df['20180102':'20180104']
```

Out[193]:

	A	B	C	D
2018-01-02	-0.815102	1.432211	-0.192513	-1.109863
2018-01-03	-0.646430	1.166314	-1.463591	0.081215
2018-01-04	0.039611	1.640956	1.564192	1.276074

In [196]:

```
#Selecting on a multi-axis by label
df.loc[:,['A','B']]
```

Out[196]:

	A	B
2018-01-01	0.999403	0.579029
2018-01-02	-0.815102	1.432211
2018-01-03	-0.646430	1.166314
2018-01-04	0.039611	1.640956
2018-01-05	0.171305	1.961174
2018-01-06	-0.559830	0.686738

In [198]:

```
#Showing label slicing, both endpoints are included
df.loc['20180102':'20180104',['A','B']]
```

Out[198]:

	A	B
2018-01-02	-0.815102	1.432211
2018-01-03	-0.646430	1.166314
2018-01-04	0.039611	1.640956

In [199]:

```
#Reduction in the dimensions of the returned object
df.loc['20180102',['A','B']]
```

Out[199]:

```
A   -0.815102
B    1.432211
Name: 2018-01-02 00:00:00, dtype: float64
```

In [200]:

```
#For getting a scalar value
df.loc[dates[0],'A']
```

Out[200]:

```
0.9994028054674479
```

In [201]:

```
#For getting fast access to a scalar
df.at[dates[0],'A']
```

Out[201]:

```
0.9994028054674479
```

In [204]:

```
#Select via the position of the passed integers
df.iloc[3]
```

Out[204]:

```
A    0.039611
B    1.640956
C    1.564192
D    1.276074
Name: 2018-01-04 00:00:00, dtype: float64
```

In [207]:

```
#By integer slices, acting similar to numpy/python
df.iloc[3:5,0:2]
```

Out[207]:

	A	B
2018-01-04	0.039611	1.640956
2018-01-05	0.171305	1.961174

In [208]:

```
#By lists of integer position locations, similar to the numpy/python style
df.iloc[[1,2,4],[0,2]]
```

Out[208]:

	A	C
2018-01-02	-0.815102	-0.192513
2018-01-03	-0.646430	-1.463591
2018-01-05	0.171305	-1.255927

In [209]:

```
#For slicing rows explicitly
df.iloc[:,1:3]
```

Out[209]:

	B	C
2018-01-01	0.579029	-1.429721
2018-01-02	1.432211	-0.192513
2018-01-03	1.166314	-1.463591
2018-01-04	1.640956	1.564192
2018-01-05	1.961174	-1.255927
2018-01-06	0.686738	-0.330407

In [212]:

```
#For getting a value explicitly
df.iloc[1,1]
```

Out[212]:

```
1.4322109433785732
```

In [213]:

```
#Using a single column’s values to select data.
df[df.A > 0]
```

Out[213]:

	A	B	C	D
2018-01-01	0.999403	0.579029	-1.429721	1.149797
2018-01-04	0.039611	1.640956	1.564192	1.276074
2018-01-05	0.171305	1.961174	-1.255927	-0.044999

In [214]:

```
#Selecting values from a DataFrame where a boolean condition is met.
df[df > 0]
```

Out[214]:

	A	B	C	D
2018-01-01	0.999403	0.579029	NaN	1.149797
2018-01-02	NaN	1.432211	NaN	NaN
2018-01-03	NaN	1.166314	NaN	0.081215
2018-01-04	0.039611	1.640956	1.564192	1.276074
2018-01-05	0.171305	1.961174	NaN	NaN
2018-01-06	NaN	0.686738	NaN	0.352707

In [215]:

```
#Using the isin() method for filtering:
df2 = df.copy()
```

In [216]:

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

In [217]:

```
df2
```

Out[217]:

	A	B	C	D	E
2018-01-01	0.999403	0.579029	-1.429721	1.149797	one
2018-01-02	-0.815102	1.432211	-0.192513	-1.109863	one
2018-01-03	-0.646430	1.166314	-1.463591	0.081215	two
2018-01-04	0.039611	1.640956	1.564192	1.276074	three
2018-01-05	0.171305	1.961174	-1.255927	-0.044999	four
2018-01-06	-0.559830	0.686738	-0.330407	0.352707	three

In [218]:

```
df2[df2['E'].isin(['two','four'])]
```

Out[218]:

	A	B	C	D	E
2018-01-03	-0.646430	1.166314	-1.463591	0.081215	two
2018-01-05	0.171305	1.961174	-1.255927	-0.044999	four