2DayPython Documentation

Release 1.0

Matt Davis, Mike Fletcher

Mar 22, 2017

Contents

Learning to Program

- You need to accomplish a task.
- You decide to do it by programming a computer.
- Now you have two tasks.

CHAPTER 1

What is Programming?

Breaking down a problem into tasks so simple even a computer, which is the most mind-bogglingly stupid thing you will ever encounter, can solve the problem. This allows you to use the mind-bogglingly stupid thing to do the task very, very fast, while you go watch cartoons.

For example, say you wanted to drop a letter in the mailbox at the end of your street. If you wanted to tell a human to do this, you would hand the letter to them and say:

- drop off this letter

but that only works because a human has enormous amounts of shared experiences (and language) that allows them to understand, plan out a course of action, and act on those actions.

For a computer, you would have to spell out the actions:

- **drop off this letter**
    - take hold of the letter
    - walk out the door
    - walk down the steps
    - turn at the sidewalk
    - proceed to the mailbox
    - open the drawer
    - put letter in drawer
    - close the drawer

well, that's obviously way more work, but it's not too bad, really. Except that a computer would likely *not* be able to accomplish the task with that level of detail:

- **drop off this letter**
    - take hold of the letter
    - **walk out the door**

```
∗ if the door is closed
    · open the door
        grasp handle
        turn handle until latch disengages
        while the door isn’t open enough
            pull (inward) on handle
            if door is going to hit you
                step back to allow door to open
        release handle
∗ while not yet through the door
    · walk forward
        lift rear foot  if feet are at same point, choose right
        swing foot forward
        plant foot
        shift weight to forward foot
```

These examples are still *far* more abstract than a computer would understand. A computer is programmed, ultimately by controlling whether particular switches are held open or closed, but few programmers today work at that level of abstraction.

Over time, programmers create “libraries” of very well defined descriptions of how to accomplish tasks such as “walk forward”, and then other programmers will simply “say” *walk_forward( )* when they want the computer to perform the task. Programming languages (such as Python) translate that human-friendly text *walk_forward(2.5)* through a *large* numbers of abstractions until it becomes a series of on-off values (bits) which control how the switches which make up a computer behave.

This process of abstracting away the details of programming is what makes it possible to accomplish so much so quickly today. While 40 years ago every tiny detail of a task might be broken down to individual on/off switch values by a programmer wishing to accomplish a task, today high level languages and ever-advancing “libraries” of solutions to particular problems mean that you can solve problems that would have been dauntingly complex just a few decades ago.

But the basics of programming have not changed. We break problems down into smaller and smaller steps, until we come to a level of abstraction the computer already understands.

CHAPTER 2

Why Python?

- Is very high level, with libraries to accomplish most common tasks
- Relatively straightforward to learn and use
- Free and open source
- Useful in many domains
  - System tasks
  - Web and database programming
  - Bioinformatics (BioPython)
  - Data analysis and visualization
- Large global community
  - Lots of big companies
  - Active support systems for both new and experienced programmers

CHAPTER 3

Contents

System Setup

- You will need a Python interpreter (Python 2.7 for the current version of these tutorials)
  - Academic users may wish to use the Enthought Python Distribution but we do not use any special features of this distribution in this lesson
- You will need a copy of this project distribution (the exercises and sample data files are included in the archive)
  - save and extract the `download.tar.gz` file into a new directory
    * `mkdir python-lesson`
    * `wget http://142.1.253.67/download.tar.gz`
    * `tar -zxvf download.tar.gz`
  - the `workshop` directory contains a local copy of this site
  - the `exercises` directory contains the exercise and source-code samples you will need to complete the session
- You will need a "command shell"
  - On Windows
    * You may use *cmd* or *powershell* (Start | Run | cmd)
    * Or you may wish to install the Cygwin environment to obtain a copy of bash (and potentially Python)
    * You will likely want to add the directory *c:Python27* to your PATH environment variable.
  - On Linux or Mac OSX you should already have *bash* available, and likely have a GUI terminal application available
- test that you can run the exercise scripts (in the `exercises` folder)

Lesson Plan

You may want to follow along with some of these examples, you can start an interactive Python prompt (an "interpreter") such as you see here by running `python` (the basic Python interpreter) or `ipython` (a more friendly interpreter).

```
$ python
Python 2.7.2+ (default, Oct  4 2011, 20:06:09)
[GCC 4.6.1] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'Hello, world!'
Hello, world!
>>>
```

You can exit the interpreter by hitting your platform's `<end of input>` key combination. On Windows this is `<ctrl-z><enter>`. On Linux or Mac OSX it is `<ctrl-d>`.

Hello World!

Exercise

- Exit the interpreter
  - we are going to make a Python script print out our traditional greeting
- *cd* to the *exercises* directory
- Edit the file *helloworld.py*
  - alter the script to print 'Hello, world!'
  - the print statement is the same as we entered into the interpreter after the `>>>` prompt
- Run the script from the command line:

```
$ ./helloworld.py
$ # or
$ python helloworld.py
```

Basics

You will want to start a new python interpreter session and follow along with these examples. If something doesn't work, or confuses you, ask.

```
$ python
```

- variables, assignment, print

```
>>> count = 5.0
>>> print 'count', count
count 5.0
>>> count = count + 3.0
>>> print 'new count',count
new count 8.0
```

- arithmetic

```
>>> count = 5.0
>>> count + 3
8.0
>>> count * 3
15.0
>>> count / 3
1.6666666666666667
>>> count ** 2 # exponentiation
25.0
>>> count ** .5
2.23606797749979
>>> count % 3 # remainder
2.0
>>> count // 3 # quotient
1.0
>>> count * 3 + 2
17.0
>>> count * (3 + 2)
25.0
```

- comparisons between numbers

```
>>> 2 > 3
False
>>> 0 < 4
True
>>> 3 >= 3
True
>>> 4 <= 6
True
>>> 8 == 8
True
>>> 8 == 9
False
```

- comparisons between strings

```
>>> "a" > "b"
False
>>> "a" < "b"
True
>>> "a" < "A"
False
>>> "z" < "Z"
False
>>> "this" > "th" # having "something more" means you are > ('i' is compared to '')
True
>>> "this" > "tho" # the first difference determines the result ('i' is compared to 'o
↪')
False
```

Note: Why is “a” > “A”?

Your computer represents the two characters with different numbers internally. Those numbers happen to be arranged such that “a” (97) is greater than “A” (65).

- variables point to values (objects), *not* to other variables

```
>>> first = 1
>>> second = 2
>>> second = first
>>> second
1
>>> first = 3
>>> first
3
>>> second
1
```

- basic types

```
>>> count = 36
>>> print count
36
>>> count / 10 # surprising?
3
>>> irrational = 3.141592653589793
>>> irrational
3.141592653589793
>>> label = "irrational, 'eh"
>>> label2 = 'count "this"'
>>> label3 = '''python has these too\n'''
>>> label4 = """but they are just a different way to write the same thing"""
>>> print label, label2, label3, label4
irrational, 'eh count "this" python has these too
but they are just a different way to write the same thing
>>> print label + label2
irrational, 'ehcount "this"
>>> None # doesn't show up
>>> print None
None
>>> print True
True
>>> print False
False
>>> True == 1
True
>>> False == 1
False
>>> False == 0
True
```

- what *type* of object is something?

```
>>> type( 0 )
<type 'int'>
>>> type( 1 )
<type 'int'>
>>> type( 1.0 )
<type 'float'>
>>> type( [] )
<type 'list'>
>>> type( False )
<type 'bool'>
>>> type( 'blue' )
<type 'str'>
```

Note: What do those *()* characters mean in *type(0)*?

We are asking a "thing" ("object") called *type* to "act" upon a single thing, which is our integer value *0*. The thing *type* has a piece of code (a "function" or "method") that tells it what to do when it is "asked to act" ("called") on a set of things ("arguments" or "parameters"). Here the set of arguments we are passing is a single value, but later on we will see how to pass multiple arguments into functions which support multiple arguments.

We'll see how to write our own functions later in this tutorial. A "method" is a function which is "attached" to an object, we'll use these throughout the tutorial, but this tutorial does not yet cover how to write our own objects.

- type conversions, each *type* normally can be "called" to create a new value of that *type*

```
>>> string = '32'
>>> string
'32'
>>> int(string)
32
>>> float(string)
32.0
>>> str( int( string ))
'32'
>>> str( float( string ))
'32.0'
```

```
>>> string = '32.6'
>>> int(string)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '32.6'
>>> float( string )
32.6
>>> int( float (string ))
32
>>> int( round( float( string ), 0 ))
33
>>> round( 32.6, 0 )
33.0
>>> round( 32.6, 1 )
32.6
```

Exercise

In the interpreter, multiply the strings '10' and '20' to get the integer result 200:

```
>>> first = '10'
>>> second = '20'
...
>>> print first * second # should print 200
```

Lists

- lists are "collections of things" which have a particular order

```
>>> integers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> integers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> integers.append( 11 )
>>> integers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11]
>>> integers.insert( 0, 12 )
>>> integers
[12, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11]
>>> len(integers)
12
>>> sorted(integers)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12]
>>> integers # why didn't integers change?
[12, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11]
>>> integers.sort()
>>> integers # the .sort() method did an "in place" sort
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12]
>>> integers.append( 'apple' )
>>> integers
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 'apple']
```

Note: What did *integers.append(11)* mean?

Here the object *integers* had a piece of code attached to it (we call these "methods") which was written to "add an object to the end of THIS list". By writing *integers.append* we "looked up" this piece of code in the *integers* list. When we added the *(11)* to the statement we asked that piece of code (*append*) to act on a single integer object *11*.

Note: Python's interactive interpreter has a *help* function that allows you to get documentation on a particular *type* of object, such as *list*. The help text normally includes all of the "methods" that are available, a description of the parameters for each method, and normally a "docstring" (human description) for the method explaining what it does, and occasionally how it does it.

```
>>> help( list ) # type <q> to exit the help
```

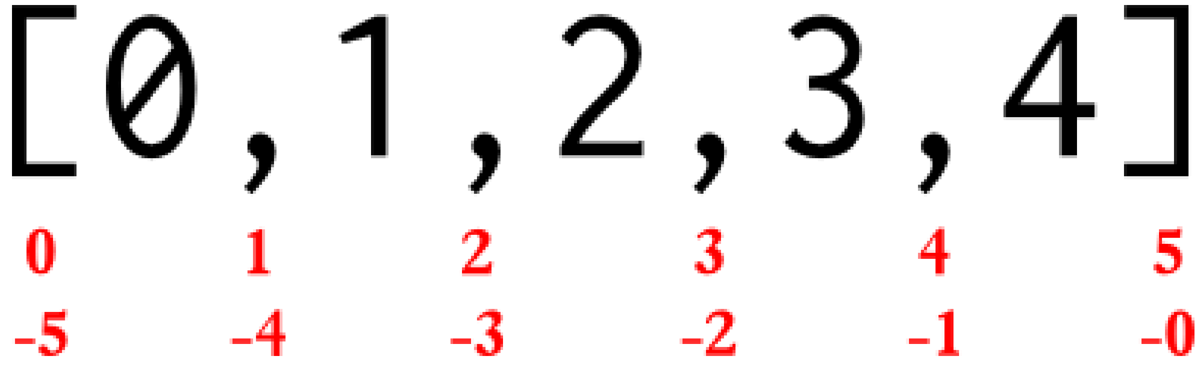
Exercise

- edit the file `exercises/basicexercise.py`
  - create, modify and display some variables
- run the file with `python basicexercise.py` from the `exercises` directory or `./basicexercise.py` if you prefer.

```
#! /usr/bin/env python
# basicexercise.py

# Create 4 variables pointing to each of the following:
#  An integer (int)
#  A (positive) floating-point number (float)
#  A list
#  A string (str)

# Print the square of the integer

# Print the square root of the float

# Append the float to the list

# Insert the string in the list at index 0

# Insert the integer in the list at index 0

# Print the list
```

List Indexing

- indexing



  - `alist[i]` looks up the index in the above scheme and gets the next item
  - `alist[-i]` looks up the index in the second line and gets the next item

```
>>> counts = [0,1,2,3,4]
>>> counts[0]
0
>>> counts[1]
1
>>> counts[2]
2
>>> counts[-1]
4
>>> counts[-2]
3
>>> counts[-4]
1
>>> counts[-5]
0
>>> counts[-6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>>
```

- list slicing
  - `alist[i:j]` looks up the index `i`, then includes all items until it reaches the index `j`

- you can leave off the index for start/end
    - `alist[:j]` retrieves all items from start (index 0) until we reach `j`, this is, conveniently, the first `j` items
    - `alist[i:]` starts at index `i` and retrieves all items until we reach the end, this "skips" the first `i` items

```
>>> counts = [1, 2, 3, 4, 5]
>>> counts
[1, 2, 3, 4, 5]
>>> counts[1:]
[2, 3, 4, 5]
>>> counts[:-1]
[1, 2, 3, 4]
>>> counts[1:-1]
[2, 3, 4]
>>> counts[99:]
[]
>>> counts[:-99]
[]
>>> counts[3:8]
[4, 5]
```

Note: Bonus Material

You can also specify a "step" in your slices:

```
>>> counts[::2] # every other item, starting at 0
[1, 3, 5]
>>> counts[1::2] # every other item, starting at 1
[2, 4]
>>> counts[::-1] # the whole list, stepping backward
[5, 4, 3, 2, 1]
>>> counts[-1:1:-1] # start at index -1, step backward while index is > 1
[5, 4, 3]
```

- convenience function for creating ranges of integers

```
>>> range( 5 )
[0, 1, 2, 3, 4]
>>> range( 2, 5 )
[2, 3, 4]
```

Exercise

- slice and dice a list

```
#! /usr/bin/env python
# basicsliceexercise.py

# we create a list of integers...
integers = range( 0, 20 )

# Print the first item of the list
# Print the last item of the list
# Print the first 5 items of the list (a slice)
# Print the last 5 items of the list (a slice)

# Print 5 items starting from index 5
```

Boolean Logic

Reduces down to the statement: "if this is True, do that, otherwise do that". Computers, being binary (on/off) machines work very easily with on/off choices such as boolean logic.

- almost any object can be tested for "boolean truth"

```
>>> bool( 0 )
False
>>> bool( 1 )
True
>>> bool( [] )
False
>>> bool( ['this'] )
True
>>> bool( 0.0 )
False
>>> bool( 1.0 )
True
>>> bool( '' )
False
>>> bool( 'this' )
True
>>> bool( None )
False
```

- if, elif, else
  - only do a given "suite" of commands if the "check" matches
  - else is for when no other check matches (and is optional)

```
>>> x = 32
>>> if x < 5:
...     print 'hello'
... elif (x+4 > 33):
...     print 'hello world'
... else:
...     print 'world'
...
hello world
```

Note: Technical Tidbit

Your computer is formed of tiny electrical switches where a current in one "wire" can prevent or allow a current from flowing in another "wire". Below all the levels of abstraction, when the computer decides "if this is True" it is checking whether a value can flow through the second "wire".

- comparisons are boolean operators

  - == (are they equal) vs = (assign value)
  - >=, <=, != (not equal)
- logical combinations allow you to string together boolean tests
  - `and`, `or`, `not`

```
>>> x = 23
>>> y = 42
>>> (x == y) or (x * 2 > y )
True
>>> (x == y) or (x > y)
False
>>> (x < y) and (y > 30)
True
>>> (x == y) or (not x > y)
True
```

Loops

- *while* something is True, keep doing "this set of things"

```
>>> x = 10
>>> while x > 0:
...     print x
...     x = x - 1
...
10
9
8
7
6
5
4
3
2
1
```

```
>>> counts = [1, 2, 3, 4, 5]
>>> i = 0
>>> while i < len(counts):
...     count = counts[i]
...     print count
...     i = i + 1
...
1
2
3
4
5
```

- loops using *for x in y* are syntactic "sugar" for that last while loop, this pattern is referred to as "iterating over" an object, and is extremely common

```
>>> counts = [1, 2, 3, 4, 5]
>>> for count in counts:
```

```
...     print count
...
1
2
3
4
5
```

- "suites" of commands, *python* is not normal here (most languages use *{}* braces or pairs of words, such as *do* and *done*)

```
for var in a,b,c,d
do
    echo "Variable is ${var}"
    ls ${var}
done
```

```
#! /usr/bin/env python
# iterforxiny.py

measurements = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

print 'Squares'
total = 0
for item in measurements:
    print item, item ** 2
    total += item **2
print 'Sum of Squares:', total
```

- the suites can "nest" with further for-loops (or other structures)

```
#! /usr/bin/env python
# iternest.py

rows = [
    ['Lightseagreen Mole', 24.0, -0.906, 0.424, -2.13, 0.0, 'green'],
    ['Indigo Stork', 51.0, 0.67, 0.742, 0.9, 9.0, 'yellow'],
]

for i,row in enumerate( rows ):
    print 'rows[{}]'.format( i )
    for measurement in row[1:-1]:
        print '  {}'.format( measurement )
```

Note: The `enumerate` function we use in the above sample can be thought of as doing this:

```
result = []
for i in range( len( rows )):
    result.append( (i,rows[i]))
return result
```

but is actually implemented in a more efficient manner.

```
#! /usr/bin/env python
# iterfilter.py

measurements = range( 30 )

print 'Odd Triple Squares'
total = 0
rest = 0
for item in measurements:
    if item == 25:
        print '25 is cool, but not an odd triple'
    elif item % 2 and not item % 3:
        print item, item ** 2
        total += item **2
print 'Sum of Odd Triple Squares:', total
```

Exercise

- construct lists by iterating over other lists
- use conditions to only process certain items in a list
- use conditions and a variable to track partial results

```
#! /usr/bin/env python
# iterexercise.py

rows = [
    ['Lightseagreen Mole', 24.0, -0.906, 0.424, -2.13, 0.0, 'green'],
    ['Springgreen Groundhog', 77.0, 1.0, -0.031, -32.27, 25.0, 'red'],
    ['Blue Marten', 100.0, -0.506, 0.862, -0.59, 16.0, 'yellow'],
    ['Red Bobcat', 94.0, -0.245, 0.969, -0.25, 36.0, 'green'],
    ['Ghostwhite Falcon', 31.0, -0.404, 0.915, -0.44, 49.0, 'green'],
    ['Indigo Stork', 51.0, 0.67, 0.742, 0.9, 9.0, 'yellow'],
]


# Create 2 lists holding the first two columns of *numeric* data
# (second and third columns)

# Print those items in the second column which are greater than 20 and less than 90

# Print the largest value in the third column
```

String Manipulation

- strip (remove whitespace or other characters)

```
>>> value = '  25.3  '
>>> value
'  25.3  '
>>> value.strip()
'25.3'
>>> quoted = '"this"'
>>> quoted
```

```
'"this"'
>>> quoted.strip('"')
'this'
```

- split, join

```
>>> row = 'Silver Deer,69,-0.115,0.993,-0.12,25,violet'
>>> components = row.split( ',' )
>>> components
['Silver Deer', '69', '-0.115', '0.993', '-0.12', '25', 'violet']
>>> print "\n".join( components )
Silver Deer
69
-0.115
0.993
-0.12
25
violet
>>> not_all_strings = [ 'Silver Goat', 45, -.333, .75, .08, 5, 'violet' ]
>>> "\n".join( not_all_strings )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 1: expected string, int found
```

- format

```
>>> count = 53
>>> mean = 37.036
>>> label = 'DMX Score'
>>> '{0},{1},{2}'.format( label, count, mean )
'DMX Score,53,37.036'
>>> '{0!r} for {1} items {2:0.2f}'.format( label, count, mean )
"'DMX Score' for 53 items 37.04"
```

Dictionaries

- a.k.a. hash-tables in other languages, have special syntax in most scripting languages
  - keys must be immutable (technically, hashable)
  - values (anything)

```
#! /usr/bin/env python
# dictdefinitions.py

dictionary = {}
dictionary2 = {
    'thar': 'thusly',
    'them': 'tharly',
}
dictionary3 = {2:3, 4:None, 5:18}
```

- you can add, remove, reassign

```
>>> dictionary = {}
>>> dictionary
{}
```

```
>>> dictionary['this'] = 'those'
>>> dictionary
{'this': 'those'}
>>> dictionary['those'] = 23
>>> dictionary
{'this': 'those', 'those': 23}
>>> len(dictionary)
2
>>> dictionary['this'] == 'those'
True
>>> del dictionary['those']
>>> dictionary
{'this': 'those'}
>>> 'those' in dictionary
False
>>> 'this' in dictionary
True
>>> dictionary['those']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'those'
```

- only one entry for each equal-hash-and-compare-equal key
  - you can thus use a dictionary to confirm/create uniqueness
  - values *must* compare equal *and* have the same "hash", this is "computer equal", not "human equal", though Python tries to make "computer equal" a bit more human e.g. with floats/ints

```
>>> dictionary = {'this':'that'}
>>> dictionary[ ' this ' ] = 'thar'
>>> dictionary
{'this': 'that', ' this ': 'thar'}
>>> dictionary[ 45 ] = 8
>>> dictionary[ 45.0 ] = 9
>>> dictionary
{'this': 'that', ' this ': 'thar', 45: 9}
>>> # Super Bonus Ask During Coffee Question: why is the key 45 and not 45.0?
```

- iterable, but un-ordered, so don't depend on the order of items

```
#! /usr/bin/env python
# dictiteration.py

dictionary = {'this':'that','those':'thar',23:18,None:5}

print 'items',dictionary.items()
print 'values',dictionary.values()
print 'keys', dictionary.keys()

for key in dictionary:
    print '{0!r} : {1}'.format( key, dictionary[key] )
```

Exercise

- loop over a list of strings, split into key: value pairs and add to dictionary

```
#! /usr/bin/env python
# dictexercise.py

rows = [
    'Dodgerblue Lemming ,30,-0.988,0.154,-6.41,36,yellow',
    ' Orangered Myotis,88,0.035,0.999,0.04,0,blue',
    '  Aquamarine Falcon,68,-0.898,0.44,-2.04,16, indigo',
    'Lightsalmon Prairie-Dog,20,0.913,0.408,2.24,16,violet ',
    'Magenta Pigeon,25,-0.132,0.991,-0.13,1, blue',
    'Peru Eagle,25,-0.132,0.991,-0.13,1, blue',
    'Peru Eagle  ,25,-0.132,0.991,-0.13,1,red ',
]
# create a dictionary that maps column 1 (name) to the last column (colour)
# strip the name of any extra whitespace, same with the colour
# for each item in the dictionary, print the name and colour
# what colour will be shown for "Peru Eagle"?
```

Reading a File

- look at `../sample_data.csv`, note how it looks like the data in the previous exercise

```
Subject,Count,DMX Score,Coda Score,Vinny Score,Zim Score,Subject Choice
Dodgerblue Lemming,30,-0.988,0.154,-6.41,36,yellow
Orangered Myotis,88,0.035,0.999,0.04,0,blue
Aquamarine Falcon,68,-0.898,0.44,-2.04,16,indigo
Lightsalmon Prairie-Dog,20,0.913,0.408,2.24,16,violet
Magenta Pigeon,25,-0.132,0.991,-0.13,1,blue
Peru Eagle,25,-0.132,0.991,-0.13,1,blue
Mintcream Caribou,10,-0.544,-0.839,0.65,4,green
Silver Deer,69,-0.115,0.993,-0.12,25,violet
Darkslateblue Ibis,34,0.529,-0.849,-0.62,4,violet
Olive Goshawk,74,-0.985,0.172,-5.74,4,blue
Lightcoral Seal,47,0.124,-0.992,-0.12,49,indigo
Red Vulture,37,-0.644,0.765,-0.84,25,yellow
Palegoldenrod Brown-Bear,11,-1.0,0.004,-225.95,9,blue
Firebrick Coyote,51,0.67,0.742,0.9,9,yellow
Thistle Bustard,84,0.733,-0.68,-1.08,16,red
Whitesmoke Lynx,57,0.436,0.9,0.48,1,orange
Beige Wolverine,90,0.894,-0.448,-2.0,4,violet
Darkorchid Grebe,85,-0.176,-0.984,0.18,25,orange
Ivory Wolf,18,-0.751,0.66,-1.14,4,blue
Fuchsia Moose,62,-0.739,0.674,-1.1,36,violet
```

- this is a standard comma separated value data-file, possibly from some survey which observed animals and subjected them to various (humane) tests which generated measurements. Let's poke around in it:

```
>>> reader = open( '../sample_data.csv', 'r') # r is for "read" mode
>>> reader
<open file '../sample_data.csv', mode 'r' at 0x...>
>>> content = reader.read()
>>> len(content)
995
>>> reader.close()
>>> lines = content.splitlines()
>>> len(lines)
21
```

```
>>> lines[0]
'Subject,Count,DMX Score,Coda Score,Vinny Score,Zim Score,Subject Choice'
>>> lines[1]
'Dodgerblue Lemming,30,-0.988,0.154,-6.41,36,yellow'
>>> lemming = lines[1]
>>> columns = lemming.split(',')
>>> columns
['Dodgerblue Lemming', '30', '-0.988', '0.154', '-6.41', '36', 'yellow']
>>> measurement = columns[2]
>>> measurement
'-0.988'
>>> type(measurement)
<type 'str'>
>>> measurement = float( measurement )
>>> measurement
-0.988
>>> type(measurement)
<type 'float'>
```

- the previous loaded the whole file into memory at one go, we could also have iterated over the file line-by-line.

```
>>> reader = open( '../sample_data.csv', 'r')
>>> header = reader.readline()
>>> header # note the '\n' character, you often need to do a .strip()!
'Subject,Count,DMX Score,Coda Score,Vinny Score,Zim Score,Subject Choice\n'
>>> for line in reader:
...     print float(line.split(',')[2])
...
-0.988
0.035
...
```

- the special file `sys.stdin` can be used to process input which is being piped into your program at the `bash` prompt (we'll see two more special pipes in *Writing (Structured) Files* below.

```
#! /usr/bin/env python
# argumentsstdin.py
import sys
header = sys.stdin.readline()
for line in sys.stdin:
    print float(line.split(',')[2])
```

```
$ cat ../sample_data.csv | ./argumentsstdin.py
-0.988
0.035
-0.898
0.913
...
```

Note: file objects keep an internal "pointer" (offset, bookmark) which they advance as you iterate through the file. Regular files on the file-system can be "rewound" or positioned explicitly. File-like objects such as pipes often cannot provide this functionality.

Exercise

- read data from `../sample_data.csv` so that you have a list of strings, one string for each line in the file
- use code from `dictexercise.py` to again map names to colours and print the colour of a "Firebrick Coyote"
- use code from `iterexercise.py` to turn data into columns and find out which animal was spotted most frequently

```
#! /usr/bin/env python
# filereadexercise.py

# read data from ../sample_data.csv

# use code from dictexercise.py to again map names to colours
# what colour is a "Firebrick Coyote"?

# for each column in the data file make a list containing
# that columns data
# which animal was seen most frequently?
# Hint: refer back to iterexercise.py and dictexercise.py
# Hint: watch out for the header row
# Hint: help(list.index)
```

Simple Functions

- previous exercise introduced code reuse
- simple functions, one returned value

```
#! /usr/bin/env python
# functionsimple.py

def double( value ):
    """returns the value multiplied by two"""
    return value * 2

def larger( first, second):
    """Return the larger of first and second"""
    if first >= second:
        return first
    else:
        return second

print 'double of the larger of 3 and 4:',double( larger( 3,4 ))
# print value # doesn't work
```

- variable scope

Exercise

- copy your code from `filereadexercise.py` into `readdata.py` and turn the code that reads data from a file into a function that returns a list of strings
- have the function take a file name as an argument

```
#! /usr/bin/env python
# readdata.py

# use your code from filereadexercise.py here. turn the code that reads
# from a file into a function that returns a list of strings from the
# file. make the function take a filename as an argument.
# Hint: make sure to call your function so the rest of your code works!
```

Functions as Building Blocks

- grouping code in small, logical chunks helps you reuse it
- docstrings

```
#! /usr/bin/env python
# functionreuse.py

def pretty_print_add(x, y):
    """
    nicely print the addition of two things
    """
    template = '{0} + {1} = {2}'
    print template.format(x, y, x + y)


pretty_print_add(8, 9)

pretty_print_add(4.5, 5.6)

pretty_print_add((1,2), (3,4))

pretty_print_add([5,6], [7,8])
```

Exercise

- in `readdata.py` group the code that makes a dictionary from the data into a function that returns the dictionary
- also put the code that makes lists into a function that returns several lists
- have both of these functions take a file name as an argument and call the file reading function you've already written
- call these functions and use the data they return to make the rest of the code work

Modules

- using code from other files, modules and importing
- put all code into functions
- __name__ == '__main__'

```python
#! /usr/bin/env python
# moduledemo1.py

from functionreuse import pretty_print_add

pretty_print_add(145, 396)
```

```python
#! /usr/bin/env python
# moduledemo2.py

from pretty_print import pretty_print_add

pretty_print_add(145, 396)
```

```python
#! /usr/bin/env python
# pretty_print.py

def pretty_print_add(x, y):
    """
    nicely print the addition of two things
    """
    template = '{0} + {1} = {2}'
    print template.format(x, y, x + y)

if __name__ == '__main__':
    pretty_print_add(8, 9)

    pretty_print_add(4.5, 5.6)

    pretty_print_add((1,2), (3,4))

    pretty_print_add([5,6], [7,8])
```

Exercise

- write a function that finds the mean of a list of numbers and use it to find the mean of each of the score columns in `sample_data.csv`
- use your functions in `readdata.py` by importing them
  - you will need to modify `readdata.py` so that it doesn't print anything when you import it

```python
#! /usr/bin/env python
# moduleexercise.py

# import relevant function from ``readdata.py``. make sure nothing is
# printed to the screen when you do this.

# write a function that calculates the mean of a list of numbers
# Hint: help(sum) and help(len)

# find the mean of each of the score columns in sample_data.csv
# and print them
# put this code in a function too
```

```
if __name__ == '__main__':
    # call just one function here so that the means are printed
```

Arguments and Return Codes

- as you will recall from the `bash` session, programs have return codes which invoking programs will check to see whether the program succeeded
- main function and the "entry point" for scripts
  - scripting languages execute their code line-by-line, so they don't have a `void main() {}` entry point as in `C`
  - putting the main actions inside a function doesn't seem that useful until you discover that most Python packaging tools can generate wrapper scripts that invoke a particular function (such as main, here)

```
#! /usr/bin/env python
# argumentsmain.py

import sys

def main():
    """Primary entry point for the script/module"""
    return 1

# A python-only idiom meaning "only execute this if we are the top-level script"
# i.e. do *not* run this if we are being imported as a module
if __name__ == "__main__":
    sys.exit( main())
```

- command line arguments, sys.argv

```
#! /usr/bin/env python
# argumentsargv.py

import sys

def print_files( files ):
    """A function another module might want to invoke"""
    for file in files:
        print file

def main():
    """Primary entry point for the script/module"""
    if sys.argv[1:]:
        print_files( sys.argv[1:] )
        return 0
    else:
        sys.stderr.write( "You need to provide file[s]\n" )
        return 1

if __name__ == "__main__":
    sys.exit( main())
```

- most real-world applications *also* want optional parameters, for those see the OptParse (for Python 2.6 and below) or ArgParse (for Python 2.7 and above) modules

Exercise

- modify your `moduleexercise.py` script take the file to process from the (bash) command line

Writing (Structured) Files

- while using `print` is fine when you are directly communicating with a user, you will often want to output data in a structured format for future processing
- files can be opened in "write" mode by passing `'w'` as the `mode` parameter
- the standard module `sys` has two pipe handles already opened for output, these are similar to the pipe handle `sys.stdin` we saw in *Reading a File*.
  - stdout – where most client programs expect your primary output
  - stderr – where most client programs expect error messages, warnings etc.

```
#! /usr/bin/env python
# outputbasic.py
import sys

rows = [
    ['Lightseagreen Mole', 24.0, -0.906, 0.424, -2.13, 0.0, 'green'],
    ['Indigo Stork', 51.0, 0.67, 0.742, 0.9, 9.0, 'yellow'],
]

def format_row( row ):
    result = []
    for item in row:
        result.append( str(item))
    return ",".join( result )

def write_rows( rows, writer ):
    for row in rows:
        writer.write( format_row( row ))
        writer.write( '\n' )

def write_file( rows, filename='' ):
    if not filename:
        write_rows( rows, sys.stdout )
    else:
        writer = open( filename,'w')
        write_rows( rows, writer )
        writer.close()

if __name__ == "__main__":
    if sys.argv[1:]:
        write_file( rows, sys.argv[1] )
    else:
        write_file( rows )
```

Exercise

- modify your `moduleexercise.py` script to write the summary information for each (numeric) column processed into a CSV file where each row is the original column label (the first row in the file) and the mean value for that row

Exceptions and Tracebacks

- so far we've ignored situations where errors occurred, but real software needs to handle errors or unexpected conditions all the time

```
>>> value = ' Aquamarine Falcon '
>>> float( value )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: could not convert string to float:  Aquamarine Falcon
```

- when functions call other functions, the system creates a "stack" of "frames", an uncaught error will, by default, print out a "traceback" of these frames
  - when something goes wrong, you use the traceback to help you find out where and what the problem was
  - in *python* the traceback is ordered from "top" to "bottom", that is, the "frame" printed first in the traceback ("<stdin>" in the example below) is the "top level" caller
  - each frame is a function which was running (not yet complete) when the uncaught error was encountered
  - in *python*, the last line of the traceback is a string representation of the `Exception` which was raised, which generally attempts to be a useful description of what went wrong

```
>>> from functionarguments import *
>>> rows = split_rows( open('../sample_data.csv').read().splitlines()[1:] )
>>> first,second = extract_columns( rows, 1, -2 )
>>> first,second = extract_columns( rows, 30 )
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "functionarguments.py", line 15, in extract_columns
    result.append( extract_column( rows, column ))
  File "functionarguments.py", line 8, in extract_column
    result.append( row[column] )
IndexError: list index out of range
```

- it is possible to catch these `Exceptions` in Python by using a special type of block around the code in which the exception may occur

```
>>> value = '  Aquamarine Falcon '
>>> try:
...     value = float( value )
... except ValueError, err:
...     value = value.strip()
...
>>> value
'Aquamarine Falcon'
```

Note: We can catch multiple Exception types using `except (ValueError,TypeError), err` instead.

Note: The syntax for catching exceptions changes between Python 2.x and 3.x, in Python 3.x the syntax becomes `except ValueError, TypeError as err`

Exercise

- does your script fail if you point it at `../bad_sample_data.csv`?
  - if not, congratulations; you pass
  - if so, what does the traceback tell you?
- (if necessary) modify your `moduleexercise.py` so that it can parse `../bad_sample_data.csv` as well as any file in the `../real_data/` directory
  - catch the case where the first column is a quoted, comma-separated name, convert the name to `first last` rather than `last, first`
  - assume that missing (numeric) values should be set equal to 0.0
  - assume that comments (lines starting with '#') and blank lines should be ignored

Bonus Exercise

- modify your script to load *multiple* files passed from the command line
- check for duplicate subject names

Using Existing Libraries

- Generally speaking, you should prefer to use pre-written modules to handle common tasks. The Python standard library and the thousands of Python packages and extensions mean that you normally would *not* write this type of low-level code yourself.

```
#! /usr/bin/env python
# reusecsv.py
import csv
lines = list(csv.reader(open('../sample_data.csv')))[1:]
```

Bonus Exercise

- rewrite your code to use the python standard csv library to parse the CSV data
- use the built-in min, max and sum functions to calculate summary information on your columns, rather than using your custom-written functions

Numpy

- numpy is a powerful package for use in scientific compuation with Python
- you can readily rewrite many of our samples (and far more involved processes) just by combining the tools Numpy already provides

```
#! /usr/bin/env python
# reusenumpy.py
from functionarguments import *
import csv, numpy
rows = list(csv.reader(open('../sample_data.csv')))[1:]

column = extract_column( rows,1 )
column = as_type(column,float)
print 'Max of column [1]',numpy.max( column )
print 'Mean of column [1]',numpy.mean( column )
print 'Median of column [1]',numpy.median( column )
print 'Standard deviation of column [1]',numpy.std( column )
```

Bonus Exercise

- using `numpy`, load the `sample_data.csv` data-set and play with the columns of data to determine what relationship the columns have to one another

Useful Python Links

The *Python Tutor Mailing List<http://mail.python.org/mailman/listinfo/tutor>* provides a friendly environment for asking questions while getting started with Python.

Core Python

- Main Python Docs
  - http://docs.python.org/
- Global Module Index
  - Built-in modules like os, sys, datetime, etc.
  - http://docs.python.org/modindex.html
- Built-in Functions
  - Built-in, always available functions like open, enumerate, zip, range, etc.
  - http://docs.python.org/library/functions.html
- String Formatting
  - The lowdown on string formatting
  - http://docs.python.org/library/string.html#formatstrings

Python in Science

- Numpy
  - Fast arrays
- SciPy
  - Minimization, fitting, solvers, statistics, and more
- matplotlib
  - 2D and 3D plotting, maps
- Astropy for Astronomy
- Biopython for Bioinformatics

- Sage for mathematical analysis

Your Instructors

Matthew Davis

- Works at the Space Telescope Science Institute
- Has used Python in science settings for 4 years
- Contributes to Astropy
- @jiffyclub
- Blog

Mike Fletcher

- Is a Design Epistemologist (Philosopher of Design)
- Has been programming since 1981
- Has been programming Python since 1995
- Python Software Foundation member
- Open Source software developer, consultant
- VRPlumber Blog

Still to Cover

- How do you solve a problem?
  - How do you figure out what is available?
  - How do you determine if something is working?
  - How do you figure out why it isn't working?
  - Where is the documentation?
  - Where do you ask questions (and how)?
  - What are the common patterns you'll see time and again?
- paths and processing large numbers of files
  - path-name composition (os.path.join, etc)
  - `for (path, directories, files) in os.walk`
- modules
  - sys, os, subprocess (often required for any real work)
  - glob, shutil (getting things done at the OS level)
  - requests (web client operations)

- testing/verification?
  - asserting your assumptions (prelude to TDD)
  - asserting pre/post conditions
  - throwing errors vs silent corrections
  - handling corner cases

Notes for Instructors

To regenerate the session site and downloads:

```
aptitude install python-sphinx
./fakedata.py
# TODO: this wasn't really supposed to get checked in...
bzr revert sample_data.csv
make html # to just build the HTML site
make download # to build the HTML site and create a download
```

To serve it on the day of the lesson:

```
$ aptitude install nginx
```

Then use this nginx config-file:

```
server {
        #listen   80; ## listen for ipv4; this line is default and implied
        #listen   [::]:80 default ipv6only=on; ## listen for ipv6

        root /usr/share/nginx/2daypython;
        index index.html index.htm;

        # Make site accessible from http://localhost/
        server_name localhost sturm.local;

        location / {
                # First attempt to serve request as file, then
                # as directory, then fall back to index.html
                try_files $uri $uri/ /index.html;
        }

}
```

And link `/usr/share/nginx/2daypython` to your _build/html directory:

```
$ sudo ln -s /home/mcfletch/2day-dev/_build/html /usr/share/nginx/2daypython
```