Using the IPython Notebook

March 31, 2015

1 Warm-up exercises

1. Write a function, `pos_neg(a, b, is_negative)`, which should return `True` if one of `a` or `b` is negative. But if `is_negative` is `True`, return true only if both `a` and `b` are negative.

2. Write a function, `extra_end(str)`, that takes a string and returns a new string formed by taking the last two characters of `str` and repeats it three times. *Bonus*: handle cases in which `str` has length less than 2.

3. Write a function, `rotate_left3(lst)`, that takes a list of 3 integers and returns the elements "rotated left", so `[1, 2, 3]` yields `[2, 3, 1]`.

```
In [1]: def pos_neg(a, b, is_negative):
            if is_negative:
                return a < 0 and b < 0
            else:
                return a < 0 or b < 0

        def extra_end(str):
            return str[-2:] * 3

        def rotate_left3(lst):
            return lst[1:] + [lst[0]]
```

```
In [2]: print pos_neg(1, -1, False)
        print extra_end("warm")
        print rotate_left3([5, 3, 2])
```

```
True
rmrmrm
[3, 2, 5]
```

2 About IPython

From *Python for Data Analysis*:

> The `IPython` project began in 2001 as Fernando Perez's side project to make a better interactive `Python` interpreter. In the subsequent 11 years it has grown into what's widely considered one of the most important tools in the modern scientific `Python` computing stack. While it does not provide any computational or data analytical tools by itself, `IPython` is designed from the ground up to maximize your productivity in both interactive computing and software development. It encourages an *execute-explore* workflow instead of the typical *edit-compile-run* workflow of many other programming languages.

The `IPython Notebook` uses an input-output programming paradigm centered around the *cell*. Excecuting a cell saves all of the data into the notebook, so you can use it later. But, you can execute a cell, write more code, and come back to the same cell to make changes, which gives the notebook an incredible amount of versatility.

Each notebook uses an execution kernel to keep track of all of the data. If you run into a challenge, say, an accidental infinite loop, you can interrupt and restart the kernel from the options. Restarting the kernel drops all of the data saved to the main memory stack, so you need to re-execute all of your cells.

3 Key features of `IPython`

Users of *Mathematica* may feel familiar with the overall layout of the `IPython Notebook`, but there are some important differences (and advantages) of using the Notebook. The `IPython Notebook` is (lovingly, but jokingly) referred to as the "poor person's *Mathematica*", which is unfair to both `IPython` and *Mathematica*. The Notebook serves a different purpose from Wolfram's product, and it does so exceptionally well.

3.1 Tab completion

From *Python for Data Analysis*:

One of the major improvements over the standard `Python` shell is *tab completion*, a feature common to most interactive data analysis environments. While entering expressions in the shell, pressing <Tab> will search the namespace for any variables (objects, functions, etc.) matching the characters you have typed so far:

```
In [1]: an_apple = 27

In [2]: an_example = 42

In [3]: an<Tab>
an_apple    and    an_example    any
```

In this example, not that `IPython` displayed both the two variables defined as well as the `Python` keyword `and` and built-in function `any`. Naturally, you can also complete methods and attributes on any object after typing a period:

```
In [3]: b = [1, 2, 3]

In [2]: b.<Tab>
b.append    b.extend    b.insert    b.remove    b.sort
b.count     b.index     b.pop       b.reverse
```

Tab completion works in many contexts outside of searching the interactive namespace and completing object or module attributes. When typing anything that looks like a file path (even in a `Python` string), pressing <Tab> will complete anything on your computer's file system matching what you've typed. Combined with the `%run` command (see later section), this functionality will undoubtedly save you many keystrokes.

Another area where tab completion saves time is in the completion of function keyword argument (arguments that include the `=` sign).

3.2 Introspection

Closing a question mark (`?`) before or after a variable will display some general information about the object:

```
In [4]: b?
```

```
Type:        list
String form: [1, 2, 3]
Length:      3
Docstring:
list() -> new empty list
list(iterable) -> new list initialized from iterable's items
```

This is referred to as *object introspection*. If the object is a function or instance method, the docstring, if defined, will also be shown. Suppose we'd written the following functions:

```
In [5]: def add_numbers(a, b):
            """
            Add two numbers together

            Returns
            -------
            the_sum : type of arguments
            """
            return a + b
```

Then using `?` shows us the docstring:

```
In [6]: add_numbers?
```

```
Type:        function
String form: <function add_numbers at 0x7facfc177488>
File:        /home/alethiometryst/mathlan/public_html/courses/python/course-material/ipynbs/<ipython-inp
Definition:  add_numbers(a, b)
Docstring:
Add two numbers together

Returns
-------
the_sum : type of arguments
```

Using `??` will also show the function's source code if possible:

```
In [7]: add_numbers??
```

```
Type:        function
String form: <function add_numbers at 0x7facfc177488>
File:        /home/alethiometryst/mathlan/public_html/courses/python/course-material/ipynbs/<ipython-inp
Definition:  add_numbers(a, b)
Source:
def add_numbers(a, b):
    """
    Add two numbers together

    Returns
    -------
    the_sum : type of arguments
    """
    return a + b
```

`?` has a final usage, which is for searching the `IPython` namespace in a manner similar to the standard `UNIX` or Windows command line. A number of characters combined with the wildcard (*) will show all names matching the wildcard expression. For example, we could get a list of all functions in the top level `NumPy` namespace containing `load`:

```
In [8]: import numpy as np
        np.*load*?

np.load
np.loads
np.loadtxt
np.pkgload
```

3.3 The %run Command

Any file can be run as a Python program inside the environment of your IPython session using the %run command.

3.4 Keyboard Shortcuts

IPython has many keyboard shortcuts for navigating the prompt (which will be familiar to users of the Emacs text editor or the UNIX bash shell). Here are the most commonly used elements:

3.4.1 Command Mode (press Esc to enable)

Command	Description	Command	Description
Enter	edit mode	Ctrl-j	move cell down
Shift-Enter	run cell, select below	a	insert cell above
Ctrl-Enter	run cell	b	insert cell below
Alt-Enter	run cell, insert below	x	cut cell
y	to code	c	copy cell
m	to markdown	Shift-v	paste cell above
r	to raw	v	paste cell below
1	to heading 1	d	delete cell (press twice)
2	to heading 2	Shift-m	merge cell below
3	to heading 3	s	save notebook
4	to heading 4	Ctrl-s	save notebook
5	to heading 5	l	toggle line numbers
6	to heading 6	o	toggle output
Up	select previous cell	Shift-o	toggle output scrolling
Down	select next cell	q	close pager
k	select previous cell	h	keyboard shortcuts
j	select next cell	i	interrupt kernel (press twice)
Ctrl-k	move cell up	0	restart kernel (press twice)

3.4.2 Edit Mode (press Enter to enable)

Command	Description	Command	Description
Tab	code completion or indent	Ctrl-Down	go to cell end

Command	Description	Command	Description
`Shift-Tab`	tooltip	`Ctrl-Left`	go one word left
`Ctrl-]`	indent	`Ctrl-Right`	go one word right
`Ctrl-[`	dedent	`Ctrl-Backspace`	del word before
`Ctrl-a`	select all	`Ctrl-Delete`	del word after
`Ctrl-z`	undo	`Esc`	command mode
`Ctrl-Shift-z`	redo	`Ctrl-m`	command mode
`Ctrl-y`	redo	`Shift-Enter`	run cell, select below
`Ctrl-Home`	go to cell start	`Ctrl-Enter`	run cell, select below
`Ctrl-Up`	go to cell start	`Alt-Enter`	run cell, insert below
`Ctrl-End`	go to cell end	`Ctrl-Shift--`	split cell
`Ctrl-s`	save notebook		

3.5 Exception and Tracebacks

If an exception is raised while executing any statement, `IPython` will by default print a full call stack trace (traceback) with a few lines of context around the position at each point in the stack.

```
In [9]: def func(a):
            return a + 2

        func(3, 4)


        ---------------------------------------------------------------------------

        TypeError                                 Traceback (most recent call last)

        <ipython-input-9-73c995299d78> in <module>()
          2     return a + 2
          3
    ----> 4 func(3, 4)


        TypeError: func() takes exactly 1 argument (2 given)
```

3.6 Magic Commands

`IPython` has many special commands, known as "magic" commands, which are designed to facilitate common tasks and enable you to easily control the behavior of the `IPython` system. A magic command is any command prefixed by the percent symbol `%`.

Magic commands can be viewed as command line programs to be run within the `IPython` system. Many of them have additional "command line" options, which can all be viewed (as you might expect) using `?`.

Magic functions can be used by default without the percent sign, as long as no variable is defined with the same name as the magic function. This feature is called *automagic* and can be enabled or disabled using `%automatic`.

Since `IPython`'s documentation is easily accessible from within the system, I encourage you to explore all of the special commands available by typing `%quickref` or `%magic`. Here are a few more of the most critical

ones for being productive in interactive computing and `Python` development in `IPython`.

Command	Description
`%quickref`	Display the `IPython` Quick Reference Card
`%magic`	Display detailed documentation for all of the available magic commands
`%debug`	Enter the interactive debugger at the bottom of the last exception traceback
`%hist`	Print command input (and optionally output) history
`%pdb`	Automatically enter debugger after any exception
`%paste`	Execute pre-formatted `Python` code from clipboard
`%cpaste`	Open a special prompt for manually pasting `Python` code to be executed
`%reset`	Delete all variables / names defined in interactive namespace
`%page OBJECT`	Pretty print the object and display it through a pager
`%run script.py`	Run a `Python` script inside `IPython`
`%prun statement`	Execute `statement` with `cProfile` and report the profiler output
`%time statement`	Report the execution time of single statement
`%timeit statement`	Run a (short) statement multiple times to compute an emsemble average execution time.
`%who, %who_ls, %whos`	Display variables defined in interactive namespace, varying info depth
`%xdel variable`	Delete a variable and clear references to the object in the `IPython` internals

3.6.1 Try it!

Which "squaring" function do you think is more efficient?

```
In [10]: def square_one(x):
             return x ** 2

         def square_two(x):
             return x * x

         %timeit square_one(500)
         %timeit square_two(500)
```

```
10000000 loops, best of 3: 167 ns per loop
10000000 loops, best of 3: 142 ns per loop
```

It's good to remember that $x \cdot x$ is always faster to compute than x^2.

`%timeit` can be used for more complicated functions. For example, consider the Fibonacci numbers, which are computed according to the following rule:

$$
\begin{aligned}
F_1 &= 1 \\
F_2 &= 1 \\
F_n &= F_{n-1} + F_{n-2}
\end{aligned}
$$

So, F_3 is the sum of F_1 and F_2, i.e. $F_3 = 1 + 1 = 2$. Then $F_4 = F_2 + F_3 = 5$, and so on. We can write a `Python` function to calculate Fibonacci numbers with two different strategies: recursion or iteration. One might ask which implementation is more efficient, so we can use `%timeit` to get a good idea.

```
In [11]: # Recursive implementation
         def fibonacci_one(n):
             if n == 1 or n == 2:
                 return 1
             else:
                 return fibonacci_one(n-1) + fibonacci_one(n-2)

         # Iterative implementation
         def fibonacci_two(n):
             a = 1
             b = 1
             if n < 3:
                 return 1
             for i in range (3, n+1):
                 c = a
                 a += b
                 b = c

             return a
```

```
In [12]: %timeit fibonacci_one(15)
         %timeit fibonacci_two(15)
```

```
1000 loops, best of 3: 186 s per loop
1000000 loops, best of 3: 1.82 s per loop
```

Remember that there are 1000 nanoseconds per microsecond. In other words, the iterative implementation is **150 times faster** than the recursive one.

3.7 Debugging

Anyone who has dealt with computers for any length of time has had to deal with bugs in their code (I'm looking at you, 161ers). Sometimes you make a (or many) small mistake(s) that you don't catch before running your code. Your program won't work properly, and you need to figure out what's causing the problem. Luckily, with `IPython`, we get to use the interactive debugger, `%debug`, which lets you step through your code to help you spot errors.

Consider the following problem. You need to write a function that takes a list of `doubles` and computes the list of its multiplicative inverses. Then, you need to plot the data (don't worry about the `matplotlib` code; it's a demonstration). For example, if you are given $[1, 2, 3, 4, 5]$, you should obtain

$$f([1,2,3,4,5]) = \left[1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}\right]$$

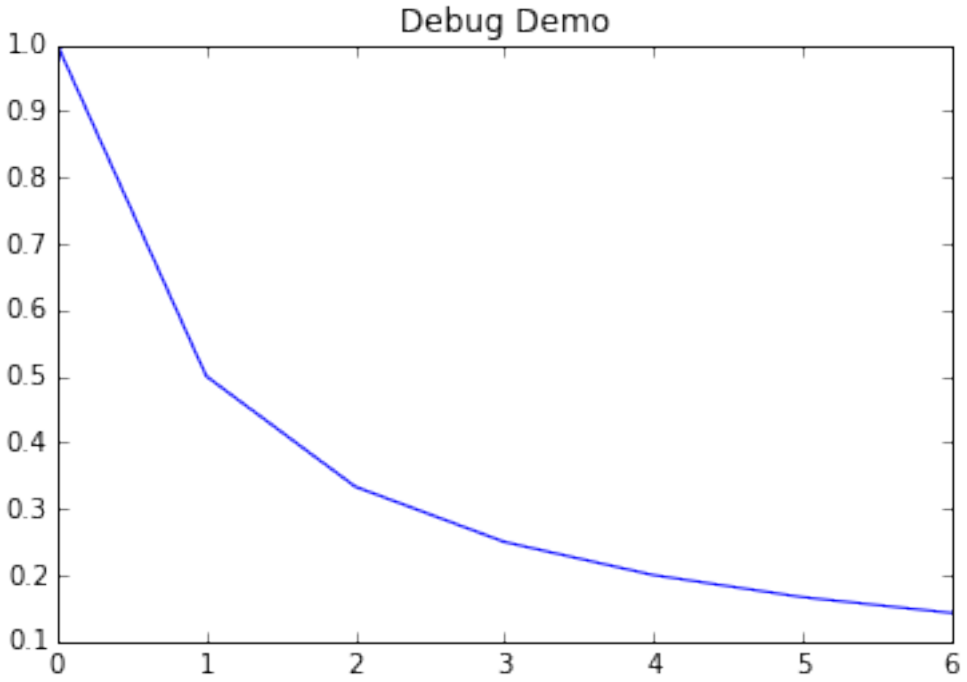
represented as `doubles`. Suppose your first attempt at a solution is as follows.

```
In [13]: import matplotlib.pyplot as plt
         %matplotlib inline
```

```
In [14]: # A simple, naive solution
         def invert(list_of_doubles):
             inverted_list = []
             for i in list_of_doubles:
                 inverted_list.append(1./i)
             return inverted_list
```

```
        # A silly, useless function
        def plot_demo(inverted_list):
            plt.plot(inverted_list)
            plt.title("Debug Demo")
```

```
In [15]: plot_demo(invert([1., 2., 3., 4., 5., 6., 7.]))
```



Everything looks good, so what's the problem?

```
In [16]: plot_demo(invert([1., 2., 3., 4., 5., 6., 7., 0.]))
```

```
---------------------------------------------------------------------------

ZeroDivisionError                         Traceback (most recent call last)

<ipython-input-16-588f355e682e> in <module>()
----> 1 plot_demo(invert([1., 2., 3., 4., 5., 6., 7., 0.]))


<ipython-input-14-b7d23d10d306> in invert(list_of_doubles)
      3     inverted_list = []
      4     for i in list_of_doubles:
----> 5         inverted_list.append(1./i)
      6     return inverted_list
      7


ZeroDivisionError: float division by zero
```

Okay, so we're getting a divide by zero... let's run `%debug` if it can enlighten us on where the functions broke.

```
In [17]: %debug

> <ipython-input-14-b7d23d10d306>(5)invert()
      4     for i in list_of_doubles:
----> 5         inverted_list.append(1./i)
      6     return inverted_list

ipdb> d
*** Newest frame
ipdb> u
> <ipython-input-16-588f355e682e>(1)<module>()
----> 1 plot_demo(invert([1., 2., 3., 4., 5., 6., 7., 0.]))

ipdb> q
```

3.7.1 Debugger commands

This example was extremely simplified, but `%debug` is a very useful command to have in your toolbelt. Here are the key commands inside the debugger to help you navigate.

Command	Action	Command	Action
`h(elp)`	Display command list	`s(tep)`	Step *into* function c
`help` *command*	Show documentation for *command*	`n(ext)`	Execute current line
`c(ontinue)`	Resume program execution	`u(p) / d(own)`	Move up/down in fu
`q(uit)`	Exit debugger without executing any more code	`a(rgs)`	Show arguments for
`b(reak)` *number*	Set a breakpoint at *number* in current file	`debug` *statement*	Invoke *statement* in
`b` *path/to/file.py:number*	Set breakpoint at line *number* in specified file	`l(ist)` *statement*	Show current positi
`w(here)`	Print full stack trace with context at current position		

4 Tips for Productive Code Development Using `IPython`

Wes McKinney has a number of helpful tips regarding using `IPython` for code development. Importantly, he says

> Writing code in a way that makes it easy to develop, debug, and ultimately *use* interactively may be a paradigm shift for many users. There are procedural details like code reloading that may require some adjustment as well as coding style concerns.
>
> As such, most of this section is more of an art than a science and will require some experimentation on your part to determine a way to write your `Python` code that is effective and productive for you. Ultimately you want to structure your code in a way that makes it easy to use iteratively and to be able to explore the results of running a program or function as effortlessly as possible.

On these lines, here are some of McKinney's tips for good code design in `Python`.

4.0.2 Flat is better than nested

- Deeply nested code makes me think about the many layers of an onion.

- When testing or debugging a function, how many layers of the onion must you peel back in order to reach the code of interest?
- Making functions and classes as decoupled and modular aas possible makes them easier to test (if you are writing unit tests), debug, and use interactively.

4.0.3 Overcome a fear of longer files

- If you come from a `Java` background, you may have been told to keep files short.
- However, while developing code using `IPython`, working with 10 small, but interconnected files (under, say, 100 lines each) is likely to cause you more headache in general than a single large file or two or three longer files.
- Fewer files means fewer modules to reload and less jumping between files while editing, too.

5 Lab: Testing Efficiency

One fundamental problem in computing is array (or list) sorting. There are many different ways to sort, each providing distinct advantages. As a general rule, the best objective way to compare sorting algorithms is their efficiency, but this can change depending on the size of the array (or list) under consideration.

This lab invites you to try to implement two common sorting algorithms:

- Bubble sort,
- Selection sort

Neither algorithm is inherently efficient, but are comparatively simple to put to code. The bubble sort works as follows:

1. Start with a list of numbers
2. Check if it is sorted
    1. If it is sorted, then return the list
    2. If not, continue
3. Compare the first and second numbers
    1. If the first number is less than the second number, continue
    2. If the second number is less than the first number, swap, and then continue
4. Shift over one element of the list, and repeat.
5. After reaching the end of the list, go back to step (2).

The selection sort works as follows:

1. Start with a list of numbers
2. Set the "swap" index to 0
3. Check if the list after the swap index is sorted
    1. If it is sorted, then return the list
    2. If not, continue
4. Cycle through the remaining list and identify the location of the smallest number
5. Exchange the minimum with the current swap index
6. Increment the swap index
7. Go back to step(2).

Your task: implement these sorting algorithms in `Python`, and use `IPython` tools to determine which algorithm is more efficient.

```
In [18]: from numpy import random

         def is_sorted(lst):
             for i in range(1,len(lst)):
                 if lst[i] < lst[i-1]:
                     return False
             return True

         def bubblesort(lst):
             while not is_sorted(lst):
                 for i in range(0, len(lst) - 1):
                     if lst[i+1] < lst[i]:
                         temp = lst[i]
                         lst[i] = lst[i+1]
                         lst[i+1] = temp
             return lst

         def selection_sort(lst):
             for i, e in enumerate(lst):
                 mn = min(range(i,len(lst)), key=lst.__getitem__)
                 lst[i], lst[mn] = lst[mn], e
             return lst

In [19]: %timeit bubblesort(random.randn(1000))
         %timeit selection_sort(random.randn(1000))

1 loops, best of 3: 362 ms per loop
10 loops, best of 3: 82.1 ms per loop
```