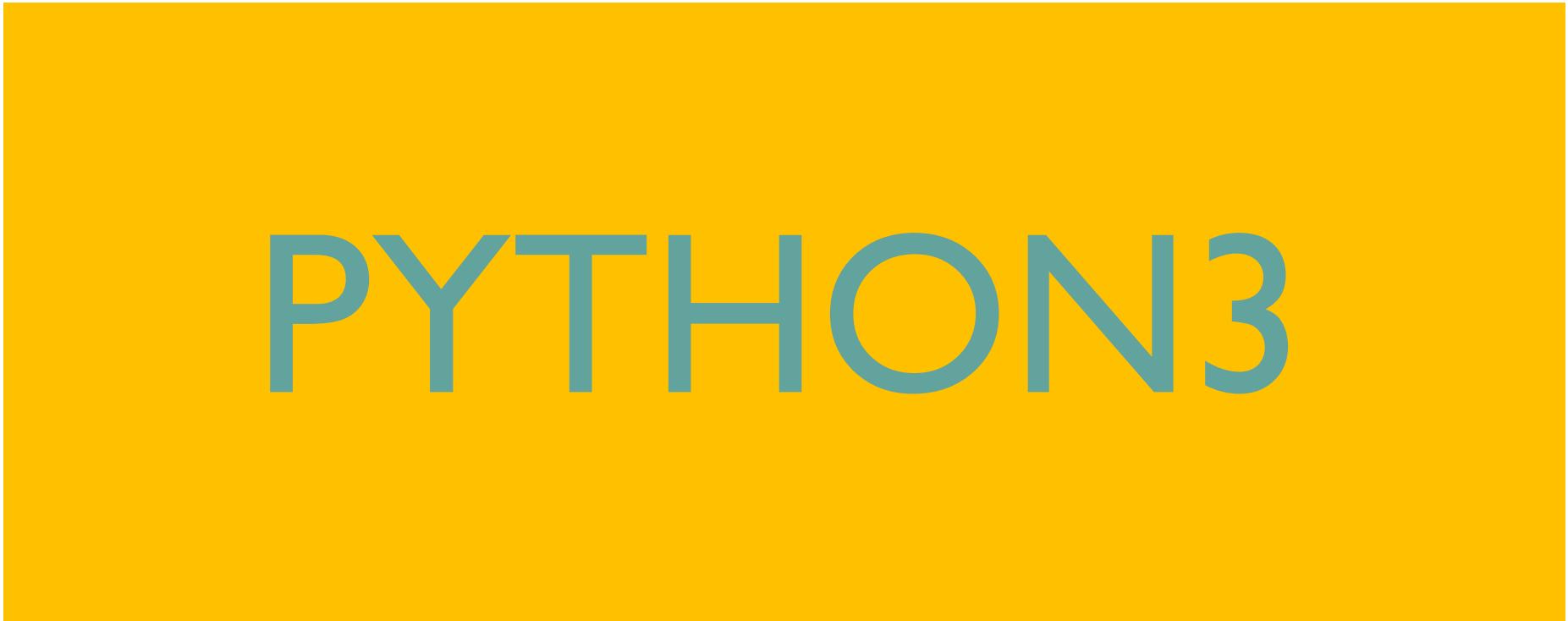




# **NHERI SIMCENTER PROGRAMMING BOOTCAMP**

JULY 22-26, 2019, AT UC BERKELEY'S RICHMOND FIELD STATION



**PYTHON3**

# INTRODUCTION

## ■ What is Python?

- Fully programmable and extendable scripting
- Platform independent (PC, Mac, Linux, Raspberri Pi, Arduino, ...)
- OpenSource (no licensing issues)
- Great prototype for C++
- Does not replace C++

# INTRODUCTION

## ■ Where do I find help?

1. Python documentation: <https://docs.python.org/3.6/index.html>
2. Python tutorial: <https://docs.python.org/3.6/tutorial/index.html>
3. Python library reference: <https://docs.python.org/3.6/library/index.html>
4. Master Google

## ■ Developer tools:

1. **PyCharm CE**
2. VS Code
3. Vim, emacs
4. Notepad, gedit

# INTRODUCTION

## ■ What is similar to C/C++?

- Object oriented
- Many data types
- Same indexing as C/C++ (0 .. N-1)
- Expandable; numerous free libraries

## ■ What is similar to Matlab?

- Easy scripting
- Can be used as calculator on steroids

# INTRODUCTION

## ■ What is different to C/C++?

- Interpreter language; no need for a compiler
- No need for header files
- Structure by indentation instead of { ... }
- No semi-colons “;”

## ■ What is different to Matlab?

- Python is primarily object oriented
- [0..n-1] instead of [1:n]
- Default parameter handling in functions is by address !!!
- Python enforces structured code

# PART I.I: DATA TYPES

Time	Topic
<b>9:00-9:45 AM</b>	<b>Python Data Types</b>
9:45-10:15 AM	Object oriented programming in Python
10:15-10:30 AM	Break
10:30-11:30 AM	File IO and data processing
11:30-1:00 PM	Lunch
1:00-2:00 PM	Data visualization in Python
2:00-3:30 PM	Programming exercise
3:30-3:40 PM	Break
3:40-4:30 PM	Subprocesses and interaction with the OS
4:30-5:00 PM	Exercise
5:00 PM	Adjourn for the day

# EXERCISE: HOW TO PLAY ALONG

- Open terminal
- \$ python3
- >>> print("Hello,World!")
  - what did you get?
- >>> print“Hello,World!”
  - what did you get?
- >>> ^D (this is Ctrl-D)

## EXERCISE: HOW TO PLAY ALONG

```
Peters-MBP-15:Presentations pmackenz$ python3
Python 3.6.8 |Anaconda custom (64-bit)| (default, Dec 29 2018, 19:04:46)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> print("Hello, World!")
Hello, World!
[>>> print "Hello, World!"
  File "<stdin>", line 1
    print "Hello, World!"^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("Hello, World!")?
[>>> print("Hello, World!")
Hello, World!
[>>> print "Hello, World!"^
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("Hello, World!")?
[>>> print("Hello, World!")
Hello, World!
```

# GETTING STARTED

## ■ My first “Hello, world!”

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

## ■ My first variables

```
>>> a = 3  
>>> b = 2  
>>> a/b  
1  
>>> b/a  
0
```

Python 2.7

# GETTING STARTED

## ■ My first variables

```
>>> a = 3
>>> b = 2
>>> a/b
1
>>> b/a
0
```

Python 2.7

```
>>> a = 3.
>>> b = 2.
>>> a/b
1.5
>>> b/a
0.6666666666666667
```

# GETTING STARTED

## ■ Basic Variable types

- Integer
- Floating point (double, real\*8, ...)
- Character
- String

```
>>> a = 2
>>> b = 3.14126
>>> c = 'A'
>>> s = "Peter Mackenzie"
```

# GETTING STARTED

## ■ Types in python

	<b>Mutable</b>	<b>Immutable</b>
Number		int, long, float
Sequence	list	tuple, str, unicode
Mapping	dict	
Other	object	

# GETTING STARTED

## ■ Int, long, float

- Int:  $2^{16}$  (0:65535 or -32767:32768)
- Long:  $2^{32}$
- Float (64bit): ~base ( $2^{53}$ ) \*  $10^{(2^{11})}$

## ■ String

- Array of characters: character ASCII(0:255)
- Unicode: list of unicode characters u(0:65535)

# GETTING STARTED

## ■ Tuple

- ( item1,[ item2 [, item3 [, item4 [...]]]])

## ■ List

- [ item1[, item2, ... , itemN] ]

# GETTING STARTED

## ■ Tuple

- ( item1,[

## ■ List

- [ item1[,

```
IDLE 1.2.2
>>> a=[1,'a','Hello, world!']
>>> b=(1,'a','aaaa')
>>> b[1]
'a'
>>> a[1]
'a'
>>> a[1]=5
>>> a
[1, 5, 'Hello, world!']
>>> b[1]=5

Traceback (most recent call last):
  File "<pyshell#8>", line 1, in <module>
    b[1]=5
TypeError: 'tuple' object does not support item assignment
```

# GETTING STARTED

## ■ Dictionary

- `dict( key1=item1 [, key2=item2 [, key3=item3 [...]]])`
- `{ 'key1':item1 [, 'key2':item2 [, 'key3':item3[, ...] ] ] }`

# GETTING STARTED

## ■ Diction

### ■ dict

```
>>> d = dict(name='Peter', age=42)
>>> d['name']
'Peter'
>>> d['age']
42

>>> d.keys()
['age', 'name']

>>> d.values()
[42, 'Peter']

>>> print ('The answer to everything is ',d['age'])
The answer to everything is 42

>>> for i in d.keys():
    print( i, d[i] )

age 42
name Peter
```

# GETTING STARTED

## ■ Sequences (tuple, list, str)

```
>>> my_list = (3.141267, 'a', "Hello, World!")  
>>> print my_list  
(3.141267, 'a', 'Hello, World!')  
  
>>> for item in my_list:  
    item  
  
3.141267  
'a'  
'Hello, World!'  
  
>>> for item in my_list:  
    print( item )  
  
3.141267  
a  
Hello, World!
```

```
>>> a='Hello, world!'  
>>> for c in a:  
                  print(c)  
  
H  
e  
l  
l  
o  
,w  
o  
r  
l  
d  
!  

```

# GETTING STARTED

## ■ An unexpected problem, or is it?

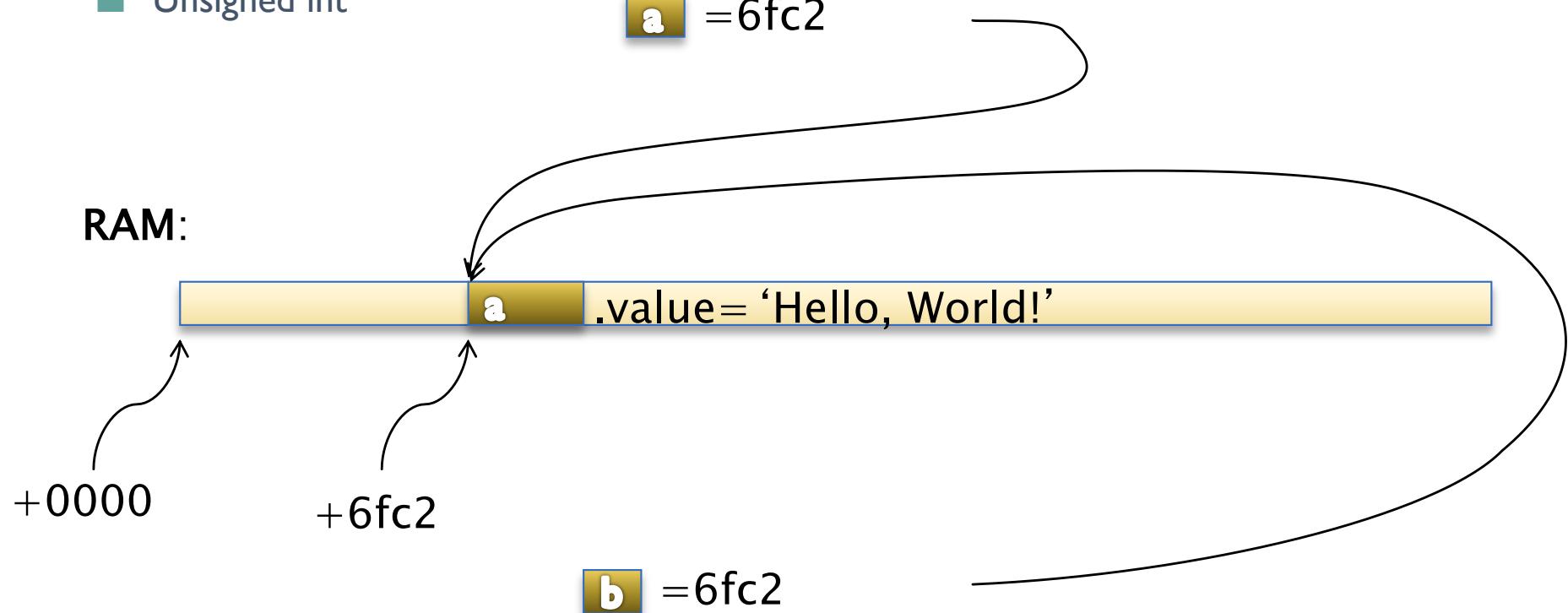
```
>>> a=["Hello","World","!"]  
  
>>> a  
['Hello', 'World', '!']  
  
>>> b=a  
  
>>> b[0]  
'Hello'  
  
>>> b[1]  
'World'  
  
>>> b[1]='Peter'  
  
>>> b  
['Hello', 'Peter', '!']  
  
>>> a  
['Hello', 'Peter', '!']
```

# OTHER TYPES

## ■ Pointers and References

### ■ Unsigned int

**b** = **a**

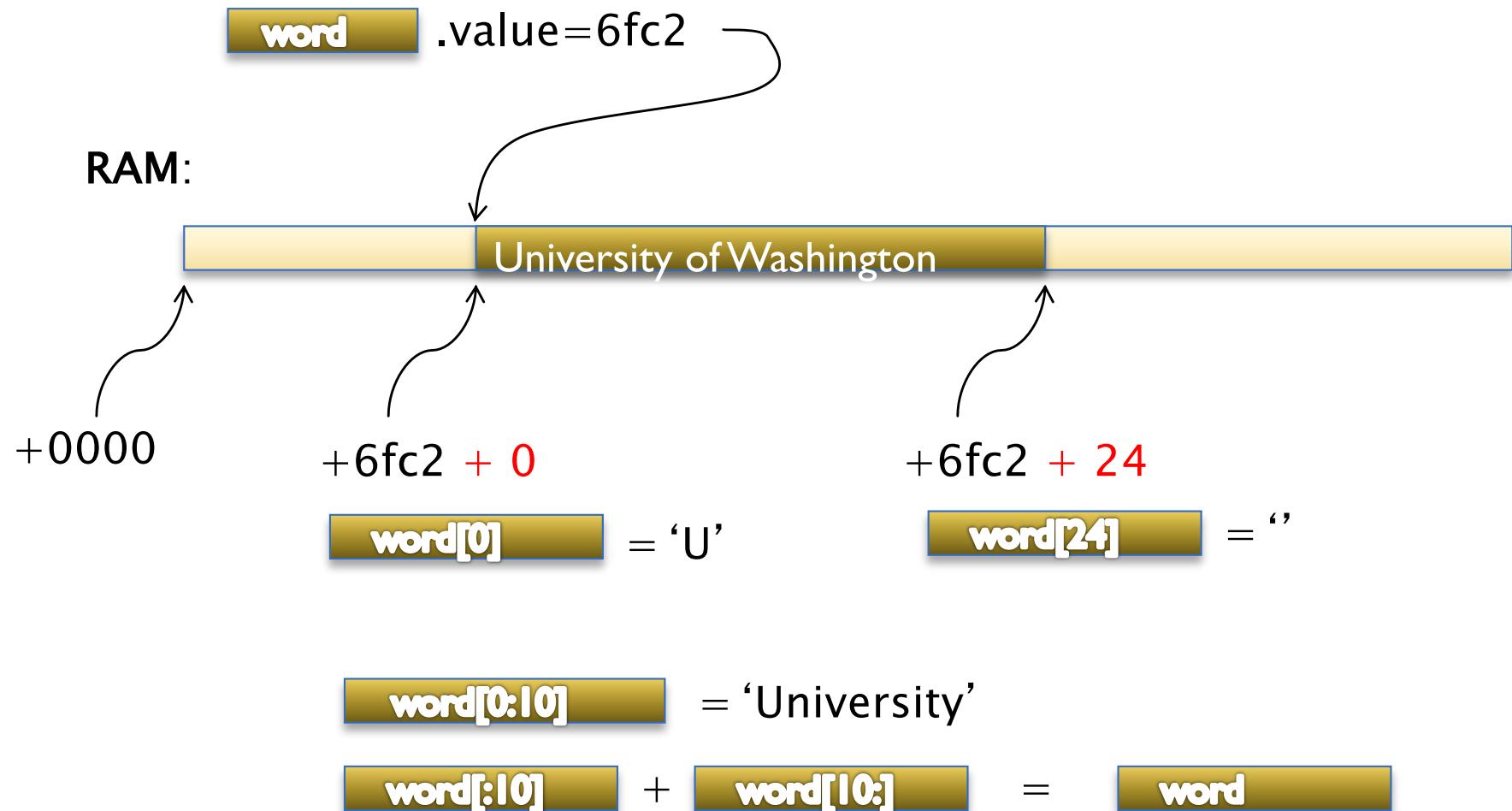


# GETTING STARTED

## Ranges and slicing

```
>>> word='University of Washington'  
>>> word  
'University of Washington'  
  
>>> len(word)  
24  
  
>>> word[1:10]  
'niversity'  
  
>>> p1 = word[1:10]  
>>> len(p1)  
9  
  
>>> list(range(1,10))  
[1, 2, 3, 4, 5, 6, 7, 8, 9]  
  
>>> word[0:10]  
'University'
```

# STRING SEGMENTS



# FLOW CONTROL

## ▶ Boolean operations

- ==
- <
- <=
- >
- >=
- !=
- not
- and, or

▶ **0 = false**

▶ **not 0 = true**

## Exercise

▶  $17 < 365/21$

▶ ‘Monday’ < ‘Friday’

▶  $(100 - 99 - 1)$

▶  $\sin( 3.14127/3 )$

# FLOW CONTROL

## Exercise

- ▶  $17 \leq 365/21$  and ‘Monday’ < ‘Friday’
- ▶ ‘Monday’ < ‘Friday’ or  $\cos(3.1427)$
- ▶  $(100 - 99)$  and  $\sin(3.14127/3)$  or ‘Friday’ > ‘Monday’
- ▶  $(100 - 99)$  and  $\sin(3.14127/3)$  and ‘Friday’ > ‘Monday’
- ▶  $(100 - 99)$  or  $\sin(3.14127/3)$  and ‘Friday’ > ‘Monday’
- ▶  $((100 - 99) \text{ or } \sin(3.14127/3))$  and ‘Friday’ > ‘Monday’
- ▶  $((100 - 99) \text{ or } \sin(3.14127/3))$  or ‘Friday’ > ‘Monday’

# CREATING LISTS

## Range

```
>>> range(3)  
range(0, 3)  
>>> list(range(3))  
[0, 1, 2]
```

```
>>> list(range(3,10))  
[3, 4, 5, 6, 7, 8, 9]
```

```
>>> list(range(3,10,2))  
[3, 5, 7, 9]
```

## Slicing

```
>>> s="SimCenter BootCamp"  
>>> s  
'SimCenter BootCamp'  
  
>>> s[3]  
'C'  
  
>>> s[0:10]  
'SimCenter' # see the extra space ?
```

```
>>> s[:10]  
'SimCenter'
```

```
>>> s[10:]  
'BootCamp'
```

```
>>> s[:10]+s[10:]  
'SimCenter BootCamp'
```

# DATA TYPES

## Scientific data types (numpy extension)

- array
- matrix

```
>>> a = [1,2,3]
>>> b = [5,6,7]
>>> a+b
[1, 2, 3, 5, 6, 7]

>>> b+a
[5, 6, 7, 1, 2, 3]
```

```
import numpy as np

>>> a = np.array([1,2,
>>> b = np.array([5,6,7])
>>> a+b
array([ 6,  8, 10])

>>> b+a
array([ 6,  8, 10])

>>> np.dot(a,b)
38

>>> np.cross(a,b)
array([-4,  8, -4])
```

```
import numpy as np

>>> a=np.matrix([1,2,3])
>>> a
matrix([[1, 2, 3]])

>>> b=np.matrix([5,6,7])
>>> a*b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
File "/Users/pmackenz/anaconda3/lib/python3.6/site-packages/numpy/matrixlib/defmatrix.py", line 215, in __mul__
    return N.dot(self, asmatrix(other))
ValueError: shapes (1,3) and (1,3) not aligned: 3 (dim 1) != 1
(dim 0)
>>> a*b.T
matrix([[38]])
```

# FUNCTIONS

## ■ Definition

```
def f(x):  
    pass
```

```
def f(x):  
    print("x = {}".format(x))
```

## ■ Parameters

### ■ Call by address effects:

- Lists are altered globally (call by address)
- Duple cannot be altered and will trigger error
- Floats and integers are handled by value -> require return

# FUNCTIONS

## ■ Definition

```
def factor(n):
    if n>1:
        return n*factor(n-1)
    else:
        return 1

# test the new function
print( factor(5) )
print("11! = {}".format(factor(11)) )
```

```
def factor(n):
    if n>1:
        return n*factor(n-1)
    else:
        return 1

def prettyPrint(k):
    s = "{}! = {}"
    print(s.format(k, factor(k)))

# test the new function
prettyPrint(5)
prettyPrint(11)
```

**RESULTS:**

5! = 120  
11! = 39916800

# FUNCTIONS

## ■ Parameters

### ■ Call by address effects:

- Lists are altered globally (call by address)
- Tuple cannot be altered and will trigger error
- Floats and integers are handled by value  
-> require a return value

# function definition

```
def f(alist):  
    retval = ""  
    if len(alist) >= 3:  
        retval = alist[2]  
        alist[2] = 'Peter'  
    return retval
```

# the executable portion

```
lst = ['BootCamp', 2019, 'Somebody']  
print(lst)  
print(f(lst))  
print(lst)
```

# FLOW CONTROL

## ■ LOOPS

# MATLAB

```
s = 10;  
H = zeros(s);  
  
for c = 1:s  
    for r = 1:s  
        H(r,c) = 1/(r+c-1);  
    end  
end
```

# Python

```
import numpy as np  
  
s = 10  
H = np.zeros((s,s))  
  
for c in range(s):  
    for r in range(s):  
        H[r,c] = 1/(r+c+1)
```

# FLOW CONTROL

## ■ LOOPS – cont'd

```
>>> lst = ['SimCenter','BootCamp', 2019]  
>>> for item in lst:  
...     print(item)
```

```
SimCenter  
BootCamp  
2019
```

```
>>> s='SimCenter BootCamp 2019'  
>>> for c in s:  
...     print(c)  
S  
i  
m  
C  
e  
n  
t  
e  
r  
B  
o  
o  
t  
C  
a  
m  
P  
2  
0  
1  
9
```

# FLOW CONTROL

## ■ IF THEN ELSE

```
# function definition

def cond(x):
    if x>0:
        print("{} is positive".format(x))
    elif x<0:
        print("{} is negative".format(x))
    else:
        print("{} is zero".format(x))

# execution
cond(3.14127)
cond(7-13)
cond(1./10.-0.1)
```

```
# function definition

def cond(x):
    b = 0
    if x>0:
        b = 1
        print("{} is positive".format(x))
    elif x<0:
        b = 2
        print("{} is negative".format(x))
    else:
        print("{} is zero".format(x))
    return b

# execution
cond(3.14127)
cond(7-13)
cond(1./10.-0.1)
```

# FLOW CONTROL

## ■ WHILE

```
def countdown(n):
    if n<1:
        return
    while n>0:
        print(n)
        n -= 1

# execution
countdown(10)
```

10  
9  
8  
7  
6  
5  
4  
3  
2  
1

# USING MODULES (LIKE LIBRARIES IN C++)

```
import numpy
```

```
a = numpy.array([1,2,3])
```

```
import numpy as np
```

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

```
from numpy import array
```

```
a = array([1,2,3])
```

- ALWAYS start code with imports
- AFTER THAT, define functions and classes
- AFTER THAT, create a main
- AFTER THAT, execute that main

# USING MODULES (LIKE LIBRARIES IN C++)

- Using import is the logic to split code into multiple files
- Have one main.py to start
- Place (groups of) functions (and classes) into their own files

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

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

```
from numpy import array  
  
a = array([1,2,3])
```

# NUMERIC ANALYSIS

- Vector, Matrix, and Tensor analysis requires multidimensional arrays.
- NumPy is a Python library for working with multidimensional arrays.
- The main data type is an array.
- An array is a set of elements, all of the same type, indexed by a vector of nonnegative integers.
- [http://www.scipy.org/Tentative\\_Numpy\\_Tutorial](http://www.scipy.org/Tentative_Numpy_Tutorial)

# NUMERIC ANALYSIS

```
>>> from numpy import *

>>> a = array( [ 10, 20, 30, 40 ] ) # create an array out of a list
>>> a
array( [10, 20, 30, 40] )

>>> b = arange( 4 ) # create an array of 4 integers, from 0 to 3
>>> b
array([0, 1, 2, 3])

>>> c = linspace(-pi,pi,3) # create an array of 3 evenly spaced samples from -pi to pi
>>> c
array( [-3.14159265, 0., 3.14159265] )
```

```
>>> d = a + b**2 # element-wise operations
>>> d
array( [10, 21, 34, 49] )
```

# DATA TYPES WITHIN AN ARRAY

## ■ ndarray.dtype

an object describing the type of the elements in the array. Elements of an array are of identical type.

- standard Python types
- bool\_
- character
- int\_
- int8
- int16
- int32
- int64
- float\_
- float8
- float16
- float32
- float64
- complex\_
- complex64
- object\_

# MULTIDIMENSIONAL ARRAYS

```
>>> from numpy import *
>>> x = ones( (3,4) )
>>> x
array( [[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]])
>>> x.shape
(3, 4)
```

$$\begin{bmatrix} [0,0] & [0,1] & [0,2] & [0,3] \\ [1,0] & [1,1] & [1,2] & [1,3] \\ [2,0] & [2,1] & [2,2] & [2,3] \end{bmatrix}$$

```
>>> x.ravel()
array([ 1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.,  1.])
```

$[0,0] \ [0,1] \ [0,2] \ [0,3] \ [1,0] \ [1,1] \ [1,2] \ [1,3] \ [2,0] \ [2,1] \ [2,2] \ [2,3]$

# MULTIDIMENSIONAL ARRAYS

## MATLAB

```
>> x = ones( 3,4 )
```

```
x =
```

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

```
>> size(x)
```

```
ans =
```

```
3 4
```

$$\begin{bmatrix} [0,0] & [0,1] & [0,2] & [0,3] \\ [1,0] & [1,1] & [1,2] & [1,3] \\ [2,0] & [2,1] & [2,2] & [2,3] \end{bmatrix}$$

```
>> x = reshape( x, 1, 12 )
```

```
x = [1. 1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
```

$$[[0,0] \quad [1,0] \quad [2,0] \quad [0,1] \quad [1,1] \quad [2,1] \quad [0,2] \quad [1,2] \quad [2,2] \quad [0,3] \quad [1,3] \quad [2,3]]$$

# MULTIDIMENSIONAL ARRAYS

```
>>> y=arange(8)  
>>> y  
array([0, 1, 2, 3, 4, 5, 6, 7])
```

```
>>> y.shape = (2,2,2)  
>>> y  
array( [[[0, 1],  
        [2, 3]],  
  
        [[4, 5],  
        [6, 7]]] )
```

```
>>> y = arange(12).reshape((3,4))  
>>> y  
array([[ 0,  1,  2,  3],  
       [ 4,  5,  6,  7],  
       [ 8,  9, 10, 11]])
```

```
>>> y  
array( [[[0, 1],  
        [2, 3]],
```

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

```
>>> y[0,0,0]
```

```
0
```

```
>>> y[0,0,1]
```

```
1
```

```
>>> y[1,0,0]
```

```
4
```

```
>>> y[1,1,1]
```

```
7
```

# ARRAY OPERATIONS

```
>>> A = array( [[1,2], [3,4]] )  
>>> w = array( [1,3] )  
  
>>> A  
array([[1, 2],  
       [3, 4]])  
  
>>> w  
array([1, 3])
```

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

$$w = \begin{bmatrix} 1 \\ 3 \end{bmatrix} \equiv [1 \quad 3]$$

```
>>> A*w  
array([[ 1,  6],  
       [ 3, 12]])
```

$$A * w = ?$$

```
>>> dot(A,w)  
array([ 7, 15])
```

$$= A \cdot w$$

```
>>> dot(w,A)  
array([10, 14])
```

$$= w \cdot A$$

```
>>> dot(w,w)  
10
```

$$= w \cdot w$$

$$w = [1 \quad 3] \rightarrow \begin{bmatrix} 1 & 3 \\ ? & ? \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 3 \\ 1 & 3 \end{bmatrix}$$

# ARRAY MANIPULATIONS

```
>>> A = arange(12).reshape(3,4)
>>> print A
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]  
  
>>> A[:2,1:3]
array([[1, 2],
       [5, 6]])  
  
>>> print A[:2,1:3]
[[1 2]
 [5 6]]  
  
>>> print A[:,2]
[2 6 10]  
  
>>> print A[0,:]
[0 1 2 3]
```

```
>>> v = array( [3,4,5] )
>>> w = array( [12,11,10] )
>>> print v
[3 4 5]
>>> print w
[12 11 10]  
  
>>> print( column_stack( (v,w) ) )
[[ 3 12]
 [ 4 11]
 [ 5 10]]  
  
>>> print( row_stack( (v,w) ) )
[[ 3  4  5]
 [12 11 10]]  
  
>>> print( hstack( (v,w) ) )
[ 3  4  5 12 11 10]  
  
>>> print( vstack( (v,w) ) )
[[ 3  4  5]
 [12 11 10]]
```

# MATRIX OBJECTS

```
from numpy import matrix
from numpy import linalg

A = matrix( [[1,2,3],[11,12,13],[21,22,23]])      # Creates a matrix.
x = matrix( [[1],[2],[3]] )                          # Creates a matrix (like a column vector).
y = matrix( [[1,2,3]] )                            # Creates a matrix (like a row vector).

Print( A.T )          # Transpose of A.
Print( A*x )          # Matrix multiplication of A and x.
Print( A.I )          # Inverse of A.

Print( linalg.solve(A, x) )    # Solve the linear equation system.
```

# MATRIX OBJECTS

```
>>> A=matrix([[1,2],[3,4]])  
>>> A  
matrix([[1, 2],  
       [3, 4]])  
  
>>> B=matrix([[2,3],[5,7.]])  
>>> B  
matrix([[ 2.,  3.],  
       [ 5.,  7.]])  
  
>>> A*B  
matrix([[ 12.,  17.],  
       [ 26.,  37.]])  
  
>>> B*A  
matrix([[ 11.,  16.],  
       [ 26.,  38.]])  
  
>>> _.dtype  
dtype('float64')
```

```
>>> C = array(A)  
  
>>> C is A  
False  
  
>>> mat(C)  
matrix([[1, 2],  
       [3, 4]])  
  
>>> C  
array([[1, 2],  
       [3, 4]])  
  
>>> v = mat([[1],[3]])  
>>> v  
matrix([[1],  
       [3]])  
  
>>> A*v  
matrix([[ 7],  
       [15]])
```

# SCIPY / MATPLOTLIB

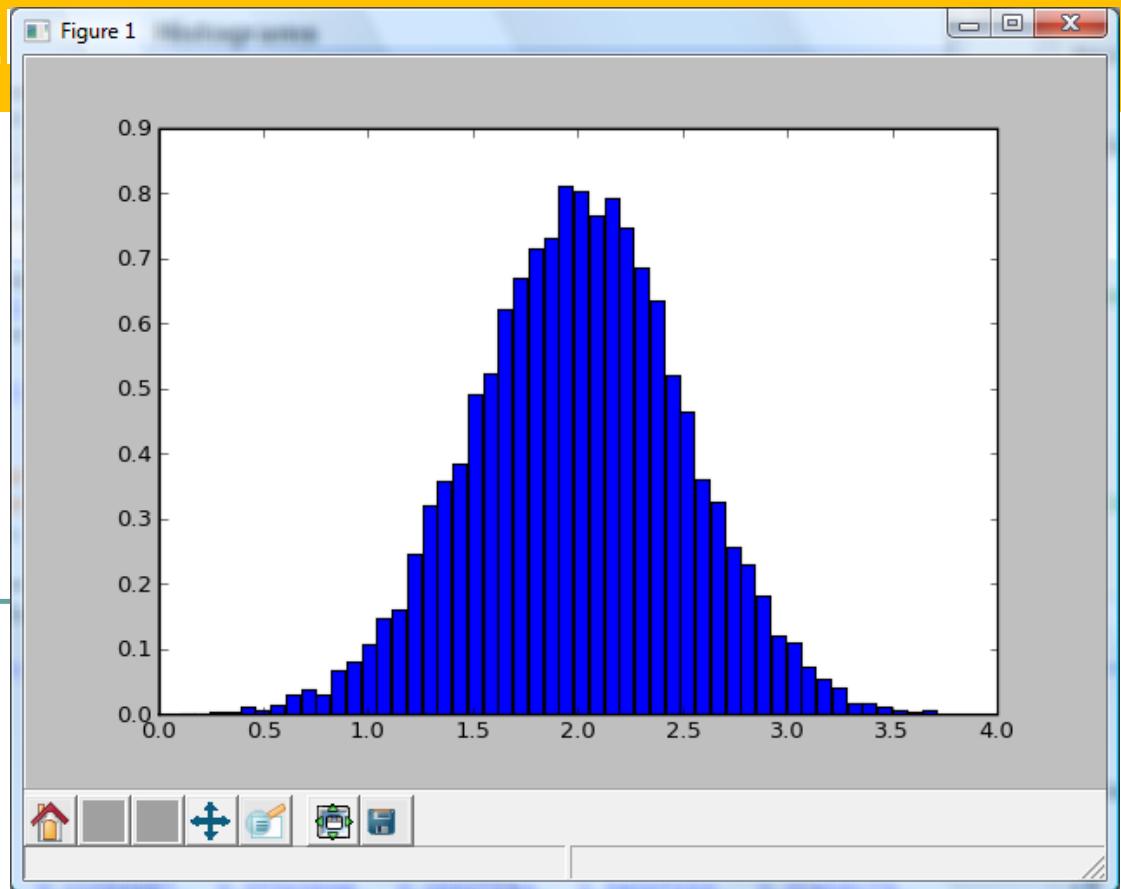
- Linear algebra  
(eig, inverse, ...)
- Statistics

```
import numpy  
from matplotlib import pyplot
```

```
mu, sigma = 2, 0.5
```

```
v = numpy.random.normal(mu,sigma,10000)
```

```
pyplot.hist(v, bins=50, normed=1)      # matplotlib version (plot)  
pyplot.show()
```



# ORGANIZING YOUR CODE

## Simple code – Single file

### ■ Structure

1. Imports first!
2. Global functions, if any, second!  
(I usually avoid these and make them a class)
3. Class definitions third!
4. def main(): second to last
5. Execute main() as last statement



# ORGANIZING YOUR CODE

## Complex code – Multiple files

- One file per function (or class)
- Name file the same as the class makes things easier
- Always have a main.py (not necessary but smart)
- Document your code

## PART I.2:

Time	Topic
9:00-9:45 AM	Python Data Types
<b>9:45-10:15 AM</b>	<b>Object oriented programming in Python</b>
10:15-10:30 AM	Break
10:30-11:30 AM	File IO and data processing
11:30-1:00 PM	Lunch
1:00-2:00 PM	Data visualization in Python
2:00-3:30 PM	Programming exercise
3:30-3:40 PM	Break
3:40-4:30 PM	Subprocesses and interaction with the OS
4:30-5:00 PM	Exercise
5:00 PM	Adjourn for the day

# PROCEDURAL PROGRAMMING

## ■ Functions

```
function val = my_function(x1, x2, ...)
```

$$f(x) : \mathbb{R}^n \rightarrow \mathbb{R}$$

- Has a return value

## ■ Procedures

```
Procedure my_procedure(x1, x2, ...)
```

- Does not have a return value

## ■ Python / C++

- All functions and procedures have return value



# PROCEDURAL PROGRAMMING

- Functions/Procedures require a certain data structure to work on
  - Difficult to reuse procedures in different codes
  - Problems with procedures from different packages = different data structure
  - Data exposed in code = can be modified by mistake
  - Package upgrades may require changes in the data structure, thus affecting the entire code
  - Difficulty developing code in a group effort

# OBJECT ORIENTED PROGRAMMING (OOP)

- Objects control both data and methods
  - Data
    - Private vs. public
  - Methods
    - Functions and procedures to manipulate own data
- Key ideas
  - Data integrity
    - User has no direct access to data
  - Easy upgrade without changes to the main code
    - Objects have a well defined interface
    - Access to data through access functions

# OBJECT ORIENTED PROGRAMMING (OOP)

## Implementation

```
""  
Created on Apr 17, 2009  
  
@author: Peter Mackenzie  
"  
  
class MyClass (object):  
    ""  
  
    classdocs  
    "  
  
    def __init__(self, params):  
        ""  
        Constructor  
        "  
  
        # variables  
        self.var1 = _some_default_value_  
  
    def my_method1(self, params):  
        pass
```

Class definition

Class constructor

Class variables (public)

Class method

# OBJECT ORIENTED PROGRAMMING (OOP)

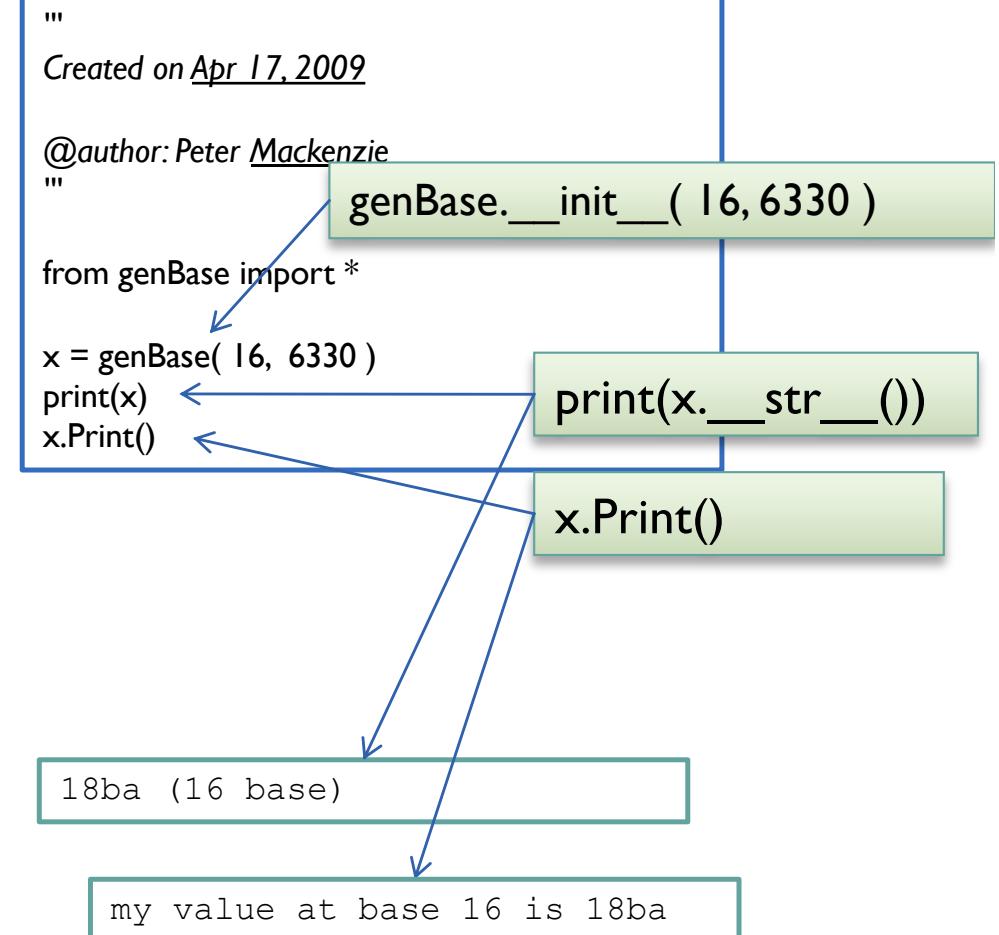
```
import problem1 as p

class genBase(object):
    """
    classdocs
    """

    def __init__(self, base = 10, value = 0):
        """
        Constructor
        """
        self.base = base
        self.value = p.to_base(base, value)

    def __str__(self):
        s = "{} ({}) base)".format(self.value,
self.base)
        return s

    def Print(self):
        print("my value at base {} is
{}".format(self.base, self.value))
```



# OBJECT ORIENTED PROGRAMMING (OOP)

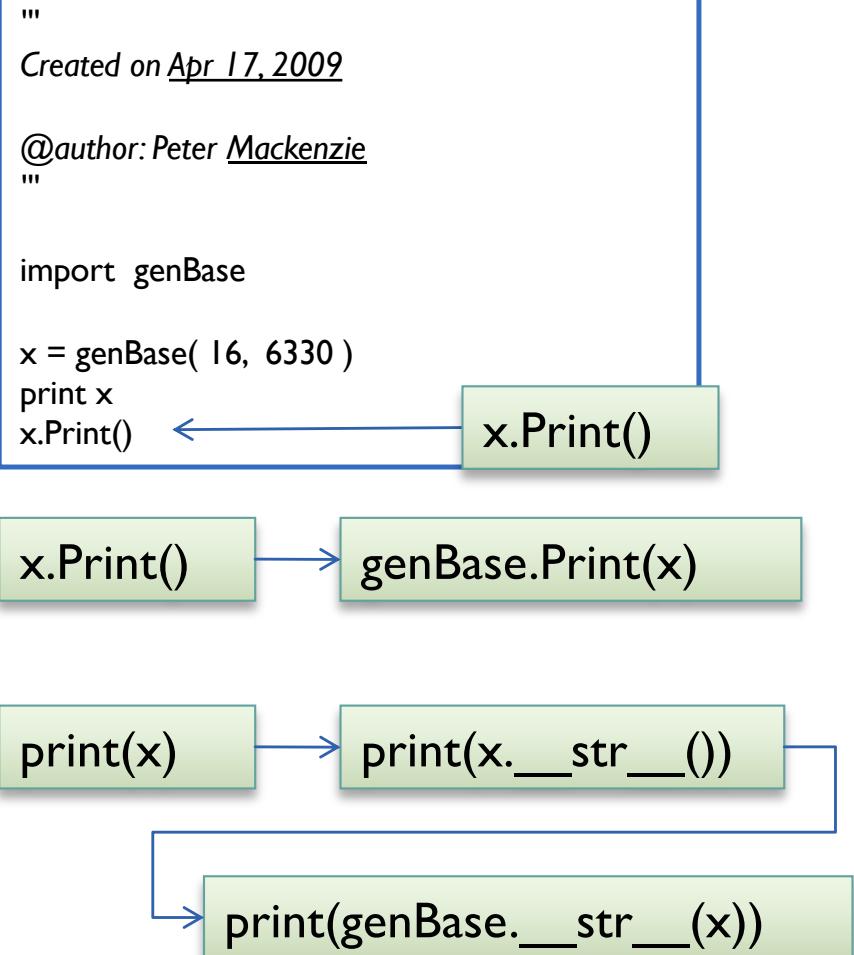
```
import problem1 as p

class genBase(object):
    """
    classdocs
    """

    def __init__(self, base = 10, value = 0):
        """
        Constructor
        """
        self.base = base
        self.value = p.to_base(base, value)

    def __str__(self):
        s = "{} ({}) base)".format(self.value,
self.base)
        return s

    def Print(self):
        print("my value at base {} is
{}".format(self.base, self.value))
```



# OBJECT ORIENTED PROGRAMMING (OOP)

```
import problem1 as p

class genBase(object):
    """
    classdocs
    """

    def __init__(self, base = 10, value = 0):
        """
        Constructor
        """
        self.base = base
        self.value = p.to_base(base, value)

    def __str__(self):
        s = "{} ({})".format(self.value, self.base)
        return s

    def Print(self):
        print("my value at base {} is {}".format(self.base, self.value))
```

```
import problem1 as p1
import problem2 as p2

class genBase(object):

    def __init__(self, base = 10, value = 0):
        """
        Constructor
        """
        v = p1.to_base(base, value)
        self.data = { 'BASE': base, 'VALUE': v }

    def __str__(self):
        s = "{} ({})".format(self.data['VALUE'], self.data['BASE'])
        return s

    def Print(self):
        print("my value at base {} is {}".format(self.data['BASE'], self.data['VALUE']))
```

# OBJECT ORIENTED PROGRAMMING (OOP)

## ■ More features: Inheritance

- Classes can be derived from one or more base classes
- The new class may
  - Use inherited data and methods
  - Add new data (variables)
  - Add new methods
  - Overload methods
  - Overload operators

# OOP: INHERITANCE

## Class definition

```
class Animal (object):
    def __init__(self, name='unknown'):
        self.name = name

    def sound(self):
        print("{} is silent".format(self.name))
```

```
class Cat (Animal):
    def sound(self):
        print("{} is meowing.format(self.name))

class Dog (Animal):
    def sound(self):
        print("{} is barking".format(self.name))
```

## Code example

```
from Animals import *

a = Animal('Hans')
b = Cat('Cleo')
c = Dog('Foster')
```

Cat.\_\_init\_\_(self, name) = Animal.\_\_init\_\_(self, name)  
Cat.sound(self, name) = Animal.sound(self, name)

Cat.\_\_init\_\_(self, name) = Animal.\_\_init\_\_(self, name)  
Cat.sound(self, name) = independent

## Class definition

```
class Animal (object):
    def __init__(self, name='unknown'):
        self.name = name

    def sound(self):
        print("{} is silent".format(self.name))

class Cat (Animal):
    def sound(self):
        print("{} is meowing".format(self.name))

class Dog (Animal):
    def sound(self):
        print("{} is barking".format(self.name))
```

## Code example

```
from Animals import *

a = Animal('Hans')
b = Cat('Cleo')
c = Dog('Foster')

a.sound()
b.sound()
c.sound()
```



# QOP: INHERITANCE

Class definition

Code example

```
class Animal (object):
    def __init__(self, name='unknown'):
        self.name = name

    def sound(self):
        print("{} is silent".format(self.name))
```

```
class Cat (Animal):
    def sound(self):
        print("{} is meowing".format(self.name))
```

```
class Dog (Animal):
    def sound(self):
        print("{} is barking".format(self.name))
```



```
from Animals import *

a = Animal('Hans')
b = Cat('Cleo')
c = Dog('Foster')

a.sound()
b.sound()
c.sound()
```



# OBJECT ORIENTED PROGRAMMING (OOP)

## ■ Overloading Methods

- Methods from a base class may be either used as is (inheritance) or replaced by an alternative method of identical name (overloading)
- Examples:
  - `Animal.__init__()`
  - `Animal.sound() – Cat.sound() – Dog.sound()`
- The object class has standard methods which can be overloaded as well

# OBJECT ORIENTED PROGRAMMING (OOP)

## ► Overloading Default Methods

- ▶ `__init__(self, args)` Constructor
  - ▶ Set defaults / initiate variables
- ▶ `__str__(self)` convert self to string
  - ▶ Used in print statements
- ▶ `__repr__(self)` represent description of self as string (used for debugging)
- ▶ `__len__(self)` define/return length of object
- ▶ `__getitem__(self, i)` get indexed item
  - ▶ Needed for sequences (similar to list or tuple)

# OBJECT ORIENTED PROGRAMMING (OOP)

## ■ Example:

- Use a list to represent a vector and implement vector addition, subtraction, and the dot product

Default  
list behavior :

```
>>> a = [1,2,3]
>>> b = [4,6,8]
>>> a
[1, 2, 3]
>>> b
[4, 6, 8]
>>> a + b
[1, 2, 3, 4, 6, 8]
>>> b + a
[4, 6, 8, 1, 2, 3]
>>> a - b
```



Traceback (most recent call last):  
File "<pyshell#6>", line 1, in <module>  
 a - b  
TypeError: unsupported operand type(s) for -: 'list' and 'list'

# OOP: OPERATOR OVERLOAD

## Class definition

```
class Vector(object):

    def __init__(self, data):
        self.v = data

    def __str__(self):
        return str(self.v)

    def __len__(self):
        return len(self.v)

    def __getitem__(self, i):
        return self.v[i]

    def __div__(self, x):
        raise NotImplemented
```

## Code example

```
from Vector import *

# define two vectors
v = Vector([1, 2, 3])
w = Vector([4, 6, 8])

print("v: {}".format(v))
print("w: {}".format(w))
```

```
v: [1, 2, 3]
w: [4, 6, 8]
```

# OBJECT ORIENTED PROGRAMMING (OOP)

## ■ Emulating numeric types: operator overload

- `__add__(self, v)`    return self + v
- `__sub__(self, v)`    return self - v
- `__mul__(self, v)`    return self \* v
- `__div__(self, v)`        return self / v
- `__pow__(self, v)`    return self \*\* v
- `__floordiv__(self, v)`    return self // v
- `__mod__(self, v)`    return self % v

# OOP: OPERATOR OVERLOAD

## Class definition

```
class Vector(object):

    def __init__(self, data):
        self.v = data

    def __str__(self):
        return str( self.v )

    def __len__(self):
        return len(self.v)

    def __getitem__(self, i):
        return self.v[i]

    def __div__(self, x):
        raise NotImplemented
```

```
def __add__(self, x):
    if type(x) == Vector:
        w = self.v[:]
        for i in range(0,len(self)):
            w[i] += x[i]
        return Vector(w)
    else:
        raise TypeError

    .
    .
    .
```

copy

# OOP: OPERATOR OVERLOADING

## Class definition

```
class Vector(object):

    def __init__(self, data):
        self.v = data

    def __str__(self):
        return str( self.v )

    def __len__(self):
        return len(self.v)

    def __getitem__(self, i):
        return self.v[i]

    def __div__(self, x):
        raise NotImplemented
```

```
def __add__(self, x):
    if type(x) == Vector:
        w = self.v[:]
        for i in range(0,len(self)):
            w[i] += x[i]
        return Vector(w)
    else:
        raise TypeError
```

```
def __sub__(self, x):
    if type(x) == Vector:
        w = self.v[:]
        for i in range(0,len(self)):
            w[i] -= x[i]
        return Vector(w)
    else:
        raise TypeError
```

```
def __mul__(self, x):
    if type(x) == Vector:
        w = 0.0
        for i in range(0,len(self)):
            w += self.v[i] * x[i]
        return w
    else:
        raise TypeError
```

copy

copy

# OOP: INHERITANCE

## Code example

```
from Vector import *\n\n# define two vectors\nv = Vector([1, 2, 3])\nw = Vector([4, 6, 8])\n\nprint("v: {}".format(v))\nprint("w: {}".format(w))\n\nx = v + w\nprint("v + w: {}".format(x))\n\ny = v - w\nprint("v - w: {}".format(y))\n\nz = v * w\nprint("v * w: {}".format(z))
```

## Output

v: [1, 2, 3]  
w: [4, 6, 8]

v + w: [5, 8, 11]

v - w: [-3, -4, -5]

v \* w: 40.0

# OOP: INHERITANCE

## Code example

```
# from Vector import *
from VectorNoisy import *

# define two vectors
v = Vector([1, 2, 3])
w = Vector([4, 6, 8])

print("v: {}".format(v))
print("w: {}".format(w))

x = v + w
print("v + w: {}".format(x))

y = v - w
print("v - w: {}".format(y))

z = v * w
print("v * w: {}".format(z))
```

```
entering __init__()
entering __init__()
entering __str__()
v: [1, 2, 3]
entering __str__()
w: [4, 6, 8]
entering __add__()
entering __len__()
entering __getitem__()
entering __getitem__()
entering __getitem__()
entering __str__()
v + w: [5, 8, 11]
entering __sub__()
entering __len__()
entering __getitem__()
entering __getitem__()
entering __getitem__()
entering __str__()
v - w: [-3, -4, -5]
entering __mul__()
entering __len__()
entering __getitem__()
entering __getitem__()
entering __getitem__()
entering __str__()
v * w: 40.0
```

# OBJECT ORIENTED PROGRAMMING (OOP)

## ■ Comparison: operator overload

- `__lt__(self, v)`      return `self < v`
- `__le__(self, v)`      return `self <= v`
- `__eq__(self, v)`      return `self == v`
- `__ne__(self, v)`      return `self != v`
- `__gt__(self, v)`      return `self > v`
- `__ge__(self, v)`      return `self >= v`
  
- `__nonzero__(self)`      return True or False

# OBJECT ORIENTED PROGRAMMING (OOP)

## ■ Emulating numeric types (binary action): operator overload

- `__and__(self, v)`      return `self & v`
- `__or__(self, v)`        return `self | v`
- `__xor__(self, v)`      return `self ^ v`
- `__lshift__(self, v)`     return `self << v`
- `__rshift__(self, v)`     return `self >> v`

## PART 2:

Time	Topic
9:00-9:45 AM	Python Data Types
9:45-10:15 AM	Object oriented programming in Python
10:15-10:30 AM	Break
10:30-11:30 AM	File IO and data processing
11:30-1:00 PM	Lunch
1:00-2:00 PM	Data visualization in Python
2:00-3:30 PM	Programming exercise
3:30-3:40 PM	Break
3:40-4:30 PM	Subprocesses and interaction with the OS
4:30-5:00 PM	Exercise
5:00 PM	Adjourn for the day

## EXCEPTION HANDLING

NO ERROR SHOULD  
HAPPEN WITHOUT  
PROPER  
EXCEPTION  
HANDLING!

# EXCEPTION HANDLING

## ■ Built in form

```
x = 1/0
```

Traceback (most recent call last):

File "C:\Users\Peter\workspace\Exceptions\src\demo.py", line 10, in <module>

  x = 1/0

ZeroDivisionError: integer division or modulo by zero

## ■ Handling the run-time error

```
try:  
    x = 1/0  
except ZeroDivisionError:  
    print("a division by zero occured")  
    raise
```

a division by zero occurred

Traceback (most recent call last):

File "C:\Users\Peter\workspace\Exceptions\src\demo.py", line 11, in <module>

  x = 1/0

ZeroDivisionError: integer division or modulo by zero

# EXCEPTION HANDLING

## ■ Exception handling

```
try:  
    a, b = 1, 0  
    x = a / b  
except ZeroDivisionError:  
    print("a division by zero occured")  
    raise
```

## ■ Traditional

```
import sys  
  
a, b = 1, 0  
  
if b == 0:  
    print("a division by zero occured")  
    sys.exit()  
else:  
    x = a / b
```

# EXCEPTION HANDLING

## ■ Exception class

```
class MyException(Exception):
    def __init__(self, val):
        self.value = val
    def __str__(self):
        return repr(self.value)

try:
    raise MyException('messed up')
except MyException, e:
    print("Exception triggered:", e.value)
```

Exception triggered: messed up

# EXCEPTION HANDLING

## Common application: file I/O

### ■ Create a file object

- `f = open(filename, mode)`
- Mode = ‘r’ , ‘w’ , ‘a’ , ’ r+’ , ‘rb’ , ‘wb’
- Raises an Exception if open fails

### ■ Read from a file object

- `line = f.readline()`

### ■ Close file object on completion

- `f.close()`

# EXCEPTION HANDLING COMMON APPLICATION: FILE I/O

```
import sys

try:
    f = open('somefile.txt','r')
except IOError:
    print("could not open file for reading")
    try:
        print("creating file ... ", end="")
        f = open('somefile.txt','w')
        f.write( 'Peter was here ;-)')
        f.close()
        f = open('somefile.txt','r')
        print("success")
    except IOError:
        print("failed")
        sys.exit()

for line in f:
    print(line)

f.close()
```

1<sup>st</sup> run (file does not exist)

could not open file for reading  
creating file ... success  
Peter was here ;-)

2<sup>nd</sup> run (file exists)

Peter was here ;-)

# EXCEPTION HANDLING

## ■ General form of the try statement

```
try:  
    statement(s)  
except Error1:  
    handling1  
except (Error2, Error3):  
    handling2  
else:  
    handing_the_unknown  
    raise  
finally:  
    execute_after_all_exceptions  
    cleanup_statements
```

# STRING MANIPULATION

## ■ String Methods:

- <https://docs.python.org/3.6/library/stdtypes.html#text-sequence-type-str>
- <https://docs.python.org/3.6/library/stdtypes.html#string-methods>

## ■ String Services:

- <https://docs.python.org/3.6/library/string.html>

# PARSING A STRING

## ■ `string.split(char)`

Escape Sequence	Meaning
<code>\newline</code>	Ignored
<code>\\"</code>	Backslash (\)
<code>\'</code>	Single quote (' )
<code>\"</code>	Double quote (" )
<code>\a</code>	ASCII Bell (BEL)
<code>\b</code>	ASCII Backspace (BS)
<code>\f</code>	ASCII Formfeed (FF)
<code>\n</code>	ASCII Linefeed (LF)
<code>\r</code>	ASCII Carriage Return (CR)
<code>\t</code>	ASCII Horizontal Tab (TAB)
<code>\v</code>	ASCII Vertical Tab (VT)
<code>\ooo</code>	ASCII character with octal value <i>ooo</i>
<code>\xhh...</code>	ASCII character with hex value <i>hh...</i>

# PARSING A STRING

- `string.strip()`
- `string.strip("")`
- `string.rstrip()`
- `string.lstrip()`
  
- `string.find('substring')`
- `if substring in string: ...`
  
- `string.replace('search string', 'replace string')`

# ALTERNATIVE TOOLS

- Export excel as CSV (comma separated values)
- csv.py
  - <https://docs.python.org/3/library/csv.html>
  - Provides parsing and splitting and cleaning in a module

# EXERCISE

- Go to Python/Exercises/ParsingFiles
- Write a code that
  - 1. Open the scores.csv file
  - 2. Sums up all the values in a line and stores the sum to a list ( `rowSum.append()` )
  - 3. Computes the column sums (`colSum`)
  - 4. Closes the file
  - 5. compute the sum of all `rowSums` and the sum of all `colSums`
  - 6. Show that both sums yield the same result (`print`)

# EXERCISE

## I. **Create a text file**

- (using notepad) that contains at least 3 lines. (name it something like ‘MySource.txt’)

## 2. **Create a program that**

- Opens that text file, reads and print it line per line
- Opens a file named ‘MyCopy.txt’ in which it writes every line it reads from your file.
- Design and implement exception handling for ‘cannot open file for reading’ and ‘cannot open file for writing’.
- Test your exception handling by triggering these exceptions (remove source file, try placing a copy in the directory where you want to write)

# PART 3.I:

Time	Topic
9:00-9:45 AM	Python Data Types
9:45-10:15 AM	Object oriented programming in Python
10:15-10:30 AM	Break
10:30-11:30 AM	File IO and data processing
11:30-1:00 PM	Lunch
<b>1:00-2:00 PM</b>	<b>Data visualization in Python</b>
2:00-3:30 PM	Programming exercise
3:30-3:40 PM	Break
3:40-4:30 PM	Subprocesses and interaction with the OS
4:30-5:00 PM	Exercise
5:00 PM	Adjourn for the day

# MATPLOTLIB

## ■ WHAT IS MATPLOTLIB?

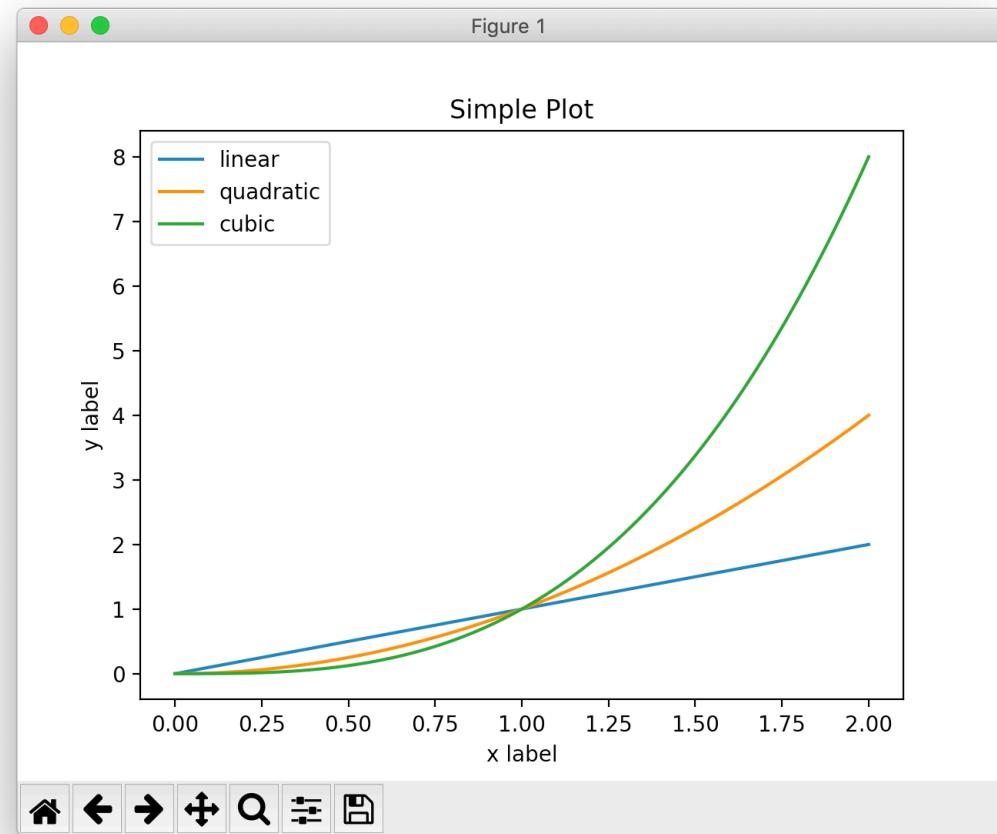
- An OpenSource library for MATLAB-like plotting
- Provides a simple to use widget – pyplot – that acts like figure() in MATLAB
- Also provides a large set of functions to customize plotting
- Integration into GUI (PyQt5)
- Provides plotting without a display (in memory, direct to file)

## ■ HOW TO GET STARTED?

- <https://matplotlib.org/tutorials/index.html>

# LINE GRAPHICS

```
import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.linspace(0, 2, 100)  
  
plt.plot(x, x, label='linear')  
plt.plot(x, x**2, label='quadratic')  
plt.plot(x, x**3, label='cubic')  
  
plt.xlabel('x label')  
plt.ylabel('y label')  
  
plt.title("Simple Plot")  
  
plt.legend()  
  
plt.show()
```



# LINE GRAPHICS – SAVING TO FILE

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

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()

plt.show()
```

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

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()

# plt.show()

plt.savefig('ex2.png')
plt.savefig('ex2.jpg')
plt.savefig('ex2.pdf')
plt.savefig('ex2b.png', dpi=300)
```

# LINE GRAPHICS

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

def my_plotter(ax, data1, data2, param_dict):
    """
    A helper function to make a graph

    Parameters
    ax : Axes. The axes to draw to
    data1 : array. The x data
    data2 : array. The y data
    param_dict : dict. Dictionary of kwargs to pass to ax.plot

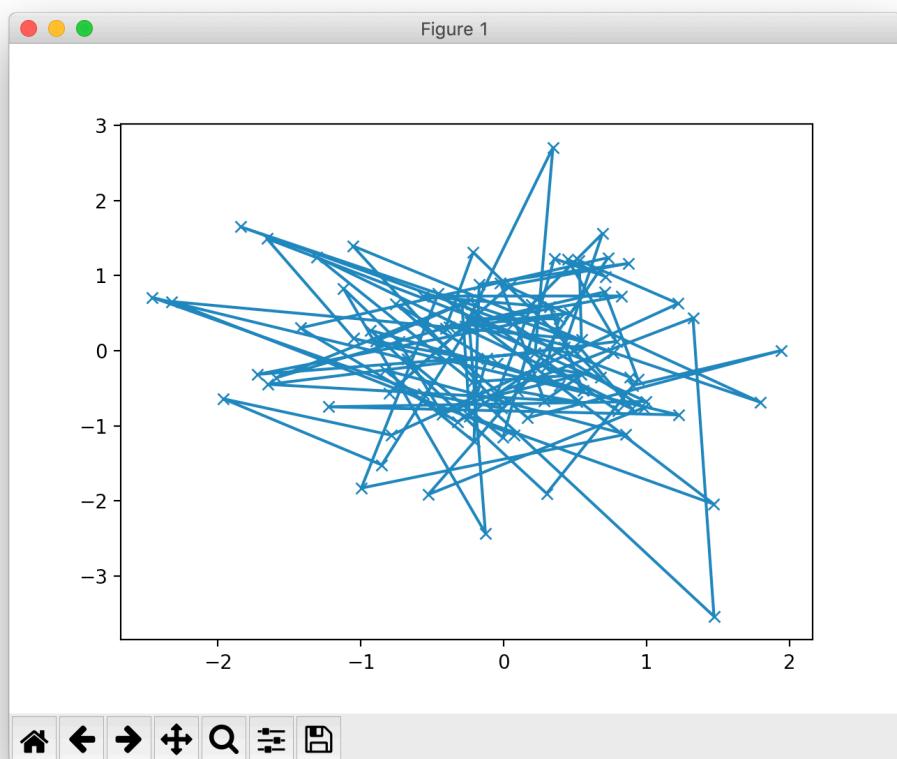
    Returns
    out : list. list of artists added
    """

    out = ax.plot(data1, data2, **param_dict)
    return out

# which you would then use as:

data1, data2 = np.random.randn(2, 100)
fig, ax = plt.subplots(1, 1)
my_plotter(ax, data1, data2, {'marker': 'x'})

plt.show()
```



# LINE GRAPHICS

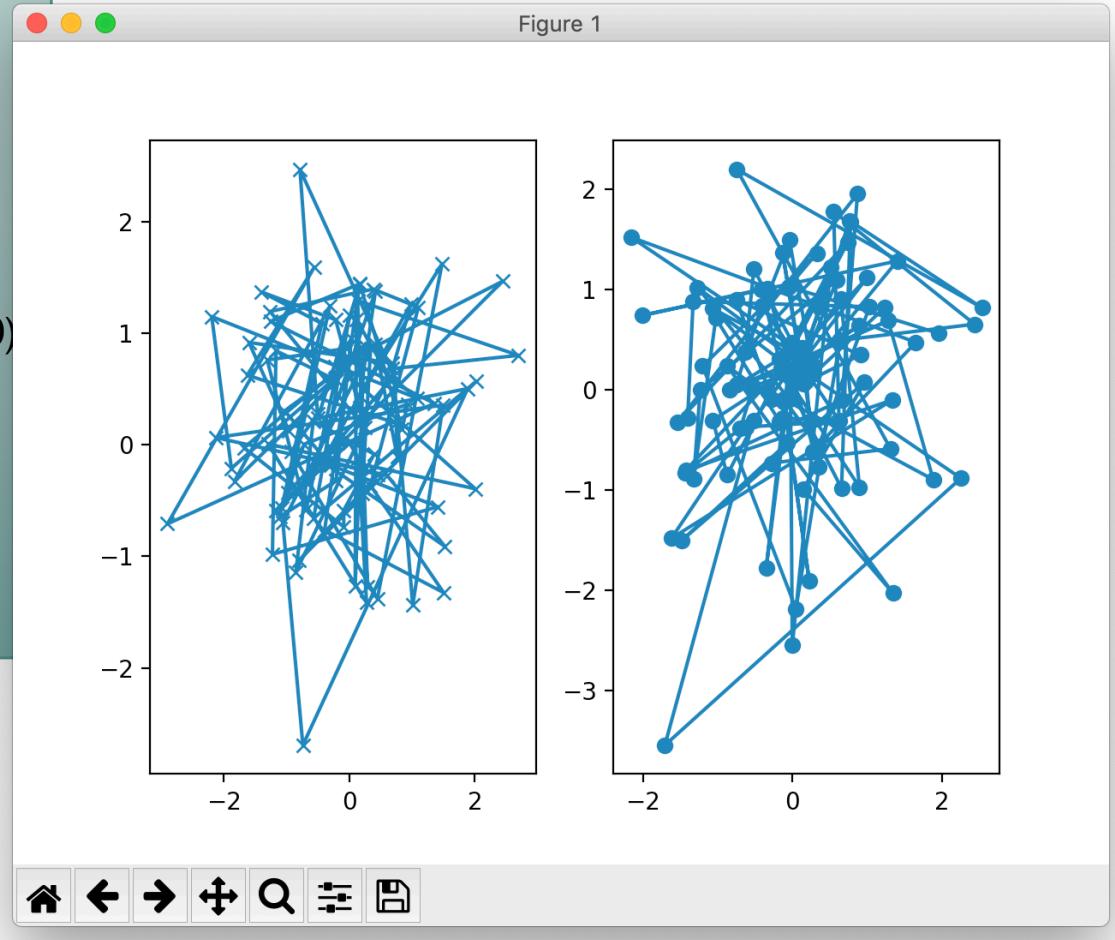
```
import numpy as np
import matplotlib.pyplot as plt

def my_plotter(ax, data1, data2, param_dict):
    out = ax.plot(data1, data2, **param_dict)
    return out

# which you would then use as:

data1, data2, data3, data4 = np.random.randn(4, 100)
fig, (ax1, ax2) = plt.subplots(1, 2)
my_plotter(ax1, data1, data2, {'marker': 'x'})
my_plotter(ax2, data3, data4, {'marker': 'o'})

plt.show()
```



# PLOTTING W/O DISPLAY – BACKENDS

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

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')

plt.title("Simple Plot")

plt.legend()

plt.show()
```

```
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

x = np.linspace(0, 2, 100)

plt.plot(x, x, label='linear')
plt.plot(x, x**2, label='quadratic')
plt.plot(x, x**3, label='cubic')

plt.xlabel('x label')
plt.ylabel('y label')
plt.title("Simple Plot")
plt.legend()

ext = 'png'
filename = 'ex5.' + ext
plt.savefig(filename)
```

## PART 3.2:

Time	Topic
9:00-9:45 AM	Python Data Types
9:45-10:15 AM	Object oriented programming in Python
10:15-10:30 AM	Break
10:30-11:30 AM	File IO and data processing
11:30-1:00 PM	Lunch
1:00-2:00 PM	Data visualization in Python
<b>2:00-3:30 PM</b>	<b>Programming exercise</b>
3:30-3:40 PM	Break
3:40-4:30 PM	Subprocesses and interaction with the OS
4:30-5:00 PM	Exercise
5:00 PM	Adjourn for the day



# EXERCISES

## PART 4.I:

Time	Topic
9:00-9:45 AM	Python Data Types
9:45-10:15 AM	Object oriented programming in Python
10:15-10:30 AM	Break
10:30-11:30 AM	File IO and data processing
11:30-1:00 PM	Lunch
1:00-2:00 PM	Data visualization in Python
2:00-3:30 PM	Programming exercise
3:30-3:40 PM	Break
3:40-4:30 PM	Subprocesses and interaction with the OS
4:30-5:00 PM	Exercise
5:00 PM	Adjourn for the day

# SUBPROCESS MANAGEMENT

- The [subprocess](#) module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.
- <https://docs.python.org/3.6/library/subprocess.html>
- <https://pymotw.com/2/subprocess/> was helpful

# SUBPROCESS – RUN()

```
subprocess.run(['ls','-l'])
```

```
$ python3 list.py
total 8
-rw-r--r-- 1 pmackenz staff 47 Jul 23 15:33 list.py
```

```
subprocess.run(args, *, stdin=None, input=None, stdout=None, stderr=None, shell=False,
cwd=None, timeout=None, check=False, encoding=None, errors=None, env=None)
```

# SUBPROCESS – POPEN()

```
import subprocess

p = subprocess.Popen(['ls', '-l'], stdout=subprocess.PIPE)

res = p.communicate()[0]

s = res.decode(encoding='utf-8', errors='strict')
slist = s.strip().split('\n')

cnt = 0

for item in slist:
    print("{}: {}".format(cnt, item))
    cnt += 1
```

# SUBPROCESS – POPEN()

```
import subprocess

p = subprocess.Popen(['greet'], stdin=subprocess.PIPE)

firstname = 'Peter'
lastname = 'Mackenzie-Helnwein'

answers    = (firstname, lastname)
answerString = "\n".join(answers)
answerBytes = answerString.encode(encoding='utf-8', errors='strict')

# send answers to STDIN
p.communicate(answerBytes)
```

# USEFUL LIBRARIES

## ■ `pathlib`

- High-level path manipulation
- <https://docs.python.org/3/library/pathlib.html#module-pathlib>

## ■ `os.path`

- Help with reading, checking, comparing file paths on various OS'
- <https://docs.python.org/3/library/os.path.html>

## PART 4.2:

Time	Topic
9:00-9:45 AM	Python Data Types
9:45-10:15 AM	Object oriented programming in Python
10:15-10:30 AM	Break
10:30-11:30 AM	File IO and data processing
11:30-1:00 PM	Lunch
1:00-2:00 PM	Data visualization in Python
2:00-3:30 PM	Programming exercise
3:30-3:40 PM	Break
3:40-4:30 PM	Subprocesses and interaction with the OS
4:30-5:00 PM	Exercise
5:00 PM	Adjourn for the day



# EXERCISE