1-0_IPython Notebooks

September 15, 2015

1 IPython Notebooks

1.1 Introduction

You've just got a basic intro to programming in python. Now, we are going to get our hands dirty using an environment called an "IPython notebook". It's a great tool for learning, but it's also used by the pros (like the many authors contributing to this material!). You can click on anything in this notebook and edit it, but...

Don't panic!

You can't break anything! You can always get another copy of any notebook from the workshop on the PS239T repository.

1.2 Navigating in IPython Notebook

IPython has more useful features for interactive use than the standard python interpreter and online systems like Codecademy, but it works in the same basic way: you type things and then execute them. We'll use it from here on out.

Unlike many graphical systems, there is no button to run your code! Instead, **you run code using Shift-Enter**. This also moves you to the next box (or "cell") for code below the one you just ran (but this may change in the future).

Try to **run the following code using Shift-Enter** now!

```
In [1]: print "Look Ma, I'm programming!"
```

```
Look Ma, I'm programming!
```

If you hit **Enter** only, IPython Notebook gives you another line in the current cell.

This allows you to compose multi-line commands and submit them to python all at once.

```
In [2]: a = 1 + 2
        print a
```

```
3
```

Control-Enter (**Cmd-ENTER** on Macs) executes the cell and does not move to the next cell.

You can enter the same line over and over again into the interpreter.

```
In [3]: i = 0
```

Try entering this cell a few times:

```
In [4]: i = i + 1
        print i
```

```
1
```

If you want to create new empty cells, you can use Insert -> Insert Cell Below or use the Insert Cell Below button at the top of the notebook. Try entering a new cell below this one.

What happens if you run this cell?

```
In [5]: print "Or am I
```

```
          File "<ipython-input-5-254da1df7681>", line 1
        print "Or am I
                     ^
    SyntaxError: EOL while scanning string literal
```

As you can see, IPython offers proper error reporting. But don't worry - Python is just trying to help fix something it can't understand!

1.3 Executing code in files

If your code is in a file, you can execute it from the IPython shell with the %run command. Execute madlib.py like so

```
In [8]: %run madlib.py
```

```
Enter a specific example for animal:
Enter a specific example for food:
Enter a specific example for city:

Once upon a time, deep in an ancient jungle,
there lived a .  This
liked to eat , but the jungle had
very little  to offer.  One day, an
explorer found the  and discovered
it liked .  The explorer took the
 back to , where it could
eat as much  as it wanted.  However,
the  became homesick, so the
explorer brought it back to the jungle,
leaving a large supply of .

The End

Press Enter to end the program.
```

We can even execute shell commands from IPython notebook using '!'

```
In [9]: !cat madlib.py
```

```
"""
String Substitution for a Mad Lib
Adapted from code by Kirby Urner
"""

story = """
```

```
Once upon a time, deep in an ancient jungle,
there lived a %(animal)s.  This %(animal)s
liked to eat %(food)s, but the jungle had
very little %(food)s to offer.  One day, an
explorer found the %(animal)s and discovered
it liked %(food)s.  The explorer took the
%(animal)s back to %(city)s, where it could
eat as much %(food)s as it wanted.  However,
the %(animal)s became homesick, so the
explorer brought it back to the jungle,
leaving a large supply of %(food)s.

The End
"""

def tellStory():
    userPicks = dict()
    addPick('animal', userPicks)
    addPick('food', userPicks)
    addPick('city', userPicks)
    print story % userPicks

def addPick(cue, dictionary):
    prompt = "Enter a specific example for %s: " % cue
    dictionary[cue] = raw_input(prompt)

tellStory()
raw_input("Press Enter to end the program.")
```

1.4 Clearing IPython

IPython remembers everything it executed, **even if it's not currently displayed in the notebook**.
To clear everything from IPython use Kernel->Restart in the menu.

```
In [10]: mystring = "And three shall be the count."

         print mystring
```

```
And three shall be the count.
```

Now use Kernel->Restart in the menu!

```
In [1]: print mystring
```

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

        NameError                                 Traceback (most recent call last)

        <ipython-input-1-16a377486261> in <module>()
    ----> 1 print mystring


        NameError: name 'mystring' is not defined
```

Note that the error message contains a recap of the input that caused the error (with an arrow, no less!) It is objecting that **mystring** is not defined, since we just reset it.

1.5 Getting Help

IPython has some nice help features. Let's say we want to know more about the integer data type. There are at least two ways to do this task:

```
In [2]: help(int)

Help on class int in module __builtin__:

class int(object)
 |  int(x=0) -> int or long
 |  int(x, base=10) -> int or long
 |
 |  Convert a number or string to an integer, or return 0 if no arguments
 |  are given.  If x is floating point, the conversion truncates towards zero.
 |  If x is outside the integer range, the function returns a long instead.
 |
 |  If x is not a number or if base is given, then x must be a string or
 |  Unicode object representing an integer literal in the given base.  The
 |  literal can be preceded by '+' or '-' and be surrounded by whitespace.
 |  The base defaults to 10.  Valid bases are 0 and 2-36.  Base 0 means to
 |  interpret the base from the string as an integer literal.
 |  >>> int('0b100', base=0)
 |  4
 |
 |  Methods defined here:
 |
 |  __abs__(...)
 |      x.__abs__() <==> abs(x)
 |
 |  __add__(...)
 |      x.__add__(y) <==> x+y
 |
 |  __and__(...)
 |      x.__and__(y) <==> x&y
 |
 |  __cmp__(...)
 |      x.__cmp__(y) <==> cmp(x,y)
 |
 |  __coerce__(...)
 |      x.__coerce__(y) <==> coerce(x, y)
 |
 |  __div__(...)
 |      x.__div__(y) <==> x/y
 |
 |  __divmod__(...)
 |      x.__divmod__(y) <==> divmod(x, y)
 |
 |  __float__(...)
 |      x.__float__() <==> float(x)
 |
 |  __floordiv__(...)
 |      x.__floordiv__(y) <==> x//y
 |
 |  __format__(...)
```

```
|
|  __getattribute__(...)
|      x.__getattribute__('name') <==> x.name
|
|  __getnewargs__(...)
|
|  __hash__(...)
|      x.__hash__() <==> hash(x)
|
|  __hex__(...)
|      x.__hex__() <==> hex(x)
|
|  __index__(...)
|      x[y:z] <==> x[y.__index__():z.__index__()]
|
|  __int__(...)
|      x.__int__() <==> int(x)
|
|  __invert__(...)
|      x.__invert__() <==> ~x
|
|  __long__(...)
|      x.__long__() <==> long(x)
|
|  __lshift__(...)
|      x.__lshift__(y) <==> x<<y
|
|  __mod__(...)
|      x.__mod__(y) <==> x%y
|
|  __mul__(...)
|      x.__mul__(y) <==> x*y
|
|  __neg__(...)
|      x.__neg__() <==> -x
|
|  __nonzero__(...)
|      x.__nonzero__() <==> x != 0
|
|  __oct__(...)
|      x.__oct__() <==> oct(x)
|
|  __or__(...)
|      x.__or__(y) <==> x|y
|
|  __pos__(...)
|      x.__pos__() <==> +x
|
|  __pow__(...)
|      x.__pow__(y[, z]) <==> pow(x, y[, z])
|
|  __radd__(...)
|      x.__radd__(y) <==> y+x
|
|  __rand__(...)
|      x.__rand__(y) <==> y&x
|
|  __rdiv__(...)
|      x.__rdiv__(y) <==> y/x
|
|  __rdivmod__(...)
|      x.__rdivmod__(y) <==> divmod(y, x)
|
|  __repr__(...)
|      x.__repr__() <==> repr(x)
|
|  __rfloordiv__(...)
|      x.__rfloordiv__(y) <==> y//x
|
|  __rlshift__(...)
|      x.__rlshift__(y) <==> y<<x
|
|  __rmod__(...)
|      x.__rmod__(y) <==> y%x
|
|  __rmul__(...)
|      x.__rmul__(y) <==> y*x
|
|  __ror__(...)
|      x.__ror__(y) <==> y|x
|
|  __rpow__(...)
|      y.__rpow__(x[, z]) <==> pow(x, y[, z])
|
|  __rrshift__(...)
|      x.__rrshift__(y) <==> y>>x
|
|  __rshift__(...)
|      x.__rshift__(y) <==> x>>y
|
|  __rsub__(...)
|      x.__rsub__(y) <==> y-x
|
|  __rtruediv__(...)
|      x.__rtruediv__(y) <==> y/x
|
|  __rxor__(...)
|      x.__rxor__(y) <==> y^x
|
|  __str__(...)
|      x.__str__() <==> str(x)
|
|  __sub__(...)
|      x.__sub__(y) <==> x-y
|
|  __truediv__(...)
|      x.__truediv__(y) <==> x/y
|
 |  __trunc__(...)
 |      Truncating an Integral returns itself.
 |
 |  __xor__(...)
 |      x.__xor__(y) <==> x^y
 |
 |  bit_length(...)
 |      int.bit_length() -> int
 |
 |      Number of bits necessary to represent self in binary.
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __new__ = <built-in method __new__ of type object>
 |      T.__new__(S, ...) -> a new object with type S, a subtype of T
```

which displays a scrolling text help, or

```
In [3]: int?
```

Which displays a shorter help summary in the magic pane at the bottom of the screen. You can minimize the magic pane when it gets in your way.

If you wanted to see all the built-in commands available for something, use the dir command. Check out all of the methods of the object "Hello world", which are shared by all objects of the str type.

```
In [4]: dir("Hello world")

Out[4]: ['__add__',
         '__class__',
         '__contains__',
         '__delattr__',
         '__doc__',
'__eq__',
'__format__',
'__ge__',
'__getattribute__',
'__getitem__',
'__getnewargs__',
'__getslice__',
'__gt__',
'__hash__',
'__init__',
'__le__',
'__len__',
'__lt__',
'__mod__',
'__mul__',
'__ne__',
'__new__',
'__reduce__',
'__reduce_ex__',
'__repr__',
'__rmod__',
'__rmul__',
'__setattr__',
'__sizeof__',
'__str__',
'__subclasshook__',
'_formatter_field_name_split',
'_formatter_parser',
'capitalize',
'center',
'count',
'decode',
'encode',
'endswith',
'expandtabs',
'find',
'format',
'index',
'isalnum',
'isalpha',
'isdigit',
'islower',
'isspace',
'istitle',
'isupper',
'join',
'ljust',
'lower',
'lstrip',
'partition',
'replace',
'rfind',
'rindex',
'rjust',
         'rpartition',
         'rsplit',
         'rstrip',
         'split',
         'splitlines',
         'startswith',
         'strip',
         'swapcase',
         'title',
         'translate',
         'upper',
         'zfill']
```

There's a method that looks important – .swapcase(). Let's see what it does:

```
In [5]: "Hello world".swapcase()
```

```
Out[5]: 'hELLO WORLD'
```

We'll be going over methods in more detail. But for now, off to the next notebook!