The background of the slide features a series of overlapping, diagonal streaks in various shades of green and yellow, creating a dynamic, abstract pattern that suggests movement and energy. The streaks are most concentrated on the left side and fan out towards the right.

# **Introduction to Computational Science**

# Contents

- Introduction
- Applications involving scientific computing
- Tools and languages to solve complex scientific problems

# Introduction to Computational Science

- Computational science is an exciting new field at the intersection of the sciences, computer science, and mathematics because much scientific investigation now involves computing as well as theory and experiment.
- Computational Science is now well recognized by the scientific community as a critical enabling discipline underpinning modern research and development in all fields of science, engineering, and medicine.

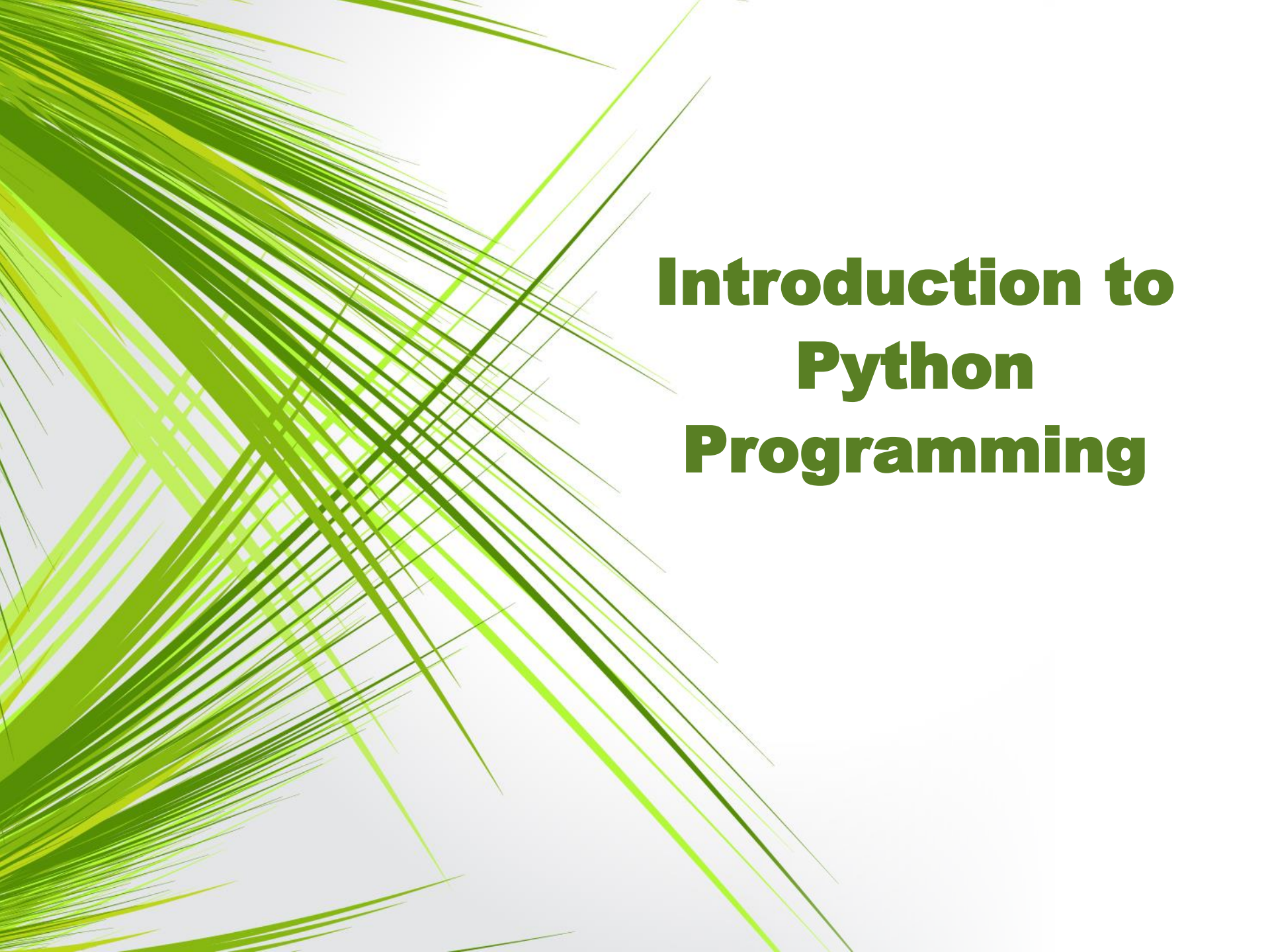


# Applications involving scientific computing

- ***Scientific Computing*** is the collection of tools, techniques, and theories required to solve on a computer mathematical models of problems in **Science** and **Engineering**.
- The applications range in scale from biomolecules, complex materials, built structures and other engineered systems, weather and climate, to galaxies, observing proteins as they bind to drug molecules, predicting the landfall of hurricanes or the spread of disease, and mapping the evolution of the universe.

# Tools and languages to solve complex scientific problems

- Python
- MatLab
- R
- etc....



# **Introduction to Python Programming**



# Contents

- Introduction
- Environmental setup
- Basic Syntax
- Data Types
- Operators
- Decision Making
- Loops
- Numbers
- Strings
- File I/O
- numPy

# What is Python?

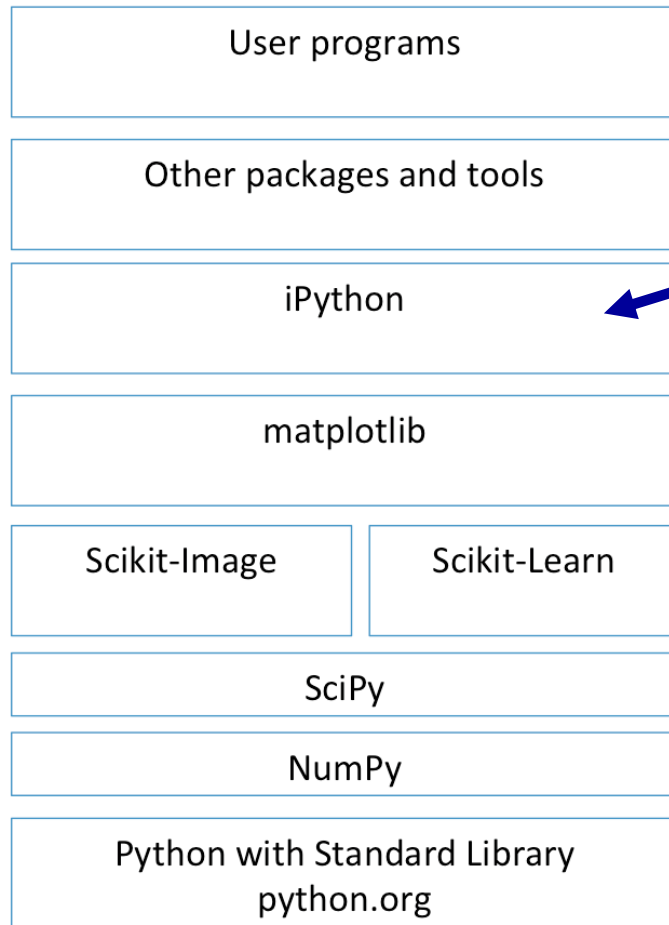
- Python is a widely used high-level, general-purpose, interpreted, dynamic programming language
- Available under the GNU General Public License (GPL)
- Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles.
- It features a dynamic type system and automatic memory management and has a large and comprehensive standard library.
- Huge community pypi (Python Package Index) containing 92K + packages
- Python Features
  - Easy-to-learn
  - Easy-to-read
  - A broad standard library
  - Databases
  - GUI Programming
  - Scientific Calculations



# Python for Science and Engineering

- Python by itself is not that useful for S&E
- Many add-ons and extensions have been created to allow python to work for S&E's, web development, big data processing, and other software stacks".

Python Stack for Scientist and Engineers



iPython is now called Jupyter



# Getting Python

Most up-to-date version is available on the official website of Python

<https://www.python.org/>

It is available for a wide variety of platforms.

# Getting Python

Various ways to start python:

## 1. Immediate Mode:

Enter python command to get following prompt

```
>>>
```

The python interpreter is ready for input

Eg. type 4+7 and press enter to get the output

To exit this mode type `exit()` or `quit()` and press enter.



# Getting Python

Various ways to start python:

## 2. Script Mode:

Used to execute Python program written in a file called a script

Python scripts have the extension .py

To execute this file in script mode we simply write *python helloWorld.py* at the command prompt

# Getting Python

Various ways to start python:

## 3. Integrated Development Environment (IDE):

IDE is a piece of software that provides useful features like code hinting, syntax highlighting and checking, file explorers etc. to the programmer for application development

IDLE is the IDE installed along with the Python programming language and is available from the official website. Others are

- Jupyter notebook
- Spyder
- PyCharm
- PyScripter
- PythonWin

Anaconda is one of the most popular Python data science platform

# The Anaconda Navigator

Anaconda Navigator

File Help

ANACONDA NAVIGATOR

Sign in to Anaconda Cloud

Home

Environments

Learning

Community






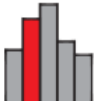


Documentation

Developer Blog

Feedback

Twitter YouTube GitHub

Applications on base (root) Channels Refresh

 jupyterlab 0.32.1 An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture. <a href="#">Launch</a>	 notebook 5.5.0 Web-based, interactive computing notebook environment. Edit and run human-readable docs while describing the data analysis. <a href="#">Launch</a>	 qtconsole 4.3.1 PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical calltips, and more. <a href="#">Launch</a>	 spyder 3.2.8 Scientific PYTHON Development Environment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features <a href="#">Launch</a>
 vscode 1.25.1 Streamlined code editor with support for development operations like debugging, task running and version control.	 glueviz 0.13.3 Multidimensional data visualization across files. Explore relationships within and among related datasets.	 orange3 3.13.0 Component based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows	 rstudio 1.1.423 A set of integrated tools designed to help you be more productive with R. Includes R essentials and notebooks.

19:22 17-08-2018



# Jupyter

The screenshot shows a web browser window with the address bar at `localhost:8888/tree`. The Jupyter logo is in the top left, and 'Quit' and 'Logout' buttons are in the top right. Below the navigation tabs ('Files', 'Running', 'Clusters'), there is a message 'Select items to perform actions on them.' and buttons for 'Upload', 'New', and a refresh icon. The main area displays a file tree for the root directory. The tree includes standard Windows folders like 'Anaconda3', 'Contacts', 'Desktop', 'Documents', 'Downloads', 'Favorites', 'Links', 'Music', 'Pictures', 'Saved Games', 'Searches', and 'Videos'. It also lists Jupyter notebooks: 'Untitled.ipynb' (1.32 kB, 11 days ago), 'Untitled1.ipynb' (2.24 kB, 10 days ago), and 'Untitled2.ipynb' (782 B, 3 days ago). The Windows taskbar at the bottom shows icons for various applications and the system clock at 19:23 on 17-08-2018.

Home x

localhost:8888/tree

Apps MQTT Websocket Cli CSE Dept Recruitment Python

jupyter

Quit Logout

Files Running Clusters

Select items to perform actions on them.

Upload New ↻

<input type="checkbox"/> 0 ▾	/	Name ↓	Last Modified	File size
<input type="checkbox"/>	📁 Anaconda3		11 days ago	
<input type="checkbox"/>	📁 Contacts		a month ago	
<input type="checkbox"/>	📁 Desktop		3 hours ago	
<input type="checkbox"/>	📁 Documents		11 days ago	
<input type="checkbox"/>	📁 Downloads		39 minutes ago	
<input type="checkbox"/>	📁 Favorites		a month ago	
<input type="checkbox"/>	📁 Links		a month ago	
<input type="checkbox"/>	📁 Music		a month ago	
<input type="checkbox"/>	📁 Pictures		a month ago	
<input type="checkbox"/>	📁 Saved Games		a month ago	
<input type="checkbox"/>	📁 Searches		a month ago	
<input type="checkbox"/>	📁 Videos		a month ago	
<input type="checkbox"/>	📄 Untitled.ipynb		11 days ago	1.32 kB
<input type="checkbox"/>	📄 Untitled1.ipynb		10 days ago	2.24 kB
<input type="checkbox"/>	📄 Untitled2.ipynb		3 days ago	782 B

EN 19:23 17-08-2018

# Spyder

The image shows the Spyder Python IDE interface. The main editor window displays a Python script named `temp.py` with the following code:

```
1 #-*- coding: utf-8 -*-
2 """
3 Spyder Editor
4
5 This is a temporary script file.
6 """
7 a=10
8 b=10
9 c=a+b
10 print(c)
11
```

The Variable explorer panel on the right shows the current state of variables:

Name	Type	Size	Value
a	int	1	10
b	int	1	10
c	int	1	20

The IPython console at the bottom shows the execution output:

```
Python 3.6.5 [Anaconda, Inc.] (default, Mar 29 2018, 13:32:41) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 6.4.0 -- An enhanced Interactive Python.

In [1]: runfile('C:/Users/CSE-5/.spyder-py3/temp.py', wdir='C:/Users/CSE-5/.spyder-py3')
20

In [2]:
```

The bottom status bar indicates the current settings: Permissions: RW, End-of-lines: CRLF, Encoding: UTF-8, Line: 1, Column: 1, Memory: 75 %.

# PyCharm

1aBarChart.py [C:\Users\CSE-5\AppData\Local\Temp\1aBarChart.py] - E:\Python\1aBarChart.py [1aBarChart.py] - PyCharm

File Edit View Navigate Code Refactor Run Tools VCS Window Help

1aBarChart.py

Add Configuration...

Project

- 1aBarChart.py
- External Libraries
  - < Python 3.6 > C:\Users\CSE-5\Anaconda3\python
- Scratches and Consoles

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 menMeans = (20, 35, 30, 35, 27)
5 menStd = (2, 3, 4, 1, 2)
6 womenMeans = (25, 32, 34, 20, 25)
7 womenStd = (3, 5, 2, 3, 3)
8
9
10 N = len(menMeans)           # number of data entries
11 ind = np.arange(N)         # the x locations for the groups
12 width = 0.35               # bar width
13
14 fig, ax = plt.subplots()
15
16 rects1 = ax.bar(ind, menMeans,           # data
17                 width,                   # bar width
18                 color='MediumSlateBlue', # bar colour
19                 yerr=womenStd,           # data for error bars
20                 error_kw={'ecolor':'Tomato', # error-bars colour
21                           'linewidth':2})   # error-bar width
22
23 rects2 = ax.bar(ind + width, womenMeans,
24                 width,
25                 color='Tomato',
26                 yerr=womenStd,
27                 error_kw={'ecolor':'MediumSlateBlue',
28                           'linewidth':2})
29
30 axes = plt.gca()
31 axes.set_ylim([0, 41])           # y-axis bounds
32
33 ax.set_ylabel('Scores')
```

Python Console Terminal 6: TODO Event Log

Updating skeletons for C:\Users\CSE-5\Anaconda3\python.exe...

50:18 CRLF UTF-8 22:20 18-08-2018



# Installing Packages

- You can install packages from the Anaconda terminal using the command:
  - `conda install <name of package>`
- For example, Seaborn is a package for Statistical Data Visualization.
  - `conda install seaborn`
- piserial is a package for communication with serial devices.
  - `conda install piserial`

# Running Python

➤ Type following code in any of IDE/python prompt

```
>>> print("Hello world!")
```

➤ Few more examples

```
>>> 5+7
```

# Python Keywords

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



# Python Identifiers

- Identifier is a name used to identify a variable, function, class, module or other object
- An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9)
- Python does not allow punctuation characters such as @, \$, and % within identifiers
- Identifiers are case sensitive

# Python Indentation

- In most other programming languages the indentation in code is for readability only, in Python the indentation is very important
- Python uses indentation to indicate a block of code
- Following code creates a block under if statement

```
if 5 > 2:  
    print("Five is greater than two!")
```

- While this code generates error

```
if 5 > 2:  
print("Five is greater than two!")
```

# Comments and help

## ➤ Python Comments:

- Comments start with a #, and Python will render the rest of the line as a comment

```
#This is a comment.  
print("Hello, World!")
```

## ➤ Docstrings

```
"""This is a  
multiline docstring."""
```

## ➤ 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('V','S','U', sep="", \t", end="X")` `#V, S, UX`



# Printing in Python

- `print("a={0}".format(a))` `#a=5`
- `print("a={0} and b={1}".format(a, b))` `#a=5 and b=2`
- `print("a={1} and b={0}".format(a, b))` `#a=2 and b=5`
- `print("a={} and b={}".format(a, b))` `#a=5 and b=2 #auto field numbering`
- `print("bin={0:b}, oct={0:o}, hex={0:x}".format(12))` `#bin=1100, oct=14, hex=c`
- `print("bin={0:b}, oct={1:o}, hex={1:x}".format(12,10))` `#bin=1100, oct=12, hex=a`
- `print("bin={:b}, oct={:o}, hex={:X}".format(12,10,10))` `#bin=1100, oct=12, hex=A`

# Printing – Padding & Alignment

- `print("a={0:d} and b={1:d}".format(a, b))` `#a=5 and b=2`
- `print("a={0:3d} and b={1:5d}".format(a, b))` `#a= 5 and b= 2`
- `print("a={0:>3d} and b={1:>5d}".format(a, b))` `#a= 5 and b= 2`
- `print("a={0:<3d} and b={1:<5d}".format(a, b))` `#a=5 and b=2`
- `print("a={:<{}}d and b={:<{}}d".format(a,3,b,5))` `#???`
- `print("a={0:03d} and b={1:05d}".format(a, b))` `#a=005 and b=00002`
- `print("a={0:^3d} and b={1:^5d}".format(a, b))` `#a= 5 and b= 2`
- `print("a={:f}".format(123.4567898))` `#a=123.456790`
- `print("a={:8.3f}".format(123.4567898))` `#a= 123.457`

# Number Formatting Types

Number Formatting Types

Type	Meaning
d	Decimal integer
c	Corresponding Unicode character
b	Binary format
o	Octal format
x	Hexadecimal format (lower case)
X	Hexadecimal format (upper case)
n	Same as 'd'. Except it uses current locale setting for number separator
e	Exponential notation. (lowercase e)
E	Exponential notation (uppercase E)
f	Displays fixed point number (Default: 6)
F	Same as 'f'. Except displays 'inf' as 'INF' and 'nan' as 'NAN'
g	General format. Rounds number to p significant digits. (Default precision: 6)
G	Same as 'g'. Except switches to 'E' if the number is large.
%	Percentage. Multiplies by 100 and puts % at the end.

# 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 and working their way from -1 at the end.
- 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 at index 2 to 4**  
                                 **# first index is inclusive while end is exclusive**
- `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 –

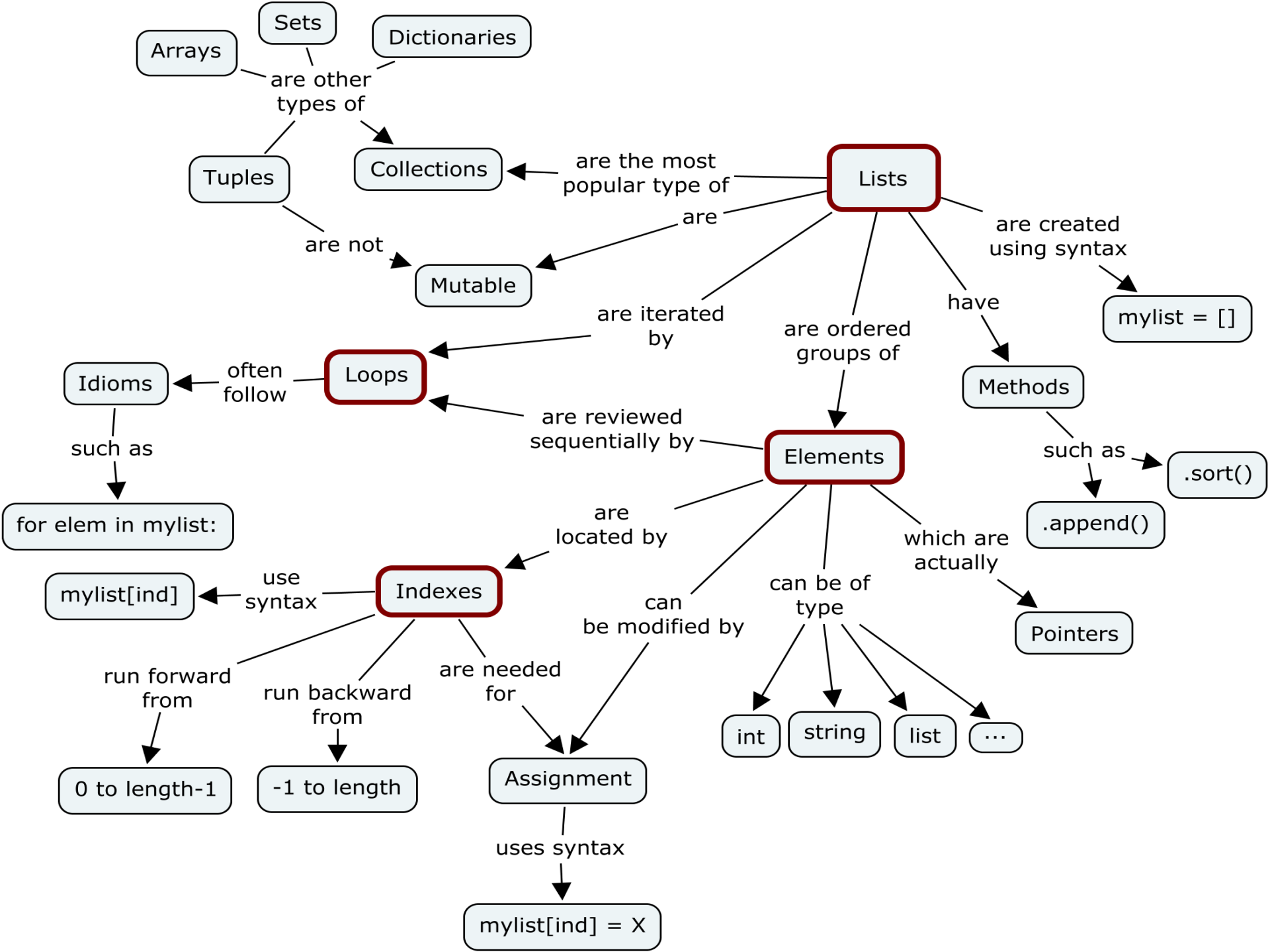
➤ `!`

➤ `ld`

➤ `Hello World!`

# Mutable and Immutable in Python

- Python represents all its data as objects that has
  - an identity (id)
  - a value (mutable or immutable)
- Some of these objects like lists and dictionaries are mutable, meaning you can change their content without changing their identity (Mutable)
- Other objects like integers, floats, strings and tuples are objects that can not be changed (Immutable). If you change the value, they also change their id.
- Some objects are mutable, meaning they can be altered. Others are immutable; they cannot be changed but rather return new objects when attempting to update
- Primitive data types are normally immutable while container and user-defined data types are mutable
- Immutable objects are fundamentally expensive to "change", because doing so involves creating a copy. Changing mutable objects is cheap



# Mutable and Immutable in Python

```
b = []
```

```
print(id(b))
```

```
140675605442000
```

```
b.append(3)
```

```
print(b)
```

```
[3]
```

```
print(id(b))
```

```
140675605442000    # SAME
```

```
a = 4
```

```
print(id(a))
```

```
6406872
```

```
a = a + 1
```

```
print(id(a))
```

```
6406899    # DIFFERENT
```



# 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.

JupyterLab

# Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

- **Casting in python is therefore done using constructor functions:**
  - `int()` - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
  - `float()` - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
  - `str()` - constructs a string from a wide variety of data types, including strings, integer literals and float literals

# Type Conversion

You can convert from one type to another with the `int()`, `float()`, and `complex()` methods:

```
y = 2.8 # float
```

```
z = 1j # complex
```

```
#convert from int to float:
```

```
a = float(x)
```

```
#convert from float to int:
```

```
b = int(y)
```

```
#convert from int to complex:
```

```
c = complex(x)
```

```
print(a,b,c)
```

**Note:** You cannot convert complex numbers into another number type.

# Standard Data Types

Python has the following data types built-in by default, in these categories:

- Text Type: `str`
- Numeric Types: `int`, `float`, `complex`
- Sequence Types: `list`, `tuple`, `range`
- Mapping Type: `dict`
- Set Types: `set`, `frozenset`
- Boolean Type: `bool`
- Binary Types: `bytes`, `bytearray`, `memoryview`



# 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 (see the jupyter notebook)

- `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 from 2nd till 3rd
- `print (list[2:])`    # Prints elements 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

- `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 from 2nd till 3rd
- `print (list[2:])`      # Prints elements 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 and List

- `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

## ➤ Sets

- A set in Mathematics refers to an unordered collection of objects without any duplicate.
- An object of type set may be created by enclosing the elements of the sets with in curly brackets.
- The major difference is that sets, unlike lists or tuples, cannot have multiple occurrences of the same element and store unordered values.
- The set objects may also created by applying the set function to lists strings and tuples.
- Unlike lists we can't access the elements of a set through indexing and slicing
- We cant apply + and \* operators on sets
- Open the jupyter notebook



# 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

- Matrix is a two-dimensional data structure
- All matrices can be considered as nested list
- `a=[[1,2,3],[4,5,6]]`
- `a[0]`, `a[1]`, `a[0][0]`, `a[0][2]`, `a[1][2]`
- Eg.

```
a = [['Vijay',80,75,85,90,95],  
      ['Vishal',75,80,75,85,100],  
      ['Priyank',80,80,80,90,95]]
```

```
print(a[0])           # ['Vijay', 80, 75, 85, 90, 95]
```

```
print(a[0][1])        # 80
```

```
print(a[1][2])        # 80
```

# 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	$a ** b = 10 \text{ to the power } 21$
//	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 = 0b00111100 and b = 13 = 0b00001101, 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.

Eg.

```
list1=[1,2,3,4,5]
list2=[6,7,8,9]
for item in list1:
    if item in list2:
        print("overlapping")
else:
    print("not overlapping")
```

# 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).

```
# Python program to illustrate the use
# of 'is' identity operator
x = 5
if (type(x) is int):
    print ("true")
else:
    print ("false")
```

# Basic Operators

## ➤ Python Operator Precedence

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

## ➤Output:

Current Letter : P

Current Letter : y

Current Letter : t

Breaked Letter : h

# 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, 76, 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 contains an even number
```

# Loops

## ➤ Using else Statement with Loops

```
num=0
while num < 10:
    num=num+1
    if num%2==0:
        print('In side the loop', num)
        break
else:
    print ('Outside the loop in else')
```

## Output:

In side the loop 2

# Numbers - Revisited

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

Need to import math

```
import math
num=-6.1
print(math.ceil(num))
```

# Numbers - Revisited

## ➤ Numbers

### ➤ Mathematical Functions

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

# Numbers - Revisited

## ➤ Numbers

### ➤ Random Number Functions

Function	Description
<code>random.choice(seq)</code>	A random item from a list, tuple, or string.
<code>random.random()</code>	A random float $r$ , such that 0 is less than or equal to $r$ and $r$ is less than 1
<code>random.seed([x])</code>	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
<code>random.shuffle(lst)</code>	Randomizes the items of a list in place. Returns None.
<code>random.uniform(x, y)</code>	A random float $r$ , such that $r$ is less than or equal to $x$ and $r$ is less than $y$
<code>random.randint(x, y)</code>	Return random integer in range $[x, y]$ , including both end points
<code>random.sample(population, k)</code>	Chooses $k$ unique random elements from a population sequence or set.

# Numbers - Revisited

## ➤ Random Number Functions

```
import random
myList = [2, 109, False, 10, "Vijay", 482, "Vishal"]
print(random.randint(0, 100))
print(random.random() * 100)
print(random.choice(myList))
print(random.randrange(0, 101, 5))
print(random.uniform(0, 100))

print(myList)
random.shuffle(myList)
print(myList)
```

### **Output**

```
39
45.26060117036571
Vijay
40
84.56477617706464
[2, 109, False, 10, 'Vijay', 482, 'Vishal']
[482, 2, 'Vishal', 'Vijay', 109, 10, False]
```



# Strings - Revisited

## ➤ Strings (Assume str to be a string variable)

Sr. No.	Methods with Description
1	<code>str.capitalize()</code> Capitalizes first letter of string.
2	<code>str.isalnum()</code> Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
3	<code>str.isalpha()</code> Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
4	<code>str.isdigit()</code> Returns true if string has at least 1 character and contains only digits and false otherwise.
5	<code>str.islower()</code> Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
6	<code>str.isspace()</code> Returns true if string contains only whitespace characters and false otherwise.

# Strings - Revisited

## ➤ Strings

Sr. No.	Methods with Description
7	<code>str.isupper()</code> Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
8	<code>len(str)</code> Returns the length of the string
9	<code>str.lower()</code> Converts all uppercase letters in string to lowercase.
10	<code>max(str)</code> Returns the max alphabetical character from the string str.
11	<code>min(str)</code> Returns the min alphabetical character from the string str.
12	<code>str.upper()</code> Converts lowercase letters in string to uppercase.

# 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
<code>len([1, 2, 3])</code>	3	Length
<code>[1, 2, 3] + [4, 5, 6]</code>	<code>[1, 2, 3, 4, 5, 6]</code>	Concatenation
<code>['Hi!'] * 4</code>	<code>['Hi!', 'Hi!', 'Hi!', 'Hi!']</code>	Repetition
<code>3 in [1, 2, 3]</code>	True	Membership
<code>for x in [1,2,3]:     print (x,end=' ')</code>	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	<code>len(list)</code> Gives the total length of the list.
2	<code>max(list)</code> Returns item from the list with max value.
3	<code>min(list)</code> Returns item from the list with min value.
4	<code>list.copy()</code> Returns a copy of the list

# Lists - Revisited

## ➤ List Methods

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

```
myList = [2, 109, False, 10, "Vijay", 482, "Vishal"]  
print(myList)
```

# Function definition is here

```
def changeme( mylist ):  
    "This changes the 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  
changeme(myList)
```

Output:

```
[2, 109, False, 10, 'Vijay', 482, 'Vishal']
```

```
Values inside the function before change: [2, 109, False, 10, 'Vijay', 482, 'Vishal']
```

```
Values inside the function after change: [2, 109, 50, 10, 'Vijay', 482, 'Vishal']
```

# 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] #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

# Python Functions

## ➤ Multiple return values

# This function returns a tuple

```
def fun():
```

```
    str = "Python is Great"
```

```
    x    = 100
```

```
    return str, x; # Return tuple, we could also  
                  # write (str, x)
```

#Call the function now

```
str, x = fun() # Assign returned tuple
```

```
print(str)
```

```
print(x)
```

### **Output:**

Python is Great

100

# 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"`

# Python Input / Output

## ➤ Printing to screen using print statement

```
print ("Python is great")
```

## ➤ Reading keyboard input

### ➤ raw\_input() - discontinued in version 3

- The raw\_input([prompt]) function reads one line from standard input and returns it as a string

```
str = raw_input("Enter your input: ")  
print("Received input is : ", str)
```

### ➤ input()

- The input([prompt]) function is equivalent to raw\_input, except that it assumes the input is a valid Python expression and returns the evaluated result to you.

```
str = input("Enter your input: ");  
print "Received input is : ", str
```

### Output:

```
Enter your input: [x*5 for x in range(2,10,2)]  
Recieved input is : [10, 20, 30, 40]
```



# File Input / Output

## ➤ Open File

```
file_obj = open(file_name [, access_mode][, buffering])
```

- file\_name – The file\_name argument is a string value that contains the name of the file that you want to access
- access\_mode – The access\_mode determines the mode in which the file has to be opened, i.e., read, write, append, etc. This is optional parameter and the default file access mode is read (r).
- buffering – If the buffering value is set to 0, no buffering takes place. If the buffering value is 1, line buffering is performed while accessing a file.

# File Input / Output

Mode	Description
r	Opens a file for reading only. The file pointer is placed at the beginning of the file. This is the default mode
rb	Opens a file for reading only in binary format
r+	Opens a file for both reading and writing
rb+	Opens a file for both reading and writing in binary format
w	Opens a file for writing only. Overwrites the file if the file exists. If the file does not exist, creates a new file for writing
wb	Opens a file for writing only in binary format
w+	Opens a file for both writing and reading
wb+	Opens a file for both writing and reading in binary format
a	Opens a file for appending. The file pointer is at the end of the file if the file exists. That is, the file is in the append mode. If the file does not exist, it creates a new file for writing
ab	Opens a file for appending in binary format
a+	Opens a file for both appending and reading
ab+	Opens a file for both appending and reading in binary format

# File Input / Output

- Once a file is opened with open function, many attributes of the file can be accessed with file\_object

Attribute	Description
file.closed	Returns true if file is closed, false otherwise.
file.mode	Returns access mode with which file was opened.
file.name	Returns name of the file
file.softspace	Returns false if space explicitly required with print, true otherwise

```
# Open a file
fo = open("test.txt", "wb")
print("Name of the file: ", fo.name)
print("Closed or not : ", fo.closed)
print("Opening mode : ", fo.mode)
print("Softspace flag : ", fo.softspace)
```

## Output

```
Name of the file:  test.txt
Closed or not :   False
Opening mode :    wb
Softspace flag :  0
```

- Close the file using file\_object.close() method

# File Reading / Writing

- The `file_object` provides access methods to access the file
- `write()` Method
  - The `write()` method writes any string to an open file

```
#Open a file
```

```
fo = open("test.txt", "w")
```

```
fo.write("Python is a great. \nYeah its great!!\n");
```

```
#Close opened file
```

```
fo.close()
```



# File Reading / Writing

## ➤ read() Method

- The read() method reads a string from an open file

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

## Output

Read String is : Python is

# File Position

## ➤ `tell()` Method

- The `tell()` method tells you the current position within the file
- The `seek(offset[, from])` method changes the current file position. The `offset` argument indicates the number of bytes to be moved. The `from` argument specifies the reference position from where the bytes are to be moved.
  - `from: 0` - means beginning of file
  - `from: 1` - current position as reference point
  - `from: 2` - end of the file as reference point

# File Position

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

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

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

## Output

```
Read String is :  Python is
Current file position :  10
Again read String is :  Python is
```

# Python Modules

- Module allows to logically organize the Python code
- Eg. Create a hello.py file with following contents

```
def say_hello( par ):  
    print "Hello : ", par  
    return
```

- import statement
- A module is loaded only once, regardless of the number of times it is imported

## Output

```
# Import module hello  
import hello
```

```
# Call defined function from module as follows  
hello.say_hello("Vijay")
```



# NumPy

---

---

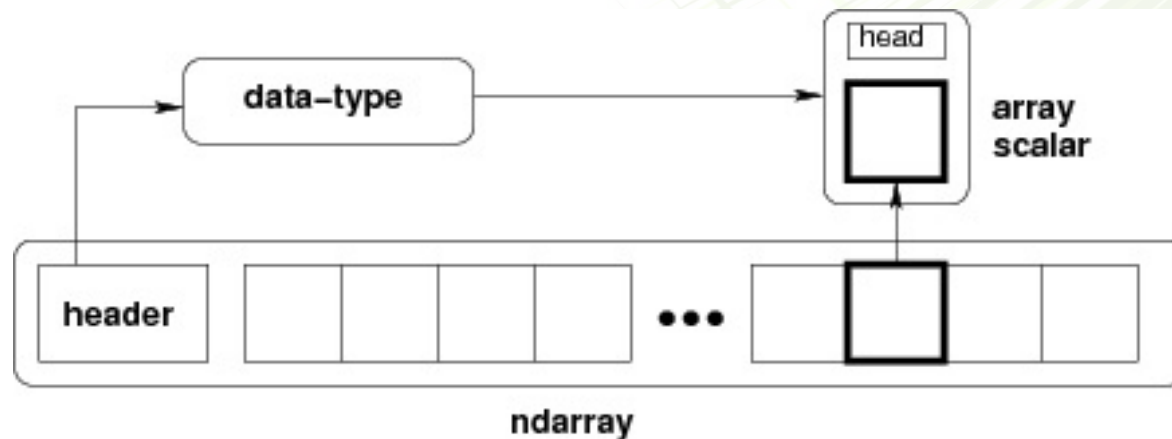
# Numpy

## ➤ 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.**
- **Install**
  - `C:\\Python34\\scripts>pip3.4 list`
  - `C:\\Python34\\scripts>pip3.4 install numpy`
  - **Already installed with Anaconda. If not, use conda install numpy**
- `import numpy as np`

# Numpy

- NumPy has an N-dimensional array type called ndarray
- Every item in an ndarray takes the same size of block in the memory. Each element in ndarray is an object of data-type object (called dtype).



- To create an ndarray
- `numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)`

# Numpy

- To create an array

```
import numpy as np
a = np.array([1,2,3])
print a
```

## **Output**

```
[1 2 3]
```

- To create multi-dimensional array

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

## **Output**

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

- To create array with minimum dimension

```
a = np.array([1, 2, 3, 4, 5], ndmin = 2)
print a
```

## **Output**

```
[[1 2 3 4 5]]
```



# Numpy

- To create an array with dtype parameter

```
a = np.array([1, 2, 3], dtype = complex)
```

```
print a
```

**Output**

```
[ 1.+0.j,  2.+0.j,  3.+0.j]
```

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

```
print a
```

**Output**

```
[[1. 2. 3.]
```

```
 [4. 5. 6.]]
```

# Numpy - datatypes (dtypes)

- NumPy supports a much greater variety of numerical types than Python does.
- `bool_`, `int8`, `int16`, `int32`, `int64`, `uint8`, `uint16`, `uint32`, `uint64`, `float16`, `float32`, `float64`, `complex64`, `complex128` etc...
- A dtype object is constructed using the following syntax :

```
numpy.dtype(object, align, copy)
```

```
dt = np.dtype([('name', 'S20'), ('no', np.int16)])
```

```
a = np.array([('VU', 4), ('ViyU', 20), ('PU', 3)], dtype = dt)
```

```
print (a)
```

```
student = np.dtype([('name', 'S20'), ('age', 'i1'), ('marks', 'f4')])
```

```
a = np.array([('abc', 21, 50), ('xyz', 18, 75)], dtype = student)
```

```
print(a)
```

```
print(a[1])
```

# 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)



# Numpy

## ➤ np.array

### ➤ Basic array operations

- `a=np.array([[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]])`
- `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]])

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

### ➤ 3-D Array

- `a=np.array([ [[11,5,14],[2,5,1]], [[11,5,14],[2,5,1]] ])`

# 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  $[[x_1, x_2, \dots], [y_1, y_2, \dots], [z_1, z_2, \dots], \dots]$  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  $i$ th and  $j$ th 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()
0.70110427435769551
```

To generate random integers in the range  $[\text{min}, \text{max})$  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]
```

# Python References

**Enthought.com** ← Location of Enthought Canopy and Enthought Python Distribution

**Docs.python.org** ← A great resource for general Python

**Docs.enthought.com** ← Release notes, installation instructions for Enthought

**Pyvideo.org** ← a repository of links to videos on Python from all the python conferences

**Training.enthought.com** ← get a Enthought account from Bob get free TRAINING!

**scipy-lectures.org** ← Tutorials on the scientific Python ecosystem

**awesome-python.com** ← A curated list of awesome Python frameworks, libraries, software and resources

**talkpython.fm** ← talk python to me podcast

**pythonbytes.fm** ← very short and to the point weekly updates about python

## Computational Science References:

[https://www.nsf.gov/attachments/118651/public/MPSAC Computational Science White Paper.pdf](https://www.nsf.gov/attachments/118651/public/MPSAC_Computational_Science_White_Paper.pdf)

<https://dl.acm.org/doi/book/10.5555/2591770>

<https://www.scicomp.uni-kl.de/about/scientific-computing/>

<https://cse.gatech.edu/content/scientific-computing-and-simulation>