Hands-on Exercise 0: Introduction to Python & Astropy

by Leo Singer (2014) and Eric Bellm (2015)

Introduction

Our hands-on exercises will use the Python programming language ([http://python.org/](http://python.org/)). No previous experience with Python is necessary!

Python is a powerful tool out of the box, but its strength comes from the huge number of external packages. The following are vital for data analysis:

- [Matplotlib](http://matplotlib.org): plotting interactive or publication-quality figures
- [Numpy](http://numpy.org): vectorized arithmetic and linear algebra
- [Scipy](http://scipy.org): curated collection of algorithms for root finding, interpolation, integration, signal processing, statistics, linear algebra, and much more
- [IPython Notebook](http://ipython.org/notebook.html), the Mathematica-like interface that you are using now, and last but not least
- [Astropy](http://astropy.org/), a community library for astronomy. See also [Robitaille et al. (2013)](http://dx.doi.org/10.1051/0004-6361/201322068).

We'll cover the basics of Python itself and then dive in to some applications to explore each of these packages.

We are using the Jupyter Notebook interface, which like Mathematica allows you to intermingle computer code with generated plots and output.

The notebook contains two main types of cells: [Markdown](https://guides.github.com/features/mastering-markdown/) cells for text, like this one...

```
In [204]: # and "code" cells for computation and output, like this one! Press Shift-Enter to execute it.
print 2+2
```

[4]

Double-click in this text cell to edit it:

WRITE YOUR NAME HERE

and then press Shift-Enter to execute it.
In [ ]: You can change the types of cells as well.
This cell is currently a code cell--change it to Markdown using the toolbar at the top of the page.

How to get Python/Matplotlib/Numpy/Scipy
Python and all of the packages that we discuss in this tutorial are open source software, so there are multiple options for installing them.

You should have already installed Python before our tutorial, but this section is here for your future reference.

**Recommended on all platforms: Anaconda, Canopy**


**TIP** These are both great Python environments for everyday use. If you need the latest version of Python in a headless environment like a cluster or web server, consider the lightweight Miniconda (http://conda.pydata.org/miniconda.html) distribution.

**Advanced: Debian/Ubuntu**

Most operating systems already come with Python. Release schedules vary, but generally the ‘testing’ versions of Debian (https://www.debian.org/) and Ubuntu (http://www.ubuntu.com) come with very up-to-date versions of Python. Debian Jessie (https://packages.debian.org/search?suite=jessie&searchon=names&keywords=astropy) and Ubuntu 13.10 (Saucy Salamander) (http://packages.ubuntu.com/utopic/python-astropy) both come with the latest version of Astropy. On these systems, you can set up a pretty complete Python environment with:

```
$ sudo apt-get install ipython-notebook python-matplotlib python-scipy python-pip python-astropy
```

**Advanced: MacPorts**

Every version of Mac OS comes with a Python interpreter, but it’s slightly easier to obtain Matplotlib and Numpy if you use a package manager such as MacPorts (http://macports.org), HomeBrew (http://brew.sh), or Fink (http://fink.sf.net). I use MacPorts (and contribute to it, too), so that’s what I suggest. Install MacPorts and then do:

```
$ sudo port install py27-pip py27-matplotlib py27-scipy py27-astropy py27-ipython +notebook
```

**How to install Python packages**

The Python community has developed a huge collection of packages for various tasks. The best place to find Python libraries is the Python Package Index (PyPI, https://pypi.python.org/) (https://pypi.python.org/).

No matter how you installed Python, you can install any Python package from PyPI using pip (https://pypi.python.org/pypi/pip). For example:

```
$ pip install astroquery
```

or

```
$ pip install --user astroquery
```

In Anaconda, you can also install some popular packages using conda (http://conda.pydata.org). On Debian or Ubuntu, you can get many Python packages with apt-get and on MacPorts with port.
Python basics

We're going to give you a whirlwind tour of Python--just enough to get you up to speed for the rest of this week's hands-on exercises. Because Python is widely used outside of astronomy, there are lots of great resources to learn more when you are ready!

The Official Python Tutorial (https://docs.python.org/2/tutorial/)

Python as a calculator

Python does what you expect when you type in math expressions:

```python
In [296]: 4-10
Out[296]: -6

In [297]: 2*3
Out[297]: 6
```

Data Types

Python has several useful data types. We've already met a couple--Python integers (int) and floating points (float) are just numbers!

Numeric and boolean literals

Python's numeric types include integers and both real and complex floating point numbers:

```python
In [298]:
a = 30 # an integer
b = 030 # WARNING! octal, =24 in decimal; probably not what you meant
c = 3.14159 # a floating point number
d = 5.1e10 # scientific notation
e = 2.5 + 5.3j # a complex number
hungry = True # boolean literal
need_coffee = False # another boolean literal
```
The expressions above assign values to the variables, so we can use (or modify) the variables instead of the values.

By the way, all of the text on a given line after the trailing hash sign (#) is a comment, ignored by Python.

The arithmetic operators in Python are similar to C, C++, Java, and so on. There is addition (and subtraction):

```
In [299]: a + c
Out[299]: 33.14159
```

Multiplication:

```
In [300]: a * e
Out[300]: (75+159j)
```

Division:

```
In [301]: a / c
Out[301]: 9.549304651466295
```

But **beware** that, like in C, C++, Java, etc., **division of integers gives you integers**:

```
In [302]: 7 / 3
Out[302]: 2
```

If you want true division, convert one or both of the operands to floating point:

```
In [303]: a = 7
b = 3
float(a) / b
Out[303]: 2.3333333333333335
```

**NOTE** that in **Python 3**, division of integers is **true division**.

If you are using Python 2.2 or higher, you can enable true division by putting the following statement at the top of your script:

```python
from __future__ import division
```

However, in **both Python 2 and Python 3**, the double-slash operator `//` represents integer division:

```
In [304]: 7 // 3
Out[304]: 2
```
The `%` sign is the remainder (modulus) operator:

In [305]: 32 % 26
Out[305]: 6

Exponentiation is accomplished with the `**` operator:

In [306]: 5 ** 3
Out[306]: 125

**Exercise**

Calculate the reciprocal of the cube root of 18:

In [307]: # COMPLETE
18**(1/3.)
Out[307]: 0.38157141418444396

**Strings**

Strings are text—they are indicated by pairs of quotes:

In [308]: hello = 'Hello world!

In the iPython notebook, executing a cell with an expression in it returns that expression, evaluated:

In [309]: hello
Out[309]: 'Hello world!

To display strings in functions, etc., we will print them:

In [310]: print 'Hello, world!
Hello, world!

This is a Python statement, consisting of the built-in command `print` and a string surrounded by single quotes.

**Note:** `print` changed from a built-in **statement** to a **function** in Python 3. If you are using Python 3, just put the arguments in parentheses. To get this behavior in previous versions of Python, put this at the top of your script:

    from __future__ import print_function
If you need single or double quotes inside a string, use the opposite kind outside the string:

```python
In [311]:
    print 'She said, "Hello, world!"'
    print 'She said, 'Hello, world!''
She said, "Hello, world!"
She said, 'Hello, world!'
```

If you need a string that contains newlines, use triple quotes (```) or triple double quotes ("""):

```python
In [312]:
    print """MIRANDA
    O, wonder!
    How many goodly creatures are there here!
    How beauteous mankind is! O brave new world
    That has such people in't!"""
```

Strings are concatenated (joined) by "adding" them:

```python
In [313]:
    'abc' + 'def'
Out[313]: 'abcdef'
```

Part of the magic of Python is that every datatype in python is an object. This means that, along with data there are methods (https://docs.python.org/2/library/stdtypes.html#string-methods) that operate on that data. Strings have many useful methods built-in. Let's use a filename as an example:

```python
In [314]:
    filename = '../data/PTF_d022683_f02_c06_u000114210_p12_refimg.fits'
```

If we want to check if a file ends with a certain suffix, we can use the .endswith() method:

```python
In [315]:
    filename.endswith('fits')
Out[315]: True
```

We can split the filename into pieces (a list--see the next section!) based on certain characters:

```python
In [316]:
    filename.split('/')
Out[316]: ['..', 'data', 'PTF_d022683_f02_c06_u000114210_p12_refimg.fits']
```

Note that we can get help on Python functions with help.
In [317]: help(filename.split)

Help on built-in function split:

    split(...)  
    S.split([sep [,maxsplit]]) --> list of strings

    Return a list of the words in the string S, using sep as the
    delimiter string. If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are removed
    from the result.

Out[318]: <function split>

Or if you are typing them in a notebook, try pressing Shift-Tab after you type the function name!

In [318]: # put your cursor after "split" below and type Shift-Tab a few times:
filename.split

Out[318]: <function split>

The dir function lists the methods of a given python object. Methods with names surrounded by __double underscores__ are internal; you can ignore them for now.
In [319]: dir(filename)
Out[319]: ['__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',]
Exercise

Replace "PTF" in the `filename` variable with your name, and make the whole string uppercase.

```python
In [320]: # COMPLETE
    filename.replace("PTF","Eric").upper()
Out[320]: '../DATA/ERIC_D022683_F02_C06_U000114210_P12_REFIMG.FITS'
```

Lists

Lists are flexible containers which can contain Python objects of any type. You can add or remove items or alter existing items. Lists are surrounded by square brackets and have items separated by commas.

```python
In [321]: some_list = ['a','b',3,10]
```

Once you have made a list, you might want to retrieve a value from it. You index a list with square brackets, **starting from zero:**

```python
In [322]: some_list[0]
Out[322]: 'a'

In [323]: some_list[1]
Out[323]: 'b'
```

You can access whole ranges of values using **slice notation:**

```python
In [324]: some_list[1:4]
Out[324]: ['b', 3, 10]
```

Or, to count backward from the end of the list, use a **negative index:**

```python
In [325]: some_list[-2]
Out[325]: 3
```

Lists can be nested:

```python
In [326]: another_list = [4,5,[6,7,8]]

In [327]: another_list[-1]
Out[327]: [6, 7, 8]
```
In [328]: another_list[-1][1]
Out[328]: 7

len gives the number of items in a list, but note that nested lists only count as one object:

In [329]: len(another_list)
Out[329]: 3

Strings can be treated just like lists of individual characters:

In [330]: person = 'Miranda'
   print person[3:6]
   and

In [331]: your_list = ['foo', 'bar', 'bat', 'baz']
   my_list = ['xyzzy', 1, 3, 5, 7]

You can change elements:

In [332]: my_list[1] = 2
   print my_list
   ['xyzzy', 2, 3, 5, 7]

Or append elements to an existing list:

In [333]: my_list.append(11)
   print my_list
   ['xyzzy', 2, 3, 5, 7, 11]

Or delete elements:

In [334]: del my_list[0]
   print my_list
   [2, 3, 5, 7, 11]

**Exercise**

Replace the second element of another_list with 7, remove the nested list, and append 12.

In [335]: # COMPLETE
   another_list[1] = 7
   del another_list[-1]
   another_list.append(12)
   another_list

Out[335]: [4, 7, 12]
Dictionaries

Sometimes, you want a collection that is like a list, but whose indices are strings or other Python values. That's a dictionary. Dictionaries are handy for any type of database-like operation, or for storing mappings from one set of values to another. You create a dictionary by enclosing a list of key-value pairs in curly braces:

```python
In [336]:
my_grb = {'name': 'GRB 130702A', 'redshift': 0.145, 'ra': (14, 29, 14.78), 'dec': (15, 46, 26.4)}
my_grb
```

```
Out[336]:
{'dec': (15, 46, 26.4),
 'name': 'GRB 130702A',
 'ra': (14, 29, 14.78),
 'redshift': 0.145}
```

You can index items in dictionaries with square braces [], similar to lists, but you provide the name of the mapping element rather than a numerical position. Dictionaries may store items in any order!

```python
In [337]: my_grb['dec']
```

```
Out[337]: (15, 46, 26.4)
```

or add items to them:

```python
In [338]: my_grb['url'] = 'http://gcn.gsfc.nasa.gov/other/130702A.gcn3'
my_grb
```

```
Out[338]:
{'dec': (15, 46, 26.4),
 'name': 'GRB 130702A',
 'ra': (14, 29, 14.78),
 'redshift': 0.145,
 'url': 'http://gcn.gsfc.nasa.gov/other/130702A.gcn3'}
```

or delete items from them:

```python
In [339]: del my_grb['url']
my_grb
```

```
Out[339]:
{'dec': (15, 46, 26.4),
 'name': 'GRB 130702A',
 'ra': (14, 29, 14.78),
 'redshift': 0.145}
```

Dictionary keys can be any immutable kind of Python object: tuples, strings, integers, and floats are all fine. Values in a dictionary can be any Python value at all, including lists or other dictionaries.
Exercise

Add two vegetable choices to the `menu_options` dictionary and replace your least-favorite entree with something better. Add another guest. Then print a string listing your choices for cheese, vegetable, entree, and dessert separated by commas.

```python
In [340]: menu_options = {
    'entrees': ['chicken', 'veggie burger', 'BBQ'],
    'cheeses': ['muenster', 'gouda', 'camembert', 'mozarella'],
    'dessert': 'icecream',
    'number_of_guests': 42
}
```

```python
In [341]: # COMPLETE
   menu_options['entrees'][0] = 'pizza'
   menu_options['veggies'] = ['corn', 'asparagus']
   menu_options['number_of_guests'] += 1
   print menu_options['cheeses'][1] + ',' + menu_options['veggies'][1] + ',' + menu_options['entrees'][0] + ',' + menu_options['dessert']
```

gouda, asparagus, pizza, icecream

The None object

Sometimes you need to represent the absence of a value, for instance, if you have a gap in a dataset. You might be tempted to use some special value like -1 or 99 for this purpose, but don’t! Use the built-in object None.

```python
In [342]: a = None
```

Conditionals

In Python, control flow statements such as conditionals and loops have blocks indicated with indentation. Any number of spaces or tabs is fine, as long as you are consistent within a block. Four spaces is the recommended standard.

You can use the `if...elif...else` statement to have different bits of code run depending on the truth or falsehood of boolean expressions. For example:
In [343]:

    a = 5

    if a < 3:
        print "i'm in the 'if' block"
        message = 'a is less than 3'
    elif a == 3:
        print "i'm in the 'elif' block"
        message = 'a is 3'
    else:
        print "i'm in the 'else' block"
        message = 'a is greater than 3'

    print message

    # i'm in the 'else' block
    # a is greater than 3

You can chain together inequalities just like in mathematical notation:

In [344]:

    if 0 < a <= 5:
        print 'a is greater than 0 but less than or equal to 5'

    # a is greater than 0 but less than or equal to 5

You can also combine comparison operators with the boolean and, or, and not operators:

In [345]:

    if (a < 6) or (a > 8):
        print 'yahoo!'

    # yahoo!

In [346]:

    if not a == 5:
        # same as a != 5
        print 'a is not 5'

Exercise:

Write a conditional expression that tests if a is an odd number less than 6:

In [347]:

    if (a < 6) and (a % 2 == 1):
        # COMPLETE
        print 'a is an odd number less than 6!'

    # a is an odd number less than 6!

The comparison operator is tests whether two Python values are not only equal, but represent the same object. Since there is only one None object, the is operator is particularly useful for detecting None.

In [348]:

    food = None

    if food is None:
        print 'No soup for you!'
    else:
        print 'Here is your', food

    # No soup for you!
Likewise, there is an `is not` operator:

```
In [349]: if food is not None:
    print 'Yum!'
```

The `in` and `not in` operators are handy for testing for membership in a string, list, or dictionary:

```
In [350]: if 3 in [1, 2, 3, 4, 5]:
    print 'indeed it is'

indeed it is
```

```
In [351]: if 'i' not in 'team':
    print 'there is no "i" in "team"

there is no "i" in "team"
```

When referring to a dictionary, the `in` operator tests if the item is among the **keys** of the dictionary.

```
In [352]: d = {'foo': 3, 'bar': 5, 'bat': 9}
if 'foo' in d:
    print 'the key "foo" is in the dictionary'

the key "foo" is in the dictionary
```

### The **for** and **while** loops

In Python, there are just two types of loops: **for** and **while**. **for** loops are useful for repeating a set of statements for each item in a collection (tuple, set, list, dictionary, or string). **while** loops are not as common, but can be used to repeat a set of statements until a boolean expression becomes false.

```
In [353]: for i in [0, 1, 2, 3]:
    print i**2

0
1
4
9
```

The built-in function `range`, which returns a list of numbers, is often handy here:

```
In [354]: for i in range(4):
    print i**2

0
1
4
9
```

Or you can have the range start from a nonzero value:
In [355]:
    for i in range(-2, 4):
       print i**2

4
1
0
1
4
9

You can iterate over the keys and values in a dictionary with .items() or .iteritems():

In [356]:
    for key, val in d.items():
       print key, '...', val**3

bat ... 729
foo ... 27
bar ... 125

The syntax of the while loop is similar to the if statement:

In [357]:
    a = 1
    while a < 5:
       a = a * 2
       print a

2
4
8

**Exercise**

Print the keys in our menu_options dictionary in uppercase, followed by the number of items in the value. You will get an error: how can you work around it?

In [358]:
    # COMPLETE
    for key, val in menu_options.items():
       if key is not 'number_of_guests':
          print key.upper(), len(val)

CHEESES 4
ENTREES 3
VEGGIES 2
DESSERT 8

**List comprehensions**

Sometimes you need a loop to create one list from another. List comprehensions make this very terse. For example, the following for loop:
In [359]:
    | a = []
    | for i in range(5):
    |     a.append(i * 10)

is equivalent to this list comprehension:

In [360]:
    | a = [i * 10 for i in range(5)]

You can even incorporate conditionals into a list comprehension. The following:

In [361]:
    | a = []
    | for i in range(5):
    |     if i % 2 == 0:
    |         # i is even
    |         a.append(i * 10)

can be written as:

In [362]:
    | a = [i * 10 for i in range(5) if i % 2 == 0]

Exercise

Return a list of the lengths (in characters--use len()) of all of the keys in menu_options except number_of_guests. Hint: menu_options.keys() returns a list of keys.

In [363]: # COMPLETE
    | [len(m) for m in menu_options.keys() if m != 'number_of_guests']

Out[363]: [7, 7, 7, 7]

Functions

Functions are created with the def statement. A function may either have or not have a return statement to send back a return value.

In [364]:
    | def square(n):
    |     return n * n
    | a = square(3)
    | print a

    | 9

If you want to return multiple values from a function, separate them by commas:
In [365]:
```python
def powers(n):
    return n**2, n**3

print powers(3)
```
```
(9, 27)
```

If a function returns multiple values, you can automatically unpack them into multiple variables:

In [366]:
```python
square, cube = powers(3)
print square
```
```
9
```

If you pass a mutable value such as a list to a function, then the function may modify that value. For example, you might implement the Fibonacci sequence like this:

In [367]:
```python
def fibonacci(seed, n):
    while len(seed) < n:
        seed.append(seed[-1] + seed[-2])
    # Note: no return statement

seed = [1, 1]
fibonacci(seed, 10)
print seed
```
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

You can also give a function’s arguments default values, such as:

In [368]:
```python
def fibonacci(seed, n=6):
    """Fill a seed list with values from the Fibonacci sequence until it has length n.""
    # If the first line of a function is a string, then that string is # used to create online help for the function.
    while len(seed) < n:
        seed.append(seed[-1] + seed[-2])
    # Note: no return statement

seed = [1, 1]
fibonacci(seed)
print seed
```
```
[1, 1, 2, 3, 5, 8]
```

If a function has a large number of arguments, it may be easier to read if you pass the arguments by keyword, as in:

In [369]:
```python
seq = [1, 1]
fibonacci(seed=seq, n=4)
```
```
(9, 27)
9
[1, 1, 2, 3, 5]
**Exercise**

Write a function that plays "FizzBuzz":

given an input number n, it prints the numbers from 1-n, unless the number is a multiple of 3, in which case print "Fizz". If the number is a multiple of 5, print "Buzz". If both, print "FizzBuzz".

run your function for n=20.

```python
In [370]: def fizzbuzz(n):
    for i in range(n):
        j = i+1
        outstr = '
        if (j % 3) == 0:
            outstr = 'Fizz'
        if (j % 5) == 0:
            outstr += 'Buzz'
        if len(outstr) > 0:
            print outstr
        else:
            print j
fizzbuzz(20)
```

1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
Fizz
19
Buzz

**Python standard library**

Python comes with an extensive standard library [http://docs.python.org/2/library/index.html](http://docs.python.org/2/library/index.html) consisting of individual modules that you can opt to use with the `import` statement.

You can import specific functions or variables with the `from x import y` syntax:

```python
In [371]: from math import pi
pi
Out[371]: 3.141592653589793
```
or import the whole module in a "namespace" with import x or import long_module as y. Then you can use all of the functions and variables by calling x.func() or y.variable.

In [372]:
   import math
   math.log(math.e)

Out[372]: 1.0

Some particularly useful parts of the Python standard library are:

- random (http://docs.python.org/2/library/random.html): random number generators
- pickle (http://docs.python.org/2/library/pickle.html): read/write Python objects into files
- os (http://docs.python.org/2/library/os.html): operating system services
- os.path (http://docs.python.org/2/library/os.path.html): file path manipulation
- sqlite3 (http://docs.python.org/2/library/sqlite3.html): SQLite database access
- subprocess (http://docs.python.org/2/library/subprocess.html): launch external processes
- email (http://docs.python.org/2/library/email.html): compose, parse, receive, or send e-mail
- pdb (http://docs.python.org/2/library/pdb.html): built-in debugger
- re (http://docs.python.org/2/library/re.html): regular expressions
- SimpleHTTPServer (http://docs.python.org/2/library/simplehttpserver.html): built-in lightweight web server
- optparse (http://docs.python.org/2/library/optparse.html): build pretty command-line interfaces
- itertools (http://docs.python.org/2/library/itertools.html): exotic looping constructs
- multiprocessing (https://docs.python.org/2/library/multiprocessing.html): parallel processing

help() and dir() are useful for finding what commands are available in the standard library.

Exercise:

Use the os module to print the current working directory. Then list its contents. (Look here (https://docs.python.org/2/library/os.path.html) for functions.)

In [373]:
   # COMPLETE
   import os
   cwd = os.getcwd()
   print cwd
   print os.listdir(cwd)

/Users/ebellm/Eric/PTF/Meetings/iPTF Summer School 2015/ztf_summerschool_2015/notebooks
['.ipynb_checkpoints', 'BaryTime.py', 'BaryTime.pyc', 'Finding_RRLyrae_candidates.ipynb', 'Identify_QSO_RRL_w_Random_Forest.ipynb', 'Introduction_to_Python_and_Astropy.ipynb', 'Making_a_Lightcurve.ipynb', 'myfile.txt', 'Period_Finding.ipynb', 'ztf_summerschool.py', 'ztf_summerschool.pyc']

Error handling

It can be important for your code to be able to handle error conditions. For example, let's say that you are implementing a sinc function:
In [374]:
    
    ```python
    def sinc(x):
        return math.sin(x) / x
    
    print sinc(0)
    ```

`ZeroDivisionError` Traceback (most recent call last)
<ipython-input-374-0e6b1b141058> in <module>(
    2        return math.sin(x) / x
    3    
----> 4    print sinc(0)

<ipython-input-374-0e6b1b141058> in sinc(x)
    1    def sinc(x):
    ----> 2        return math.sin(x) / x
    3    
    4    print sinc(0)

`ZeroDivisionError`: float division by zero

Oops! We know that by definition sinc(0) = 1, so we should catch this error:

In [375]:
    
    ```python
    def sinc(x):
        try:
            result = math.sin(x) / x
        except ZeroDivisionError:
            result = 1
        return result
    
    print sinc(0)
    ```

1

**Reading and writing files**

The built-in `open` function opens a file and returns a file object that you can use to read or write data. Here's an example of writing data to a file:

In [376]:
    
    ```python
    myfile = open('myfile.txt', 'w') # open file for writing
    myfile.write("red 1\n")
    myfile.write("green 2\n")
    myfile.write("blue 3\n")
    myfile.close()
    ```

And here is reading it:
In [377]:
    
    d = {}  # create empty dictionary
    
    for line in open('myfile.txt', 'r'):  # open file for reading
        color, num = line.split()  # break apart line by whitespace
        num = int(num)  # convert num to integer
        d[color] = num  #
    
    print d

{'blue': 3, 'green': 2, 'red': 1}

Numpy & Matplotlib

Numpy provides array operations and linear algebra to Python. A Numpy array is a bit like a Python list, but arrays usually contain only numbers and support elementwise arithmetic. For example:

In [378]:
    
    import numpy as np
    x = np.array([1, 2, 3, 4, 5])
    y = 2 * x
    print y

[ 2  4  6  8 10]

By default, multiplication is elementwise:

In [379]:
    x * y

Out[379]:
array([ 2,  8, 18, 32, 50])

To perform matrix multiplication, either convert arrays to np.matrix or use np.dot:

In [380]:
    np.dot(x, y)

Out[380]:
110

Numpy arrays may have any number of dimensions:

In [381]:
    
    x = np.array([[[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
    x

Out[381]:
array([[[1, 2, 3],
        [4, 5, 6],
        [7, 8, 9]]])

An array has a certain number of dimensions denoted .ndim:

In [382]:
    x.ndim

Out[382]:
2
and the dimensions’ individual lengths are given by .shape:

In [383]: `x.shape`
Out[383]: (3, 3)

and the total number of elements by .size:

In [384]: `x.size`
Out[384]: 9

You can also perform comparison operations on arrays...

In [385]: `x > 5`
Out[385]: array([[-False, False, False],
                [-False, False,  True],
                [-  True,  True,  True]], dtype=bool)

Although a boolean array doesn’t directly make sense in an if statement:

In [386]: ```
   if x > 5:
      print 'oops'
```--------> 1 if x > 5:
                 2 print 'oops'

ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()

In [387]: ```
   if np.any(x > 5):
      print 'at least some elements are greater than 5'
```at least some elements are greater than 5

You can use conditional expressions like indices:

In [388]: ```
   x[x > 5] = 5
   x
```Out[388]: array([[1, 2, 3],
               [4, 5, 5],
               [5, 5, 5]])

Or manipulate individual rows:
In [389]:
"x[1, :] = -1"

Out[389]:
array([[ 1,  2,  3],
       [-1, -1, -1],
       [ 5,  5,  5]])

Or individual columns:

In [390]:
"x[:, 1] += 100"

Out[390]:
array([[  1, 102,   3],
       [ -1,  99,  -1],
       [  5, 105,   5]])

Other useful features include various random number generators:

In [391]:
"r = np.random.randn(10000)"

And statistical functions like mean and standard deviation:

In [392]:
"print np.mean(r), np.std(r)"

-0.00248717988281 0.998729525905

Matplotlib is the standard python plotting package. Line plots, histograms, scatter plots, and error bar plots are the most useful:

In [393]:
"import matplotlib.pyplot as plt"

%matplotlib inline
x = np.linspace(-10, 10)
y = 1 / (1 + np.exp(x))
plt.plot(x, y)
plt.xlabel('X coordinate')
plt.ylabel('Y coordinate')
plt.title('Title')

Out[393]:
<matplotlib.text.Text at 0x11b5badd0>
In [394]: plt.hist(np.random.randn(10000))

Out[394]: (array([  5.,  60.,  422., 1309., 2483., 2852., 1914.,  757., 172.,  26.]),
array([-3.90733764, -3.16195146, -2.41656528, -1.67117909, -0.92579291,
    -0.18040673,  0.56497945,  1.31036563,  2.05575182,  2.801138 ,
        3.54652418]),<a list of 10 Patch objects>)

In [395]: plt.scatter(np.random.rand(500),np.random.randn(500),alpha=0.5,color='red')

Out[395]: <matplotlib.collections.PathCollection at 0x10eb70c10>
In [396]: plt.errorbar(np.arange(10), np.arange(10), yerr=np.ones(10), alpha=0.5, fmt='_')

Out[396]: <Container object of 3 artists>

Exercise

Generate \( x = 100 \) random numbers between 0 and 2 \( \pi \), and make a scatter plot of \( x \) vs \( y = \sin(x) \). Color all the points with absolute value greater than 0.8 red and the others black.

In [397]:
x = np.random.rand(100)*2*np.pi
y = np.sin(x)
w = np.abs(y) > 0.8
plt.scatter(x[w], y[w], color='red')
plt.scatter(x[-w], y[-w], color='black')

Out[397]: <matplotlib.collections.PathCollection at 0x10ec87cd0>

Astropy
Astropy is a core Python package for astronomy. It is formed from the merger of a number of other Python astronomy packages, but also contains a lot of original code. Core features include:

- `astropy.constants`, `astropy.units`: Physical constants, units, and unit conversion
- `astropy.time`: Manipulation of dates and times
- `astropy.coordinates`: Representation of and conversion between astronomical coordinate systems
- `astropy.table`: Tables and gridded data
- `astropy.io.fits`: Manipulating FITS files
- `astropy.io.ascii`: Manipulating ASCII tables of many different formats
- `astropy.io.votable`: Virtual Observatory tables
- `astropy.wcs`: World Coordinate System transformations
- `astropy.cosmology`: Cosmological calculations
- `astropy.stats`: Astrostatistics
- `astropy.modeling`: multi-D model fitting Swiss army knife

The Astropy project also has several "affiliated packages" (http://www.astropy.org/affiliated/index.html) that have similar design but are maintained separately, including:

- APLPy (http://aplpy.github.io): High-level astronomical map making with Matplotlib
- WCSAxes (http://wcsaxes.readthedocs.org/): Nuts-and-bolts astronomical mapmaking for Matplotlib experts
- Photutils (http://photutils.readthedocs.org/en/latest/): Aperture photometry
- Astroquery (http://astroquery.readthedocs.org/en/latest/): Query astronomical databases

Let’s experiment by opening up a P48 image. We’ll need several modules from the Astropy package for this exercise.

```python
In [398]:
import astropy.coordinates as coords
import astropy.units as u
import astropy.io.fits
import astropy.stats
import astropy.table
import astropy.wcs
```

We just have time to scratch the surface of what astropy is capable of. `astropy.constants` and `astropy.units` are both quite useful:

```python
In [399]:
import astropy.constants as const
import astropy.units as u

const.c
```

```
Out[399]: 2.9979246e+8 m/s
```

Unit conversion is as simple as adding `.to(unit)`:

```python
In [400]:
const.c.to('AU/year')
```

```
Out[400]: 63241.077 AU/yr
```
Exercise

Use astropy.constants to calculate the mean density of Earth in g/cm^3.

In [401]:

```python
(const.M_earth/(4./3.*np.pi*const.R_earth**3.)).to('g/cm^3') # COMPLETE
```

Out[401]:

```
5.4967986 g/cm^3
```

Let's work with one of the PTF coadded images we'll use later in the hands-on sessions.

In [402]:

```python
image_filename = '../data/PTF_Refims_Files/PTF_d022683_f02_c06_u000114210_p12_refimg.fits'
hdulist = astropy.io.fits.open(image_filename)
```

Out[402]:

```
[<astropy.io.fits.hdu.image.PrimaryHDU at 0x10ebcb910>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ecf7f90>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ed9cf50>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10edcfed0>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ee01e50>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ee34dd0>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ee67d50>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ee99cd0>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10eeccc50>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ef06bd0>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ef3cb50>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ef73ad0>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10ef979d0>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10efaaa50>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10f01048d0>,
 <astropy.io.fits.hdu.image.ImageHDU at 0x10f13c850>]
```

Let's grab the first HDU ('header data unit') of this FITS file:

In [403]:

```python
hdu = hdulist[0]
hdu.scale('int16','old') # this file needs to be rescaled, for some reason
```

Then let's take a look at the contents of the header:
In [404]: hdu.header
Out[404]: SIMPLE = T / Fits standard
BITPIX = 16 / FOUR-BYTE SINGLE PRECISION FLOATING POINT
NAXIS = 2 / STANDARD FITS FORMAT
NAXIS1 = 2500 / STANDARD FITS FORMAT
NAXIS2 = 4600 / STANDARD FITS FORMAT
EXTEND = T
ORIGIN = 'Palomar Transient Factory' / Origin of these image data
CREATOR = 'Infrared Processing and Analysis Center' / Creator of this FITS file
TELESCOP= 'P48 ' / Name of telescope
INSTRUME='PTF/MOSAIC' / Instrument name
OBSERVER= 'Kulkarni/PTF' / Observer name and project
CCDID = '6 ' / CCD number (0..11)
DATE = '2013-08-15T16:44:40' / File creation date (YYYY-MM-DDThh:mm:ss UT)
REFERENC= 'http://www.astro.caltech.edu/ptf' / URL of PTF website

/ PROPOSAL INFORMATION

PTFPRPI = 'Kulkarni' / PTF Project PI
PTFFIELD= '22683 ' / PTF unique field ID

/ TIME AND EXPOSURE INFORMATION

FILTER = 'R ' / Filter name
FILTERID= '2 ' / Filter ID
OBSTYPE = 'object ' / Image type (dark,science,bias,focus)
IMGTYP = 'coadd ' / Image type (dark,science,bias,focus)
MINMJD = 56147.2896199999 / [day] MJD of earliest obs. in co-add
MAXMJD = 56177.2724000001 / [day] MJD of latest obs. in co-add

/ PHOTOMETRY

BUNIT = 'DN ' / Data number (analog-to-digital units or ADU)
IMAGEZPT= 27. / [mag] Photometric zero point
Now let's plot the image data. But let's use sigma-clipping to pick a nice scale for the image.

```
In [405]:
clipped_data = astropy.stats.sigma_clip(hdu.data.flatten())
mid = np.median(clipped_data)
std = np.std(clipped_data - mid)

In [406]:
plt.figure(figsize=(20, 10))
plt.imshow(hdu.data, vmin=0, vmax=mid+std, cmap='binary')
plt.xlabel('pixel $x$')
plt.ylabel('pixel $y$')

Out[406]: <matplotlib.text.Text at 0x10f15df90>
```

Let's look at a star that will quickly become our favorite: it has coordinates $\alpha_{2000}, \delta_{2000} = (312.503802, -0.706603)$.

We'll open the catalog file from IPAC and find this object.
In [407]:
catalog_filename = image_filename.replace('refimg.fits','sexcat.ctlg')
catalog_table = astropy.table.Table.read(catalog_filename)
catalog_table
<table>
<thead>
<tr>
<th>NUMBER</th>
<th>FLAGS</th>
<th>XWIN_IMAGE</th>
<th>YWIN_IMAGE</th>
<th>X_WORLD</th>
<th>Y_WORLD</th>
<th>XPEAK_IMAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>333.717659537</td>
<td>277.099189095</td>
<td>312.504342037</td>
<td>-1.16812313268</td>
<td>334</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>718.343547332</td>
<td>276.028666957</td>
<td>312.396712767</td>
<td>-1.16815611831</td>
<td>718</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1506.51889909</td>
<td>277.533914272</td>
<td>312.17507288</td>
<td>-1.1677071056</td>
<td>1509</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1565.16614655</td>
<td>276.977910615</td>
<td>312.158801397</td>
<td>-1.1678816403</td>
<td>1565</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1714.38609432</td>
<td>277.959449923</td>
<td>312.116929942</td>
<td>-1.1676616581</td>
<td>1714</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1395.03388768</td>
<td>278.003429169</td>
<td>312.206330327</td>
<td>-1.16749516627</td>
<td>1395</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1924.34484358</td>
<td>278.652661268</td>
<td>312.05801388</td>
<td>-1.16728946129</td>
<td>1924</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>392.896834628</td>
<td>277.573327341</td>
<td>312.487767619</td>
<td>-1.16804321021</td>
<td>393</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>692.733950419</td>
<td>276.832207281</td>
<td>312.403784888</td>
<td>-1.16813573524</td>
<td>692</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>16918</td>
<td>3</td>
<td>1108.52800856</td>
<td>3907.1484174</td>
<td>312.287597349</td>
<td>-0.149664892923</td>
<td>1109</td>
</tr>
<tr>
<td>16919</td>
<td>0</td>
<td>378.42815944</td>
<td>3920.77847876</td>
<td>312.492397567</td>
<td>-0.14598410437</td>
<td>378</td>
</tr>
<tr>
<td>16920</td>
<td>0</td>
<td>1103.90142612</td>
<td>3922.52710115</td>
<td>312.288910771</td>
<td>-0.145349965476</td>
<td>1104</td>
</tr>
<tr>
<td>16921</td>
<td>0</td>
<td>1910.7909822</td>
<td>3978.20569629</td>
<td>312.062533709</td>
<td>-0.129605317631</td>
<td>1911</td>
</tr>
<tr>
<td>16922</td>
<td>0</td>
<td>945.303659751</td>
<td>3950.45516614</td>
<td>312.333143999</td>
<td>-0.137574128833</td>
<td>945</td>
</tr>
<tr>
<td>16923</td>
<td>0</td>
<td>1483.75966018</td>
<td>3888.31123698</td>
<td>312.182308698</td>
<td>-0.15487344919</td>
<td>1484</td>
</tr>
<tr>
<td>16924</td>
<td>0</td>
<td>882.982973811</td>
<td>3910.28642829</td>
<td>312.350875233</td>
<td>-0.148852964152</td>
<td>883</td>
</tr>
<tr>
<td>16925</td>
<td>0</td>
<td>1498.07568872</td>
<td>3899.63978906</td>
<td>312.178281288</td>
<td>-0.154509998909</td>
<td>1498</td>
</tr>
<tr>
<td>16926</td>
<td>1</td>
<td>1619.18473736</td>
<td>3914.79065531</td>
<td>312.144213823</td>
<td>-0.147397053561</td>
<td>1619</td>
</tr>
<tr>
<td>16927</td>
<td>0</td>
<td>2233.24652751</td>
<td>3913.26757496</td>
<td>311.972057737</td>
<td>-0.147759703299</td>
<td>2233</td>
</tr>
</tbody>
</table>
The astropy SkyCoord code enables efficient crossmatching between positions.

Let’s create a coordinates object to represent the target that we are searching for:

```python
In [408]:
target_coord = astropy.coordinates.SkyCoord(312.503802, -0.706603, unit=u.deg)
target_coord
```

Out[408]: `<SkyCoord (ICRS): (ra, dec) in deg
            (312.503802, -0.706603)>`

Now, let’s match this position against the source catalog:

```python
In [409]:
catalog_coords = astropy.coordinates.SkyCoord(
    ra=catalog_table['ALPHAWIN_J2000'],
    dec=catalog_table['DELTAWIN_J2000'])

    # Do source matching
    index, separation, distance = target_coord.match_to_catalog_sky(catalog_coords)

    index, separation
```

Out[409]: `(array(6907), <Angle [ 0.00012469] deg>)`

In uncrowded PTF fields like this one, we typically use a match radius of 1.5". separation is an astropy Angle, which has units, so we can easily test if the matched object is the correct one:

```python
In [410]:
    separation < 1.5*u.arcsec

Out[410]: array([ True], dtype=bool)
```

Here is the closest-matching row:

```python
In [411]:
    index = np.asscalar(index)
    matching_row = catalog_table[index]
    matching_row['MAG_AUTO']

Out[411]: 18.14098
```
Let's look at where these catalog sources appear on the image. We'll need to convert the RA, Dec in the table to pixel coordinates. That's where the World Coordinate System (WCS) transformation comes in.
In [414]:
plt.figure(figsize=(10, 10))
plt.imshow(hdu.data, vmin=mid-std, vmax=mid+std, cmap='binary')
plt.xlim(matching_row['XWIN_IMAGE'] - 128, matching_row['XWIN_IMAGE'] + 128)
plt.ylim(matching_row['YWIN_IMAGE'] - 128, matching_row['YWIN_IMAGE'] + 128)

# Note: last argument is 'origin': FITS standard uses 1-based Fortran-like
# indexing, but Python uses 0-based C-like indexing. In this case, we are
# aligning these locations to a Python array, so we want 0-based indexing.
x, y = wcs.all_world2pix(catalog_coords.ra.deg, catalog_coords.dec.deg, 0)
plt.scatter(x, y, facecolor='none', edgecolor='red')

# I happen to know that these images are upside down; use APLPy or the like t
# o automatically orient North upward
plt.gca().invert_yaxis()
plt.xlabel('pixel $x$')
plt.ylabel('pixel $y$')

Out[414]: <matplotlib.text.Text at 0x116bb52d0>

For more sophisticated plotting of astronomical images, WCSAxes or APLPy are useful modules.
Exercise

The catalog file above is for R-band. Load the g-band catalog also and crossmatch both catalogs against each other using a 1.5\degree match radius. Scatter plot g-r color against r-band magnitude. (You'll want to adjust the x & y limits to zoom in on the data.)

```python
In [ ]:
g_catalog_filename = '../data/PTF_Refims_Files/PTF_d022683_f01_c06_u000097743_p12_sexcat.ctlg'
g_catalog_table = astropy.table.Table.read(g_catalog_filename)
g_catalog_coords = astropy.coordinates.SkyCoord(  # COMPLETE
    ra=g_catalog_table['ALPHAWIN_J2000'],
    dec=g_catalog_table['DELTAWIN_J2000'])
indexes, separations, distances = g_catalog_coords.match_to_catalog_sky(  # COMPLETE
catalog_coords)
wmatch = separations < 1.5*u.arcsec  # COMPLETE
gmags = g_catalog_table['MAG_AUTO'][wmatch]
rmags = catalog_table['MAG_AUTO'][indexes][wmatch]
```

```python
In [416]: plt.scatter(rmags, gmags-rmags, alpha=0.1)  # COMPLETE
plt.xlim(14, 23)  # COMPLETE
plt.ylim(0, 4)  # COMPLETE
plt.xlabel('R-band magnitude')
plt.ylabel('g-R magnitude')
```

Out[416]: `<matplotlib.text.Text at 0x11befb5d0>`