Phys 443
Computational Physics I

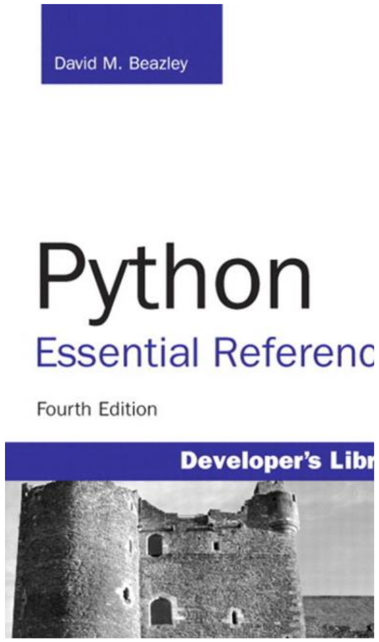
Python: Tutorial

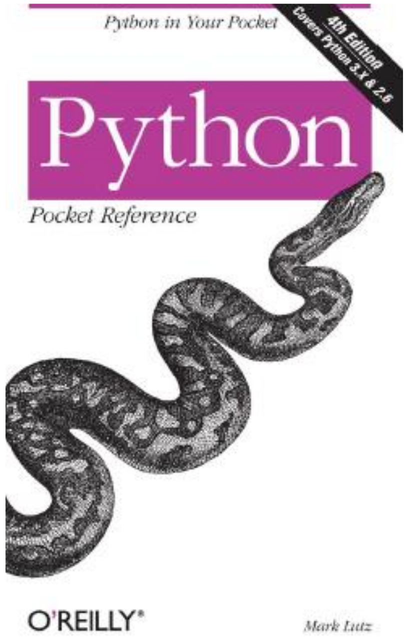
Python Basics

- Python is a high-level programming language
  - Open source and community driven
  - a standard distribution includes many modules
  - Dynamic typed
  - Source can be compiled or run just-in-time
  - Similar to perl, tcl, ruby

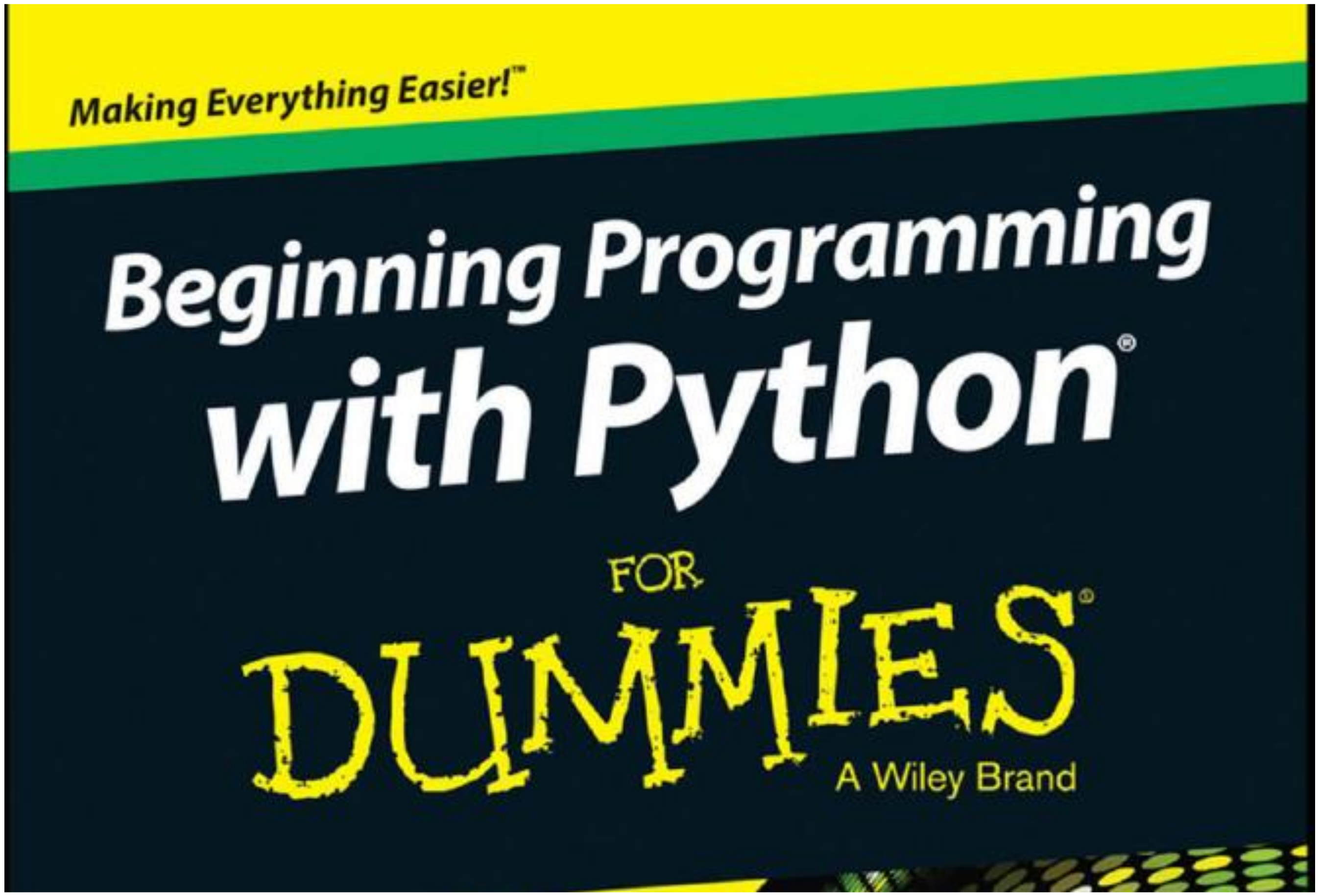
Python Documantation

- Books for reference are:
  - Python: Essential Reference(4th ed.) by David M. Beazley.
  - Python Pocket Reference by Mark Lutz.



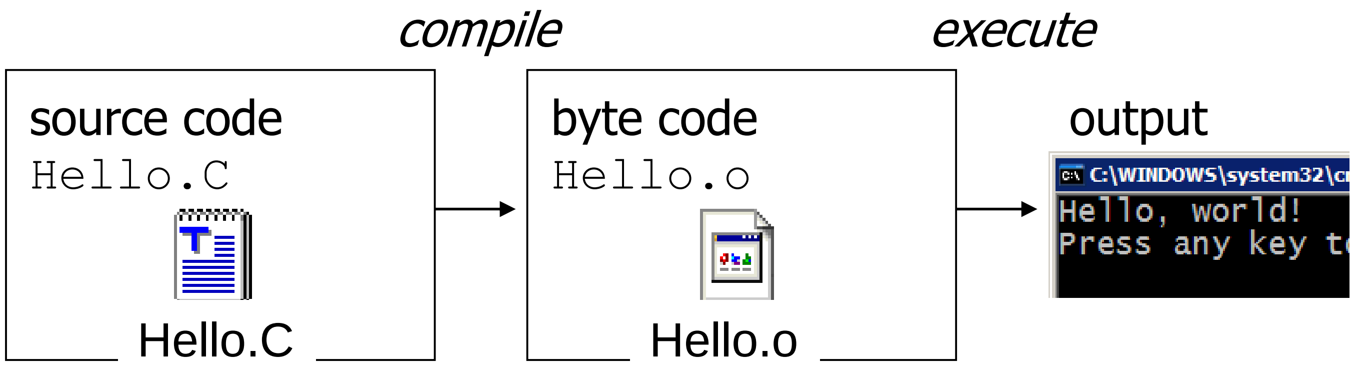


Python Documantation


Making Everything Easier!™
Beginning Programming with Python®
FOR DUMMIES®
A Wiley Brand

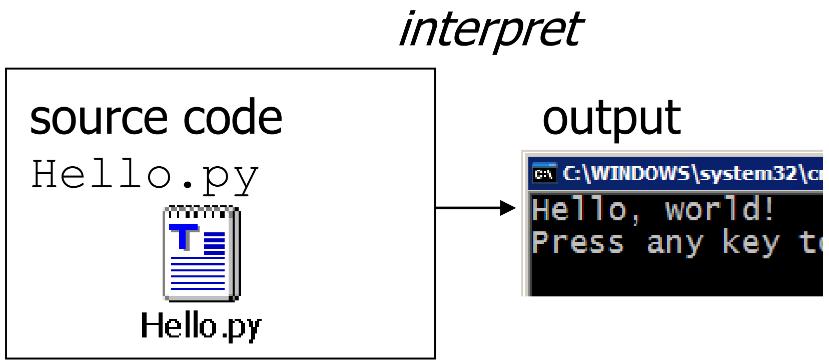
Compiling and interpreting

- Many languages require you to *compile* (translate) your program into a form that the machine understands.



```
Hello, world!
Press any key t
```

- Python is instead directly *interpreted* into machine instructions.



```
Hello, world!
Press any key t
```

Python Basics

- Python is an interpreted programming language with strong dynamic typing.
  - Interpreted: program instructions are executed directly rather than being compiled into machine code; compare to C, C++, .
  - Dynamic Typing: data types of variables (int, float, string, etc.), are determined on the fly as the program runs.
  - Strong Typing: converting a value of one type to another (e.g., string to int) is not done automatically.

Python offers fast and flexible development, and you can copy programs between machines without worrying about compatibility. **But it's slower than compiled Fortran, C, or C++ programs.**

Python Basics

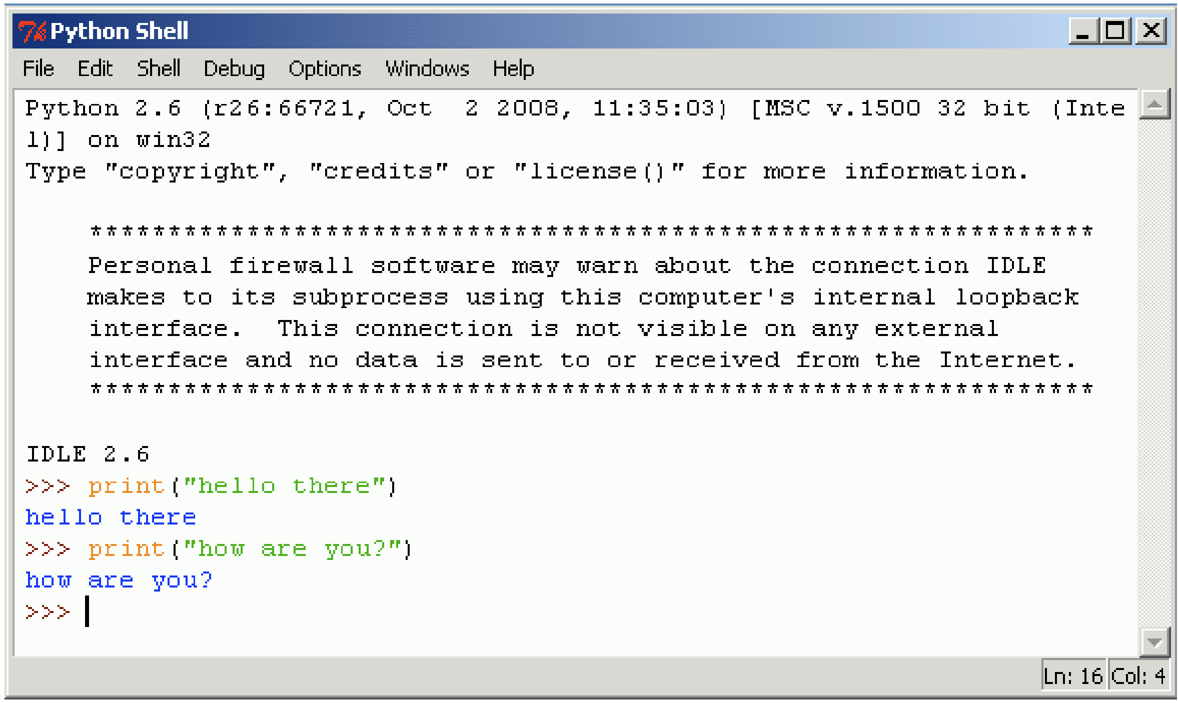
- The interpreter provides an interactive environment to play with the language
- Results of expressions are printed on the screen

```
>>> 3 + 7
10
>>> 3 < 15
True
>>> 'print me'
'print me'
>>> print 'print
me'
print me
>>>
```

Python Basics

- Elements separated by commas print with a space between them
- A comma at the end of the statement (print ‘hello’,) will not print a newline character

```
>>> print ("hello")
hello
>>> print ("hello", "there")
hello there
```



```
Python 2.6 (r26:66721, Oct  2 2008, 11:35:03) [MSC v.1500 32 bit (Inte
l)] on win32
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 2.6
>>> print("hello there")
hello there
>>> print("how are you?")
how are you?
>>> 
```

Python Basics

- Python is an object oriented language
- Practically everything can be treated as an object
- "hello world" is a string
- Strings, as objects, have methods that return the result of a function on the string

Python Basics

- You can run python programs from files, just like perl or shell scripts, by typing "`python program.py`" at the command line. The file can contain just the python commands.
- Or, one can invoke the program directly by typing the name of the file, "program.py", if it has as a first line something like "`#!/usr/bin/python`" (like a shell script... works as long as the file has execute permissions set)
- Alternatively, you can enter a python shell and run python commands interactively, by typing "python"

Python Basics

The ‘#’ starts a line comment

```
# This program prints important messages.
print("Hello, world! ")
print()              # blank line
```

```
>>> 'this will print'
'this will print'
>>> #'this will not'
>>>
```

- The \ character at the end of a line of Python code signifies that the next line is a continuation of the current line.
- One common special character is the newline command, \n, which forces any subsequent text to be printed on a new line.

```
>>> print("Hello \n there.")
Hello
there.
```

Python Basics

- `numpy`: random numbers, arrays, linear algebra.
- `scipy:` statistical tests, special functions, integration, curve fitting, minimization.
- `matplotlib:` plotting: xy plots, error bars, contour plots, histograms.

Python Operators

- + addition
- - subtraction
- / division
- ** exponentiation
- % modulus (remainder after division)
- x//y   integer part of *x* divided by *y*

Python has incorporated operators like +=, but ++ (or --) do not work in Python

```
>>>6*3  <enter>
```

Arithmetic Operators

One can combine the several operators together

- *x+2*y* $\longrightarrow$ $x+2y$
- *3*x**2* $\longrightarrow$ $3x^2$
- *x/2*y* $\longrightarrow$ $\frac{1}{2}(xy)$

- Python cannot solve the equation like
  - *2*x = y*

It should be written as

  - *x = y/2*

Python Arithmatics

- Python understands numbers and standard arithmetic. You can use Python as a very powerful calculator if you want. It can also store value in variables. Try this:

```
>>>x = 4
>>>y = 16
>>>x*y
>>>x**y
>>>y/x
>>>x**y**x
```

- That last one may take a moment : Python is actually calculating the value of $4^{(16)^4}$, which is a rather huge number.

Python Arithmatics

```
>>> 2 + 3*4
>>>     ?
```

If you expected the last answer to be 20

```
>>> (2+3)* 4
```

- Python uses the normal precedence of arithmetic operations: Multiplications and divisions are done before addition and subtraction, unless there are parentheses.

Python Arithmatics

```
>>>2*2
>>>2**3
>>>10%3
>>>1.0/2.0
>>>1/2
```

Output:

```
>>>4
>>>8
>>>1
>>>0.5
>>>0
```

- Note the difference between floating point division and integer division in the last two lines

Arithmetic Operators

Statement like

$x = x+1$

is an assigment. It does not make sense matematically, Python takes current value of x, adds one and stores in variable x.

So consider

```
>>>x=0
>>>print(x)
>>>x=x**2-2
>>>print(x)
```

This won't solve equation $x= x^2-2$ but instead assigns

$0^2-2 = -2$ *to x*

Python Basics

```
>>>print("Hello")
>>>print(3.14159)
>>>print("%s, %.2f" % ("Hello", 3.14159))
```

The resulting output:

```
Hello
3.14159
Hello, 3.14
```

```
x = 4                              # x is an int
print(" x = %d" % x)               # prints 4
print(" 2x = %d" % (2*x))          #    "      8
print("x^2 = %d" % (x**2))         #    "      16
y = 5.                             # y is a float
print(" y = %g" % y)               # prints 5
print("y/2 = %g" % (y/2))          # " 2.5
z = "mystring"                     # z is a string
print(" z = %s" % z)
u = z + x                          # raise a TypeError
```

Arithmetic Operators

There are some short cuts.

x+=1	add one to x
x-=4	subtract 4 from x
x=-2.4*	multiply x by -2.4
*x/=5*y*	divide x by 5y
x//=3.4	divide x by 3.4 and round down

Python: Variables

- Most variables do not need to be declared as real, integer, string, etc.

- The type of a variable is defined by the value assigned to it.

- The type of a variable can change throughout the program execution.

- The opposite of dynamically-typed is statically-typed. Fortran and C are examples of statically-typed languages.

Python: Variables

- Variable name can be longer…
- Type of variables

- **Boolean:** True, False
- **Integers:** positive and negative numbers
- **Float:** real numbers
- **Complex:** complex numbers such as 1+2j, j = √-1
- Integers take less space then the float.
- Floats take less space than the complex

Python: Everything is object

- Everything means everything, including <u>functions</u> and <u>classes</u>
- <u>Data type</u> is a property of the object and not of the variable

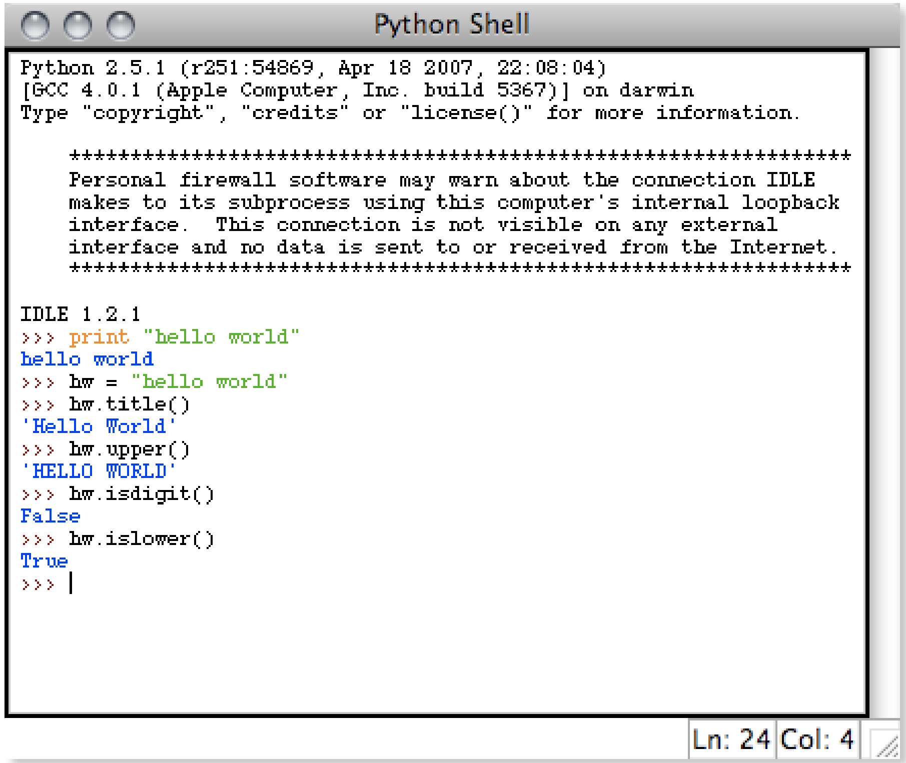
```
>>> x = 7
>>> x
7
>>> x = 'hello'
>>> x
'hello'
>>>
```

String Methods

- The string held in your variable remains the same
- The method returns an altered string
- Changing the variable requires reassignment
  - hw = hw.upper()
  - hw now equals “HELLO WORLD”

String Methods

- Assign a string to a variable
  - In this case “hw”
  - hw.title()
  - hw.upper()
  - hw.isdigit()
  - hw.islower()



```
Python 2.5.1 (r251:54869, Apr 18 2007, 22:08:04)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)] on darwin
Type "copyright", "credits" or "license()" for more information.

    ****************************************************************
    Personal firewall software may warn about the connection IDLE
    makes to its subprocess using this computer's internal loopback
    interface.  This connection is not visible on any external
    interface and no data is sent to or received from the Internet.
    ****************************************************************

IDLE 1.2.1
>>> print "hello world"
hello world
>>> hw = "hello world"
>>> hw.title()
'Hello World'
>>> hw.upper()
'HELLO WORLD'
>>> hw.isdigit()
False
>>> hw.islower()
True
>>> |
```

Arithmetic Operators

Feature of python: it can assign multiple variables in a single line.

- *x,y= 1, 2.5* is the same as
- *x=1*
- *y=2.5*

More complicated example

- *x,y = 2*z+1,(x+y)/3*

All of the right side is evaluated before assigning to the left.

Python: Boolean

- The Boolean data type has two values: True and False (note capitalization).
  - True also has a numerical value of 1
  - False also has a numerical value of 0

```
>>> True == 1
True
>>> True == 2
False
>>> False == 1
False
>>> False == 0
True
```

Python: Integer

- There are two integer data types in Python:
  - –Integer
    - Ranges from approximately
    - -2147483648 to +2147483647
    - Exact range is machine dependent
  - –Long integer
    - Unlimited except by the machine’s available memory

Python: Integer

That there are two types of integers is pretty transparent to the user (a long integer is denoted by having an L after the number.)

```
>>> a = 34
>>> a
34
>>> b = 34*20000000000000000000
>>> b
680000000000000000000L
```

Python: Floating-point Data Type

- All floating-point numbers are 64 bit (double-precision)
- Scientific notation is the same as in other

Languages.

–Either lower or upper case (e or E) can be used.

```
>>> a = 67000000000000000000.0
>>> a
6.7e+19
>>> b = 2E3
>>> b
2000.0
```

Python: Floating-point Data Type

- int(x) converts x to an integer
- float(x) converts x to a floating point
- The interpreter shows a lot of digits

```
>>> 1.23232
1.2323200000000001
>>> print 1.23232
1.23232
>>> 1.3E7
13000000.0
>>> int(2.0)
2
>>> float(2)
2.0
```

Python: Complex Data Type

- Complex numbers such as 4.5 + *i*8.2 are denoted 4.5 + 8.2j
  - Either lower-case or upper-case j or J may be used to denote the imaginary part.
  - The complex data type has some built-in attributes and methods to retrieve the real part, the imaginary part, and to compute the conjugate.

Python: Compex Data Type

```
>>> c = 3.4 + 5.6j
>>> c
(3.4+5.6j)
>>> c.real
3.4
>>> c.imag
5.6
>>> c.conjugate()
(3.4-5.6j)
```

Python: Variables

```
>>>pi = 3.1415926
>>>message = "Hello, world"
>>>i = 2+2

>>>print(type(pi))
>>>print(type(message))
>>>print(type(i))
```

```
Output:
<type 'float'>
<type 'str'>
<type 'int'>
```

Python: Variables

- Can contain letters, numbers, and underscores
- Must begin with a letter
- Cannot be one of the reserved Python keywords: and, as, assert, break, class, continue, def, del, elif, else, except, exec, finally, for, from, global, if, import, in, is, lambda, not, or, pass, print, raise, return, try, while, with, yield

Python: Coparison Operators

- == : is equal to
- != : not equal to
- > : greater than
- < : less than
- >= : greater than or equal to
- <= : less than or equal to

- Can “chain” comparisons:

```
>>> a = 42
>>> 0 <= a <= 99
True
```

Python Logical Operators

and, or, not

```
>>> 2+2==5 or 1+1==2
True
>>> 2+2==5 and 1+1==2
False
>>> not(2+2==5) and 1+1==2
True
```

Note: We do NOT use &&, ||, !, as in C!

Python: Strings

- Strings can be delimited by single or double quotes
- Python uses Unicode, so strings are not limited to ASCII characters
- An empty string is denoted by having nothing between string delimiters (e.g., "   ")
- Can access elements of strings with [], with indexing starting from zero:
  >>> “snakes”[3]
  'k'
- Note: can't go other way --- can't set “snakes”[3] = 'p' to change a string; strings are *immutable*
- a[-1] gets the *last* element of string a (negative indices work through the string backwards from the end)

```
mystring=“hello world!”

        mystring[0] -> “h”        mystring[1] -> “e”

        mystring[2] -> “l”        mystring[-1] -> “!”
```

Python Basics: Strings

```
>>>"hello"+"world"         "helloworld"# concatenation
>>>"hello"*3               "hellohellohello" # repetition
>>>"hello"[0]              "h"          # indexing
>>>"hello"[-1]             "o"          # (from end)
>>>"hello"[1:4]            "ell"        # slicing
>>>len("hello")            5            # size
>>>"hello" < "jello"       1            # comparison
>>>"e" in "hello"          1            # search
```

Python: Strings

```
>>> s = '012345'
>>> s[3]
'3'
>>> s[1:4]
'123'
>>> s[2:]
'2345'
>>> s[:4]
'0123'
>>> s[-2]
'4'
```

- **len**(String) – returns the number of characters in the String
- **str**(Object) – returns a String representation of the Object

```
>>> len(x)
6
>>>
str(10.3)
'10.3'
```

Python: Strings

- Type conversion:

```
>>> int(“42”)
 42
 >>> str(20.4)
 '20.4'
```

- Compare strings with the equal operator, == (like in C and C++):

```
>>>a = “hello”
  >>>b = “hello”
  >>>a == b
  >>>True
>>>location = “Ankara“ + “ODTÜ”
 >>>location
 AnkaraODTU
```

Python: Strings

- Strings are classes with many built-in methods. Those methods that create new strings need to be assigned (since strings are immutable, they cannot be changed in-place).

```
S.capitalize()
S.center(width)
S.count(substring [, start-idx [,
  end-idx]])
S.find(substring [, start [, end]]))
S.isalpha(), S.isdigit(),
  S.islower(), S.isspace(),
  S.isupper()
S.join(sequence)
```

Python: Replace method

- Doesn't really replace (strings are immutable) but makes a new string with the replacement performed:

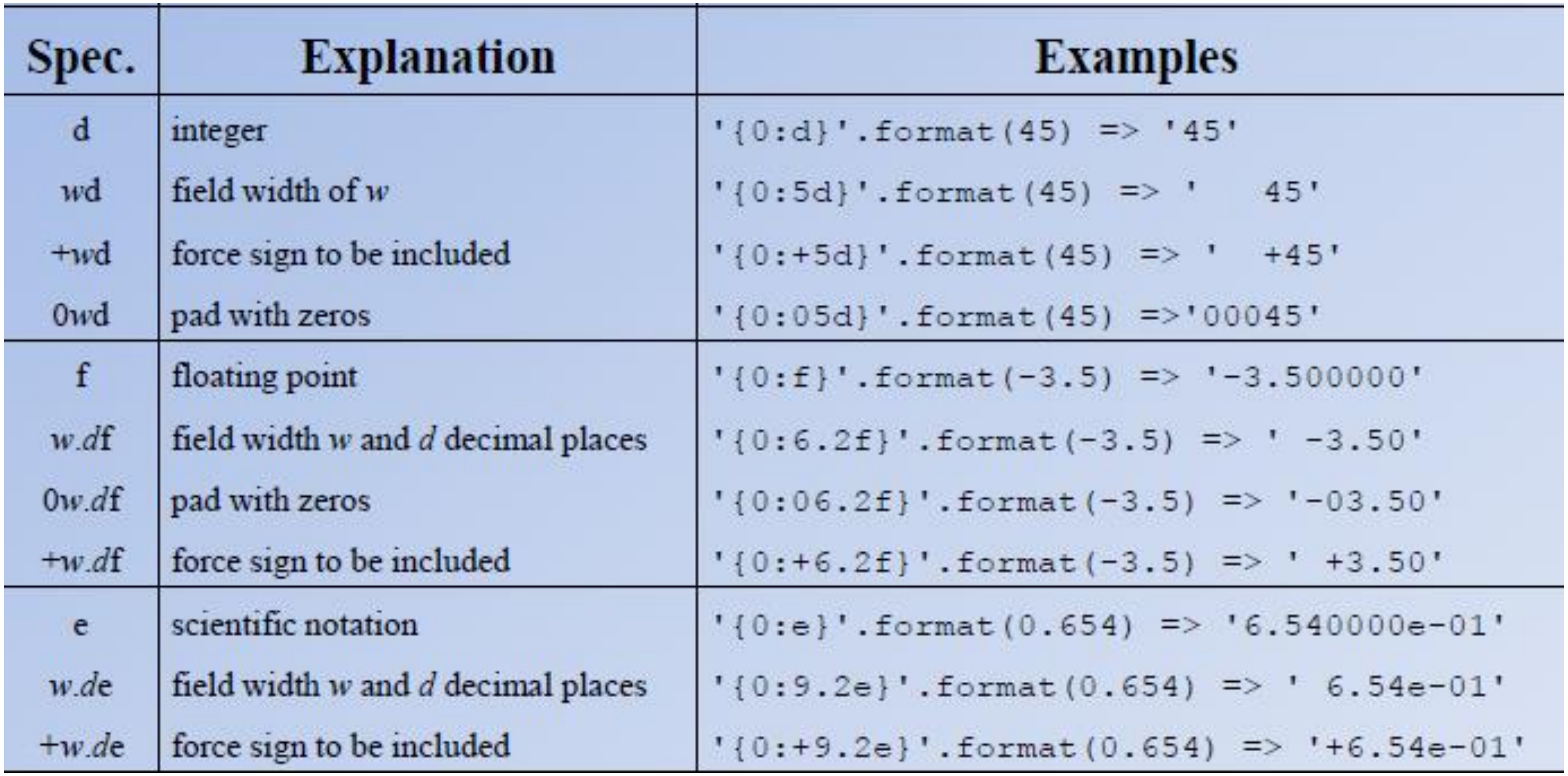
```
>>> a = “abcdefg”
>>> b = a.replace('c', 'C')
>>> b
abCdefg
>>> a
abcdefg
```

Python: Formatted string

- Strings are formatted in Python using the .format() method.

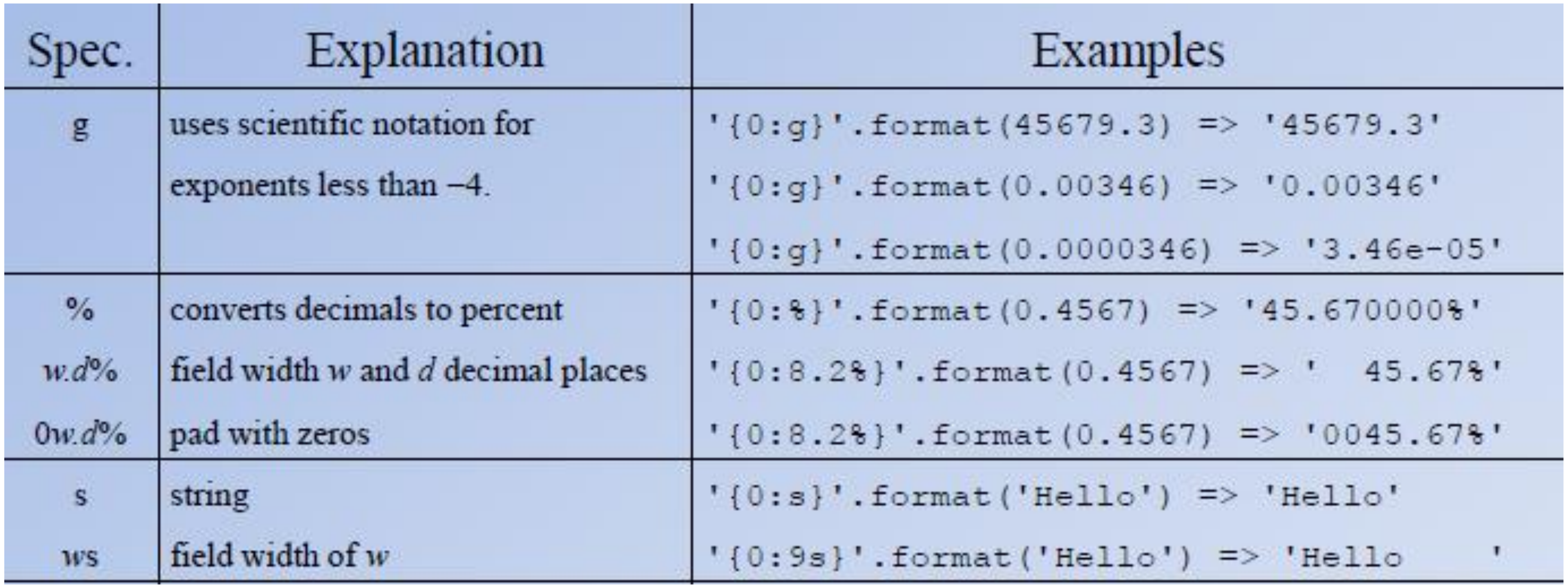
```
>>> x = 654.589342982
>>> s = 'The value of x is {0:7.2f}'.format(x)
>>> s
'The value of x is 654.59'
```

Python: Formatted string



Spec.	Explanation	Examples
d	integer	`'{0:d}'.format(45) => '45'`
wd	field width of *w*	`'{0:5d}'.format(45) => '   45'`
+wd	force sign to be included	`'{0:+5d}'.format(45) => '  +45'`
0wd	pad with zeros	`'{0:05d}'.format(45) =>'00045'`
f	floating point	`'{0:f}'.format(-3.5) => '-3.500000'`
*w.d*f	field width *w* and *d* decimal places	`'{0:6.2f}'.format(-3.5) => ' -3.50'`
0*w.d*f	pad with zeros	`'{0:06.2f}'.format(-3.5) => '-03.50'`
+*w.d*f	force sign to be included	`'{0:+6.2f}'.format(-3.5) => ' +3.50'`
e	scientific notation	`'{0:e}'.format(0.654) => '6.540000e-01'`
*w.d*e	field width *w* and *d* decimal places	`'{0:9.2e}'.format(0.654) => ' 6.54e-01'`
+*w.d*e	force sign to be included	`'{0:+9.2e}'.format(0.654) => '+6.54e-01'`

Python: Formatted string



Spec.	Explanation	Examples

g	uses scientific notation for exponents less than –4.	`'{0:g}'.format(45679.3) => '45679.3'`
		`'{0:g}'.format(0.00346) => '0.00346'`
		`'{0:g}'.format(0.0000346) => '3.46e-05'`

%
w.d%
0*w.d*%
	converts decimals to percent
	field width *w* and *d* decimal places
	pad with zeros
		`'{0:%}'.format(0.4567) => '45.670000%'`
		`'{0:8.2%}'.format(0.4567) => '  45.67%'`
		`'{0:8.2%}'.format(0.4567) => '0045.67%'`

s
*w*s
	string
	field width of *w*
		`'{0:s}'.format('Hello') => 'Hello'`
		`'{0:9s}'.format('Hello') => 'Hello    '`

Python Objects

- Lists (mutable sets of strings)
  - `var = [] # create list`
  - `var = [‘one’, 2, ‘three’, ‘banana’]`
- Tuples (immutable sets)
  - `var = (‘one’, 2, ‘three’, ‘banana’)`
- Dictionaries (associative arrays or ‘hashes’)
  - `var = {} # create dictionary`
  - `var = {‘lat’: 40.20547, ‘lon’: -74.76322}`
  - `var[‘lat’] = 40.2054`
- Each has its own set of methods

Python: List

- Think of a list as a stack of cards, on which your information is written
- The information stays in the order you place it in until you modify that order
- Methods return a string or subset of the list or modify the list to add or remove components
- Written as var[*index*], index refers to order within set (think card number, starting at 0)
- You can step through lists as part of a loop

Python: List

Lists offer the very convenient ability to store groups of related values in a single data structure.

- Ordered collection of data
- Data can be of different types
- Lists are *mutable*
- Issues with shared references and mutability
- Same subset operations as Strings

```
m = [5,6,7,8,9]    # A list of integers
n = range(5,10)    # A list n=m made with range()
print(m[0])        # print element 0: 5
print(n[1])        # print element 1: 6
print(n[-1])       # print last element: 9
print(n)           # prints [5, 6, 7, 8, 9]
```

Elements are indexed 0; 1; 2; : : : from the front, or -1,-2,-3 ... from the back.

List Methods

- Adding to the List
  - var[*n*] = *object*
    - replaces *n* with *object*
  - var.append(*object*)
    - adds *object* to the end of the list
- Removing from the List
  - var[*n*] = []
    - empties contents of card, but preserves order
  - var.remove(*n*)
    - removes card at *n*
  - var.pop(*n*)
    - removes *n* and returns its value

Phython: The += operator for lists

```
>>> a = [1, 3, 5]
>>> a += [7]              # a += 7 fails
>>> a
[1, 3, 5, 7]
>>> a += [“the-end”]    # put this
in [] also!
>>>a
[1, 3, 5, 7, 'the-end']
```

Python: List

```
>>> x = [1,'hello', (3 + 2j)]
>>> x
[1, 'hello', (3+2j)]
>>> x[2]
(3+2j)
>>> x[0:2]
[1, 'hello']
```

Python: List

Modifying Content of the list

- **x[i] = a** reassigns the ith element to the value a
- Since x and y point to the same list object, *both* are changed
- The method **append** also modifies the list

```
>>> x = [1,2,3]
>>> y = x
>>> x[1] = 15
>>> x
[1, 15, 3]
>>> y
[1, 15, 3]
>>> x.append(12)
>>> y
[1, 15, 3, 12]
```

Python: List

Modifying Content of the list

- The method **append** modifies the list and returns **None**
- List addition (**+**) returns a new list

```
>>> x = [1,2,3]
>>> y = x
>>> z = x.append(12)
>>> z == None
True
>>> y
[1, 2, 3, 12]
>>> x = x + [9,10]
>>> x
[1, 2, 3, 12, 9, 10]
>>> y
[1, 2, 3, 12]
>>>
```

Python: Functions and Methods for Lists

- len(*ls*) returns the number of items in the list *ls*
- del *ls*[*i*:*j*] deletes items at indices *i* through *j*-1
- *ls*.append(*elem*) adds element *elem* to end of list
- *ls*.extend(*elems*) adds the multiple elements *elems* to the end of the list. Note that *elems* must also be in the form of a list or tuple.

Python: Functions and Methods for Lists

- *ls*.count(*target*) this method returns the number of instances of *target* contained in the list.

- *ls*.index(*target*) this method returns the first index of the list that contains *target*. If optional *i* and *j* are given, it returns first index of occurrence in the range *i* through *j*-1.

- *ls*.insert(*i*, *elem*) this method inserts *elem* at index *i*

- *ls*.pop(*i*) this method returns element at index *i* and also removes element from the list.

Python: Functions and Methods for Lists

- *ls*.remove(*target*) this method removes first occurrence of target from the list

- *ls*.reverse() this method reverses the list in place

- *ls*.sort() this method sorts the list in place. If keyword reverse= True then it also reverses the results of the sort.

- Note that the reverse() and sort() methods both change(mutate) the actual list. They don’t just return a copy.

List

```
from math import sqrt

  r = [ 1.0, 1.5, -2.2 ]

  length = sqrt( r[0]**2 + r[1]**2 +
  r[2]**2 )

  print(length)
```

List

You can change elements of a list via

```
r=[1.0,1.5,-2.2]
r[1] =3.5
print(r)
```

List

```
from math import *
“””"Take logarithm of each element in the
list”””
r = [ 1.0, 1.5, 2.2 ]
logr = list(map(log,r))
print(logr)
r.append(1)
r.pop(2)
print(r)
```

List

Can also create an empty list with

```
r=[ ]
#Then you may add later.
r.append(1.0)
r.append(1.5)
r.append(-2.2)
print(r)
```

Python: Tuples

- Like a list, tuples are iterable arrays of objects
- Tuples are immutable –
  once created, unchangeable
- To add or remove items, you must redeclare
- Example uses of tuples
  - County Names
  - Land Use Codes
  - Ordered set of functions

Python: Tuples

Tuples are the same as lists but cannot be modified once initialized. They are “read-only” data structures.

- Tuples are *immutable* versions of lists
- One strange point is the format to make a tuple with one element:
- ‘,’ is needed to differentiate from the mathematical expression (2)

```
>>> x = (1,2,3)
>>> x[1:]
(2, 3)
>>> y = (2,)
>>> y
(2,)
>>>
```

Python List/Tuples

- Lists and tuples are both collections of values or objects.
  – The data types of the objects within the list do not have to be the same.

- Lists are denoted with square brackets, while tuples are denoted with parentheses.

```
>>> l = [4.5, -7.8, 'pickle', True, None, 5]
>>> t = (4.5, -7.8, 'pickle', True, None, 5)
```

$\longrightarrow$ list

$\longrightarrow$ tuple

Python List/Tuples

- Lists can be modified after they are created.

– Lists are *mutable*.

- Tuples cannot be modified after they are created.

– Tuples are *immutable*

Python: Dictionaries

- Dictionaries are sets of key & value pairs
- Allows you to identify values by a descriptive name instead of order in a list
- Keys are unordered unless explicitly sorted
- Keys are unique:
  - var[‘item’] = “apple”
  - var[‘item’] = “banana”
  - print var[‘item’] prints just banana

Python: Dictionaries

- Dictionaries are Python's most powerful data collection
- Dictionaries allow us to do fast database-like operations in Python
- Dictionaries are mutable
- Use the { } marker to create a dictionary
- Use the : marker to indicate `key:value` pairs

```
>>> d = {1 : 'hello', 'two' : 42, 'blah' : [1,2,3]}
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['blah']
[1, 2, 3]
```

Python: Dictionaries

- Entries can be changed by assigning to that entry

```
>>> d
{1: 'hello', 'two': 42, 'blah': [1, 2, 3]}
>>> d['two'] = 99
>>> d
{1: 'hello', 'two': 99, 'blah': [1, 2, 3]}
```

- Assigning to a key that does not exist adds an entry

```
>>> d[7] = 'new entry'
>>> d
{1: 'hello', 7: 'new entry', 'two': 99, 'blah':
[1, 2, 3]}
```

Python: Dictionaries

- Copying Dictionaries and Lists

- The built-in **list** function will copy a list
- The dictionary has a method called **copy**

```
>>> l1 = [1]
>>> l2 = list(l1)
>>> l1[0] = 22
>>> l1
[22]
>>> l2
[1]
```

```
>>> d = {1 : 10}
>>> d2 = d.copy()
>>> d[1] = 22
>>> d
{1: 22}
>>> d2
{1: 10}
```

Python: Dictionaries

- Lists, Tuples, and Dictionaries can store any type (including other lists, tuples, and dictionaries!)
- Only lists and dictionaries are mutable

`key in my_dict`

does the key exist in the dictionary

`my_dict.clear()` – empty the dictionary

`my_dict.update(yourDict)` – for each key in yourDict, updates my_dict with that key/value pair

`my_dict.copy` - shallow copy

`my_dict.pop(key)` – remove key, return value

Dictionaries: Views are iterable

```
for key in my_dict:
    print(key)
```

– prints all the keys

```
for key,value in my_dict.items():
    print (key,value)
```

– prints all the key/value pairs

```
for value in my_dict.values():
    print (value)
```

– prints all the values

Python: Indentation and Blocks

- Python uses whitespace and indents to denote blocks of code
- Lines of code that begin a block end in a colon:
- Lines within the code block are indented at the same level
- To end a code block, remove the indentation
- You'll want blocks of code that run only when certain conditions are met

Python: Files

- Files are manipulated by creating a file object
  - `f = open("points.txt", "r")`
- The file object then has new methods
  - `print f.readline() # prints line from file`
- Files can be accessed to read or write
  - `f = open("output.txt", "w")`
  - `f.write("Important Output!")`
- Files are iterable objects, like lists

Python: Input

- The **raw_input**(string) method returns a line of user input as a string
- The parameter is used as a prompt
- The string can be converted by using the conversion methods **int**(string), **float**(string), etc.

Python: Input

input.py

```
print ("What's your name?")
name = raw_input("> ")

print ("What year were you born?")
birthyear = int(raw_input("> "))

print ("Hi %s! You are %d years old!"
 % (name, 2018 - birthyear))
```

```
~: python input.py
What's your name?
> Murat
What year were you born?
>1990
Hi Murat! You are 28 years old!
```

Python: Input

inflobj = open('data', 'r')	Open the file 'data' for input
S = inflobj.read()	Read whole file into one String
S = inflobj.read(N)	Reads N bytes (N >= 1)
L = inflobj.readlines()	Returns a list of line strings

Python: Output

outflobj = open(‘data’, ‘w’)	Open the file ‘data’ for writing
outflobj.write(S)	Writes the string S to file
outflobj.writelines(L)	Writes each of the strings in list L to file
outflobj.close()	Closes the file

Conditional Branching

- if and else

```
if variable == condition:
        #do something based on v == c
else:
        #do something based on v != c
```

- elif allows for additional branching

if *condition*:

elif *another condition*:

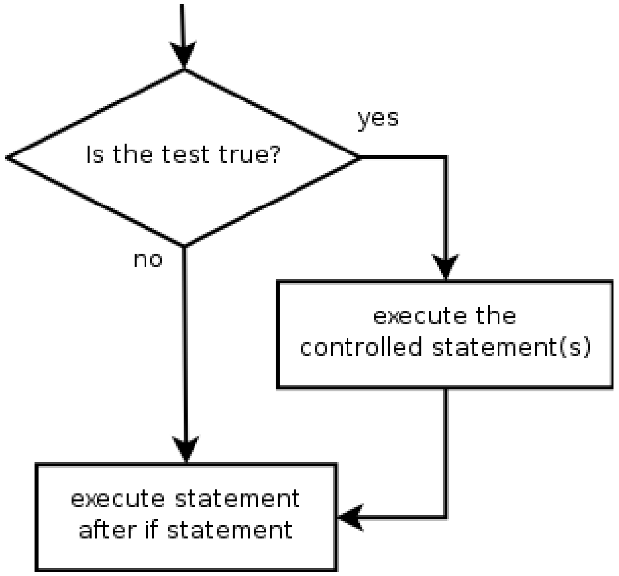
…

else: #none of the above

If statement

- **if statement**: Executes a group of statements only if a certain condition is true. Otherwise, the statements are skipped.
  - Syntax:

```
if condition:
    statements
```



Is the test true?
yes
no
execute the controlled statement(s)
execute statement after if statement

- Example:

```
gpa = 3.4
if gpa > 2.0:
    print ("Your application is accepted.")
```

Python *if* Operators

if_sample.py

```
if (1+1==2):
    print ("1+1==2")
    print ("I always thought so!")
else:
    print ("My understanding of math must 
be faulty!")
```

Note that Body of the function should be indented

Python: *if* statement

In file ifstatement.py

```
import math
x = 30
if x <= 15:
    y = x + 15
elif x <= 30:
    y = x + 30
else :
    y = x
print (" y = ",)
print (math.sin(y))
```

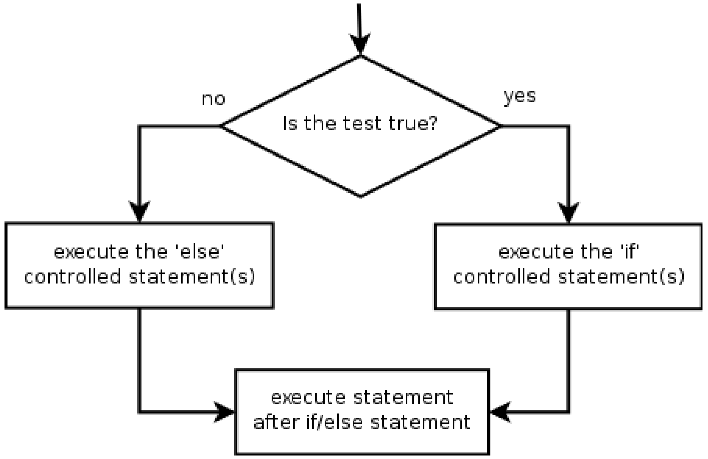
In interpreter

```
>>> import ifstatement
>>> y =  0.999911860107
```

If/else

- **`if/else` statement**: Executes one block of statements if a certain condition is True, and a second block of statements if it is False.
  - Syntax:

```
if condition:
    statements
else:
    statements
```



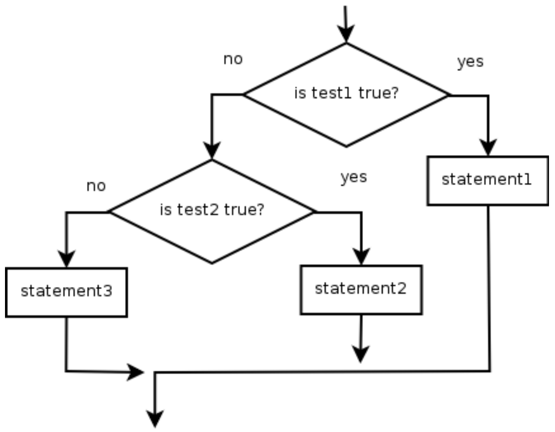
no
Is the test true?
yes
execute the 'else' controlled statement(s)
execute the 'if' controlled statement(s)
execute statement after if/else statement

- Example:

```
gpa = 1.4
if gpa > 2.0:
    print ("Welcome to Mars University!«)
else:
    print ("Your application is denied.")
```

- Multiple conditions can be chained with `elif` ("else if"):

```
if condition:
    statements
elif condition:
    statements
else:
    statements
```



Python *elseif* Operators

Equivalent of “else if” in C

Example elif_sample.py:

```
x = 3
if (x == 1):
    print("one")
elif (x == 2):
    print("two")
else:
    print("many")
```

Python: *while* statement

- **while loop**: Executes a group of statements as long as a condition is True.
  - good for *indefinite loops* (repeat an unknown number of times
- Syntax:

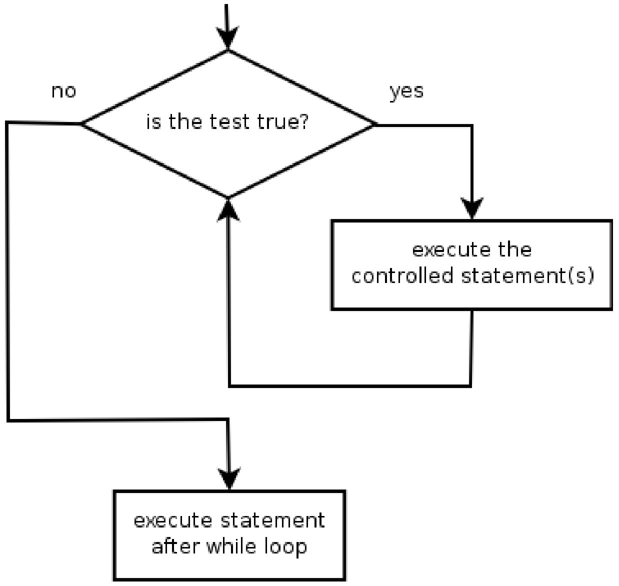
```
while condition:
    statement
```

- Example:

```
number = 1
while number < 200:
    print number,
    number = number * 2
```

  - Output:

```
1 2 4 8 16 32 64 128
```


no
yes
is the test true?
execute the controlled statement(s)
execute statement after while loop

Flow control within loops

General structure of a loop:

```
while <statement> (or for <item> in <object>):
   <statements within loop>
   if <test1>: break        # exit loop now
   if <test2>: continue     # go to top of loop now
   if <test3>: pass         # does nothing!
else:
   <other statements> # if exited loop without
                      # hitting a break
```

Python: *while* statement

Initialize the variables for the loop

As long as x is less than 10, we continue around the loop.

```
x = 1
while x < 10:
    print(x)
    x = x + 1
```

In *whileloop.py*

```
>>> import whileloop
1
2
3
4
5
6
7
8
9
>>>
```

In interpreter

Python: *while else* statement

- The optional **else** runs only if the loop exits normally (not by break)

```
x = 1

while x < 3 :
    print (x)
    x = x + 1
else:
    print ("hello")
```

Run from the command line

```
$ : python whileelse.py
1
2
hello
```

`In whileelse.py`

Looping with *For*

- For allows you to loop over a block of code a set number of times
- For is great for manipulating lists:

```
a = ['cat', dog',‘hello']

for x in a:
      print x, len(x)
```

Results:

```
cat 3
dog 3
hello 5
```

Python: loops

- Example for.py

```
for i in range(3):
    print(i)
```

  output:
   0, 1, 2
- range(n) returns a list of integers from 0 to n-1. range(0,10,2) returns a list 0, 2, 4, 6, 8

```
>>> for i in range(2, 6):
...     print(i)
2
3
4
5
>>> for i in range(15, 0, -5):
...     print(i)
15
10
5
```

Python: loops

- **For** loops also may have the optional **else** clause

```
for x in range(5):
    print(x)
    break
else :
    print("i got here")
```

elseforloop.py

```
$: python elseforloop.py
1
```

Python: *loops*

```
for x in [1,7,13,2] :
    print(x)
```

forloop1.py

```
$: python forloop1.py
1
7
13
2
```

```
for x in range(5) :
    print(x)
```

forloop2.py

```
$: python forloop2.py
0
1
2
3
4
```

`range(N) generates a list of numbers [0,1, …, n-1]`

Python: *Sum*

```
# compute sum of number 1-10
# initialize variable for sum
m = 0.
# make a list of numbers 1-10:
numbers = range(1,11)
# now add in each value in list
for i in numbers:
     m = m + i
# print the result
print m
```

The variable “m” is initialized with value of 0.0

Numbers is a list: [1,2,3,4,5,6,7,8,9,10]

First pass through loop:
i = 1 so m = 0 + 1 = 1

Next pass through loop:
i = 2 so m = 1 + 2 = 3
Third pass through loop:
i = 3 so m = 3 + 3 = 6

Python: Function

square.py

```
def square(n):
    return n*n
```

- Body of the function should be indented consistently (4 spaces is typical in Python)
- >>>print("The square of 3 is),
  >>>print square(3)

  Output:
  The square of 3 is 9

Python: Function

```
def max(x,y) :
    if x < y :
        return x
    else :
        return y
```

functionbasics.py

```
>>> import functionbasics
>>> max(3,5)
5
>>> max('hello', 'there')
'there'
>>> max(3, 'hello')
'hello'
```

Python: Function

- Functions are objects
- The same reference rules hold for them as for other objects

```
>>> x = 10
>>> x
10
>>> def x () :
...     print 'hello'
>>> x
<function x at 0x619f0>
>>> x()
hello
>>> x = 'blah'
>>> x
'blah'
```

Python: Function

```
def foo(f, a) :
    return f(a)

def bar(x) :
    return x * x
```

funcasparam.py

```
>>> from funcasparam import *
>>> foo(bar, 3)
9
```

Note that the function foo takes two parameters and applies the first as a function with the second as its parameter

Python: Function

Functions Inside Functions

- Since they are like any other object, you can have functions inside functions

```
def foo (x,y) :
    def bar (z) :
        return z * 2
    return bar(x) + y
```

funcinfunc.py

```
>>> from funcinfunc import *
>>> foo(2,3)
7
```

Python: Function

Functions Returning Functions

```
def foo (x) :
    def bar(y) :
        return x + y
    return bar
# main
f = foo(3)
print f
print f(2)
```

funcreturnfunc.py

```
$: python funcreturnfunc.py
<function bar at 0x612b0>
5
```

Python: Function

- Call by name
- Any positional arguments must come before named ones in a call

```
>>> def foo (a,b,c) :
...      print a, b, c
...
>>> foo(c = 10, a = 2, b = 14)
2 14 10
>>> foo(3, c = 2, b = 19)
3 19 2
```

Python: Function

- Example: fahToCel.py

```
def fahrenheitTocelsius(f):
    cel = (f- 32)/1.8
     return cel
```

```
>>>print("The Celsius of 100 k is),
>>>print(fahrenheitTocelsius(100))
```

Output:

```
The Celsius of 100 k is 37.777
```

Scope

scope.py

```
a = 5                   # global

def func(b):
   c = a + b
   return c

print func(4)                # gives
4+5=9
print(c)                  # not defined
```

Scope

scope.py

```
a = 5                # global

def func(b):
  global c
  c = a + b
  return c

print func(4)  # gives 4+5=9
print(c)       # now it's defined (9)
```

Python: Built-in functions

- Several useful built-in functions.

```
>>>print pow(2,3)
>>>print abs(-14)
>>>print max(1,-5,3,0)
```

- Output:

```
8
14
3
```

Functions of function

- Example funcfunc.py

```
def iseven(x,f):
    if (f(x) == f(-x)):
        return True
    else:
        return False

def square(n):
    return n*n

def cube(n):
    return n*n*n

print iseven(2,square)
print iseven(2,cube)
```

- Output:

```
True
False
```

Python: *Example*

The sequence 1, 1, 2; 3, 5, 8, 13, 21, 34, 55,... is defined by the linear homogeneous recurrence relation

$$\boxed{F_n = F_{n-1} + F_{n-2};}$$

where n = 0, 1, 2, 3,..... and $F_0 = F_1 = 1$.

Write a Python function that generates F_n given n.
Write a program that generates all the Fibonacci numbers below 1000

Python: *Example*

This implementation is a recursive function, i.e., a function that calls itself. It's an easy way to implement the Fibonacci sequence

```
def fib(n):
"""Generate term n of the Fibonacci series."""
     if n <= 1:
     # if n = 0 or 1: return 1
          return 1
     else:
          # else if n = 2, 3, ...: call fib
          return fib(n-1) + fib(n-2)
```

Python: *Example*

This more efficient implementation contains two internal state variables a and b, which keep track of the terms F_{n-1} and F_{n-2}.

```
def fib(n):
     """Generate the Fibonacci series."""
     a, b = 0, 1
     while n > 0: # loop condition
          a, b, n = b, a+b, n-1 # update step
     return b
```

We loop over n terms in the series by decrementing the function argument n and terminating the loop when n = 0.

File Proccessing

- Reading the entire contents of a file:

variableName `= open("`**_filename_**`").read()`

Example:

```
file_text = ope(”data.txt").read()
```

File Proccessing

- Creating file object
  - Syntax: file_object = open(filename, mode)
    - `Input = open(“d:\inventory.dat”, “r”)`
    - `Output = open(“d:\report.dat”, “w”)`
- Manual close
  - Syntax: close(file_object)
    - `close(input)`
- Reading an entire file
  - Syntax: string = file_object.read()
    - `content = input.read()`
  - Syntax: list_of_strings = file_object.readlines()
    - `lines = input.readline()`

Python: reading files

name = open("**filename**")

– opens the given file for reading, and returns a file object

name.read() - file's entire contents as a string

```
>>> f = open("hours.txt")
>>> f.read()
'123 Susan 12.5 8.1 7.6 3.2\n
456 Brad 4.0 11.6 6.5 2.7 12\n
789 Jenn 8.0 8.0 8.0 8.0 7.5\n'
```

Python: reading files

`name``.readline()` - next line from file as a string

– Returns an empty string if there are no more lines in the file

`name``.readlines()` - file's contents as a list of lines

```
>>> f = open("hours.txt")
>>> f.readline()
'123 Susan 12.5 8.1 7.6 3.2\n'

>>> f = open("hours.txt")
>>> f.readlines()
['123 Susan 12.5 8.1 7.6 3.2\n',
'456 Brad 4.0 11.6 6.5 2.7 12\n',
'789 Jenn 8.0 8.0 8.0 8.0 7.5\n']
```

Python: reading files

- A file object can be the target of a for ... in loop
- A template for reading files in Python:

```
for line in open("filename"):
    statements
```

```
>>> for line in open("hours.txt"):
...     print(line.strip())   # strip() removes \n

123 Susan 12.5 8.1 7.6 3.2
456 Brad 4.0 11.6 6.5 2.7 12
789 Jenn 8.0 8.0 8.0 8.0 7.5
```

Python: writing files

```
name = open("filename", "w")      # write
name = open("filename", "a")      # append
```

- opens file for write (deletes any previous contents) , or
- opens file for append (new data is placed after previous data)

`name.write(str)` - writes the given string to the file

`name.close()` - closes file once writing is done

```
>>> out = open("output.txt", "w")
>>> out.write("Hello, world!\n")
>>> out.write("How are you?")
>>> out.close()

>>> open("output.txt").read()
'Hello, world!\nHow are you?'
```

Python: Module Basic

- There are over 200 modules in the Standard Library
- Consult the Python Library Reference Manual, included in the Python installation and/or available at http://www.python.org

Python: import

- An import statement does three things:
  - Finds the file for the given module
  - Compiles it to bytecode
  - Runs the module's code to build any objects (top-level code, e.g., variable initialization)

- The module name is only a simple name; Python uses a module search path to find it. It will search: (a) the directory of the top-level file, (b) directories in the environmental variable PYTHONPATH, (c) standard directories, and (d) directories listed in any .pth files (one directory per line in a plain text file); the path can be listed by printing sys.path

Python: import

- Import brings in a whole module; you need to qualify the names by the module name (e.g., sys.argv)
- "import from" copies names from the module into the current module; no need to qualify them (note: these are copies, not links, to the original names)

```
from module_x import junk
junk()  # not module_x.junk()

from module_x import * # gets all top-level
                       # names from
module_x
```

Python: Math

Python comes with packages which can be imported

One important packages is “math”

If you want to use the logarithm function then write

```
from math import log
```

At the top of your program

(not necessary but makes for better programs)

After doing this

```
>>>x= log(2.5)
```

which calculates natural logarithms of 2.5

Python: import

```
import random

guess = random.randint(1,100)
print(guess)
dinner = random.choice([“meatloaf”,
“pizza”, “chicken pot pie”])
print(dinner)
```

Python: Math

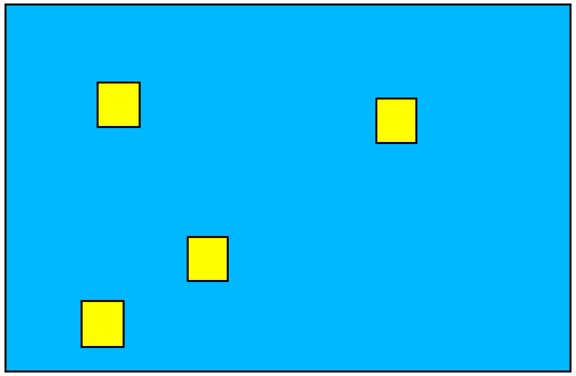
```
from math import *
```

Function name	Description
`ceil`(**value**)	rounds up
`cos`(**value**)	cosine, in radians
`degrees`(**value**)	convert radians to degrees
`floor`(**value**)	rounds down
`log`(**value**, **base**)	logarithm in any base
`log10`(**value**)	logarithm, base 10
`max`(**value1**, **value2, ...**)	largest of two (or more) values
`min`(**value1**, **value2, ...**)	smallest of two (or more) values
`radians`(**value**)	convert degrees to radians
`round`(**value**)	nearest whole number
`sin`(**value**)	sine, in radians
`sqrt`(**value**)	square root
`tan`(**value**)	tangent

Constant	Description
`e`	2.7182818...
`pi`	3.1415926...

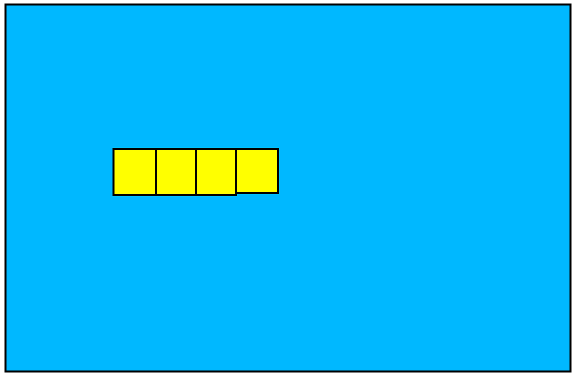
Python: lists/arrays

- Python Lists
  - Lists can have any type of data
  - Locations of List items in memory is not predictable. This limits mathematical use of Lists as arrays



List Data

- NumPy Arrays
  - All data have same type
  - All data are together in memory



NumPy Array Data

NumPy: Arrays

- NumPy creates a homogeneous, multidimensional, array.
  - Elements all have same type.
  - Each dimension of the array is called an axis.
  - Number of axes is called the “rank” of the array.
  - Positive numbers are used to index the data in the array, beginning with index 0, with one number for each axis of the array.

NumPy: Arrays

```
In [1]: import numpy as np

In [2]: a = np.array([[1,2,3],[4,5,6],[7,8,9]])

In [3]: a
Out[3]:
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [4]: type(a)
Out[4]: numpy.ndarray

In [5]: a.ndim
Out[5]: 2

In [6]: a.dtype
Out[6]: dtype('int64')

In [7]: a.shape
Out[7]: (3, 3)
```

Import NumPy

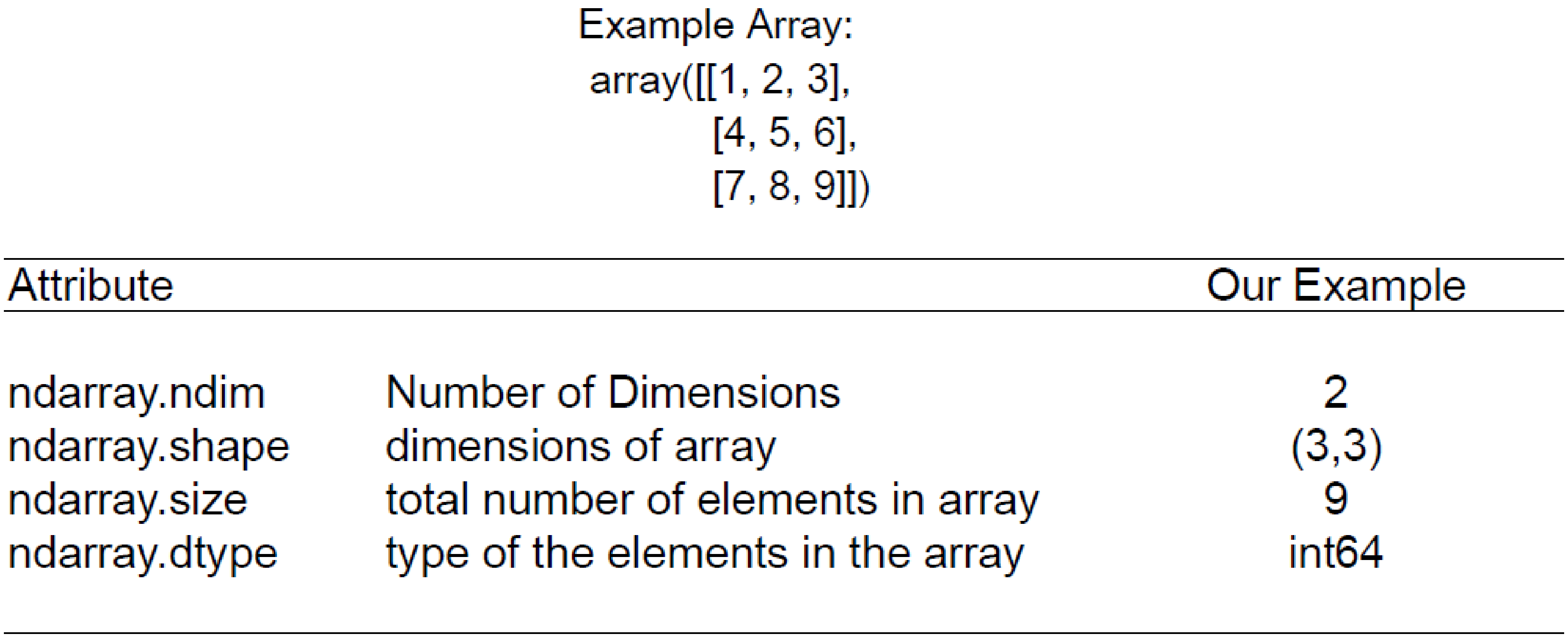
Create a 2D NumPy array

Show the array

Check "type": it is a numpy.ndarray

Check
- Dimensions
- Data Type
- Shape

NumPy: Array Attributes

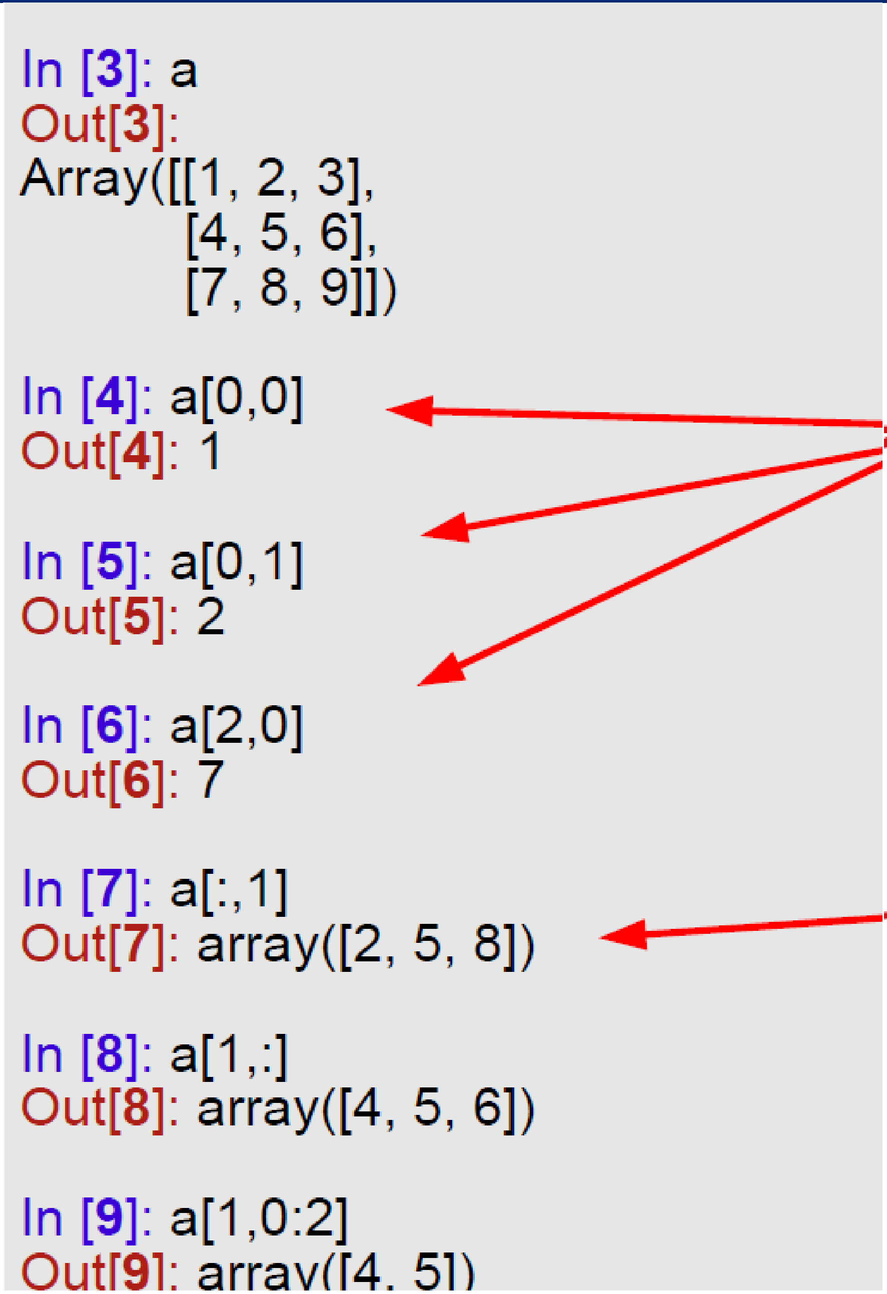


Example Array:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Attribute		Our Example
ndarray.ndim	Number of Dimensions	2
ndarray.shape	dimensions of array	(3,3)
ndarray.size	total number of elements in array	9
ndarray.dtype	type of the elements in the array	int64

NumPy: Array Attributes

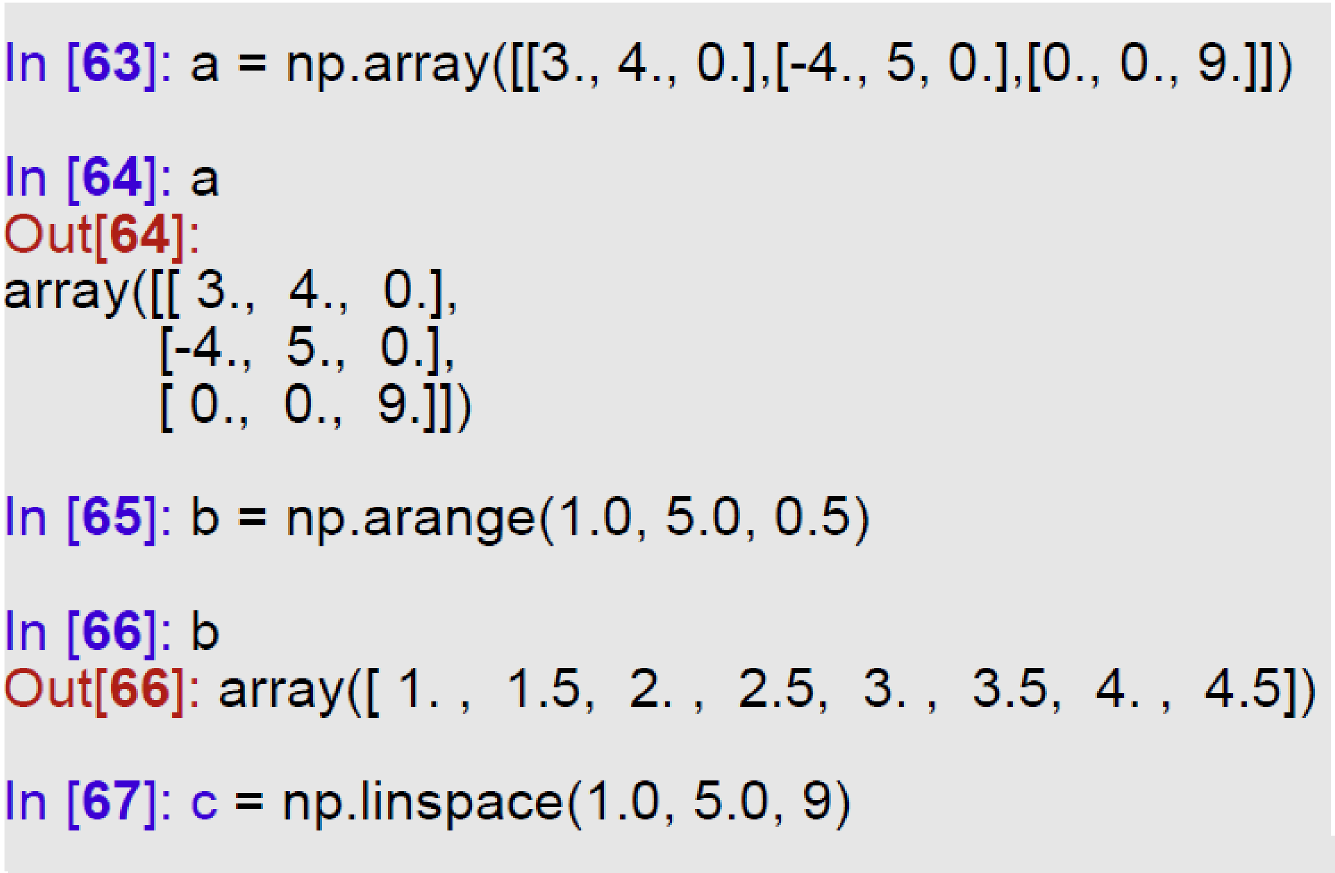


```
In [3]: a
Out[3]:
Array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])

In [4]: a[0,0]
Out[4]: 1

In [5]: a[0,1]
Out[5]: 2

In [6]: a[2,0]
Out[6]: 7

In [7]: a[:,1]
Out[7]: array([2, 5, 8])

In [8]: a[1,:]
Out[8]: array([4, 5, 6])

In [9]: a[1,0:2]
Out[9]: array([4, 5])
```

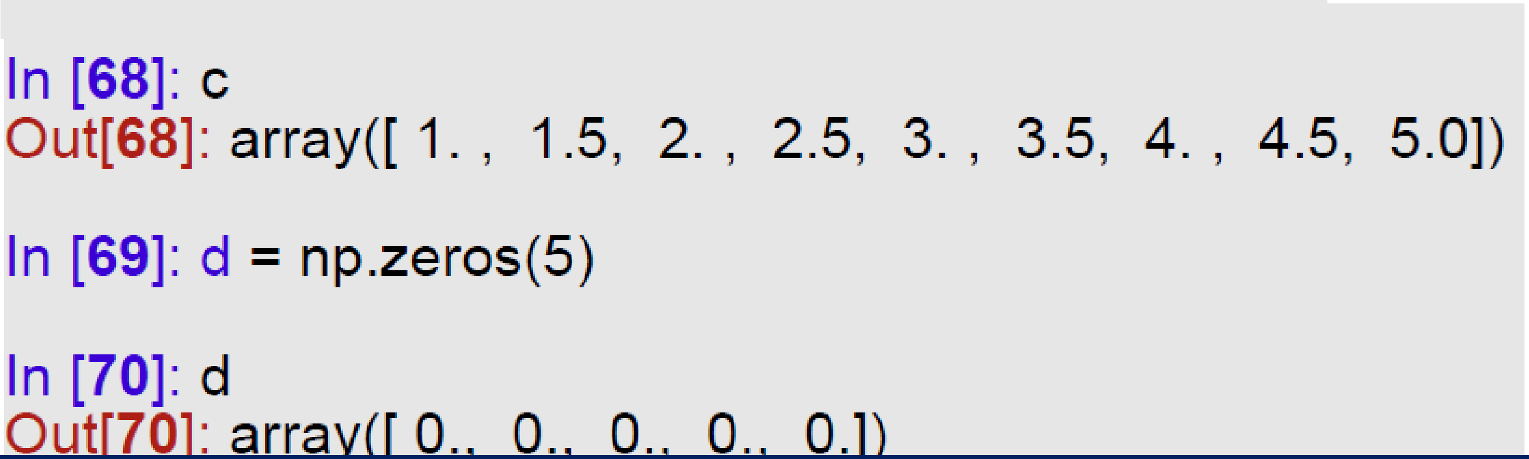
Row index is First
Column index is Second

Use of colon operator to
get a “slice” of the array.

NumPy: Creating Arrays



```
In [63]: a = np.array([[3., 4., 0.],[-4., 5, 0.],[0., 0., 9.]])

In [64]: a
Out[64]:
array([[ 3.,  4.,  0.],
       [-4.,  5.,  0.],
       [ 0.,  0.,  9.]])

In [65]: b = np.arange(1.0, 5.0, 0.5)

In [66]: b
Out[66]: array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5])

In [67]: c = np.linspace(1.0, 5.0, 9)
```



```
In [68]: c
Out[68]: array([ 1. ,  1.5,  2. ,  2.5,  3. ,  3.5,  4. ,  4.5,  5.0])

In [69]: d = np.zeros(5)

In [70]: d
Out[70]: array([ 0.,  0.,  0.,  0.,  0.])
```

NumPy: Methods for Creating Arrays

Method	What it does
numpy.array([...])	enter array values directly
numpy.arange(first,last,step)	creates array spaced by “step” beginning at “first” and ending when value is equal to or greater than “last”
numpy.linspace(first,last,n)	creates an evenly spaced array with n elements beginning at “first” and ending with “last”.
numpy.zeros(n)	creates an array of n zeros
numpy.zeros((n,m))	creates a 2D array of zeros with n rows, m columns

NumPy: Methods for Creating Arrays

- Add (+), Subtract (-), Multiply (*),Divide (/), and exponentiation (**) all operate element by element – they are NOT linear algebra operations.

A = np.array([1,2])   A+B = [4, 6]

B = np.array([3,4])   A * B = [3, 8]

A**B = [1, 16]

- Use np.dot to do proper multiplication according to linear algebra rules:

np.dot(A,B) = A[0]*B[0]+A[1]*B[1] = 11

np.sin(A) computes the sin of all the elements.

List/array

Alternative to list is an array. Differences with respect to list.

1) # of elements in array is fixed. Cannot add or remove.
2) Elements must all be of the same type. Can not mix or cannot change.

Advantage of array over list.

3) Can be two dimensional

4) They behave like vectors or matrices. Can add and substruct.

5) Much faster than lists.

Array functions are part of package *numPy*

Matplotlib and PyPlot

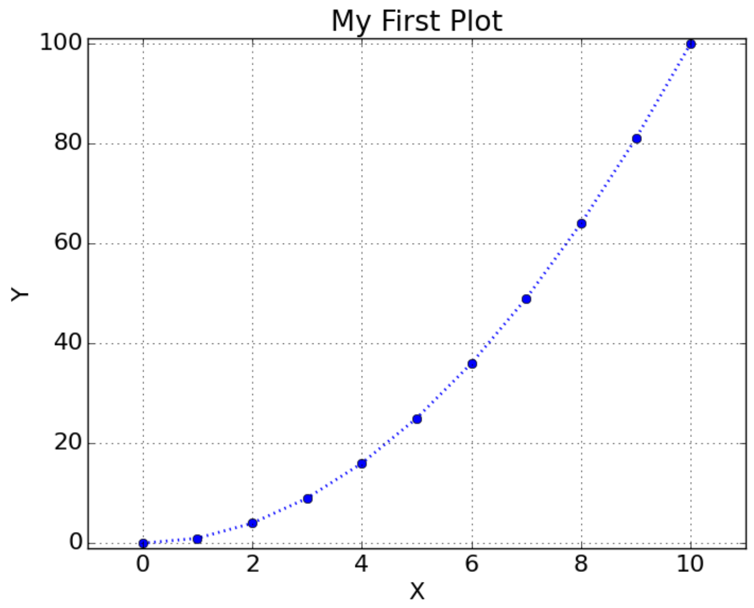
Matplotlib is a library for 2D plotting

- Can be used in scripts or interactively
- Uses NumPy arrays

- PyPlot is a collection of methods within Matplotlib which allow user to construct 2D plots easily and interactively

Matplotlib : making plot

```
import numpy as np
import matplotlib.pyplot as pl
# make a numpy array
X = np.linspace(0.,10.,11)
Y = X*X # Y array is X squared
pl.ion() # turns on interactive
  plotting
pl.plot(X,Y,'bo:') # plots large
  dots
# connected by dotted lines
pl.xlabel('X')
pl.ylabel('Y')
pl.title('My First Plot')
pl.axis([-1,11,-1,101]) # sets the
  dimensions
pl.grid() # draws dotted lines on
  major “ticks”
```



Matplotlib : Interactive Mode Plotting

- Interactive mode updates a plot each time a new command is issued.

- Turn on interactive mode with method:

```
pl.ion()
```

- Turn off interactive mode with method:

```
pl.ioff()
```

- When interactive mode is not on, you enter all pyplot commands and then use
- the method pl.show() to see the figure.

NB: `pl.show()` waits for you to close the plot figure window before you can proceed

The PyPlot Plot Method

- `pl.plot(X,Y,'CLM')`
    - X is X array for plot
    - Y is Y array for plot
    - X and Y must have same number of points
    - String `'clm'` tells how to make the plot:
        - C indicates the color
        - L indicates the line style:
            - - -- : -. omit symbol for no line
- M indicates marker style
    - . + 0 * x s d ^ v > < p h
    - None = no symbol

The PyPlot Plot Method

- `pl.xlabel('name of x axis')` - prints a label along the x-axis
- `pl.ylabel('name of y axis')` - prints a label along the y-axis
- `pl.title('title for plot')` - writes a title across `the top of the graph`
- `pl.axis([xmin, xmax, ymin, ymax])` - sets limits for plot with array shown
- `pl.grid('on')` – turn on grid lines

PyPlot Figures

- Matplotlib allows you to use one or more “figures” for making graphs.
- To start plotting in a figure, we use the figuremethod e.g.:

  `pl.figure(1)`

  - In interactive mode, this opens figure 1 and shows window on screen. Ready to start accepting plot commands.
  - The figure number can be any integer > 1

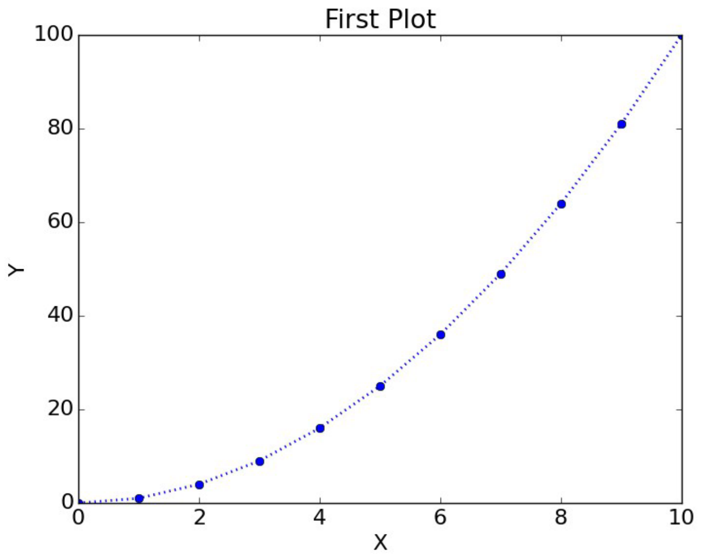
- “Close” a figure when done:

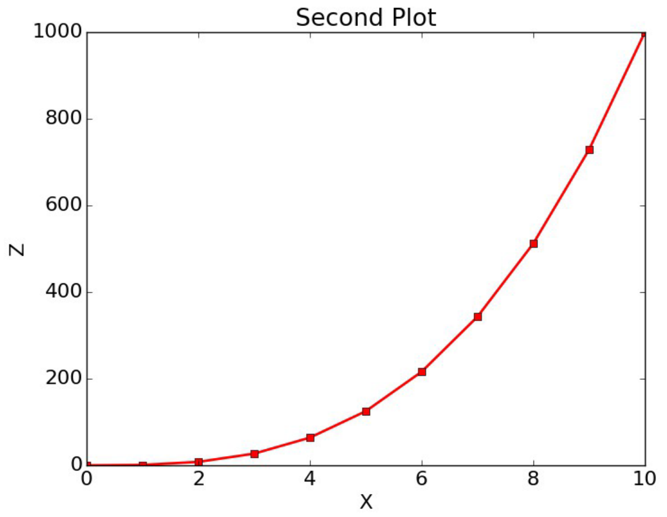
  `pl.close(1)` # closes figure 1

  `pl.close('all')` # closes ALL open figures

Multiple Figure Example

```
X = np.linspace(0.,10.,11)
Y = X*X # Y array is X squared
Z = X*X*X # Z array is X cubed
pl.ion() # turns on interactive plottir
pl.figure(1)
pl.plot(X,Y,'bo:') # plots large blue c
# connected by dotted lines
pl.xlabel('X')
pl.ylabel('Y')
pl.title('First Plot')
pl.figure(2)
pl.plot(X,Z,'rs-') # plots large red squ
# connected by solid lines
pl.xlabel('X')
pl.ylabel('Z')
pl.title('Second Plot')
```



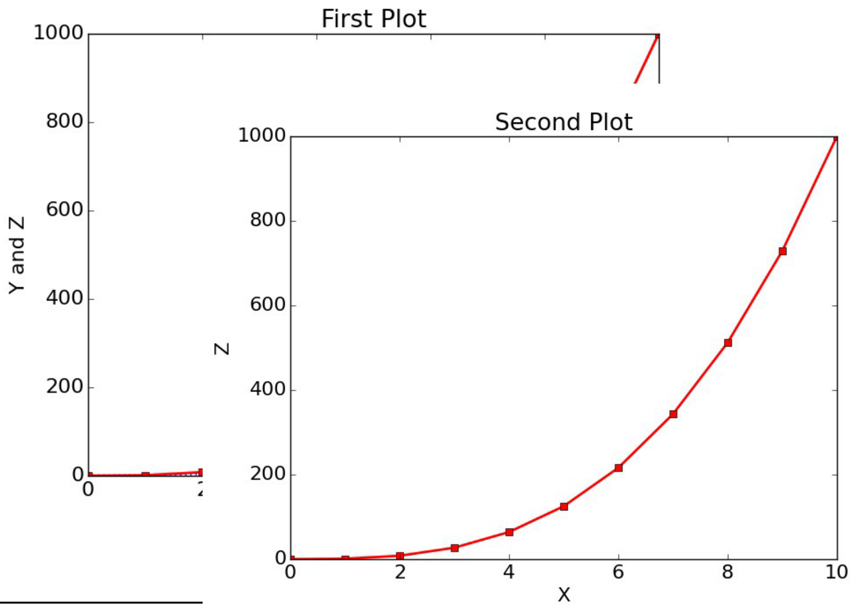


Multiple Figure Example

```
X = np.linspace(0.,10.,11)
Y = X*X # Y array is X squared
Z = X*X*X # Z array is X cubed
pl.ion() # turns on interactive plotting
pl.figure(1)
pl.plot(X,Y,'bo:')
pl.plot(X,Z,'rs-') # hold is 'on' by default
# so this line is added
pl.xlabel('X')
pl.ylabel('Y and Z')
pl.title('First Plot')
pl.hold('off') # turn off hold for
# second figure
pl.figure(2)
pl.plot(X,Y,'bo:')
pl.plot(X,Z,'rs-') # this redraws wit
# the first graph
pl.xlabel('X')
pl.ylabel('Z')
pl.title('Second Plot')
```

The hold method allows us to control whether a call to the plot method will redraw the graph

By default hold is 'on' when we start up, so additional lines will be added to the graph.



Multiple Figure Example

```
# demo program for PH281 - fall.py
# calculate position of a falling ball and plot
# x(t) = x(0) - 1/2 g t**2
# F.P. Schloerb
# import packages
import numpy as np
import matplotlib.pyplot as pl
# define an array of times for calculation
t = np.linspace(0.,10.,11)
# set value of gravitational acceleration
g = 9.8 # in m/s**2
# get the initial height from the user
h = input('Enter initial height of ball (m): ')
# compute the location of the ball
x = h - 0.5 * g * t**2
# plot result
pl.ion()
pl.plot(t,x,'o')
pl.xlabel('time (s)')
pl.ylabel('position (m)')
pl.title('Falling Ball')
# print result
print 'Here are the results (new style):'
print ' t x'
for i in range(len(t)):
print '{0:5.2f} {1:8.2f}'.format( t[i], x[i])
```

Comments about Script

Import Packages

Initialize Parameters

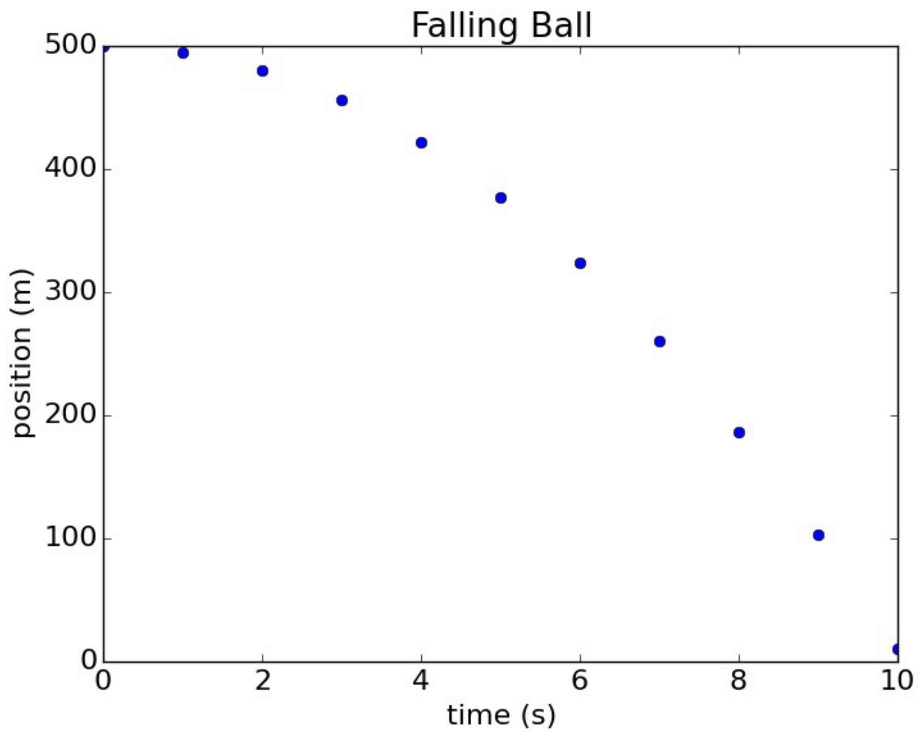
Calculate

Display Results
Plot Graph

Print Table

Multiple Figure Example

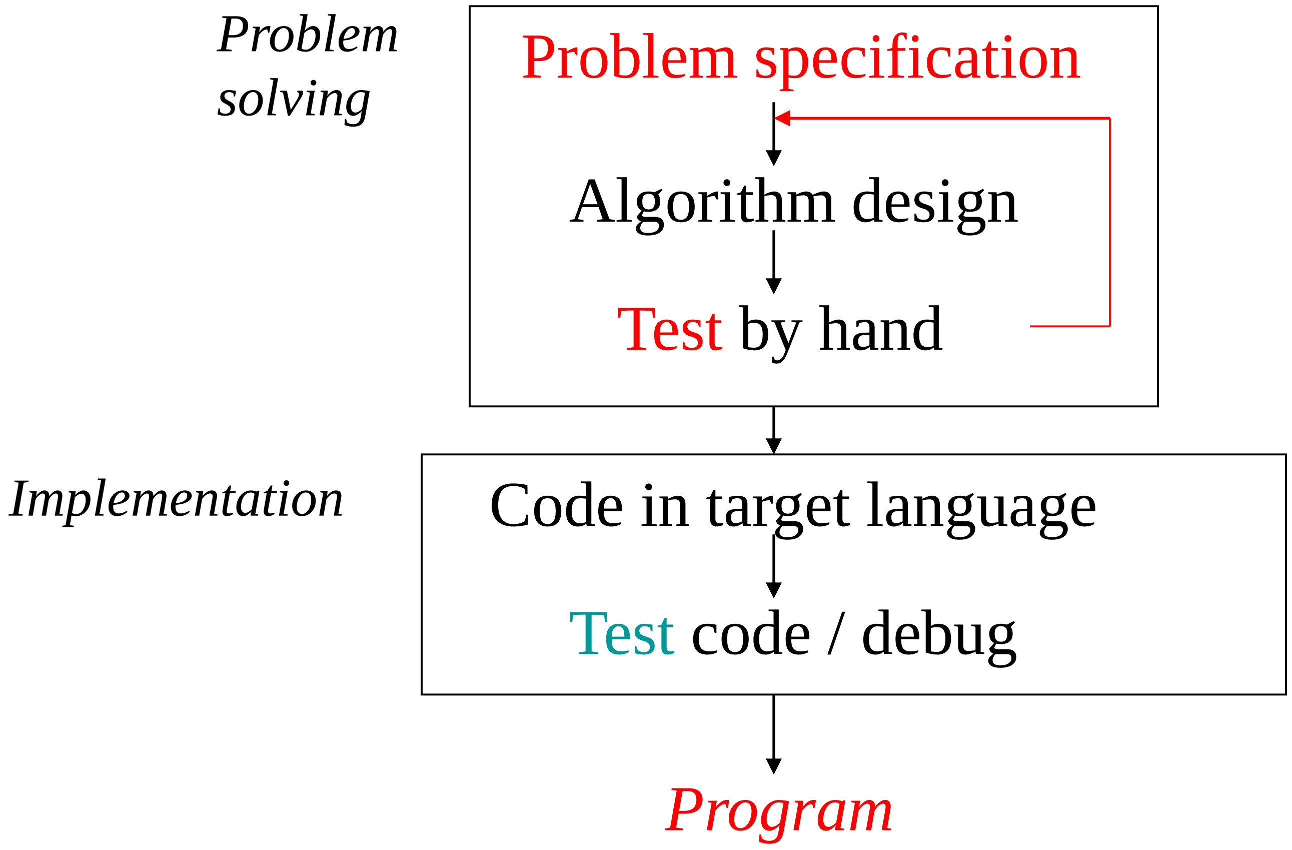
```
>>> run fall.py fall.py
Enter initial height of ball (m):
500
Here are the results (new style):
t x
0.00 500.00
1.00 495.10
2.00 480.40
3.00 455.90
4.00 421.60
5.00 377.50
6.00 323.60
7.00 259.90
8.00 186.40
9.00 103.10
10.00 10.00
```



Good programming style

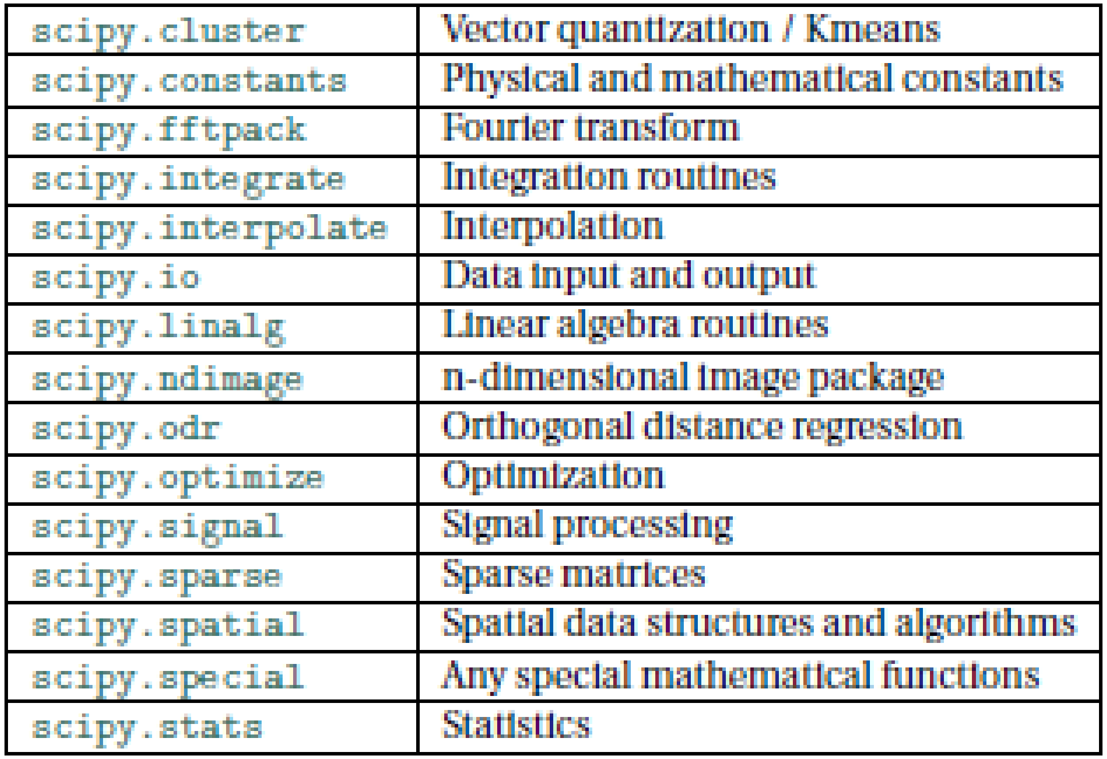
1. Use commands
2. Use meaningful variable names
3. Use the right type of variables
4. Import functions first
5. Give physical constants names
6. Print out partial results throughout the program.

Program Development



Scipy: modules

scipy is composed of task-specific sub-modules:



`scipy.cluster`	Vector quantization / Kmeans
`scipy.constants`	Physical and mathematical constants
`scipy.fftpack`	Fourier transform
`scipy.integrate`	Integration routines
`scipy.interpolate`	Interpolation
`scipy.io`	Data input and output
`scipy.linalg`	Linear algebra routines
`scipy.ndimage`	n-dimensional image package
`scipy.odr`	Orthogonal distance regression
`scipy.optimize`	Optimization
`scipy.signal`	Signal processing
`scipy.sparse`	Sparse matrices
`scipy.spatial`	Spatial data structures and algorithms
`scipy.special`	Any special mathematical functions
`scipy.stats`	Statistics

They all depend on numpy, but are mostly independent of each other.
The standard way of importing Numpy and these Scipymodules is:

Scipy: modules

They all depend on numpy, but are mostly independent of each other. The standard way of importing Numpy and these Scipymodules is:

```
>>> import numpy as np
>>> from scipy import stats # same for other sub-modules
```

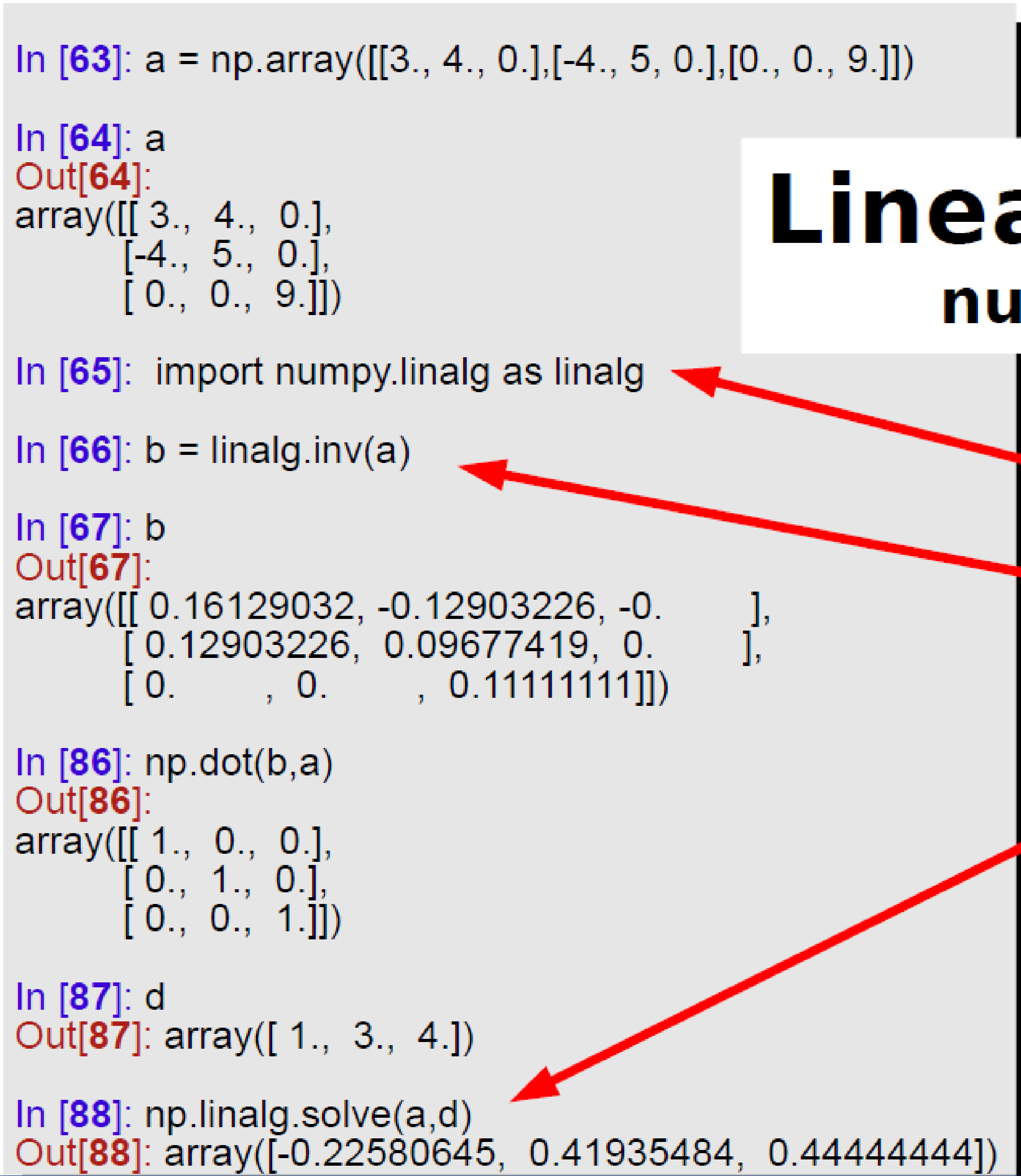
Scipy: Linear algebra

The scipy.linalg module provides standard linear algebra operations, relying on an underlying efficient implementation (BLAS, LAPACK).

The scipy.linalg.det() function computes the determinant of a square matrix:

The scipy.linalg.inv() function computes the inverse of a square matrix:

NumPy: Linear Algebra



```
In [63]: a = np.array([[3., 4., 0.],[-4., 5, 0.],[0., 0., 9.]])

In [64]: a
Out[64]:
array([[ 3.,  4.,  0.],
       [-4.,  5.,  0.],
       [ 0.,  0.,  9.]])

In [65]:  import numpy.linalg as linalg

In [66]: b = linalg.inv(a)

In [67]: b
Out[67]:
array([[ 0.16129032, -0.12903226, -0.        ],
       [ 0.12903226,  0.09677419,  0.        ],
       [ 0.        ,  0.        ,  0.11111111]])

In [86]: np.dot(b,a)
Out[86]:
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])

In [87]: d
Out[87]: array([ 1.,  3.,  4.])

In [88]: np.linalg.solve(a,d)
Out[88]: array([-0.22580645,  0.41935484,  0.44444444])
```

Linear Algebra

numpy.linalg

Import “linalg”

inv() inverts the 2D array

solve(a,d) f nds p, the solution to

$$d = a\ p$$

Linear Equations

In many instances, `numpy` arrays can be thought of as matrices.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

Matrix: Determinants

- The determinant of an array is found by using the det() function from the scipy.linalg module.

```
>>> import numpy as np
>>> import scipy.linalg as slin
>>> a = np.array()[[3,-5,8],[-
4,5,0],[0.,0.,9.]]
>>> a
array([[ 3, -5, 8],
[-1, 2, 3],
[-5, -6, 2]])
>>> slin.det(a)
259.0
```

Matrix: Trace

- The trace of an array is found by using the trace() function from numpy

```
>>> import numpy as np
>>> a = np.array()[[3,-5,8],[-1,2,3],[-5,-6,2]]
>>>a
array([[ 3, -5, 8],
[-1, 2, 3],
[-5, -6, 2]])
>>> np.trace(a)
7
```

Matrix: Trace

- Offset traces can also be computed.

```
>>> import numpy as np
>>> a = np.array()[[3,-5,8],[-1,2,3],[-5,-6,2]]
>>> a
array([[ 3, -5, 8],
[-1, 2, 3],
[-5, -6, 2]])
>>> np.trace(a,-1)
-7
>>> np.trace(a,1)
-2
```

Matrix: Inverse

Transpose of a matrix is computed from numpy.transpose() function.

```
>>> import numpy as np
>>> a = np.array()[[3,-5,8],[-1,2,3],[-5,-6,2]]
>>>a
array([[ 3, -5, 8],
[-1, 2, 3],
[-5, -6, 2]])
>>> np.transpose(a)
array([[ 3, -1, -5],
[-5, 2, -6],
[ 8, 3, 2]])
```

NumPy Matrix Objects

- NumPy also has matrix objects that are an extension of arrays.
- These matrix objects have built in methods for determinant and inverse.

Solving the Matrix Equation

- Solving Systems of Equations
- A system of linear, algebraic equations can be written in matrix form.

$$\begin{array}{c} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_n \end{array} \qquad \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

- The 3×3 matrix is called the coefficient matrix
- The right-hand side is a vector.

Solving the Matrix Equation

- We will illustrate these methods for the simple system of equations:

$$
\begin{aligned}
4x-5y+8z&=4\\
2x-8y+7z&=0\\
-5x+8y&=-5
\end{aligned}
$$

scipy.linalg.solve()

This method takes the coefficient matrix and the right - hand side vector as arguments and return a vector with the solutions.

```
>>> import  numpy as np
>>> import numpy.linalg as slin
>>> cm = np.array([[4, -5, 8], [2, -8, 7], [-5, 8, 0]])
>>> rhs= np.array([4,0,-5])
>>> soln= np.slin.solve(cm,rhs)
>>> solnarray([ 1.53112033, 0.33195021, 0.05809129]
```

Eigenvalue problems

```
import  numpy as np # A = [[4, 2],
                    # [2, 5]]
A = np.array([[4, 2], [2, 5]])
v, U = np.linalg.eig(A)  # v = eigenvalues
                         # U = eigenvectors
 D = np.dot(np.dot(U.T, A), U)
```

- The function linalg.eig calculates the eigenvalues and eigenvectors of the real symmetric 2x 2 matrix A. We then diagonalize A by calculating the product.

$$D = U^T.A.U$$

Python: random walk

```
import numpy as np
import matplotlib.pyplot as pl
from numpy.random import RandomState

n = 1000 # number of steps r = RandomState()
p = np.zeros(n)

p[0] = 0.0

for i in range(n-1):
  if (r.rand() >= 0.5):
      p[i+1] = p[i] +1.
  else:
      p[i+1] = p[i] -1.
```

initialize array for number of steps

start at position = 0

loop through n-1 steps

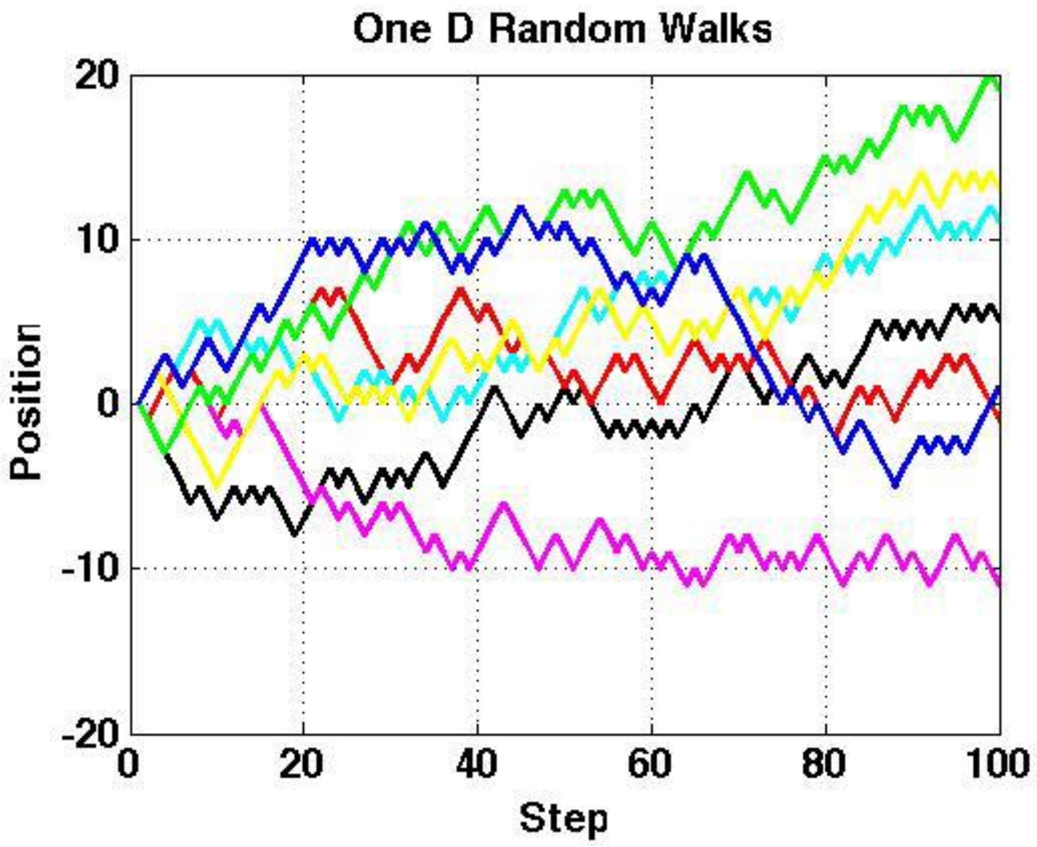
rand is uniformly distributed: 0->1

take forward step if > 0.5

take backward step if < 0.5

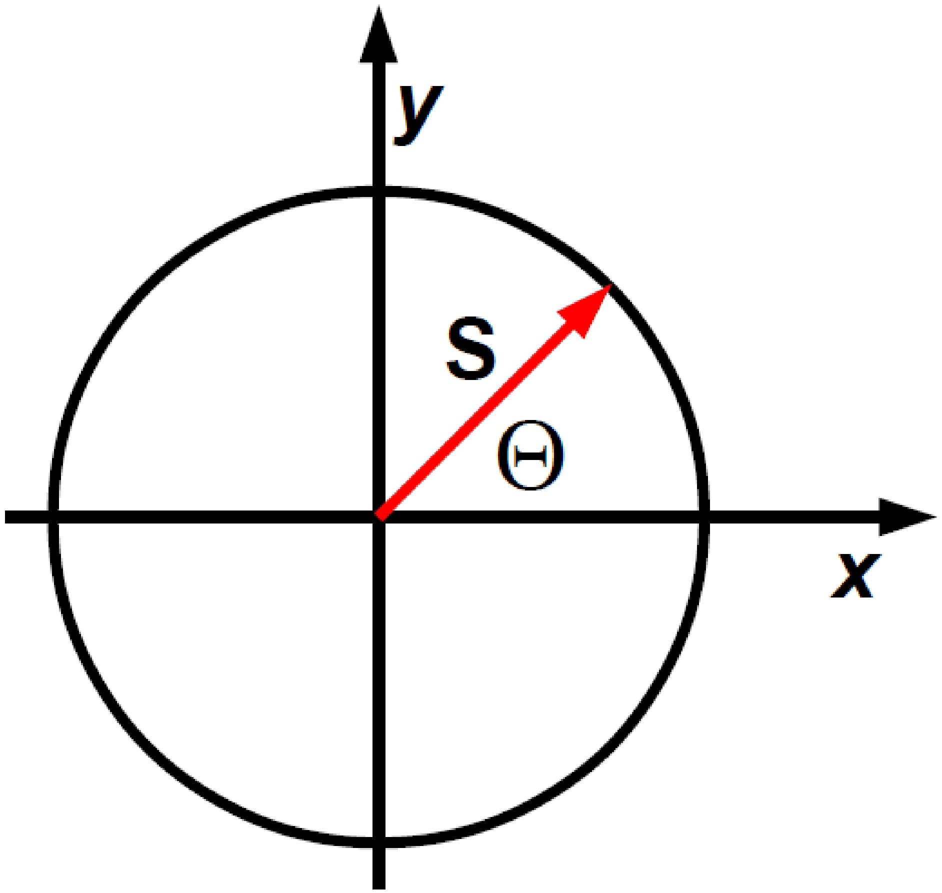
Python: random walk

```
a-) Write a code for 7 Random Walks each 100 steps
b-) Plot them position versus step
```



Python: Hw-I

```
Write a python code for 2D Random walk
```



Θ is random number: 0 - 2π

s is step size

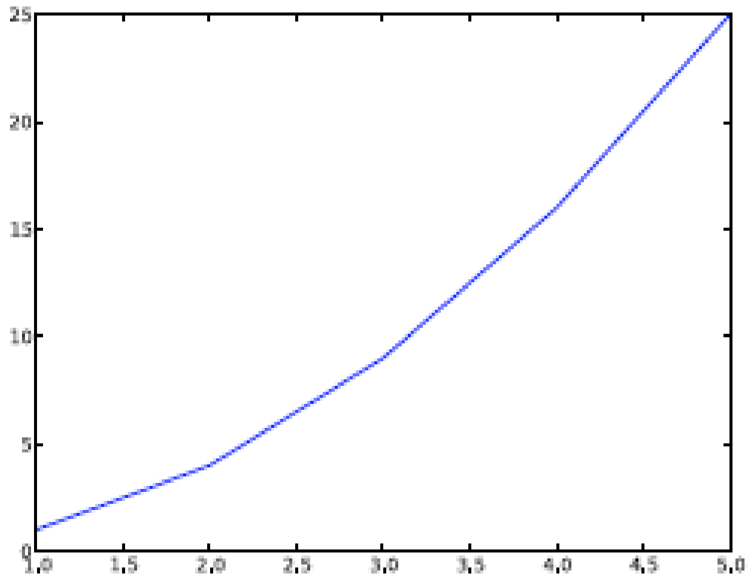
$x_{step} = s \cos\Theta$

$y_{step} = s \sin\Theta$

Plotting Examples

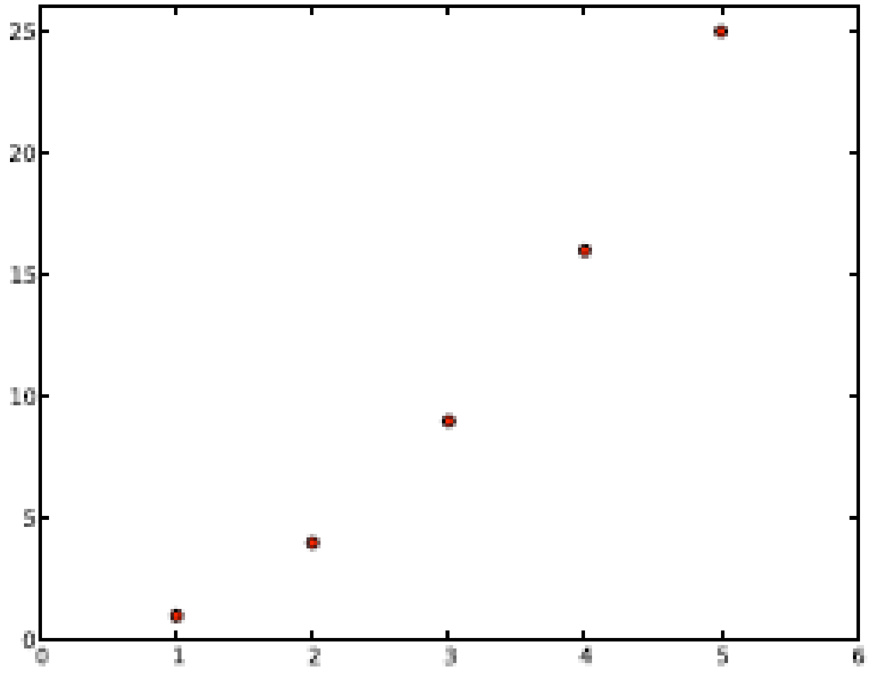
Python: Line plot

```
import numpy as np
import pylab as pl
#Make an array of x values
x=[1,2,3,4,5]
#Make an array of y values
y=[1,4,9,16,25]
#use pylab to plot x and y
pl.plot(x,y)
#Show the plot on the
screen
pl.show()
#Save the plot in pdf
format
pl.savefig('test.pdf')
```

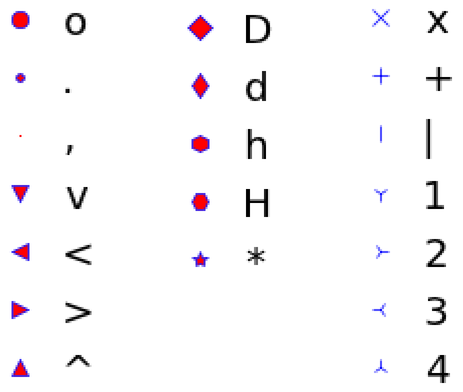


Python: Scatter plot

```
import numpy as np
import pylab as pl
#Make an array of x
values
x=[1,2,3,4,5]
#Make an array of y
values
y=[1,4,9,16,25]
#use pylab to plot x and
y as red circle
pl.plot(x,y,’ro’)
#Show the plot on the
screen
pl.show()
#Save the plot in pdf
format
pl.savefig(‘test.pdf’)
```

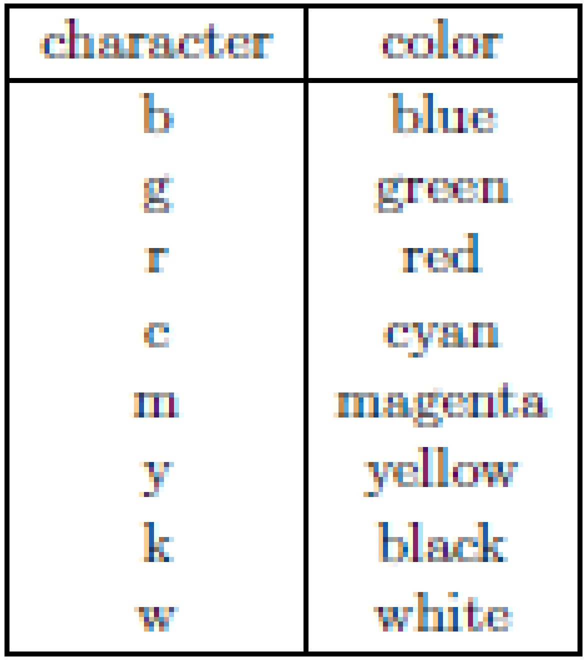


Marker list



Scatter plot

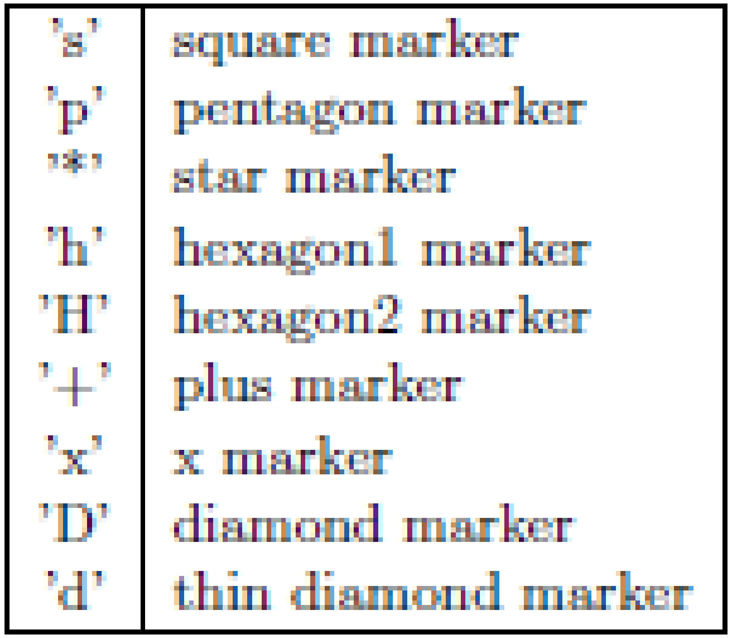
```
import numpy as np
import pylab as pl
#Make an array of x
values
x=[1,2,3,4,5]
#Make an array of y
values
y=[1,4,9,16,25]
#use pylab to plot x and
y as red circle
pl.plot(x,y,'r')
#Show the plot on the
screen
pl.show()
#Save the plot in pdf
format
pl.savefig('test.pdf')
```



character	color
b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Python: Scatter plot

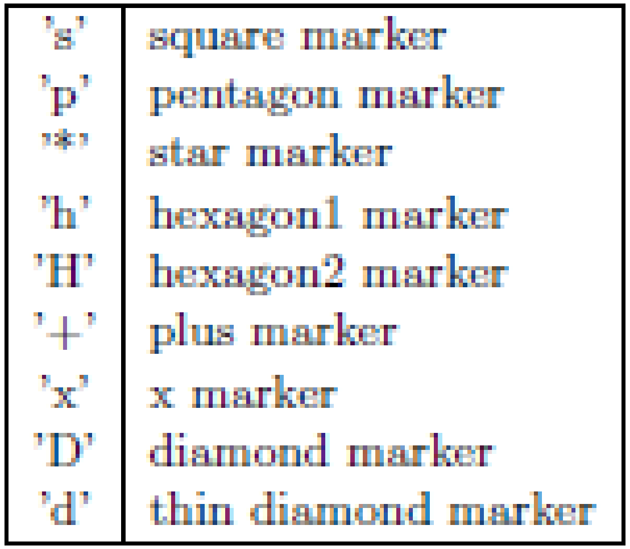
```
import numpy as np
import pylab as pl
#Make an array of x
values
x=[1,2,3,4,5]
#Make an array of y
values
y=[1,4,9,16,25]
#use pylab to plot x and
y as red circle
pl.plot(x,y,’b*’)
#Show the plot on the
screen
pl.show()
#Save the plot in pdf
format
pl.savefig(‘test.pdf’)
```



's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin diamond marker

Python: Plot style

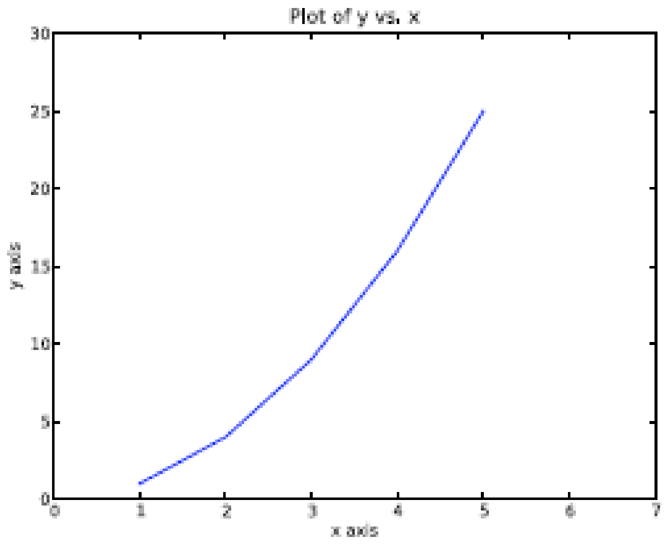
```
#label the axes
pl.xlabel('put text here')
pl.ylabel('put text here)
#set a title
pl.title('My first Plot')
#Set the x and y ranges
pl.xlim(x_low,x_high)
pl.ylim(y_low,y_high)
```



's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin diamond marker

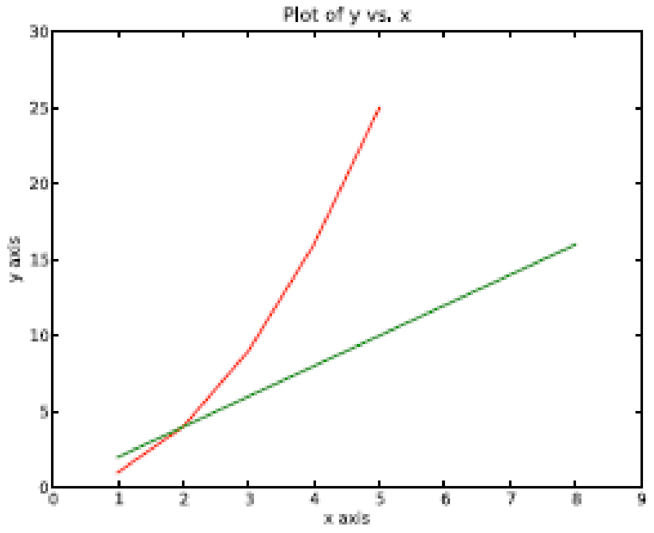
Pylab plot

```
import numpy as np
import pylab as pl
#Make an array of x values
x=[1,2,3,4,5]
#Make an array of y values
y=[1,4,9,16,25]
#make plot
pl.plot(x,y)
#Set title
pl.title('Plot of y vs x')
#Set labels
pl.xlabel('x axis')
pl.ylabel('y axis')
#Set the limits
pl.xlim(0.0,7.0)
pl.ylim(0.0,30.)
#use pylab to plot x and y as red
circle
pl.show()
```



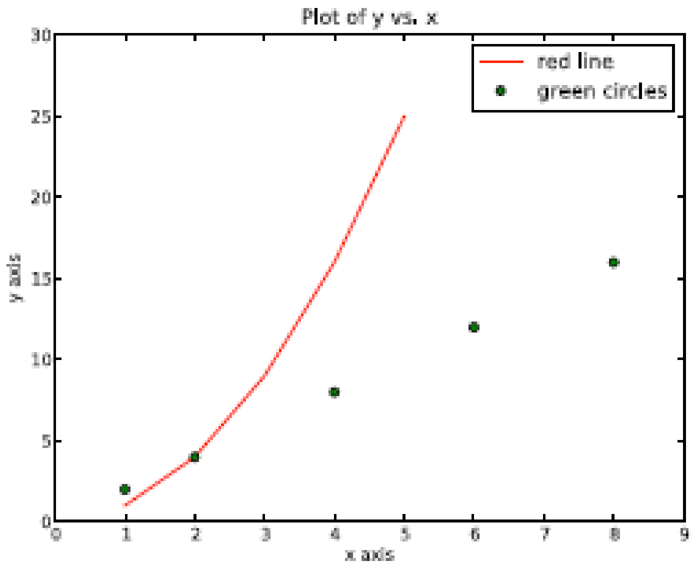
Pylab plot

```
import numpy as np
import pylab as pl
#Make x ,y  arrays for plot
x1=[1,2,3,4,5]
y1=[1,4,9,16,25]

x2=[1,2,4,6,8]
y2=[2,4,8,12,16]
#use pylab to plot x and y
pl.plot(x1,y1,'r')
pl.plot(x2,y2,'g')
#Set title
pl.title('Plot of y vs x')
#Set labels
pl.xlabel('x axis')
pl.ylabel('y axis')
#Set the limits
pl.xlim(0.0,9.0)
pl.ylim(0.0,30.)
#use pylab to plot x and y as red
circle
pl.show()
```



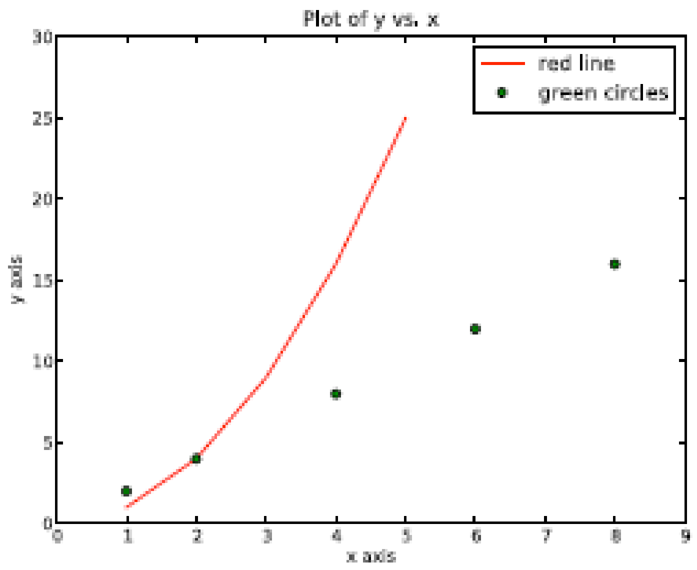
Pylab plot

```
import numpy as pn
import pylab as pl
#Make x ,y  arrays for plot
x1=[1,2,3,4,5]
y1=[1,4,9,16,25]
x2=[1,2,4,6,8]
y2=[2,4,8,12,16]
#use pylab to plot x and y
pl.plot(x1,y1,'r')
pl.plot(x2,y2,'go')
#Set title
pl.title('Plot of y vs x')
#Set labels
pl.xlabel('x axis')
pl.ylabel('y axis')
#Set the limits
pl.xlim(0.0,9.0)
Pl.ylim(0.0,30.)
#make legend
pl.legend([plot1,plot2],('red lines','green
circles'),'best',numpoints=1)
#use pylab to plot x and y as red circle
pl.show()
```



Pylab plot

```
import numpy as pn
import pylab as pl
#Make x ,y  arrays for plot
x1=[1,2,3,4,5]
y1=[1,4,9,16,25]
x2=[1,2,4,6,8]
y2=[2,4,8,12,16]
#use pylab to plot x and y
pl.plot(x1,y1,'r')
pl.plot(x2,y2,'go')
#Set title
pl.title('Plot of y vs x')
#Set labels
pl.xlabel('x axis')
pl.ylabel('y axis')
#Set the limits
pl.xlim(0.0,9.0)
pl.ylim(0.0,30.)
#make legend
pl.legend([plot1,plot2],('red
lines','green
circles'),'best',numpoints=1)
#use pylab to plot x and y as red circle
pl.show()
```

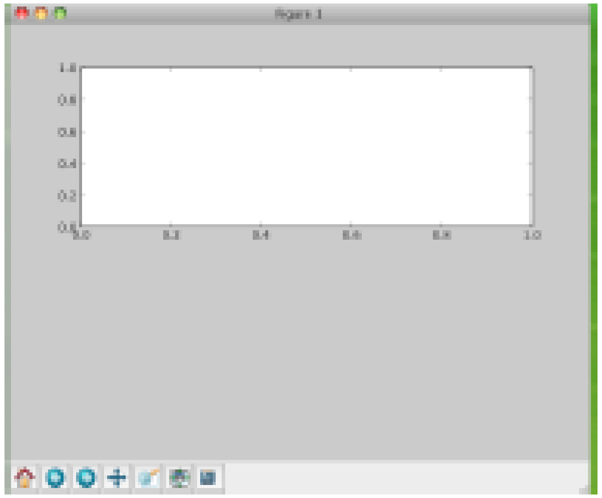


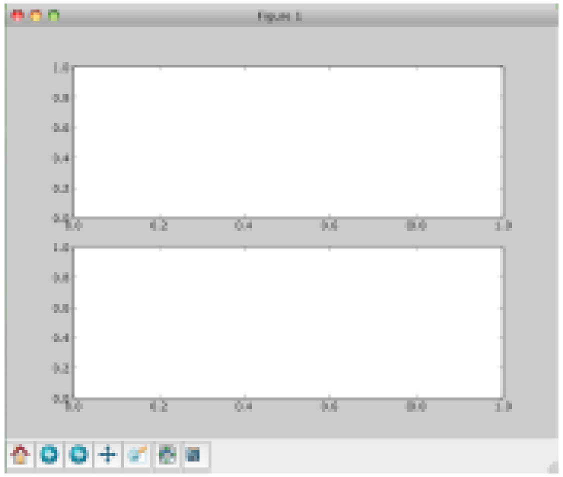
'upper right', 'upper left', 'center', 'lower left', 'lower right'.

Text	Number	
best	0	
upper right	1	
upper left	2	
lower left	3	
lower right	4	
right	5	
center left	6	= left
center right	7	= right
lower center	8	
upper center	9	
center	10	

Python: Plot

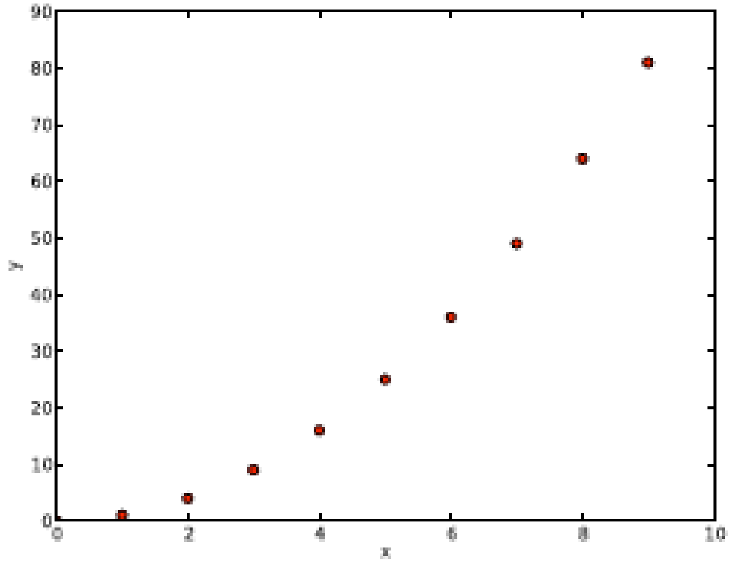
```
#you will have a figure with 2
rows 1 column
pl.subplot(211)


#you will have a figure with 2
rows 2 column
pl.subplot(212)


#reset the margins and gaps
between subplot
pl.subplots_adjust(left=0.08,rig
ht=0,95,wspace=0.25,hspace=0.45)
```





Reading data from ascii files

```
import numpy as np
import pylab as pl
data=np.loadtxt('fakedata.txt)
#plot the first column as x, and
second column as y
pl.plot(data[:,0],data[:,1],'ro')
pl.xlabel('x')
pl.ylabel('y')
pl.xlim(0.0,10.)
pl.show()
```



```
# fakedata.txt
0    0
1    1
2    4
3    9
4    16
5    25
6    36
7    49
8    64
9    81
0    0
1    1
2    4
3    9
4    16
5    25
6    36
7    49
8    64
9    81
```

Plot Histogram

```
import numpy as np
import pylab as pl
data=np.loadtxt('fakedata.txt)
#plot the first column as x,
and second column as y
pl.hist(data)
pl.xlabel('data')
pl.ylabel('arbitary')
pl.hist(data, color='green')
pl.xlim(0.0,100.)
pl.ylim(0.0,10.)
pl.hist(data, bins=20)
pl.show()
```

```
# fakedata.txt
0
1
4
9
16
25
36
49
64
81
25
25
16
49
16
64
25
25
 9
16
36
36
25
```

Plotting with pyplot

```
import numpy as np
import matplotlib.pyplot as plt
xvals = np.arange(-2, 1, 0.01) # Grid of 0.01 spacing
from -2 to 10
yvals = np.cos(xvals) # Evaluate function on xvals
plt.plot(xvals, yvals) # Create line plot with yvals
                 # against xvals
plt.show() # Show the figure
```

Plotting with pyplot

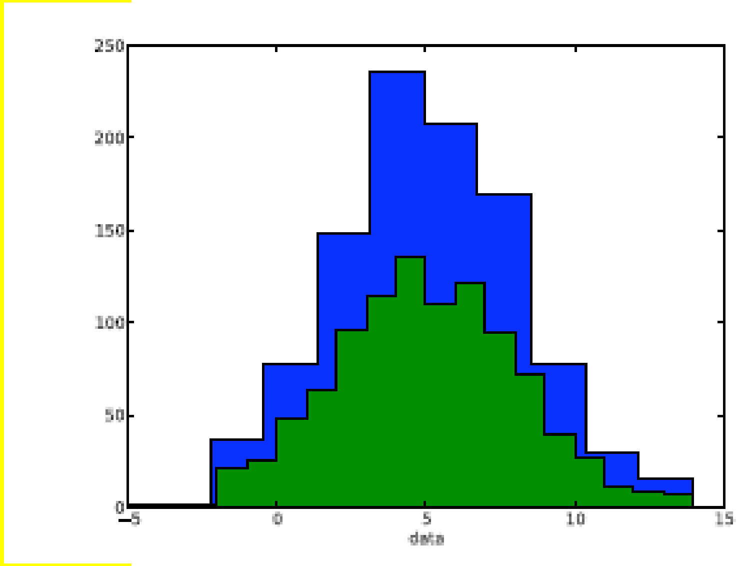
```
import numpy as np
import matplotlib.pyplot as plt
plt.figure() # Create a new figure window
xlist = np.linspace(-2.0, 1.0, 100) # Create 1-D arrays
for x,y dimensions
ylist = np.linspace(-1.0, 2.0, 100)
X,Y = np.meshgrid(xlist, ylist) # Create 2-D grid
                          #xlist,ylist values
Z = np.sqrt(X**2 + Y**2) # Compute function values on
                    # the grid
plt.contour(X, Y, Z, [0.5, 1.0, 1.2, 1.5], colors = 'k',
linestyles = 'solid')
plt.show()
```

Plotting with pyplot

```
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(0, 10, 1000)
y = np.power(x, 2)
plt.plot(x, y)
plt.show()
```

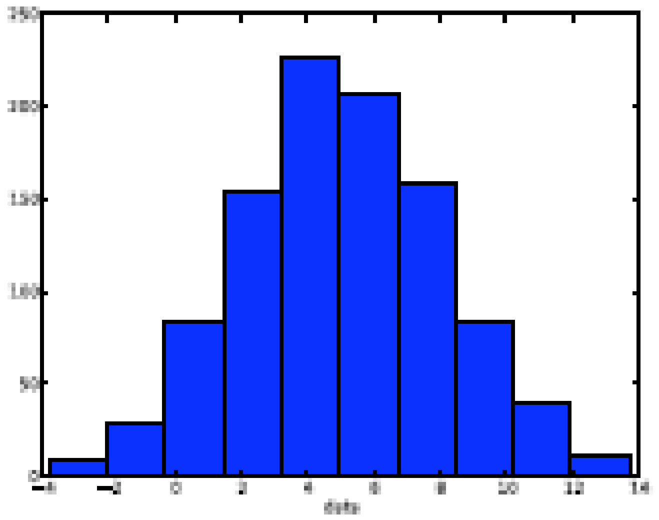
Pylab plot

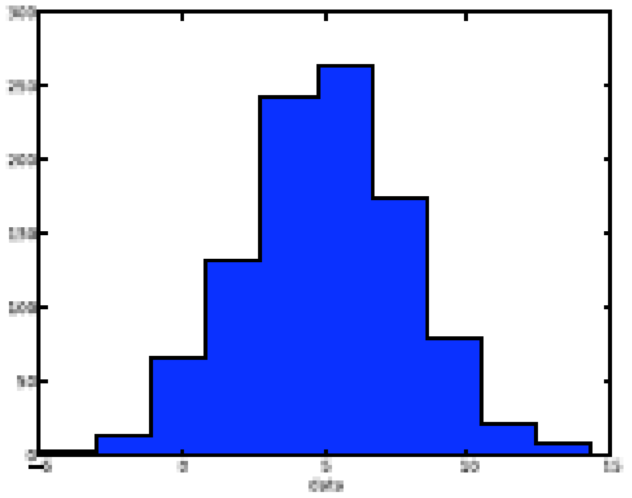
```
import numpy as np
import pylab as pl
#generate random number  usinng
#random.normal
mean= 5.0
rms=3.0
num=10000
data=np.random.normal(mean,rms,num)
bins=np.arange(-5.,16.,1.0)
#make a histogram of the data array
pl.hist(data,bins,histtype='stepfilled')
#Se the label
pl.xlabel('MC data')
pl.show()
```



Pylab plot

```
import numpy as np
import pylab as pl
#generate random number  using
#random.normal
mean= 5.0
rms=3.0
num=10000
data=np.random.normal(mean,rms,num)

#make a histogram of the data array
pl.hist(data)
#pl.hist(data,histtype=‘stepfilled’)
#Set the label
pl.xlabel(‘MC data’)
pl.show()
```


data


data