Basics of Python and Numpy

Introduction

- Python is an interpreted, interactive, object-oriented, and high-level programming language.

- Python Features
  - Easy-to-learn
  - Easy-to-read
  - A broad standard library
  - Databases
  - GUI Programming

Introduction

- Python Comments: #

- Help in Python: help(topic)
  - If no argument is given, the interactive help system starts on the interpreter console. If the argument is a string, then the string is looked up as the name of a module, function, class, method, keyword, or documentation topic, and a help page is printed on the console.

Printing in Python

- Syntax: print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)
- print("Hello World") #Hello World
- a=5
- b=2
- print(a) #5
- print(a, b) # 5 2
- print(a) # 5
  print(b) # 2
- print("Value of a =", a)
- print ("Value of b =", b)

Standard Data Types

- Python has five standard data types –
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary

Standard Data Types

- Numbers
  - int
    - All integers in Python3 are represented as long integers. Hence there is no separate number type as long.
    - Integers in Python 3 are of unlimited size.
  - float
  - complex
    - A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

Standard Data Types

- Numbers
  - Examples

int	float	complex
10	0.0	3.14j
100	15.20	45.j
-786	-21.9	9.322e1-36j
0o70	32.3e18	.876j
-0o470	-90.	-.6545+0J
-0x260	-32.54e100	3e1+26J
0x69	70.2E-12	4.53e1-7j

Standard Data Types

- Strings
  - Strings in Python are identified as a contiguous set of characters represented in the quotation marks.
  - Python allows for either pairs of single or double quotes.
  - Subsets of strings can be taken using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the string.
  - The plus (+) sign is the string concatenation operator and the asterisk (*) is the repetition operator.
  - Trying to access elements beyond the length of the string results in an error.

Standard Data Types

- Strings
  - str = 'Hello World!'

  - print (str) # Prints complete string
  - print (str[0]) # Prints first character of the string
  - print (str[2:5]) **# Prints characters starting from 3rd to 5th**
  - print (str[2:]) **# Prints string starting from 3rd character**
  - print (str * 2) # Prints string two times
  - print (str + "TEST") # Prints concatenated string

  - This will produce the following result –

  - Hello World!
  - H
  - llo
  - llo World!
  - Hello World!Hello World!
  - Hello World!TEST

Standard Data Types

- Strings
  - str = 'Hello World!'

  - print (str[-1])
  - print (str[-3:-1])
  - print (str[-12:])

  - This will produce the following result –

  - !
  - ld
  - Hello World!

Standard Data Types

- Strings
  - Python strings **cannot be changed** — they are **immutable**.
  - Therefore, assigning to an indexed position in the string results in an error
  - I.e. str[0] = 'J' results in an error. However, str="welcome" works.

Standard Data Types

- List
  - A list contains items separated by commas and enclosed within square brackets ([]).
  - To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.
  - The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way from -1 at the end.
  - The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.
  - Unlike strings, which are immutable, **lists are a mutable** type, i.e. it is possible to change their content.
  - Trying to access/assign elements beyond the length of the list results in an error.

Standard Data Types

- List
  - list = ['abcd', 786 , 2.23, 'john', 70.2]
  - tinylist = [123, 'john']

  - print (list) # Prints complete list
  - print (list[0]) # Prints first element of the list
  - print (list[1:3]) # Prints elements starting from 2nd till 3rd
  - print (list[2:]) # Prints elements starting from 3rd element
  - print (tinylist * 2) # Prints list two times
  - print (list + tinylist) # Prints concatenated lists

  - This produce the following result –

  - ['abcd', 786, 2.23, 'john', 70.2]
  - abcd
  - [786, 2.23]
  - [2.23, 'john', 70.2]
  - [123, 'john', 123, 'john']
  - ['abcd', 786, 2.23, 'john', 70.2, 123, 'john']

Standard Data Types

- Tuples
  - A tuple is another sequence data type that is similar to the list.
  - A tuple consists of a number of values separated by commas.
  - Unlike lists, however, tuples are enclosed within parentheses.
  - The main differences between lists and tuples are: Lists are enclosed in brackets ([ ]) and their elements and size can be changed, while tuples are enclosed in parentheses (( )) and cannot be updated.
  - Tuples can be thought of as **read-only lists/immutable lists**.

Standard Data Types

- Tuples
  - tuple = ('abcd', 786 , 2.23, 'john', 70.2)
  - tinytuple = (123, 'john')

  - print (tuple) # Prints complete tuple
  - print (tuple[0]) # Prints first element of the tuple
  - print (tuple[1:3]) # Prints elements starting from 2nd till 3rd
  - print (tuple[2:]) **# Prints elements starting from 3rd element**
  - print (tinytuple * 2) # Prints tuple two times
  - print (tuple + tinytuple) # Prints concatenated tuple

  - This produce the following result –

  - ('abcd', 786, 2.23, 'john', 70.2)
  - abcd
  - (786, 2.23)
  - (2.23, 'john', 70.2)
  - (123, 'john', 123, 'john')
  - ('abcd', 786, 2.23, 'john', 70.2, 123, 'john')

Standard Data Types

- Tuples
  - tuple = ('abcd', 786 , 2.23, 'john', 70.2)
  - list = ['abcd', 786 , 2.23, 'john', 70.2]
  - tuple[2] = 1000 # Invalid syntax with tuple
  - list[2] = 1000 # Valid syntax with list

Standard Data Types

- Dictionary
  - Dictionaries consist of key-value pairs.
  - A dictionary key can be almost any Python type, but are usually numbers or strings.
  - Values, on the other hand, can be any arbitrary Python object.
  - Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).
  - Dictionaries have no concept of order among elements.
  - It is incorrect to say that the elements are "out of order"; they are simply unordered.
  - Dictionaries are **mutable**.

Standard Data Types

- Dictionary
  - dict = {}
  - dict['one'] = "This is one"
  - dict[2] = "This is two"
  - tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
  - print (dict['one']) # Prints value for 'one' key
  - print (dict[2]) # Prints value for 2 key
  - print (tinydict) # Prints complete dictionary
  - print (tinydict.keys()) # Prints all the keys
  - print (tinydict.values()) # Prints all the values
  - This produce the following result –
  - This is one
  - This is two
  - {'dept': 'sales', 'code': 6734, 'name': 'john'}
  - ['dept', 'code', 'name']
  - ['sales', 6734, 'john']

Input Statement

- a=input(“Enter a:”)
- a=int(input(“Enter a:”));
- a=eval(input("Enter three values:"))
- a, b, c=eval(input(“Enter a, b, c:”))

Matrices

- a=[
  [1,2,3],
  [4,5,6]
  ]

- a[0], a[1], a[0][0], a[0][2], a[1][2]

Note: Not Recommended as the len will be 2.

Basic Operators

- Types of Operator
  - Arithmetic Operators
  - Comparison (Relational) Operators
  - Assignment Operators
  - Logical Operators
  - Bitwise Operators
  - Membership Operators
  - Identity Operators

Basic Operators

- Arithmetic Operators
  - Assume variable a holds 10 and variable b holds 21, then –

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 31
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -11
* Multiplication	Multiplies values on either side of the operator	a * b = 210
/ Division	Divides left hand operand by right hand operand	b / a = 2.1
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 1
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 21
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity):	9//2 = 4 and 9.0//2.0 = 4.0

Basic Operators

- Comparison Operators
  - Assume variable a holds 10 and variable b holds 20, then-

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 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.

Basic Operators

- Assignment Operators
  - Assume variable a holds 10 and variable b holds 20, then-

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+=	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-=	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*=	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/=	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%=	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**=	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//=	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Basic Operators

- Bitwise Operators
  - Assume a = 60 = 0011 1100 and b = 13 = 0000 1101, then-

Operator	Description	Example
&	Operator copies a bit to the result if it exists in both operands	(a & b) (means 0000 1100)
|	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)
^	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<<	The left operands value is moved left by the number of bits specified by the right operand.	a << = 2 (means 1111 0000)
>>	The left operands value is moved right by the number of bits specified by the right operand.	a >> = 2 (means 0000 1111)

Basic Operators

- Logical Operators
  - Assume a = True (Case Sensitive) and b = False (Case Sensitive), then-

Operator	Description	Example
and	If both the operands are true then condition becomes true.	(a and b) is False.
or	If any of the two operands are non-zero then condition becomes true.	(a or b) is True.
not	Used to reverse the logical state of its operand.	Not(a and b) is True.

Basic Operators

- Membership Operators
  - Python's membership operators test for membership in a sequence, such as strings, lists, or tuples.

  - There are two membership operators as explained below

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here "in" results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here "not in" results in a 1 if x is not a member of sequence y.

Basic Operators

- Identity Operators
  - Identity operators compare the memory locations of two objects.
  - There are two Identity operators explained below:

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here "is" results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here "is not" results in 1 if id(x) is not equal to id(y).

Basic Operators

- Python Operator Precedence

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	Bitwise exclusive `OR' and regular `OR'
<= < > >=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Decision Making

- Simple if
  - if expression:
    statement(s)

```
var1 = 100
if var1:
   print ("1 - Got a true expression value")
   print (var1)

var2 = 0
if var2:
   print ("2 - Got a true expression value")
   print (var2)
print ("Good bye!")
```

Output:

```
1 - Got a true expression value
100
Good bye!
```

Decision Making

- if else
  - if expression:
    statement(s)
    else:
    statement(s)

```
amount=int(input("Enter amount: "))
if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
else:
    discount=amount*0.10
    print ("Discount",discount)

print ("Net payable:",amount-discount)
```

Decision Making

- if else

Output:
Enter amount: 600
Discount 30.0
Net payable: 570.0

Enter amount: 1200
Discount 120.0
Net payable: 1080.0

Decision Making

- elif Statement

```
if expression1:
   statement(s)
elif expression2:
   statement(s)
elif expression3:
   statement(s)
else:
   statement(s)
```

Decision Making

- elif Statement

```
amount=int(input("Enter amount: "))
if amount<1000:
    discount=amount*0.05
    print ("Discount",discount)
elif amount<5000:
    discount=amount*0.10
    print ("Discount",discount)
else:
    discount=amount*0.15
    print ("Discount",discount)

print ("Net payable:",amount-discount)
```

Decision Making

- elif Statement

```
Enter amount: 600
Discount 30.0
Net payable: 570.0

Enter amount: 3000
Discount 300.0
Net payable: 2700.0

Enter amount: 6000
Discount 900.0
Net payable: 5100.0
```

Decision Making

- Nested if

```
if expression1:
   statement(s)
   if expression2:
      statement(s)
   elif expression3:
      statement(s)
   else:
      statement(s)
elif expression4:
   statement(s)
else:
   statement(s)
```

Decision Making

- Nested if

```
num=int(input("enter number"))
if num%2==0:
    if num%3==0:
        print ("Divisible by 3 and 2")
    else:
        print ("divisible by 2 not divisible by 3")
else:
    if num%3==0:
        print ("divisible by 3 not divisible by 2")
    else:
        print  ("not Divisible by 2 not divisible by 3")
```

Loops

- While Loop

```
while expression:
   statement(s)

count = 0
while count < 9:
   print ('The count is:', count)
   count = count + 1

print ("Good bye!")
```

Loops

- While Loop

```
The count is: 0
The count is: 1
The count is: 2
The count is: 3
The count is: 4
The count is: 5
The count is: 6
The count is: 7
The count is: 8
Good bye!
```

Loops

- for Loop

```
for iterating_var in sequence:
   statements(s)

for var in list(range(5)):
  print (var)
```

Output:

```
0
1
2
3
4
```

Loops

- for Loop

```
for letter in 'Python':     # traversal of a string sequence
   print ('Current Letter :', letter)
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : h
Current Letter : o
Current Letter : n
```

Loops

- for Loop

```
fruits = ['banana', 'apple',  'mango']
for fruit in fruits:        # traversal of List sequence
   print ('Current fruit :', fruit)

print ("Good bye!")
```

Output:

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Loops

- for Loop
  - Iterating by Sequence Index

```
fruits = ['banana', 'apple',  'mango']
for index in range(len(fruits)):
   print ('Current fruit :', fruits[index])

print ("Good bye!")
```

Output:

```
Current fruit : banana
Current fruit : apple
Current fruit : mango
Good bye!
```

Loops

- Break Statement

```
for letter in 'Python':
   if letter == 'h':
      break
   print ('Current Letter :', letter)
```

Output:
Current Letter : P
Current Letter : y
Current Letter : t

Loops

- Continue Statement

```
for letter in 'Python':
   if letter == 'h':
      continue
   print ('Current Letter :', letter)
```

Output:

```
Current Letter : P
Current Letter : y
Current Letter : t
Current Letter : o
Current Letter : n
```

Loops

- Using else Statement with Loops
  - Python supports to have an else statement associated with a loop statement
  - If the else statement is used with a for loop, the else block is executed only if for loops terminates normally (and not by encountering break statement).
  - If the else statement is used with a while loop, the else statement is executed when the condition becomes false.

Loops

- Using else Statement with Loops

```
numbers=[11,33,55,39,55,75,37,21,23,41,13]

for num in numbers:
    if num%2==0:
        print ('the list contains an even number')
        break
else:
    print ('the list does not contain even number')
```

Output:
the list does not contain even number

Numbers - Revisited

- Numbers
  - Number Type Conversion
    - Type int(x) to convert x to a plain integer.
    - Type float(x) to convert x to a floating-point number.
    - Type complex(x) to convert x to a complex number with real part x and imaginary part zero.
    - Type complex(x, y) to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions.

Numbers - Revisited

- Numbers
  - Mathematical Functions

Function	Returns (description)
abs(x)	The absolute value of x: the (positive) distance between x and zero.
math.ceil(x)	The ceiling of x: the smallest integer not less than x
math.exp(x)	The exponential of x: e^x
math.floor(x)	The floor of x: the largest integer not greater than x
math.log(x)	The natural logarithm of x, for x> 0
math.log10(x)	The base-10 logarithm of x for x> 0 .

Numbers - Revisited

- Numbers
  - Mathematical Functions

Function	Returns (description)
max(x1, x2,...)	The largest of its arguments: the value closest to positive infinity
min(x1, x2,...)	The smallest of its arguments: the value closest to negative infinity
pow(x, y)	The value of x**y.
round(x [,n])	x rounded to n digits from the decimal point.
math.sqrt(x)	The square root of x for x > 0

Strings - Revisited

- Strings (**Assume str to be a string variable**)

Sr. No.	Methods with Description
1	str.capitalize() Capitalizes first letter of string. Not in Place
2	str.isalnum() Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
3	str.isalpha() Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
4	str.isdigit() Returns true if string has at least 1 character and contains only digits and false otherwise.
5	str.islower() Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
6	str.isspace() Returns true if string contains only whitespace characters and false otherwise.

Strings - Revisited

- Strings

Sr. No.	Methods with Description
7	str.isupper() Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
8	len(str) Returns the length of the string
9	str.lower() Converts all uppercase letters in string to lowercase. Not in Place.
10	max(str) Returns the max alphabetical character from the string str.
11	min(str) Returns the min alphabetical character from the string str.
12	str.upper() Converts lowercase letters in string to uppercase. Not in Place.

Lists - Revisited

- Delete List Elements

```
list = ['physics', 'chemistry', 1997, 2000]
print (list)
del list[2]
print ("After deleting value at index 2 : ", list)
```

Output:

```
['physics', 'chemistry', 1997, 2000]
After deleting value at index 2 :  ['physics', 'chemistry', 2000]
```

Lists - Revisited

- Basic List Operations

Python Expression	Results	Description
len([1, 2, 3])	3	Length
[1, 2, 3] + [4, 5, 6]	[1, 2, 3, 4, 5, 6]	Concatenation
['Hi!'] * 4	['Hi!', 'Hi!', 'Hi!', 'Hi!']	Repetition
3 in [1, 2, 3]	True	Membership
for x in [1,2,3] : print (x,end=' ')	1 2 3	Iteration

Lists - Revisited

- Built in List Functions and Methods (**assume list to be name of the variable**)

Sr.	Function with Description
1	**len(list)** **Gives the total length of the list.**
2	**max(list)** **Returns item from the list with max value.**
3	**min(list)** **Returns item from the list with min value.**
4	**list.copy()** **Returns a copy of the list**

Lists - Revisited

- List Methods

SN	Methods with Description
1	list.append(obj) Appends object obj to list. Returns None.
2	list.count(obj) Returns count of how many times obj occurs in list
3	list.index(obj) Returns the lowest index in list that obj appears
4	list.insert(index, obj) Inserts object obj into list at offset index
5	list.pop() Removes and returns last object or obj from list
6	list.remove(obj) Removes first instance of obj from list
7	list.reverse() Reverses objects of list in place
8	list.sort() Sorts objects of list in place

Python Functions

- Defining a Function

```
def functionname( parameters ):
   "function_docstring"
   function_suite
   return [expression]

def printme( str ):
   "This prints a passed string into this function"
   print (str)
   return
```

Python Functions

- Pass by reference vs value
  - All parameters (arguments) in the Python language are passed by reference.
  - It means if you change what a parameter refers to within a function, the change also reflects back in the calling function.

Python Functions

- Pass by reference vs value

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   print ("Values inside the function before change: ", mylist)
   mylist[2]=50
   print ("Values inside the function after change: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Output:

```
Values inside the function before change:  [10, 20, 30]
Values inside the function after change:  [10, 20, 50]
Values outside the function:  [10, 20, 50]
```

Python Functions

- Pass by reference vs value

```
# Function definition is here
def changeme( mylist ):
   "This changes a passed list into this function"
   mylist = [1,2,3,4]              # This would assign new reference in mylist
   print ("Values inside the function: ", mylist)
   return

# Now you can call changeme function
mylist = [10,20,30]
changeme( mylist )
print ("Values outside the function: ", mylist)
```

Output:
Values inside the function: [1, 2, 3, 4]
Values outside the function: [10, 20, 30]

Python Functions

- Global vs. Local Variables
  - Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

  - This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions.

Python Functions

- Global vs. Local Variables

```
total = 0 # This is a global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
   total = arg1 + arg2; # Here total is local variable.
   print ("Inside the function local total : ", total)
   return

# Now you can call sum function
sum( 10, 20 )
print ("Outside the function global total : ", total )
```

Output:

```
Inside the function local total :  30
Outside the function global total :  0
```

Python Functions

- Global vs. Local Variables

```
total = 0 # This is global variable.
# Function definition is here
def sum( arg1, arg2 ):
   # Add both the parameters and return them."
    global total
    total = arg1 + arg2;
    print ("Inside the function local total : ", total)
    return

# Now you can call sum function
sum( 10, 20 )
print ("Outside the function global total : ", total )
```

Output:
Inside the function local total : 30
Outside the function global total : 30

Note: You can also return multiple values, e.g. return x, y

Miscellaneous

- del var_name
- del var1, var2
- type(5)
- type(5.6)
- type(5+2j)
- type("hello")
- type(['h','e'])
- type(('h','e'))
- Multiple Assignments
  - a = b = c = 1
  - a, b, c = 1, 2, "john"

Numpy

- Numpy (Numeric/Numerical Python)
  - **Numpy is an open-source add-on module that provides common mathematical and numerical routines** as pre-compiled fast functions
  - It provides **basic routines for manipulating large arrays and matrices of numeric data.**
  - import numpy as np
  - C:\\Python34\scripts>pip3.4 list
  - C:\\Python34\scripts>pip3.4 install numpy

Numpy

- np.array
  - Collection of same type of elements
  - One dimensional array

```
>>> a = np.array([1, 4, 5, 8], float)
>>> a
array([ 1.,  4.,  5.,  8.])
>>> type(a)
<type 'numpy.ndarray'>
```

```
>>> a[:2]
array([ 1.,  4.])
>>> a[3]
8.0
>>> a[0] = 5.
>>> a
array([ 5.,  4.,  5.,  8.])
```

Numpy

- np.array
  - Two dimensional array

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a[1,:]
array([ 4.,  5.,  6.])
>>> a[:,2]
array([ 3.,  6.])
>>> a[-1:,-2:]
array([[ 5.,  6.]])
```

```
>>> a.shape
(2, 3)
```

```
>>> a.dtype
dtype('float64')
```

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> len(a)
2
```

Numpy

- np.array
  - Two dimensional array: reshape() & copy()

```
>>> a = np.array(range(10), float)
>>> a
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9.])
>>> a = a.reshape((5, 2))
>>> a
array([[ 0.,  1.],
       [ 2.,  3.],
       [ 4.,  5.],
       [ 6.,  7.],
       [ 8.,  9.]])
>>> a.shape
(5, 2)
```

```
>>> a = np.array([1, 2, 3], float)
>>> b = a
>>> c = a.copy()
>>> a[0] = 0
>>> a
array([0., 2., 3.])
>>> b
array([0., 2., 3.])
>>> c
array([1., 2., 3.])
```

Strange - Shape is a settable property and it is a tuple and you can concatenate the dimension.

Numpy

- np.array
  - Two dimensional array: reshape(), transpose() & flatten()

```
>>> a = np.array(range(6), float).reshape((2, 3))
>>> a
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
>>> a.transpose()
array([[ 0.,  3.],
       [ 1.,  4.],
       [ 2.,  5.]])
```

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> a.flatten()
array([ 1.,  2.,  3.,  4.,  5.,  6.])
```

Numpy

- np.array
  - Two dimensional array: concatenate()

Two or more arrays can be concatenated together using the `concatenate` function with a tuple of the arrays to be joined:

```
>>> a = np.array([1,2], float)
>>> b = np.array([3,4,5,6], float)
>>> c = np.array([7,8,9], float)
>>> np.concatenate((a, b, c))
array([1., 2., 3., 4., 5., 6., 7., 8., 9.])
```

Numpy

- np.array
  - Two dimensional array: concatenate()

If an array has more than one dimension, it is possible to specify the axis along which multiple arrays are concatenated. By default (without specifying the axis), NumPy concatenates along the first dimension:

```
>>> a = np.array([[1, 2], [3, 4]], float)
>>> b = np.array([[5, 6], [7,8]], float)
>>> np.concatenate((a,b))
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=0)
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.],
       [ 7.,  8.]])
>>> np.concatenate((a,b), axis=1)
array([[ 1.,  2.,  5.,  6.],
       [ 3.,  4.,  7.,  8.]])
```

Numpy

- np.array
  - Other ways to create array

The arange function is similar to the range function but returns an array:

```
>>> np.arange(5, dtype=float)
array([ 0.,  1.,  2.,  3.,  4.])
>>> np.arange(1, 6, 2, dtype=int)
array([1, 3, 5])
```

```
>>> np.ones((2,3), dtype=float)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.zeros(7, dtype=int)
array([0, 0, 0, 0, 0, 0, 0])
```

Numpy

- np.array
  - Array mathematics

When standard mathematical operations are used with arrays, they are applied on an element-by-element basis. This means that the arrays should be the same size during addition, subtraction, etc.:

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([6., 4., 9.])
>>> a - b
array([-4., 0., -3.])
>>> a * b
array([5., 4., 18.])
>>> b / a
array([5., 1., 2.])
>>> a % b
array([1., 0., 3.])
>>> b**a
array([5., 4., 216.])
```

For two-dimensional arrays, multiplication remains elementwise and does *not* correspond to matrix multiplication. There are special functions for matrix math that we will cover later.

```
>>> a = np.array([[1,2], [3,4]], float)
>>> b = np.array([[2,0], [1,3]], float)
>>> a * b
array([[2., 0.], [3., 12.]])
```

Numpy

- np.array
  - Array mathematics

Errors are thrown if arrays do not match in size:

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([4,5], float)
>>> a + b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

Numpy

- np.array
  - Array mathematics - Broadcasting

However, arrays that do not match in the number of dimensions will be *broadcasted* by Python to perform mathematical operations. This often means that the smaller array will be repeated as necessary to perform the operation indicated. Consider the following:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> b = np.array([-1, 3], float)
>>> a
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> b
array([-1.,  3.])
>>> a + b
array([[ 0.,  5.],
       [ 2.,  7.],
       [ 4.,  9.]])
```

Here, the one-dimensional array b was broadcasted to a two-dimensional array that matched the size of a. In essence, b was repeated for each item in a, as if it were given by

```
array([[-1.,  3.],
       [-1.,  3.],
       [-1.,  3.]])
```

Numpy

- np.array
  - Array mathematics - Broadcasting

Python automatically broadcasts arrays in this manner. Sometimes, however, how we should broadcast is ambiguous. In these cases, we can use the `newaxis` constant to specify how we want to broadcast:

```
>>> a = np.zeros((2,2), float)
>>> b = np.array([-1., 3.], float)
>>> a
array([[ 0.,  0.],
       [ 0.,  0.]])
>>> b
array([-1., 3.])
>>> a + b
array([[-1.,  3.],
       [-1.,  3.]])
>>> a + b[np.newaxis,:]
array([[-1.,  3.],
       [-1.,  3.]])
>>> a + b[:,np.newaxis]
array([[-1., -1.],
       [ 3.,  3.]])
```

Numpy

- np.array
  - Array mathematics

In addition to the standard operators, NumPy offers a large library of common mathematical functions that can be applied elementwise to arrays. Among these are the functions: `abs`, `sign`, `sqrt`, `log`, `log10`, `exp`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`, `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, and `arctanh`.

Numpy

- np.array
  - Array mathematics

```
>>> a = np.array([1, 4, 9], float)
```

```
>>> np.sqrt(a)
array([ 1.,  2.,  3.])
```

```
>>> a = np.array([1.1, 1.5, 1.9], float)
>>> np.floor(a)
array([ 1.,  1.,  1.])
>>> np.ceil(a)
array([ 2.,  2.,  2.])
>>> np.rint(a)
array([ 1.,  2.,  2.])
```

```
>>> np.pi
3.1415926535897931
>>> np.e
2.7182818284590451
```

Numpy

- np.array
  - Array iteration

```
>>> a = np.array([1, 4, 5], int)
>>> for x in a:
...   print x
... <hit return>
1
4
5
```

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for x in a:
...   print x
... <hit return>
[ 1.  2.]
[ 3.  4.]
[ 5.  6.]
```

```
>>> a = np.array([[1, 2], [3, 4], [5, 6]], float)
>>> for (x, y) in a:
...   print x * y
...  <hit return>
2.0
12.0
30.0
```

Numpy

- np.array
  - Basic array operations

```
>>> a = np.array([2, 4, 3], float)
```

```
>>> np.sum(a)
9.0
>>> np.prod(a)
24.0
```

  - np.mean(a)
  - np.var(a)
  - np.std(a)
  - np.min(a)
  - np.max(a)
  - np.argmin(a)
  - np.argmax(a)
  - np.sort(a) (not in place)

Numpy

- np.array
  - Basic array operations
    - a=np.array([[1,2],[3,4]])

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

    - np.mean(a) #2.5
    - np.mean(a,axis=0) #array([2., 3.]) #column wise
    - np.mean(a,axis=1) #array([1.5, 3.5]) #row wise
    - b=np.array([[11,5,14],[2,5,1]])

```
[
    [11, 5, 14],
    [2, 5, 1]
]
```

    - np.sort(b)

```
# array([[ 5, 11, 14],
         [ 1,  2,  5]])
```

    - np.sort(b,axis=1)

```
# array([[ 5, 11, 14],
         [ 1,  2,  5]])
```

    - np.sort(b,axis=0)

```
# array([[ 2, 5, 1],
         [ 11,  5,  14]])
```

Numpy

- np.array
  - Basic array operations

```
>>> a = np.array([1, 1, 4, 5, 5, 5, 7], float)
>>> np.unique(a)
array([ 1.,  4.,  5.,  7.])
```

Numpy

- np.array
  - Comparison Operators & Value Testing

```
>>> a = np.array([1, 3, 0], float)
>>> b = np.array([0, 3, 2], float)
>>> a > b
array([ True, False, False], dtype=bool)
```

```
>>> a == b
array([False,  True, False], dtype=bool)
>>> a <= b
array([False,  True,  True], dtype=bool)
```

The results of a Boolean comparison can be stored in an array:

```
>>> c = a > b
>>> c
array([ True, False, False], dtype=bool)
```

Arrays can be compared to single values using broadcasting:

```
>>> a = np.array([1, 3, 0], float)
>>> a > 2
array([False,  True, False], dtype=bool)
```

Numpy

- np.array
  - Comparison Operators & Value Testing

The any and all operators can be used to determine whether or not any or all elements of a Boolean array are true:

```
>>> c = np.array([ True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

Numpy

- np.array
  - Where Function

The where function forms a new array from two arrays of equivalent size using a Boolean filter to choose between elements of the two. Its basic syntax is `where(boolarray, truearray, falsearray)`:

```
>>> a = np.array([1, 3, 0], float)
>>> np.where(a != 0, 1 / a, a)
array([ 1.        ,  0.33333333,  0.        ])
```

Broadcasting can also be used with the where function:

```
>>> np.where(a > 0, 3, 2)
array([3, 3, 2])
```

Numpy

- np.array
  - Checking for NaN and Inf

It is also possible to test whether or not values are NaN ("not a number") or finite:

```
>>> a = np.array([1, np.NaN, np.Inf], float)
>>> a
array([  1.,  NaN,  Inf])
>>> np.isnan(a)
array([False,  True, False], dtype=bool)
>>> np.isfinite(a)
array([ True, False, False], dtype=bool)
```

Numpy

- np.array
  - Array Item Selection & Manipulation

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> a >= 6
array([[ True, False],
       [False,  True]], dtype=bool)
>>> a[a >= 6]
array([ 6.,  9.])
```

Notice that sending the Boolean array given by a>=6 to the bracket selection for a, an array with only the True elements is returned. We could have also stored the selector array in a variable:

```
>>> a = np.array([[6, 4], [5, 9]], float)
>>> sel = (a >= 6)
>>> a[sel]
array([ 6.,  9.])
```

Numpy

- np.array
  - Vector and Matrix Mathematics

```
>>> a = np.array([1, 2, 3], float)
>>> b = np.array([0, 1, 1], float)
>>> np.dot(a, b)
5.0
```

The dot function also generalizes to matrix multiplication:

```
>>> a = np.array([[0, 1], [2, 3]], float)
>>> b = np.array([2, 3], float)
>>> c = np.array([[1, 1], [4, 0]], float)
>>> a
array([[ 0.,  1.],
       [ 2.,  3.]])
>>> np.dot(b, a)
array([  6.,  11.])
>>> np.dot(a, b)
array([  3.,  13.])
>>> np.dot(a, c)
array([[  4.,   0.],
       [ 14.,   2.]])
>>> np.dot(c, a)
array([[ 2.,  4.],
       [ 0.,  4.]])
```

Numpy

- np.array
  - Vector and Matrix Mathematics

```
>>> a = np.array([1, 4, 0], float)
>>> b = np.array([2, 2, 1], float)
```

```
>>> np.cross(a, b)
array([ 4., -1., -6.])
```

Numpy

- np.array
  - Statistics

The median can be found:

```
>>> a = np.array([1, 4, 3, 8, 9, 2, 3], float)
>>> np.median(a)
3.0
```

The correlation coefficient for multiple variables observed at multiple instances can be found for arrays of the form [[x1, x2, ...], [y1, y2, ...], [z1, z2, ...], ...] where x, y, z are different observables and the numbers indicate the observation times:

```
>>> a = np.array([[1, 2, 1, 3], [5, 3, 1, 8]], float)
>>> c = np.corrcoef(a)
>>> c
array([[ 1.        ,  0.72870505],
       [ 0.72870505,  1.        ]])
```

Here the return array c[i,j] gives the correlation coefficient for the ith and jth observables. Similarly, the covariance for data can be found:

```
>>> np.cov(a)
array([[ 0.91666667,  2.08333333],
       [ 2.08333333,  8.91666667]])
```

Numpy

- np.array
  - Random Numbers

An array of random numbers in the half-open interval [0.0, 1.0) can be generated:

```
>>> np.random.rand(5)
array([ 0.40783762,  0.7550402 ,  0.00919317,  0.01713451,  0.95299583])
```

The `rand` function can be used to generate two-dimensional random arrays, or the `resize` function could be employed here:

```
>>> np.random.rand(2,3)
array([[ 0.50431753,  0.48272463,  0.45811345],
       [ 0.18209476,  0.48631022,  0.49590404]])
>>> np.random.rand(6).reshape((2,3))
array([[ 0.72915152,  0.59423848,  0.25644881],
       [ 0.75965311,  0.52151819,  0.60084796]])
```

To generate a single random number in [0.0, 1.0),

```
>>> np.random.random()
0.7011042743576955 1
```

To generate random integers in the range [min, max) use `randint(min, max)`:

```
>>> np.random.randint(5, 10)
9
```

Numpy

- np.array
  - Random Numbers

The random module can also be used to randomly shuffle the order of items in a list. This is sometimes useful if we want to sort a list in random order:

```
>>> l = range(10)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> np.random.shuffle(l)
>>> l
[4, 9, 5, 0, 2, 7, 6, 8, 1, 3]
```

Saving and Loading Numpy Array

```
# Single array saving and loading
x = np.arange(10)

#save
np.save('outfile', x)

#load
x = np.load('outfile.npy')

print(x)
```

Saving and Loading Numpy Array

```
# Multiple array saving and loading

x = np.arange(10)
y = np.random.randint(1, 10, (2, 3))

#save
np.savez('outfile', x, y)  # or np.savez('outfile', x = x, y = y)

#load
dict = np.load('outfile.npz')
x = dict['arr_0']          # or x = dict['x']
y = dict['arr_1']          # or y = dict['y']

print(x, y)
```

Disclaimer

- Content of this presentation is not original and it has been prepared from various sources for teaching purpose.