# Introduction to Python, Jupyter Notebook, NumPy and pandas

# Manmeet Singh

Scientist C

Centre for Climate Change Research

Indian Institute of Tropical Meteorology, Pune, India

Adapted from High Performance Scientific Computing , AMath 483/583 Class Notes, Spring Quarter, 2013, University of Washington

Python is an object oriented general-purpose language Advantages:

- Can be used interactively from a Python shell (similar to Matlab)
- Can also write scripts to execute from Unix shell
- Little overhead to start programming
- Powerful modern language
- Many modules are available for specialized work
- Good graphics and visualization modules
- Easy to combine with other languages (e.g. Fortran)
- Open source and runs on all platforms

Disadvantage: Can be slow to do certain things, such as looping over arrays.

- Code is interpreted rather than compiled
- Need to use suitable modules (e.g. NumPy) for speed.

Can easily create custom modules from compiled code written in Fortran, C, etc.

Can also use extensions such as Cython that makes it easier to mix Python with C code that will be compiled.

Python is often used for high-level scripts that e.g., download data from the web, run a set of experiments, collate and plot results.

Nearly everything in Python is an object of some class.

The class description tells what data the object holds (attributes) and what operations (methods or functions) are defined to interact with the object.

Every "variable" is really just a pointer to some object. You can reset it to point to some other object at will.

So variables don't have "type" (e.g. integer, float, string). (But the objects they currently point to do.)

# **Object-oriented language**

```
>>> x = 3.4
>>> print id(x), type(x)
# id() returns memory address
8645588 <type 'float'>
>>> x = 5
>>> print id(x), type(x)
8401752 <type `int'>
>>> x = [4, 5, 6]
>>> print id(x), type(x)
1819752 <type `list'>
>> x = [7, 8, 9]
>>> print id(x), type(x)
1843808 <type `list' > Page 5
```

# **Object-oriented language**

```
>>> x = [7,8,9]
>>> print id(x), type(x)
1843808 <type `list'>
>>> x.append(10)
>>> x
[7, 8, 9, 10]
>>> print id(x), type(x)
1843808 <type `list'>
```

Note: Object of type 'list' has a method 'append' that changes the object. A list is a mutable object.

# **Object-oriented language**

```
>>> x = [1, 2, 3]
>>> print id(x), x
1845768 [1, 2, 3]
>>> y = x
>>> print id(y), y
1845768 [1, 2, 3]
>>> y.append(27)
>>> y
[1, 2, 3, 27]
>>> x
[1, 2, 3, 27]
```

Note: x and y point to the same object!

```
>>> x = [1, 2, 3]
>>> print id(x), x
1845768 [1, 2, 3]
>>> y = list(x) # creates new list object
>>> print id(y), y
1846488 [1, 2, 3]
>>> y.append(27)
>>> y
[1, 2, 3, 27]
>>> x
[1, 2, 3]
```

```
If type (x) in [int, float], then setting y = x
creates a new object y pointing to a new location.
```

```
>>> x = 3.4
>>> print id(x), x
8645588 3.4
>>> y = x
>>> print id(y), y
8645572 3.4
>>> y = y+1
>>> print id(y), y
8645572 4.4
>>> print id(x), x
8645588 3.4
```

### The elements of a list can be any objects

```
(need not be same type):
```

```
>>> L = [3, 4.5, 'abc', [1,2]]
```

Indexing starts at 0:

```
>>> L[0]
3
>>> L[2]
'abc'
>>> L[3]
[1, 2]
>>> L[3][0] # element 0 of L[3]
1
```



Lists have several built-in methods, e.g. append, insert, sort, pop, reverse, remove, etc.

```
>>> L = [3, 4.5, 'abc', [1,2]]
>>> L2 = L.pop(2)
>>> L2
'abc'
>>> L
[3, 4.5, [1, 2]]
```

Note: L still points to the same object, but it has changed.

In IPython: Type L. followed by Tab to see all attributes and methods.

# Lists and tuples

```
>>> L = [3, 4.5, 'abc']
>>> L[0] = 'xy'
>>> T.
['xy', 4.5, 'abc']
A tuple is like a list but is immutable:
>>> T = (3, 4.5, 'abc')
>>> T[0]
```

```
3
```

```
>>> T[0] = 'xy'
```

Traceback (most recent call last):

```
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item
assignment
```

When you start Python it has a few basic built-in types and functions. To do something fancier you will probably import modules. Example: to use square root function:

>>> from numpy import sqrt
>>> sqrt(2.)
1.4142135623730951

When type import modname, Python looks on its search path for the file modname.py.

You can add more directories using the Unix environment variable PYTHONPATH.

Or, in Python, using the sys module:

```
>>> import sys
>>> sys.path # returns list of directories
['', '/usr/bin', ....]
>>> sys.path.append('newdirectory')
```

The empty string " in the search path means it looks first in the current directory.

# **Python modules**

#### Different ways to import:

- >>> from numpy import sqrt
- >>> sqrt(2.)
- 1.4142135623730951
- >>> from numpy import \*
- >>> sqrt(2.)
- 1.4142135623730951
- >>> import numpy
- >>> numpy.sqrt(2.)
- 1.4142135623730951
- >>> import numpy as np
- >>> np.sqrt(2.)
- 1.4142135623730951

Many tools are available for plotting numerical results.

Some open source Python options:

- matplotlib for 1d plots and 2d plots (e.g. pseudocolor, contour, quiver)
- Mayavi for 3d plots (curves, surfaces, vector fields)

Mayavi is easiest to get going by installing the Enthought Python Distribution (EPD), which is available for many platforms. (Also includes NumPy, SciPy, matplotlib.) Open source packages developed by National Labs...

- Vislt (https://wci.llnl.gov/simulation/computer-codes/visit/)
- ParaView (https://www.paraview.org/)

Harder to get going, but designed for large-scale 3d plots, distributed data, adaptive mesh refinement results, etc.:

Each have stand-alone GUI and also Python scripting capabilities.

Based on VTK (Visualization Tool Kit, https://vtk.org/).



- Shell is a program where users can type commands.
- With the shell, it's possible to invoke complicated programs like climate modeling software or simple commands that create an empty directory with only one line of code.
- The most popular Unix shell is Bash (the Bourne Again SHell so-called because it's derived from a shell written by Stephen Bourne).
- Bash is the default shell on most modern implementations of Unix and in most packages that provide Unix-like tools for Windows.

http://swcarpentry.github.io/shell-novice/

When the shell is first opened, you are presented with a prompt, indicating that the shell is waiting for input.

\$

Let's try our first command, which will list the contents of the current directory

\$ ls

Desktop Downloads Movies Pictures Documents Library Music Public

### pwd shows you where you are

\$ pwd

/home/manmeet

## cd is used to change the directory

\$ cd Documents

Lists in Python are quite general, can have arbitrary objects as elements.

Addition and scalar multiplication are defined for lists, but not what we want for numerical computation, e.g.

Multiplication repeats:

[2.0, 3.0, 2.0, 3.0]

Addition concatenates:

#### Instead, use NumPy arrays:

```
>>> import numpy as np
>>> x = np.array([2., 3.])
>>> 2*x
array([ 4., 6.])
```

## Other operations also apply component-wise:

```
>>> np.sqrt(x) * np.cos(x) * x**3
array([ -4.708164 , -46.29736719])
Note: * is component-wise multiply
```

Unlike lists, all elements of an np.array have the same type

>>> np.array([1, 2, 3]) # all integers array([1, 2, 3])

>>> np.array([1, 2, 3.]) # one float

array([ 1., 2., 3.]) # they're all floats!

Can explicitly state desired data type:

```
>>> x = np.array([1, 2, 3], dtype=complex)
>>> print x
[ 1.+0.j, 2.+0.j, 3.+0.j]
>>> (x + 1.j) * 2.j
array([-2.+2.j, -2.+4.j, -2.+6.j])
```

```
>>> A = np.array([[1.,2], [3,4], [5,6]])
>>> A
array([[ 1., 2.],
       [ 3., 4.],
       [ 5., 6.]])
>>> A.shape
(3, 2)
>>> A.T
array([[ 1., 3., 5.],
       [2., 4., 6.]])
>>> x = np.array([1., 1.])
>>> x.T
array([ 1., 1.])
```

```
>>> A
array([[ 1., 2.],
       [ 3., 4.],
       [ 5., 6.]])
>>> x
array([ 1., 1.])
>>> np.dot(A,x) # matrix-vector product
array([ 3., 7., 11.])
>>> np.dot(A.T, A) # matrix-matrix product
array([[ 35., 44.],
       [ 44., 56.]])
```

For Linear algebra, may instead want to use numpy.matrix:

```
>>> A = np.matrix([[1.,2], [3,4], [5,6]])
>>> A
matrix([[ 1., 2.],
      [ 3., 4.],
      [ 5., 6.]])
```

Or, Matlab style (as a string that is converted):

```
>>> A = np.matrix("1.,2; 3,4; 5,6")
>>> A
matrix([[ 1., 2.],
      [ 3., 4.],
      [ 5., 6.]])
```

# NumPy matrices for vectors and matrices

Note: vectors are handled as matrices with 1 row or column:

```
>>> x = np.matrix("4.;5.")
>>> x
matrix([[ 4.],
         [5.1]
>>> x.T
matrix([[ 4., 5.]])
>>> A*x
matrix([[ 14.],
         [ 32.],
         [ 50.]])
```

But note that indexing into x requires two indices:

```
>>> print x[0,0], x[1,0]
4.0 5.0 Page 26
```

For linear algebra matrix may be easier (and more like Matlab), but vectors need two subscripts!

For most other uses, arrays more natural, e.g.

>>> 
$$y = x^{*}5 - 2.*sqrt(x)*cos(x) # 100 values$$

>>> plot(x,y)

np.linspace returns an array, which is what is needed here.

We will always use arrays.

See http://www.scipy.org/NumPy\_for\_Matlab\_Users

The rank of an array is the number of subscripts it takes:

```
>>> A = np.ones((4,4))
>>> A
array([[ 1., 1., 1., 1.],
      [ 1., 1., 1., 1.],
      [ 1., 1., 1., 1.],
      [ 1., 1., 1., 1.]])
>>> np.rank(A)
2
```

Warning: This is not the rank of the matrix in the linear algebra sense (dimension of the column space)!

# Rank of an matrix in Linear Algebra

G <b>()</b> gle	rank of a matrix
	All Books 🖬 Images 🗈 Videos 🗐 News 🗄 More Settings Tools
	About 14,10,00,000 results (0.57 seconds)
	The maximum number of linearly independent vectors in a matrix is equal to the number of non-zero rows in its row echelon matrix. Therefore, to find the rank of a matrix, we simply transform the matrix to its row echelon form and count the number of non-zero rows. $\begin{pmatrix} z & z & z & z & z & z & z & z & z & z $
	Matrix Rank - Stat Trek

Scalars have rank 0:

```
>>> z = np.array(7.)
>>> z
array(7.0)
```

NumPy arrays of any dimension are supported, e.g. rank 3:

```
>>> T = np.ones((2,2,2))
>>> T
array([[[ 1., 1.],
       [ 1., 1.]],
       [[ 1., 1.]],
       [[ 1., 1.]])
>>> T[0,0,0]
1.0
```

```
>>> A = np.array([[1., 2.], [3, 4]])
   >>> A
   array([[ 1., 2.],
          [3., 4.]])
   >>> b = np.dot(A, np.array([8., 9.]))
   >>> b
   array([ 26., 60.])
Now solve Ax = b:
   >>> from numpy.linalg import solve
   >>> solve(A,b)
   array([ 8., 9.])
```

>>> from numpy.linalg import eig >>> eig(A) # returns a tuple (evals, evecs) (array([-0.37228132, 5.37228132]), array([-0.82456484, -0.41597356],[0.56576746, -0.90937671]]))>>> evals, evecs = eig(A) # unpacks tuple >>> evals array([-0.37228132, 5.37228132]) >>> evecs array([[-0.82456484, -0.41597356], [0.56576746, -0.90937671]])

# **Quadrature (numerical integration)**

Estimate 
$$\int_0^2 x^2 dx = 8/3$$

- >>> from scipy.integrate import quad
- >>> def f(x):
- ... return  $x^*2$

•••

- >>> quad(f, 0., 2.)
- (2.6666666666666667, 2.960594732333751e-14)

returns (value, error estimate).

Other keyword arguments to set error tolerance, for example.

In the last example, f is so simple we might want to just include its definition directly in the call to quad.

We can do this with a lambda function:

>>> f = lambda x: x\*\*2 >>> f(4) 16

This defines the same f as before. But instead we could do:

>>> quad(lambda x: x\*\*2, 0., 2.)

(2.666666666666667, 2.960594732333751e-14)

Python modules often end with a section that looks like:

- if \_\_\_\_\_name\_\_\_ == "\_\_\_\_main\_\_\_":
- # some code

This code is **not** executed if the file is imported as a module, only if it is run as a script, e.g. by...

```
$ python filename.py
>>> execfile("filename.py")
In[1]: run filename.py
```

# **Interactive Python, Indentation & Wrapping Lines**

#### **Interactive Python**

The IPython shell is generally recommended for interactive work in Python (see <a href="http://ipython.org/documentation.html">http://ipython.org/documentation.html</a>), but for most examples we'll display the >>> prompt of the standard Python shell.

Normally multiline Python statements are best written in a text file rather than typing them at the prompt, but some of the short examples below are done at the prompt. If type a line that Python recognizes as an unfinished block, it will give a line starting with three dots, like:



print("oops!")
else:
 print("this is what we expect")

this is what we expect

Once done with the full command, typing return alone at the ... prompt tells Python we are done and it executes the command.

#### Indentation

Most computer languages have some form of begin-end structure, or opening and closing braces, or some such thing to clearly delinieate what piece of code is in a loop, or in different parts of an if-then-else structure like what's shown above. Good programmers generally also indent their code so it is easier for a reader to see what is inside a loop, particularly if there are multiple nested loops. But in most languages this is indentation is just a matter of style and the begin-end structure of the language determines how it is actually interpreted by the computer.

In Python, indentation is everything. There are no begin-end's, only indentation. Everything that is supposed to be at one level of a loop must be indented to that level. Once the loop is done the indentation must go back out to the previous level. There are some other rules you need to learn, such as that the "else" in and if-else block like the above has to be indented exactly the same as as the "if". See if\_else for more about this.

How many spaces to indent each level is a matter of style, but you must be consistent within a single code. The standard is often 4 spaces.

#### Wrapping lines

In Python normally each statement is one line, and there is no need to use separators such as the semicolon used in some languages to end a line. One the other hand you can use a semicolon to put several short statements on a single line, such as:

In [5]: x = 5; print(x)

5

It is easiest to read codes if you avoid this in most cases.

If a line of code is too long to fit on a single line, you can break it into multiple lines by putting a backslash at the end of a line:

### **Comments and Strings**

If a line of code is too long to fit on a single line, you can break it into multiple lines by putting a backslash at the end of a line:

```
In [6]: y = 3 + \
... 4
In [7]: y
Out[7]: 7
```

### Comments

Anything following a # in a line is ignored as a comment (unless of course the # appears in a string):

```
In [8]: s = "This # is part of the string" # this is a comment
```

In [9]: s

```
Out[9]: 'This # is part of the string'
```

There is another form of comment, the docstring, discussed below following an introduction to strings.

### Strings

Strings are specified using either single or double quotes:

In [10]: s = 'some text'
s = "some text"

are the same. This is useful if you want strings that themselves contain quotes of a different type.

You can also use triple double quotes, which have the advantage that they such strings can span multiple lines:

```
In [11]: s = """Note that a ' doesn't end
   ... this string and that it spans two lines"""
```

In [12]: s

Out[12]: "Note that a ' doesn't end\nthis string and that it spans two lines"

## **Docstrings, Python scripts and Python objects**

#### In [13]: print(s)

Note that a ' doesn't end this string and that it spans two lines

When it prints, the carriage return at the end of the line show up as "n". This is what is actually stored. When we "print s" it gets printed as a carriage return again.

You can put "n" in your strings as another way to break lines:

#### In [14]: print("This spans \n two lines")

This spans two lines

### Docstrings

Often the first thing you will see in a Python script or module, or in a function or class defined in a module, is a brief description that is enclosed in triple quotes. Although ordinarily this would just be a string, in this special position it is interpreted by Python as a comment and is not part of the code. It is called the docstring because it is part of the documentation and some Python tools automatically use the docstring in various ways. See ipython for one example. Also the documentation formatting program Sphinx that is used to create these class notes can automatically take a Python module and create thml or latex documentation for it by using the docstrings, the original purpose for which Sphinx was developed. See Sphinx documentation for more about this.

It's a good idea to get in the habit of putting a docstring at the top of every Python file and function you write.

### **Running Python scripts**

Most Python programs are written in text files ending with the .py extension. Some of these are simple scripts that are just a set of Python instructions to be executed, the same things you might type at the >>> prompt but collected in a file (which makes it much easier to modify or reuse later). Such a script can be run at the Unix command line simply by typing "python" followed by the file name.

See Python scripts and modules for some examples. The section Importing modules also contains important information on how to "import" modules, and how to set the path of directories that are searched for modules when you try to import a module.

### Python objects

Python is an object-oriented language, which just means that virtually everything you encounter in Python (variables, functions, modules, etc.) is an object of some class. There are many classes of objects built into Python and in this course we will primarily be using these pre-defined classes. For large-scale programming projects you would probably define some new classes, which is easy to do. (Maybe an example to come...)

The type command can be used to reveal the type of an object:

### **Docstrings, Python scripts and Python objects**

In [15]:	<pre>import numpy as np print(type(np))</pre>
	<class 'module'=""></class>
[16]:	<pre>print(type(np.pi))</pre>
	<class 'float'=""></class>
[n [17]:	<pre>print(type(np.cos))</pre>
	<class 'numpy.ufunc'=""></class>
	We see that np is a module, np.pi is a floating point real number, and np.cos is of a special class that's defined in the numpy module.
	The linspace command creates a numerical array that is also a special numpy class:
In <mark>[18</mark> ]:	x = np.linspace(0, 5, 6)
[n [19]:	print(x)
	[0. 1. 2. 3. 4. 5.]
[n [20]:	<pre>print(type(x))</pre>
	<class 'numpy.ndarray'=""></class>
	Objects of a particular class generally have certain operations that are defined on them as part of the class definition. For example, NumPy numerical arrays have a max method defined, which we can use on x in one of two ways:
In [21]:	np.max(x)
Out[21]:	5.0
In [22]:	x.max()
Out[22]:	5.0

The first way applies the method max defined in the numpy module to x. The second way uses the fact that x, by virtue of being of type numpy.ndarray, automatically has a max method which can be invoked (on itself) by calling the function x.max() with no argument. Which way is better depends in part on what you're doing.

### **Declaring Variables**

#### Here's another example:

In [23]:	L = [0, 1, 2]
In [24]:	type(L)
Out[24]:	list
In [25]:	L.append(4)
In [26]:	L
Out[26]:	[0, 1, 2, 4]

L is a list (a standard Python class) and so has a method append that can be used to append an item to the end of the list.

### **Declaring variables?**

In many languages, such as Fortran, you must generally declare variables before you can use them and once you've specified that x is a real number, say, that is the only type of things you can store in x, and a statement like x = 'string' would not be allowed.

In Python you don't declare variables, you can just type, for example:

In [27]:	x = 3.4
In [28]:	2*x
Out[28]:	6.8
In [30]:	x = 'string'
In [31]:	2*x
Out[31]:	'stringstring'
In [32]:	x = [4, 5, 6]
In [33]:	2*x
Out[33]:	[4, 5, 6, 4, 5, 6]

Here x is first used for a real number, then for a character string, then for a list. Note, by the way, that multiplication behaves differently for objects of different type (which has been specified as part of the definition of each class of objects).

In Fortran if you declare x to be a real variable then it sets aside a particular 8 bytes of memory for x, enough to hold one floating point number. There's no way to store 6 characters or a list of 3 integers in these 8 bytes.

In Python it is often better to think of x as simply being a pointer that points to some object. When you type "x = 3.4" Python creates an object of type float holding one real number and points x to that. When you type x ='string' it creates a new object of type str and now points x to that, and so on.

### Lists

We have already seen lists in the example above.

Note that indexing in Python always starts at 0:

In [34]:	L = [4,5,6]	
In [35]:	L[0]	
Out[35]:	4	
In [36]:	L[1]	
Out[36]:	5	
	Elements of a list need not all have	the same type. For example, here's a list with 5 elements:
In [37]:	L = [5, 2.3, 'abc', [4,'b'	], np.cos]
	Here's a way to see what each elen	nent of the list is, and its type:
In [39]:	<pre>for index,value in enumera     print('L[%s] is %16s</pre>	te(L): %s' % (index,value,type(value)))
	L[0] is 5	<class 'int'=""></class>
	L[1] is 2.3	
	L[2] is abc	<class 'str'=""></class>
	L[3] is [4, 'b']	<class 'list'=""></class>
	L[4] is <ufunc 'cos'=""></ufunc>	<class 'numpy.ufunc'=""></class>

Note that L[3] is itself a list containing an integer and a string and that L[4] is a function.

### **Copying Objects**

One nice feature of Python is that you can also index backwards from the end: since L[0] is the first item, L[-1] is what you get going one to the left of this, and wrapping around (periodic boundary conditions in math terms):

```
In [41]: for index in [-1, -2, -3, -4, -5]:
    print('L[%s] is %l6s' % (index, L[index]))
```

L[-1] is <ufunc 'cos'> L[-2] is [4, 'b'] L[-3] is abc L[-4] is 2.3 L[-5] is 5

In particular, L[-1] always refers to the last item in list L.

### **Copying objects**

One implication of the fact that variables are just pointers to objects is that two names can point to the same object, which can sometimes cause confusion. Consider this example:

```
In [42]: x = [4,5,6]

In [43]: y = x

In [44]: y

Out[44]: [4, 5, 6]

In [45]: y.append(9)

In [47]: y

Out[47]: [4, 5, 6, 9]

So far nothing too surprising. We initialized y to be x and then we appended another list element to y. But take a look at x:

In [48]: x

Out[48]: [4, 5, 6, 9]

We didn't really append 9 to y, we appended it to the object y points to, which is the same object x points to!

Failing to pay attention to this sort of thing can lead to programming nightmares.

What if we really want y to be a different object that happens to be initialized by copying x? We can do this by:
```

# **Copying Objects**

In [49]:	x = [4,5,6]
In [50]:	y = list(x)
In [51]:	У
Out[51]:	[4, 5, 6]
In [52]:	y.append(9)
In [53]:	y .
Out[53]:	[4, 5, 6, 9]
In [54]:	x
Out[54]:	[4, 5, 6]
	This is what we want. Here list(x) creates a new object, that is a list, using the elements of the list x to initialize it, and y points to this new object. Changing this object doesn't change the one x pointed to. You could also use the copy module, which works in general for any objects:
In [55]:	<pre>import copy y = copy.copy(x)</pre>
	Sometimes it is more complicated, if the list x itself contains other objects. See http://docs.python.org/library/copy.html for more information. There are some objects that cannot be changed once created (immutable objects, as described further below). In particular, for floats and integers, you can do things like:
In [56]:	x = 3.4 y = x y = y+1 y
Out[56]:	4.4
In [57]:	×
Out[57]:	3.4
	Here changing y did not change x, luckily. We don't have to explicitly make a copy of x for y in this case. If we did, writing any sort of numerical code in Python would be a nightmare.

# Mutable and Immutable objects

	We didn't because the command:
In [58]:	y = y+1
	above is not changing the object y points to, instead it is creating a new object that y now points to, while x still points to the old object.
	For more about built-in data types in Python, see http://docs.python.org/release/2.5.2/ref/types.html.
	Mutable and Immutable objects
	Some objects can be changed after they have been created and others cannot be. Understanding the difference is key to understanding why the examples above concerning copying objects behave as they do.
	A list is a mutable object. The statement:
In [59]:	x = [4,5,6]
	above created an object that x points to, and the data held in this object can be changed without having to create a new object. The statement
	y = x
	points y at the same object, and since it can be changed, any change will affect the object itself and be seen whether we access it using the pointer x or y.
	We can check this by:
In [60]:	id(x)
Out[60]:	140418456523464
In [61]:	id(y)
Out[61]:	140417799512208
	The id function just returns the location in memory where the object is stored. If you do something like x[0] = 1, you will find that the objects' id's have not changed, they both point to the same object, but the data stored in the object has changed.
	Some data types correspond to immutable objects that, once created, cannot be changed. Integers, floats, and strings are immutable:
In [62]:	s = "This is a string"
In [66]:	s[0]
Out[66]:	'T'

### **Mutable and Immutable objects**

```
In [64]: s[0] = 'b'
                                                         Traceback (most recent call last)
          TypeError
          <ipython-input-64-502f3bf853al> in <module>
          ---> 1 s[0] = 'b'
          TypeError: 'str' object does not support item assignment
In [67]: id(s)
Out[67]: 140417799495088
In [68]: s = "New string"
In [69]: id(s)
Out[69]: 140417799516592
          What happened to the old object? It depends on whether any other variable was pointing to it. If not, as in the example above, then Python's garbage
          collection would recognize it's no longer needed and free up the memory for other uses. But if any other variable is still pointing to it, the object will still exist,
          e.g.
In [70]: s2=s
In [72]: id(s2) # same object as s above
Out[72]: 140417799516592
In [73]: s = "Yet another string" # creates a new object
In [74]: id(s)
                                         # s now points to new object
Out[74]: 140417799494512
In [76]: id(s2)
                                         # s2 still points to the old one
Out[76]: 140417799516592
```

### **Tuples and Iterators**

### Tuples

We have seen that lists are mutable. For some purposes we need something like a list but that is immuatable (e.g. for dictionary keys, see below). A tuple is like a list but defined with parentheses (...) rather than square brackets [..]:

- In [77]: t = (4,5,6)
- In [78]: t[0]
- Out[78]: 4
- In [79]: t[0] = 9

TypeError Traceback (most recent call last)
<ipython-input-79-9a6f3bfc01f7> in <module>
----> 1 t[0] = 9

TypeError: 'tuple' object does not support item assignment

### Iterators

We often want to iterate over a set of things. In Python there are many ways to do this, and it often takes the form:

```
In []: for A in B:
    # do something, probably involving the current A
```

In this construct B is any Python object that is iterable, meaning it has a built-in way (when B's class was defined) of starting with one thing in B and progressing through the contents of B in some hopefully logical order.

Lists and tuples are iterable in the obvious way: we step through it one element at a time starting at the beginning:

```
In [83]: for i in [3, 7, 'b']:
    print("i is now ", i)
```

i is now 3 i is now 7 i is now b

### range and enumerate

```
range
           In numerical work we often want to have i start at 0 and go up to some number N, stepping by one. We obviously don't want to have to construct the list [0, 1,
           2, 3, ..., N] by typing all the numbers when N is large, so Python has a way of doing this:
In [87]: range(7)
Out[87]: range(0, 7)
           NOTE: The last element is 6, not 7. The list has 7 elements but starts by default at 0, just as Python indexing does. This makes it convenient for doing things
           like:
In [88]: L = ['a', 8, 12]
In [89]: for i in range(len(L)):
                print("i = ", i, " L[i] = ", L[i])
           i = 0 L[i] = a
           i = 1 L[i] = 8
           i = 2 L[i] = 12
           Note that len(L) returns the length of the list, so range(len(L)) is always a list of all the valid indices for the list L.
           enumerate
           Another way to do this is:
In [91]: for i,value in enumerate(L):
                print("i = ",i, " L[i] = ",value)
           i = 0 L[i] = a
          i = 1 L[i] = 8
           i = 2 L[i] = 12
           range can be used with more arguments, for example if you want to start at 2 and step by 3 up to 20:
In [94]: range(2,20,3)
Out[94]: range(2, 20, 3)
           Note that this doesn't go up to 20. Just like range(7) stops at 6, this list stops one item short of what you might expect.
```

NumPy has a linspace command that behaves like Matlab's, which is sometimes more useful in numerical work, e.g.:

In [95]:	np.lins	pace(	2,20,	7)					
Out[95]:	array([	2.,	5.,	8.,	11.,	14.,	17.,	20.])	

This returns a NumPy array with 7 equally spaced points between 2 and 20, including the endpoints. Note that the elements are floats, not integers. You could use this as an iterator too.

If you plan to iterate over a lot of values, say 1 million, it may be inefficient to generate a list object with 1 million elements using range. So there is another option called xrange, that does the iteration you want without explicitly creating and storing the list:

for i in xrange(1000000):

# do something

does what we want.

## **Python scripts and modules**

### Python scripts and modules

A Python script is a collection of commands in a file designed to be executed like a program. The file can of course contain functions and import various modules, but the idea is that it will be run or executed from the command line or from within a Python interactive shell to perform a specific task. Often a script first contains a set of function definitions and then has the main program that might call the functions.

#### script1.py

```
0.0.0.
In [96]:
         Sample script to print values of a function at a few points.
          H H H
         import numpy as np
         def f(x):
             11.11.11
             A quadratic function.
             y = x^{**2} + 1.
             return y
         print("
                  ×
                              f(x)")
         for x in np.linspace(0,4,3):
             print("%8.3f %8.3f" % (x, f(x)))
              X
                       f(x)
            0.000
                      1.000
            2.000
                      5.000
```

17.000

The main program starts with the print statement.

There are several ways to run a script contained in a file.

At the Unix prompt:

4.000

\$ python	<pre>script1.py</pre>
х	f(x)
0.000	1.000
2.000	5.000
4.000	17.000

# Python scripts and modules

	From within P	, and the second s
In [101]:	exec(open(	"scriptl.py").read())
	x	f(x)
	0.000	1.000
	2.000	5.000
	4.000	17.000
	From within IF	Python, using either execfile as above, or run:
In <mark>[10</mark> 2]:	run script	1.ру
	x	f(x)
	0.000	1.000
	2.000	5.000
	4.000	17.000
	Or, you can in	nport the file as a module (see Importing modules below for more about this):
In [103]:	Or, you can in import scr	
In [103]:	<pre>import scr x</pre>	ipt1 f(x)
In [103]:	import scr x 0.000	ipt1 f(x) 1.000
In [103]:	x 0.000 2.000	ipt1 f(x) 1.000 5.000
In [103]:	import scr x 0.000	ipt1 f(x) 1.000
In [ <mark>103]:</mark>	x 0.000 2.000 4.000 Note that this	ipt1 f(x) 1.000 5.000 17.000
	x 0.000 2.000 4.000 Note that this	<pre>ipt1     f(x)     1.000     5.000     17.000 also gives the same output. Whenever a module is imported, any statements that are in the main body of the module are executed when it is ddition, any variables or functions defined in the file are available as attributes of the module, e.g.,</pre>
	x 0.000 2.000 4.000 Note that this imported. In a script1.f(	<pre>ipt1     f(x)     1.000     5.000     17.000 also gives the same output. Whenever a module is imported, any statements that are in the main body of the module are executed when it is ddition, any variables or functions defined in the file are available as attributes of the module, e.g.,</pre>
In [104]: Dut[104]:	x 0.000 2.000 4.000 Note that this imported. In a script1.f(	<pre>ipt1     f(x)     1.000     5.000     17.000 also gives the same output. Whenever a module is imported, any statements that are in the main body of the module are executed when it is ddition, any variables or functions defined in the file are available as attributes of the module, e.g., 4)</pre>

## Python scripts and modules

	Note there are some differences between executing the script and importing it. When it is executed as a script, it is as if the commands were typed at command line. Hence:
In [106]:	exec(open("script1.py").read())
	x f(x)
	0.000 1.000
	2.000 5.000
	4.000 17.000
In [107]:	
Out[107]:	<functionmainf(x)></functionmainf(x)>
In [108]:	np

In this case f and np are in the namespace of the interactive session as if we had defined them at the prompt.

## Writing scripts for ease of importing

```
Writing scripts for ease of importing
           The script used above as an example contains a function f(x) that we might want to be able to import without necessarily running the main program. This
           can be arranged by modifying the script as follows:
           script2.py
In [109]: """
            Sample script to print values of a function at a few points.
            The printing is only done if the file is executed as a script, not if it is
            imported as a module.
            11.11.11
            import numpy as np
            def f(x):
                A quadratic function.
                11.11
                y = x^{**2} + 1.
                return y
            def print table():
                print("
                                      f(x)")
                           X
                for x in np.linspace(0,4,3):
                    print("%8.3f %8.3f" % (x, f(x)))
            if name == " main ":
                print table()
                 х
                          f(x)
               0.000
                          1.000
               2.000
                        5.000
               4.000 17.000
           When a file is imported or executed, an attribute name is automatically set, and has the value main only if the file is executed as a script, not if it is imported
           as a module. So we see the following behavior:
```

\$ python	<pre>script2.py </pre>
х	f(x) < br >
0.000	1.000
2.000	5.000
4.000	17.000

# **Reloading Modules**

	as with script1.py, but:
n [110]:	<pre>import script2 # does not print table</pre>
n [112]:	<pre>script2name # not 'main'</pre>
ut[112]:	'script2'
n [113]:	<pre>script2.f(4)</pre>
ut[113]:	17.0
n [114]:	<pre>script2.print_table()</pre>
	x f(x) 0.000 1.000 2.000 5.000 4.000 17.000
	Deleading medules
	Reloading modules
	Reloading modules When you import a module, Python keeps track of the fact that it is imported and if it encounters another statement to import the same module will not bother to do so again (the list of modules already import is in sys.modules). This is convenient since loading a module can be time consuming. So if you're debugging a script using execfile or run from an IPython shell, each time you change it and then re-execute it will not reload numpy, for example.
	When you import a module, Python keeps track of the fact that it is imported and if it encounters another statement to import the same module will not bother to do so again (the list of modules already import is in sys.modules). This is convenient since loading a module can be time consuming. So if you're
	When you import a module, Python keeps track of the fact that it is imported and if it encounters another statement to import the same module will not bother to do so again (the list of modules already import is in sys.modules). This is convenient since loading a module can be time consuming. So if you're debugging a script using execfile or run from an IPython shell, each time you change it and then re-execute it will not reload numpy, for example.
n [115]:	When you import a module, Python keeps track of the fact that it is imported and if it encounters another statement to import the same module will not bother to do so again (the list of modules already import is in sys.modules). This is convenient since loading a module can be time consuming. So if you're debugging a script using exectile or run from an IPython shell, each time you change it and then re-execute it will not reload numpy, for example. Sometimes, however, you want to force reloading of a module, in particular if it has changed (e.g. when we are debugging it). Suppose, for example, that we modify script2.py so that the quadratic function is changed from $y = x^2 + 1$ to $y = x^2 + 10$ . If we make this change and
	When you import a module, Python keeps track of the fact that it is imported and if it encounters another statement to import the same module will not bother to do so again (the list of modules already import is in sys.modules). This is convenient since loading a module can be time consuming. So if you're debugging a script using execfile or run from an IPython shell, each time you change it and then re-execute it will not reload numpy, for example. Sometimes, however, you want to force reloading of a module, in particular if it has changed (e.g. when we are debugging it). Suppose, for example, that we modify script2.py so that the quadratic function is changed from $y = x^2 + 1$ to $y = x^2 + 10$ . If we make this change and then try the following (in the same Python session as above, where script2 was already imported as a module):

We have to use the reload command to see the change we want:

# **Reloading Modules**

	We have to us	ise the reload command to see the change we want:
In [118]:	<pre>from impor reload(scr</pre>	rtlib <mark>import</mark> reload ript2)
Out[118]:	<module 's<="" td=""><td><pre>script2' from '/home/manmeet/Documents/teri/script2.py'&gt;</pre></td></module>	<pre>script2' from '/home/manmeet/Documents/teri/script2.py'&gt;</pre>
	script2.pr	rint_table()
In [119]:	1.1	
In [119]:	×	f(x)
In [119]:	50 D	f(x) 10.000
In [119]:	×	

## **Command Line Arguments**

```
Command line arguments
           We might want to make this script a bit fancier by adding an optional argument to the print_table function to print a different number of points, rather than the
           3 points shown above.
           The next version has this change, and also has a modified version of the main program that allows the user to specify this value n as a command line
           argument:
           script3.py
In [121]: ***
           Modification of script2.py that allows a command line argument telling how
           many points to plot in the table.
           Usage example: To print table with 5 values:
              python script3 5
           10.01.00
           import numpy as np
           def f(x):
               0.0.0
               A quadratic function.
               y = x^{**2} + 1.
               return y
           def print table(n=3):
               print(" x
                                      f(x)")
                for x in np.linspace(0,4,n):
                    print("%8.3f %8.3f" % (x, f(x)))
           if __name__ == "__main__":
               What to do if the script is executed at command line.
               Note that sys.argv is a list of the tokens typed at the command line.
                0.01.01
               import sys
               print("sys.argv is ",sys.argv)
               if len(sys.argv) > 1:
                    try:
                        n = int(sys.argv[1])
                        print table(n)
                    except:
                        print("*** Error: expect an integer n as the argument")
                else:
                    print table()
```

sys.argv is ['/home/manmeet/anaconda3/envs/py35/lib/python3.6/site-packages/ipykernel\_launcher.py', '-f', '/home/m anmeet/.local/share/jupyter/runtime/kernel-a6904343-3430-4b67-a126-523d7a5b7e3c.json'] \*\*\* Error: expect an integer n as the argument

## **Command Line Arguments**

### Note that:

- The function sys.argv from the sys module returns the arguments that were present if the script is executed from the command line. It is a list of strings, with sys.argv[0] being the name of the script itself, sys.argv[1] being the next thing on the line, etc. (if there were more than one command line argument, separated by spaces).
- We use int(sys.argv[1]) to convert the argument, if present, from a string to an integer.
- We put this conversion in a try-except block in case the user gives an invalid argument.

Sample output:

_						
\$	python	script3.py				
	х	f(x)				
	0.000	1.000				
	2.000	5.000				
	4.000	17.000				
\$	python	script3.py	5			
	X	f(x)				
	0.000	1.000				
	1.000	2.000				
	2.000	5.000				
	3.000	10.000				
	4.000	17.000				
\$	python	script3.py	5.2			
**	** Erro	r: expect an	n integer	n as	the	argument

## **Importing modules**

### Importing modules

When Python starts up there are a certain number of basic commands defined along with the general syntax of the language, but most useful things needed for specific purposes (such as working with webpages, or solving linear systems) are in modules that do not load by default. Otherwise it would take forever to start up Python, loading lots of things you don't plan to use. So when you start using Python, either interactively or at the top of a script, often the first thing you do is import one or more modules.

A Python module is often defined simply by grouping a set of parameters and functions together in a single .py file. See Python scripts and modules for some examples.

Two useful modules are os and sys that help you interact with the operating system and the Python system that is running. These are standard modules that should be available with any Python implementation, so you should be able to import them at the Python prompt:

In [122]: import os, sys

Each module contains many different functions and parameters which are the methods and attributes of the module. Here we will only use a couple of these. The getcwd method of the os module is called to return the "current working directory" (the same thing pwd prints in Unix), e.g.:

In [123]:	os.getcwd()
Out[123]:	'/home/manmeet/Documents/teri'

Note that this function is called with no arguments, but you need the open and close parens. If you type "os.getcwd" without these, Python will instead print what type of object this function is:

In	[124]	:	os.getcwd	
----	-------	---	-----------	--

Out[124]: <function posix.getcwd()>

## **The Python Path**

### **The Python Path**

The sys module has an attribute sys.path, a variable that is set by default to the search path for modules. Whenever you perform an import, this is the set of directories that Python searches through looking for a file by that name (with a .py extension). If you print this, you will see a list of strings, each one of which is the full path to some directory. Sometimes the first thing in this list is the empty string, which means "the current directory", so it looks for a module in your working directory first and if it doesn't find it, searches through the other directories in order:

#### In [125]: print(sys.path)

['/home/manmeet/anaconda3/envs/py35/lib/python36.zip', '/home/manmeet/anaconda3/envs/py35/lib/python3.6', '/home/manmeet/anaconda3/envs/py35/lib/python3.6/lib-dynload', '', '/home/manmeet/anaconda3/envs/py35/lib/python3.6/site-packages', '/home/manmeet/anaconda3/envs/py35/lib/python3.6/site-packages/IPython/extensions', '/home/manmeet/.ipython']

If you try to import a module and it doesn't find a file with this name on the path, then you will get an import error:

In [126]: import junkname

ModuleNotFoundError Traceback (most recent call last) <ipython-input-126-477437cb618a> in <module> ----> 1 import junkname

ModuleNotFoundError: No module named 'junkname'

When new Python software such as NumPy or SciPy is installed, the installation script should modify the path appropriately so it can be found. You can also add to the path if you have your own directory that you want Python to look in, e.g.:

sys.path.append(newdirectory)

will append the directory indicated to the path. To avoid having to do this each time you start Python, you can set a Unix environment variable that is used to modify the path every time Python is started. First print out the current value of this variable:

\$ echo \$PYTHONPATH

It will probably be blank unless you've set this before or have installed software that sets this automatically. To append the above example directory to this path:

# **Python strings**

	Python strings
	String formatting
	Often you want to construct a string that incorporates the values of some variables. This can be done using the form format % values where format is a string that describes the desired format and values is a single value or tuple of values that go into various slots in the format.
	This is best learned from some examples:
In [156]:	x = 45.6
In [157]:	s = "The value of x is %s" % x
In [158]:	s
Out[158]:	'The value of x is 45.6'
	The %s in the format string means to convert x to a string and insert into the format. It will use as few spaces as possible.
In [159]:	s = "The value of x is %21.14e" % x
In [160]:	s
Out[160]:	'The value of x is 4.56000000000000000000000000000000000000
	In the case above, exponential notation is used with 14 digits to the right of the decimal point, put into a field of 21 digits total. (You need at least 7 extra characters to leave room for a possible minus sign as well as the first digit, the decimal point, and the exponent such as e+01.
In [161]:	<b>y</b> = -0.324876
In [162];	s = "Now x is %8.3f and y is %8.3f" % (x,y)
In [163]:	s
Out[163]:	'Now x is 45.600 and y is -0.325'
	In this example, fixed notation is used instead of scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide. Note that y has been rounded.
	In the last example, two variables are inserted into the format string.

### Other forms of import

This appends another directory to the search path already specified (if any). You can repeat this multiple times to add more directories, or put something like:

```
export PYTHONPATH=$PYTHONPATH:dir1:dir2:dir3
```

in your .bashrc file if there are the only 3 personal directories you always want to search.

### Other forms of import

If all we want to use from the os module is getcwd, then another option is to do:

- In [127]: from os import getcwd
- In [128]: getcwd()
- Out[128]: '/home/manmeet/Documents/teri'

In this case we only imported one method from the module, not the whole thing. Note that now getcwd is called by just giving the name of the method, not module.method. The name getcwd is now in our namespace. If we only imported getcwd and tried typing "os.getcwd()" we'd get an error, since it wouldn't find os in our namespace.

You can rename things when you import them, which is sometimes useful if different modules contain different objects with the same name. For example, to compare how the sqrt function in the standard Python math module compares to the numpy version:

```
In [129]: from math import sqrt as sqrtm
from numpy import sqrt as sqrtn
```

In [130]: sqrtm(-1.)

```
ValueError Traceback (most recent call last)
<ipython-input-130-4b23e2676c45> in <module>
----> 1 sqrtm(-1.)
```

ValueError: math domain error

# Other forms of import

In [131]:	<pre>sartn(-1.)</pre>
	/home/manmeet/anaconda3/envs/py35/lib/python3.6/site-packages/ipykernel_launcher.py:1: RuntimeWarning: invalid valu e encountered in sqrt ""Entry point for launching an IPython kernel.
Out[131]:	nan
	The standard function gives an error whereas the numpy version returns nan, a special numpy object representing "Not a Number".
	You can also import a module and give it a different name locally. This is particularly useful if you import a module with a long name, but even for numpy many examples you'll find on the web abbreviate this as np (see Numerics in Python):
In [132]:	<pre>import numpy as np theta = np.linspace(0., 2*np.pi, 5)</pre>
In [133]:	theta
Out[133]:	array([0. , 1.57079633, 3.14159265, 4.71238898, 6.28318531])
In [134]:	np.cos(theta)
Out[134]:	array([ 1.00000000e+00, 6.1232340e-17, -1.60000000e+00, -1.8369702e-16, 1.0000000e+00])
In [135]:	<pre>from numpy import pi, linspace, cos theta = linspace(0., 2*pi, 5)</pre>
In [136]:	theta
Out[136]:	array([0. , 1.57079633, 3.14159265, 4.71238898, 6.28318531])
In [137]:	cos(theta)
Out[137]:	array([ 1.0000000e+00, 6.1232340e-17, -1.6000000e+00, -1.8369702e-16, 1.0000000e+00])
	If you're going to be using lots of things form numpy you might want to import everything into your namespace:
In [138]:	from numpy import *
	Then linspace, pi, cos, and several hundred other things will be available without the prefix.
	When writing code it is often best to not do this, however, since then it is not clear to the reader (or even to the programmer sometimes) what methods or attributes are coming from which module if several different modules are being used. (They may define methods with the same names but that do very different things, for example.)

# **Python functions**

	Python functions
	Functions are easily defined in Python using def, for example:
In [146];	<pre>def myfcn(x): import numpy as np y = np.cos(x) * np.exp(x) return y</pre>
In [147]:	myfcn(0.)
Out[147]:	1.0
In [148]:	myfcn(1.)
Out[148]:	1.4686939399158851
	As elsewhere in Python, there is no begin-end notation except the indentation. If you are defining a function at the command line as above, you need to input a blank line to indicate that you are done typing in the function.
	Defining functions in modules
	Except for very simple functions, you do not want to type it in at the command line in Python. Normally you want to create a text file containing your function and import the resulting module into your interactive session.
	If you have a file named myfile.py for example that contains:
def myfcn(x): impo	rt numpy as np y = np.cos(x) * np.exp(x) return y
	and this file is in your Python search path (see python_path), then you can do:
In [150]:	from myfile import myfcn
In [151]:	myfcn(0.)
Out[151]:	1.0
In [152]:	myfcn(1.)
Out[152]:	1.4686939399158851
	In Python a function is an object that can be manipulated like any other object.

### Lambda functions

### Lambda functions

Some functions can be easily defined in a single line of code, and it is sometimes useful to be able to define a function "on the fly" using "lambda" notation. To define a function that returns 2\*x for any input x, rather than:

### In [153]: **def** f(x):

return 2\*x

we could also define f via:

In [154]: f = lambda x: 2\*x

You can also define functions of more than one variable, e.g.:

In [155]: g = lambda x,y: 2\*(x+y)

# **Jupyter Notebook**

jupyter <mark>n</mark>	tebook_tutorial Last Checkpoint: 5 hours ago (unsaved changes)	é	Logout
e Edit Vie	v Insert Cell Kernel Widgets Help	Trusted	Python 3 O
+ & 4	A A A Run ■ C → Code T		
In [160]:	S.		
	'The value of x is 4.56000000000000000000000000000000000000		
	n the case above, exponential notation is used with 14 digits to the right of the decimal point, put into a field of 21 digits total. (You n characters to leave room for a possible minus sign as well as the first digit, the decimal point, and the exponent such as e+01.	eed at least 7	extra
In [161]:	<b>y</b> = -0.324876		
In [162]:	s = "Now x is %8.3f and y is %8.3f" % (x,y)		
In [163]:	S		
Out[163]:	'Now x is 45.600 and y is -0.325'		
	n this example, fixed notation is used instead of scientific notation, with 3 digits to the right of the decimal point, in a field 8 character been rounded.	s wide. Note	that y has
	n the last example, two variables are inserted into the format string.		
	Jupyter Notebook		
	The Jupyter notebook is fairly new and changing rapidly. Install the notebook by		
	\$ pip install jupyter		
	Then start the notebook via:		
	\$ jupyter-notebook		

## **Jupyter Notebook - File Menu**

File Edit View	Insert Cell Kernel Widgets Help	Trusted	Python 3
New Notebook	↑ ↓ N Run ■ C ≫ Code ▼ 550		
Make a Copy Save as Rename	e value of x is 4.56000000000000000000000000000000000000		
Save and Checkpoint	e case above, exponential notation is used with 14 digits to the right of the decimal point, put into a field of 21 digits total. (You ne acters to leave room for a possible minus sign as well as the first digit, the decimal point, and the exponent such as e+01.	ed at least 7	' extra
Save and Checkpoint	acters to leave room for a possible minus sign as well as the first digit, the decimal point, and the exponent such as e+01.	ed at least 7	' extra
<u>,</u> , , , , , , , , , , , , , , , , , , ,	acters to leave room for a possible minus sign as well as the first digit, the decimal point, and the exponent such as e+01.	ed at least 7	' extra
Revert to Checkpoint  Print Preview	acters to leave room for a possible minus sign as well as the first digit, the decimal point, and the exponent such as e+01.	ed at least 7	' extra

In this example, fixed notation is used instead of scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide. Note that y has been rounded.

In the last example, two variables are inserted into the format string.

### Jupyter Notebook

The Jupyter notebook is fairly new and changing rapidly. Install the notebook by

\$ pip install jupyter

Then start the notebook via:

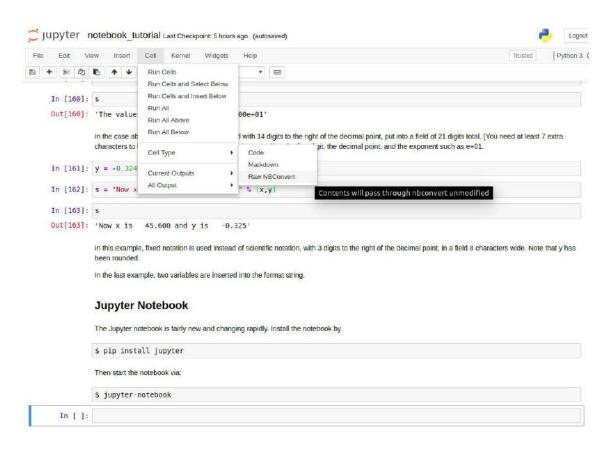
\$ jupyter-notebook

In [ ]:

# Jupyter Notebook - Kernel

e Edit Vi	lew insert Cell	Kernel Widgets H	Help Trusted	Pytho
+ % 2 In [160]: Out[160]:	A      A	Interrupt Restart Restart & Clear Output Restart & Run All Reconnect Shutdown Change kernel	+01' +14 digits to the right of the decimal point, put into a field of 21 digits total. (You need at least 7 as well as the first digit, the decimal point, and the exponent such as e+01.	' extra
In [161]:	y = -0.324876			
In [162]:	s = "Now x is %8.	3f and y is %8.3f"	‰ (x,y)	
In [163]:	5			
Out[163]:		0 and y is -0.325	2	
Out[163]:	'Now x is 45.60(		, scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide. Note	that y ha
Out[163];	'Now x is 45.600 In this example, fixed no been rounded.		scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide, Note	that y ha
Out[163]:	'Now x is 45.600 In this example, fixed no been rounded.	otation is used instead of variables are inserted int	scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide, Note	that y ha
Out[163]:	Now x is 45.600 In this example, fixed no been rounded. In the last example, two Jupyter Noteb	otation is used instead of variables are inserted int	scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide, Note	that y ha
Out[163]:	Now x is 45.600 In this example, fixed no been rounded. In the last example, two Jupyter Noteb	otation is used instead of variables are inserted int rook stairly new and changing	scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide, Note to the format string,	that y ha
Out[163]:	"Now x is 45.600 in this example, fixed no been rounded. In the last example, two Jupyter Noteb The Jupyter notebook is	otation is used instead of variables are inserted int <b>OOK</b> stairly new and changing yter	scientific notation, with 3 digits to the right of the decimal point, in a field 8 characters wide, Note to the format string,	that y ha

### **Jupyter Notebook - Cell**



# **Summary statistics**

>>> pima.describe()

ī	Unnamed: 0	npreg	glu	bp	skin	bmi	$\setminus$
count	332.000000	332.000000	332.000000	332.000000	332.000000	332.000000	
mean	166.500000	3.484940	119.259036	71.653614	29.162651	33.239759	
std	95.984374	3.283634	30.501138	12.799307	9.748068	7.282901	
min	1.000000	0.00000	65.000000	24.000000	7.000000	19.400000	
25%	83.750000	1.000000	96.000000	64.000000	22.000000	28.175000	
50%	166.500000	2.00000	112.000000	72.000000	29.000000	32.900000	
75%	249.250000	5.000000	136.250000	80.00000	36.000000	37.200000	
max	332.000000	17.000000	197.000000	110.000000	63.000000	67.100000	
	ped	aç	ge				
count	332.000000	332.00000	00				
mean	0.528389	31.31620	65				
std	0.363278	10.63622	25				
min	0.085000	21.00000	00				
25%	0.266000	23.00000	00				
50%	0.440000	27.00000	00				
75%	0.679250	37.00000	00				
max	2.420000	81.00000	00				



"Python Data Analysis Library"

Mature library for data analysis

Developed from http://pandas.pydata.org/

Main author Wes McKinney has written a 2012 book (McKinney, 2012)

A better Numpy: keep track of variable names, better indexing, easier linear modeling.

A better R: Access to more general programming language.

### Get some data from R

Get a standard dataset, Pima, from R:

\$ R

- > library(MASS)
- > write.csv(Pima.te, "pima.csv")

### pima.csv now contains comma-separated values:

"", "npreg", "glu", "bp", "skin", "bmi", "ped", "age", "type"
"1", 6, 148, 72, 35, 33.6, 0.627, 50, "Yes"
"2", 1, 85, 66, 29, 26.6, 0.351, 31, "No"
"3", 1, 89, 66, 23, 28.1, 0.167, 21, "No"
"4", 3, 78, 50, 32, 31, 0.248, 26, "Yes"
"5", 2, 197, 70, 45, 30.5, 0.158, 53, "Yes"
"6", 5, 166, 72, 19, 25.8, 0.587, 51, "Yes"

Back in Python:

```
>>> import pandas as pd
```

```
>>> pima = pd.read csv("pima.csv")
```

"pima" is now what Pandas call a DataFrame object. This object keeps track of both data (numerical as well as text), and column and row headers.

Lets use the first columns and the index column:

```
>>> import pandas as pd
>>> pima = pd.read_csv("pima.csv", index_col=0)
```

For example, you can see the first two rows or the three last rows:

```
>>> pima[0:2]
```

npreg glu bp skin bmi ped age type

1 6 148 72 35 33.6 0.627 50 Yes

```
2 1 85 66 29 26.6 0.351 31 No
```

```
>>> pima[-3:]
```

npreg glu bp skin bmi ped age type

330 10 101 76 48 32.9 0.171 63 No

331 5 121 72 23 26.2 0.245 30 No

```
332 1 93 70 31 30.4 0.315 23 No
```

Notice that this is not an ordinary numerical matrix: We also got text (in the "type" column) within the "matrix"!

# Indexing the columns

See a specific column, here 'bmi' (body-mass index): >>> pima["bmi"] 1 33.6 2 26.6 3 28.1 4 31.0 [here I cut out several lines] 330 32.9 331 26.2 332 30.4 Name: bmi, Length: 332 The returned type is another of Pandas Series object, — another of the fundamental objects in the library: >>> type(pima["bmi"]) <class 'pandas.core.series.Series'>

# **Conditional indexing**

Get the fat people (those with BMI above 30):

```
>>> pima.shape
```

```
(332, 9)
```

```
>>> pima[pima["bmi"]>30].shape
```

(210, 9)

### Row and column conditional indexing

import pandas as pd

from pylab import \*

```
df = pd.DataFrame(rand(10,5), columns=["A", "B", "C", "D", "E"])
```

```
df.ix[:, df.ix[0, :]<0.5]
```

These variations do not work

```
df[:, df[0]<0.5]
```

```
df[:, df[:1]<0.5]
```

```
df.ix[:, df[:1]<0.5]
```

# **Conditional Indexing**

fre	om pylab	das <mark>as</mark> import taFrame	*	0,5), colu
df	.ix[:, d	lf.ix[ <mark>0</mark> ,	:]<0.5]	
	А	В	с	E
0	0.159699	0.123944	0.483297	0.426953
1	0.083522	0.004395	0.459733	0.604684
2	0.409187	0.050444	0.014837	0.544117
3	0.760465	0.759616	0.062657	0.413452
4	0.851981	0.696130	0.061952	0.562658
5	0.278712	0.249822	0.231442	0.672320
6	0.713089	0.562738	0.585630	0.897577
7	0.204244	0.539397	0.031190	0.702550
8	0.929349	0.096289	0.755836	0.047888
9	0.266317	0.798710	0.603293	0.911597

### **Constructing a DataFrame**

```
In [182]: # intialise data of lists.
    data = {'Name':['Tom', 'nick', 'krish', 'jack'], 'Age':[20, 21, 19, 18]}
    # Create DataFrame
    df = pd.DataFrame(data)
    # Print the output.
    df
```

### Out[182]:

	Name	Age
0	Tom	20
1	nick	21
2	krish	19
3	jack	18

1.22000000 H 2000000

## Filling missing data

### Out[183]:

	First Score	Second Score	Third Score
0	100.0	30.0	0.0
1	90.0	45.0	40.0
2	0.0	56.0	80.0
3	95.0	0.0	98.0

### http://pandas.pydata.org/

The canonical book "Python for data analysis" (McKinney, 2012).

Will it Python?: Porting R projects to Python, exemplified through scripts from Machine Learning for Hackers (MLFH) by Drew Conway and John Miles White.

# Thank you!