Intro to Scientific Python (2018-01-23)

March 26, 2018

1 Introduction to Scientific Python

Boston University, PY 355

Date: January 2018

Lecturer: Chris Laumann

This short crash course draws heavily on a number of great resources from around the web. To really learn Python, it is best to spend a weekend and work through some of the many tutorials. Some good resources:

- The Python Tutorial
- Python Scientific Lecture Notes
- A Crash Course in Python for Scientists

Google is your friend. There's lots of documentation for Python and its many packages. I've included links to the main websites below but there's many other sources of information.

2 Why Python?

2.0.1 Simple, well-structured, general-purpose language

- Readability great for quality control and collaboration
- Code how you think: many books now use python as pseudocode

2.0.2 High-level

- Rapid development
- Do complicated things in few lines

2.0.3 Interactive

- Rapid development and exploration
- No need to compile, run, debug, revise, compile
- Data collection, generation, analysis and publication plotting in one place

2.0.4 Speed

- With some experience in good coding, plenty fast
- Your development time is more important than CPU time
- Not as fast as C, C++, Fortran but these can be easily woven in where necessary

2.0.5 Vibrant community

- Great online documentation / help available
- Open source

2.0.6 Rich scientific computing libraries

- Don't reinvent the wheel!

3 Scientific Python Key Components

The core pieces of the scientific Python platform are:

Python, the language interpreter - Many standard data types, libraries, etc - Python 3.6 is the current version but 2.7 is still maintained - Python 3 is mostly compatible with Python 2, but we will stick to 3

Jupyter: notebook based (in browser) interface - Builds on **IPython**, the interactive Python shell - Interactive manipulation of plots - Easy to use basic parallelization - Lots of useful extra bells and whistles for Python

Numpy, powerful numerical array objects, and routines to manipulate them. - Work horse for scientific computing - Basic linear algebra (np.linalg) - Random numbers (np.random)

Scipy, high-level data processing routines. - Signal processing (scipy.signal) - Optimization (scipy.optimize) - Special functions (scipy.special) - Sparse matrices and linear algebra

Matplotlib, plotting and visualization - 2-D and basic 3-D interactive visualization - "Publication-ready" plots - LaTeX labels/annotations automagically

Mayavi, 3-D visualization - For more sophisticated 3-D needs (won't discuss)

3.0.1 Importing the Scientific Environment

The simplest way to use the basic scientific libraries (numpy, scipy) and plotting tools (matplotlib) in jupyter is to execute the following command at the beginning of your notebook:

```
In [4]: %pylab notebook

Populating the interactive namespace from numpy and matplotlib
```

Commands beginning with % our IPython 'magic' commands. This one sets up a bunch of matplotlib back end and imports numpy into the global namespace. We will do this in all of our class notebooks. To manually import the numpy library, one could also use

```
In [5]: import numpy as np
```

which will make all of the numpy functions, such as array() and sin(), available as np.array(), np.sin(), ...

4 Jupyter Workflow

4.0.1 Two primary workflows:

1. Work in a Jupyter/IPython notebook. Write code in cells, analyze, plot, etc. Everything stored in **.ipynb** file.
2. Write code in **.py** files using a text editor and run those within the IPython notebook or from the shell.

We still stick to the first.

While you are using a notebook, there is a **kernel** running which actually executes your commands and stores your variables, etc. If you quit/restart the kernel, all variables will be forgotten and you will need to re-execute the commands that set them up. This can be useful if you want to reset things. The input and output that is visible in the notebook is saved in the notebook file.

Note: .py files are called **scripts** if they consist primarily of a sequence of commands to be run and **modules** if they consist primarily of function definitions for import into other scripts/notebooks.

4.0.2 Notebook Usage

Two modes: editing and command mode.

Press escape to go to command mode. Press return to go into editing mode on selected cell.

In command mode: 1. Press h for a list of keyboard commands. 2. Press a or b to create a new cell above or below the current. 3. Press m or y to convert the current cell to markdown or code. 4. Press shift-enter to execute. 5. Press d d to delete the current cell. (Careful!)

In editing mode: 1. Press tab for autocomplete 2. Press shift-tab for help on current object 3. Shift-enter to execute current cell

Two types of cells: 1. Markdown for notes (like this) 2. Code for things to execute

4.0.3 Exercise

Try editing this markdown block to make it more interesting.

4.0.4 Exercise

Execute the next block and then create a new block, type x. and press tab and shift-tab.

```
In [3]: x = 10
```

4.0.5 Exercise

Run this stuff.

```
In [4]: print('Hello, world!')
```

```
Hello, world!
```

```
In [5]: "Hello, world!"
```

```
Out[5]: 'Hello, world!'

In [6]: 2.5 * 3

Out[6]: 7.5

In [7]: 3**3

Out[7]: 27

In [8]: 3 + 3

Out[8]: 6

In [9]: "ab" + "cd"

Out[9]: 'abcd'

In [10]: "Hello" == 'Hello'

Out[10]: True
```

5 Variables and Objects

Everything in memory in Python is an object. Every object has a type such as int (for integer), str (for strings) or ndarray (for numpy arrays). Variables can reference objects of any type and that type can change.

The equals sign in programming does not mean 'is equal to' as in math. It means **'assign the object on the right to the variable on the left'**.

```
In [11]: a = 3

In [12]: a

Out[12]: 3

In [13]: type(a)

Out[13]: int

In [14]: a+a

Out[14]: 6

In [15]: 2+a

Out[15]: 5

In [9]: a = array([1,2])

In [10]: a
```

```
Out[10]: array([1, 2])

In [11]: type(a)

Out[11]: numpy.ndarray

In [16]: # All objects have properties, accessible with a .
         a.shape

Out[16]: (2,)

In [14]: a+a

Out[14]: array([2, 4])

In [15]: 2+a

Out[15]: array([3, 4])

In [ ]: a = "Hello, world!"

In [ ]: a

In [ ]: type(a)

In [ ]: a+a

In [ ]: 2+a
```

5.0.1 Overloading

Operators and functions will try to execute no matter what type of objects are passed to them, but they may do different things depending on the type. + adds numbers and concatenates strings.

5.0.2 Variables as References

All variables are **references** to the objects they contain. Assignment does not make copies of objects.

```
In [89]: a = array([1,2])
         a

Out[89]: array([1, 2])

In [90]: b = a
         b

Out[90]: array([1, 2])

In [19]: b[0] = 0
         b

Out[19]: array([0, 2])

In [20]: a

Out[20]: array([0, 2])
```

6 Types of Objects

6.1 Basic Types

1. **Numeric**
2. Integer: -1, 0, 1, 2, ...
3. Float: 1.2, 1e8
4. Complex: 1j, 1. + 2.j
5. Boolean: True, False
6. **Strings**, "hi"
7. Tuples, (2,7, "hi")

- Ordered collection of other objects, represented by parentheses
- can't change after creation (*immutable*)

3. **Lists**, [0,1,2,"hi", 4]

- Ordered collection of other objects, represented by square brackets
- can add/remove/change elements after creation (*mutable*)

4. Dictionaries, {'hi': 3, 4: 7, 'key': 'value'}
5. **Functions**, def func()

6.2 Common Scientific Types

6. **NumPy arrays**, array([1,2,3])

- Like lists but all entries have same type

7. Sparse arrays, scipy.sparse

7 Basic Types: Numeric

There are 4 numeric types: - int: positive or negative integer - float: a 'floating point' number is a real number like 3.1415 with a finite precision - complex: has real and imaginary part, each of which is a float - bool: two 'Boolean' values, True or False

```
In [ ]: a = 4
        type(a)

In [21]: c = 4.
         type(c)

Out[21]: float

In [22]: a = 1.5 + 0.1j
         type(a)

Out[22]: complex
```

```
In [23]: a.real

Out[23]: 1.5

In [24]: a.imag

Out[24]: 0.1

In [25]: flag = (3>4)
         flag

Out[25]: False

In [26]: type(flag)

Out[26]: bool

In [27]: type(True)

Out[27]: bool

In [28]: # Type conversion
         float(1)

Out[28]: 1.0
```

8 Basic Types: Strings

Strings are **immutable** sequences of characters. This means you can't change a character in the middle of a string, you have to create a new string.

Literal strings can be written with single or double-quotes. Multi-line strings with triple quotes. 'Raw' strings are useful for embedding LaTeX because they treat backslashes differently.

```
In [88]: 'Hello' == "Hello"

Out[88]: True

In [ ]: a = """This is a multiline string.
        Nifty, huh?"""

In [ ]: a

In [29]: print("\nu")


u


In [ ]: print(r"\nu")

In [30]: a = 3.1415
```

```
In [31]: # Simple formatting (type convert to string)
         "Blah " + str(a)

Out[31]: 'Blah 3.1415'

In [32]: # Old style string formatting (ala sprintf in C)
         "Blah %1.2f, %s" % (a, "hi")

Out[32]: 'Blah 3.14, hi'

In [33]: # New style string formatting
         "Blah {:1.2f}, {}".format(a, "hi")

Out[33]: 'Blah 3.14, hi'
```

9 Basic Types: Lists

Python lists store **ordered** collections of arbitrary objects. They are efficient maps **from index to values**. Lists are represented by square brackets [ ].

Lists are **mutable**: their contents can be changed after they are created.

It takes time O(1) to: 1. Lookup an entry at given index. 2. Change an item at a given index. 3. Append or remove (pop) from the end of the list.

It takes time O(N) to: 1. Find items by value if you don't know where they are. 2. Remove items from near the beginning of the list.

You can also grab arbitrary **slices** from a list efficiently.

Lists are 0-indexed. This means that the first item in the list is at position 0 and the last item is at position N-1 where N is the length of the list.

```
In [34]: days_of_the_week = ["Sunday","Monday","Tuesday",
                            "Wednesday","Thursday","Friday"]

In [ ]: days_of_the_week[0]

In [ ]: # The slice from 2 to 5 (inclusive bottom, exclusive top)
        days_of_the_week[2:5]

In [ ]: days_of_the_week[-1]

In [ ]: # every other day
        days_of_the_week[0:-1:2]

In [ ]: # every other day (shorter)
        days_of_the_week[::2]

In [ ]: # Oops!
        days_of_the_week.append("Saturday")

In [ ]: days_of_the_week[-1]

In [ ]: days_of_the_week[5] = "Casual Friday"
```

```
In [ ]: days_of_the_week

In [ ]: # Get the length of the list
        len(days_of_the_week)

In [ ]: # Sort the list in place
        days_of_the_week.sort()

In [ ]: days_of_the_week
```

Remember tab completion Every thing in Python (even the number 10) is an object. Objects can have methods which can be accessed by the notation a.method(). Typing a. allows you to see what methods an object a supports. Try it now with days_of_the_week:

```
In [ ]: days_of_the_week.
```

Each item is arbitrary: You can have lists of lists or lists of different types of objects.

```
In [ ]: aList = ["zero", 1, "two", 3., 4.+0j]
        aList

In [ ]: listOfLists = [[1,2], [3,4], [5,6,7], 'Hi']

In [ ]: listOfLists[2][1]
```

10 Numpy Arrays

Numpy arrays store **multidimensional arrays** of objects of a fixed type. The type of an array is a **dtype**, which is a more refined typing system than Python provides. They are efficient maps **from indices (i,j) to values**. They have **minimal memory overhead**.

Arrays are **mutable**: their contents can be changed after they are created. However, their size and dtype, once created cannot be efficiently changed (requires a copy).

Arrays are good for: 1. Representing matrices and vectors (**linear algebra**) 2. Storing grids of numbers (**plotting, numerical analysis**) 3. Storing data series (**data analysis**)

Arrays are not good for: 1. Applications that require growing/shrinking the size. 2. Heterogenous objects. 3. Non-rectangular data.

Arrays are 0-indexed.

We will talk about applications 1 and 2 today.

```
In [35]: # A vector is an array with 1 index
         a = array([1/sqrt(2), 0, 1/sqrt(2)])
         a

Out[35]: array([ 0.70710678,  0.        ,  0.70710678])

In [36]: a.shape

Out[36]: (3,)

In [37]: a.dtype
```

```
Out[37]: dtype('float64')

In [38]: a.size

Out[38]: 3

In [39]: # We access the elements using [ ]
         a[0]

Out[39]: 0.70710678118654746

In [40]: a[0] = a[0]*2

In [41]: a

Out[41]: array([ 1.41421356,  0.        ,  0.70710678])
```

We create a 2D array (that is a matrix) by passing the array() function a list of lists of numbers in the right shape.

```
In [42]: # A matrix is an array with 2 indices
         B = array( [[ 1, 0, 0],
                     [ 0, 0, 1],
                     [ 0, 1, 0]] )
         B

Out[42]: array([[1, 0, 0],
                [0, 0, 1],
                [0, 1, 0]])

In [43]: B.shape

Out[43]: (3, 3)

In [44]: B.dtype

Out[44]: dtype('int64')

In [45]: B.size

Out[45]: 9

In [46]: B[0,0]

Out[46]: 1
```

10.0.1 Exercise

Change the last row of B to have a 2 instead of a 1 in the middle position.

Warning! There is also a type called 'matrix' instead of 'array' in numpy. This is specially for 2-index arrays but is being removed from Numpy over the next two years because it leads to bugs. **Never use matrix(), only array()**

10.1 Basic Linear Algebra

There are two basic kinds of multiplication of arrays in Python:

1. **Element-wise multiplication:** a*b multiplies arrays of the same shape element by element.
2. **Dot product:** a@b forms a dot product of two vectors or a matrix product of two rectangular matrices.

Mathematically, for vectors,

$$a@b = \sum_i a[i]b[i]$$

while for 2D arrays (matrices),

$$A@B[i,j] = \sum_k A[i,k]B[k,j]$$

```
In [47]: a = array([1,  1]) / sqrt(2)
         b = array([1, -1]) / sqrt(2)

In [48]: a

Out[48]: array([ 0.70710678,  0.70710678])

In [49]: b

Out[49]: array([ 0.70710678, -0.70710678])

In [50]: a@a

Out[50]: 0.99999999999999978

In [51]: # Compute the length of a
         norm(a)

Out[51]: 0.99999999999999989

In [52]: sqrt(a@a)

Out[52]: 0.99999999999999989

In [53]: a@b

Out[53]: 0.0
```

There are many, many more functions for doing linear algebra operations numerically provided by numpy and scipy. We will use some of them as we go on in the course.

11 Basic Plotting

The second primary use of numpy array's is to hold grids of numbers for analyzing and plotting. In this case, we consider a long 1D array with length N as representing the values of the x and y axis of a plot, for example.

Let's plot a sine wave:

```
In [6]: # create an equally spaced array of 100 numbers
        # from -2pi to 2pi
        x = linspace(-2*pi, 2*pi, 100)

        # evaluate a function at each point in x and create a
        # corresponding array
        y = 0.5*sin(x)

        figure()
        plot(x,y)
        grid()
        xlabel(r'$x$')
        ylabel(r'$0.5 \sin(x)$')
```

```
<IPython.core.display.Javascript object>
```

```
<IPython.core.display.HTML object>
```

```
Out[6]: Text(0,0.5,'$0.5 \\sin(x)$')
```

11.1 Some Common Arrays

```
In [55]: arange(2, 10, 2)
```

```
Out[55]: array([2, 4, 6, 8])
```

```
In [61]: # compact notation for the previous
         # r_ creates a 'row vector', notice the [ ] rather than ( )
         r_[2:10:2]
```

```
Out[61]: array([2, 4, 6, 8])
```

```
In [56]: linspace(2, 10, 5)
```

```
Out[56]: array([  2.,   4.,   6.,   8.,  10.])
```

```
In [57]: # compact notation for the previous
         r_[2:10:5j]
```

```
Out[57]: array([  2.,   4.,   6.,   8.,  10.])
```

```
In [58]: ones((2,2))

Out[58]: array([[ 1.,  1.],
                [ 1.,  1.]])

In [59]: zeros((3,2))

Out[59]: array([[ 0.,  0.],
                [ 0.,  0.],
                [ 0.,  0.]])

In [60]: eye(3)

Out[60]: array([[ 1.,  0.,  0.],
                [ 0.,  1.,  0.],
                [ 0.,  0.,  1.]])

In [62]: diag([1,2,3])

Out[62]: array([[1, 0, 0],
                [0, 2, 0],
                [0, 0, 3]])

In [63]: np.random.rand(2,2)

Out[63]: array([[ 0.03440661,  0.71990721],
                [ 0.34235528,  0.72901571]])
```

11.2 Array DTypes

Every element in a numpy array has the same type. These are called **dtypes** because they are more specific data types than the basic python number system -- they allow you to control how many bytes are used for each number. We will typically not need to worry about this but you may need to be aware that the system is a bit different from the basic numeric types.

1. Integers: int16 ('i2'), int32 ('i4'), int64 ('i8'), ...
2. Unsigned: uint32 ('u4'), uint64 ('u8'), ...
3. Float: float16 ('f2'), float32 ('f4'), float64 ('f8'), ...
4. Boolean: bool
5. Fixed Length Strings: 'S1', 'S2', ...

```
In [64]: array([1,0]).dtype

Out[64]: dtype('int64')

In [65]: array([1.,0]).dtype

Out[65]: dtype('float64')

In [67]: dtype('i4')

Out[67]: dtype('int32')
```

12 Control Flow

The flow of a program is the order in which the computer executes the statements in the code. Typically, this is in order from top to bottom. However, there are many cases where we want to change the flow in some way. For example, we might want to divide two numbers but only if the divisor is not zero. Or we might want to iterate: repeat a block of code many times for each value in some list. The commands which allow these are called control flow commands.

WARNING: Python cares about **white space**! You must **INDENT CORRECTLY** because that's how Python knows when a block of code ends.

Typically, people indent with 4 spaces per block but 2 spaces or tabs are okay. They must be consistent in any block.

12.0.1 If/elif/else

```
In [68]: if 2>3:
             print("Yep")
             print("It is")

         elif 3>4:
             print("Not this one either.")

         else:
             print("Not")
             print("At all")
```

```
Not
At all
```

12.0.2 For Loops

For loops *iterate* through elements in a collection. This can be a list, tuple, dictionary, array or any other such collection.

These are the most *Pythonic* way to think about iterations.

```
In [69]: for i in range(5):
             j = i**3
             print("The cube of " + str(i) + " is " + str(j))
```

```
The cube of 0 is 0
The cube of 1 is 1
The cube of 2 is 8
The cube of 3 is 27
The cube of 4 is 64
```

```
In [70]: for day in days_of_the_week:
             print("Today is " + day)
```

```
Today is Sunday
Today is Monday
Today is Tuesday
Today is Wednesday
Today is Thursday
Today is Friday
```

Enumerate to get index and value of iteration element

```
In [71]: words = ('your', 'face', 'is', 'beautiful')

         for (i, word) in enumerate(words):
             print(i, word)
```

```
0 your
1 face
2 is
3 beautiful
```

12.0.3 While Loops

Repeats a block of code while a condition holds true.

```
In [ ]: x = 5

        while x > 0:
            print("Bark " + str(x))
            x -= 1
```

13 Functions

Any code that you call multiple times with different values should be wrapped up in a function. For example:

```
In [72]: def square(x):
             """Return the square of x."""
             return x*x

In [73]: square?

In [74]: square(9)

Out[74]: 81

In [75]: def printAndSquare(x):
             """Print the square of x and return it."""
             y = x**2
             print(y)
             return y
```

```
In [76]: printAndSquare?

In [77]: printAndSquare(8)

64


Out[77]: 64
```

13.0.1 Functions are Objects

Functions are just like any object in Python:

```
In [78]: type(square)

Out[78]: function
```

Make another variable refer to the same function:

```
In [79]: a = square

In [80]: a(5)

Out[80]: 25
```

A function being passed to another function.

```
In [86]: def test():
             print("In Test!")
             return

         def callIt(fun):
             print("In callIt!")
             fun()
             return

In [87]: callIt(test)

In callIt!
In Test!
```