Introduction to Python

An introduction to Python for just about anyone

- *(with at least a little programming experience)*

(http://continuum.io)

Taught by **Ian Stokes-Rees** ijstokes@continuum.io (mailto:ijstokes@continuum.io)

- Twitter: @ijstokes (http://twitter.com/ijstokes)
- About.Me: http://about.me/ijstokes (http://about.me/ijstokes)
- LinkedIn: http://linkedin.com/in/ijstokes (http://linkedin.com/in/ijstokes)

Workshop Website: http://j.mp/python-intro-2h (http://j.mp/python-intro-2h)

Setup

1. Grab some food or drink (we may have lots of people, so don't be greedy!)
2. Download and install the Anaconda Python Distribution (http://continuum.io/downloads) from Continuum Analytics
    - It is about 230 MB, but contains a lot of really useful tools and extra libraries beyond the Python Standard Library
    - In a pinch, Linux and Mac users: you already have Python on your machine; Windows users, you can get the Official Python Distribution from python.org (http://python.org) which is only 30 MB
    - Failing that, I have some USB keys ...
3. Introduce yourself to your neighbour
    - programming is more fun with friends!
    - probably too many people for me to help you or answer questions during workshop: peer-to-peer learning! collaborative problem solving!
4. Download the workshop material:
    - GitHub
        - repository ()
        - zip file ()
    - Wakari
        - web view ()
        - zip file ()
5. Unzip the material somewhere sensible (e.g. your home directory)
    - Windows:

```
c:\> cd %HOMEPATH%
c:ijstokes> unzip %HOMEPATH%\Downloads\python-intro-2h.zip
```

    - Mac:

```
$ cd ~
$ unzip ~/Downloads/python-intro-2h.zip
```

    - *nix: you're on your own
6. Check your Python version and fire up your interactive interpreter:

```
$ cd ~/python-intro-2h
$ python -V
$ python
Python 2.7.8 |Continuum| (default, Aug 21 2014, 15:21:46)
[GCC 4.2.1 (Apple Inc. build 5577)] on darwin
>>>
```

Types and Expressions

```
In [85]: # basic math
         3 + 7
```

```
Out[85]: 10
```

In [86]:
```
15 * 4
```

Out[86]: 60

In [90]:
```
15.5 / 2.5
```

Out[90]: 6.2

In [92]:
```
2**10
```

Out[92]: 1024

In [94]:
```
(3 + 7) * 2.2
```

Out[94]: 22.0

In [95]:
```
7 / 2 # integer math!
```

Out[95]: 3

In [96]:
```
# expressions
'hello' + 'everyone'
```

Out[96]: 'helloeveryone'

In [97]:
```
'python love' + 'python love' + 'python love'
```

Out[97]: 'python lovepython lovepython love'

In [98]:
```
'python love' * 3
```

Out[98]: 'python lovepython lovepython love'

In [99]:
```
# references (you may know them as "variables")
a = 3
```

In [100]:
```
a
```

Out[100]: 3

In [101]:
```
type(a)
```

Out[101]: int

In [102]:
```
a + 7
```

Out[102]: 10

In [103]:
```
day = 'Tuesday'
```

In [104]:
```
day
```

Out[104]: 'Tuesday'

In [105]:
```
type(day)
```

Out[105]: str

In [106]:
```
day + ' is a great day to learn Python'
```

Out[106]:
```
'Tuesday is a great day to learn Python'
```

In [107]:
```
temp = 45.7
```

In [108]:
```
temp
```

Out[108]:
```
45.7
```

In [109]:
```
type(temp)
```

Out[109]:
```
float
```

In [110]:
```
temp - 15.0 # maybe tomorrow!
```

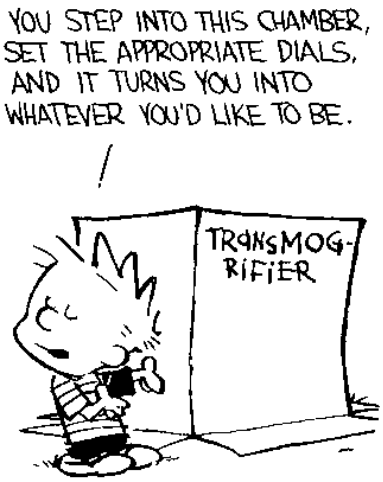
Out[110]:
```
30.700000000000003
```

In []:
```
# types: int, float, str
```

You should be doing your best to follow along in your own interactive interpreter session

In []:
```
# quick review from vanilla Python interpreter
# dir() (at start and end)
```

Some other languages provide *transmogrifcation* magic:



But not Python:

- dynamically typed, but ...
- strongly typed

In [111]:
```
a = '3'
```

In [112]:
```
type(a)
```

Out[112]:
```
str
```

In [114]:
```
# comment on exception
a + 7
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-114-cd3dfc25b26e> in <module>()
----> 1 a + 7

TypeError: cannot concatenate 'str' and 'int' objects
```

In [ ]:
```
# make a prediction! what will happen
a * 7
```

In [115]:
```
age = 39
```

In [116]:
```
'I am ' + age + ' years old'
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-116-3fd6b688018d> in <module>()
----> 1 'I am ' + age + ' years old'

TypeError: cannot concatenate 'str' and 'int' objects
```

In [ ]:
```
# type conversion
```

In [117]:
```
v = int(a)
```

In [118]:
```
v
```

Out[118]: 3

In [120]:
```
type(v)
```

Out[120]: int

In [121]:
```
v + 7
```

Out[121]: 10

In [122]:
```
v * 7
```

Out[122]: 21

In [123]:
```
'I am ' + str(age) + ' years old'
```

Out[123]: 'I am 39 years old'

In [124]:
```
sentence = 'I am ' + str(age) + ' years old'
```

In [125]:
```
sentence
```

Out[125]: 'I am 39 years old'

In [126]:
```
result = v * 7
```

In [127]:
```
result
```

Out[127]: 21

References and Objects

All **things** in Python are objects:

- numbers
- strings
- lists (aka arrays)
- functions
- exceptions
- classes
- instances
- modules (aka libraries)

Objects all have these characteristics:

- type
- value (or *state*)
- no scope (autonomous)
  - think "heap" rather than "stack"
  - imagine the interpreter has a single bag filled with all objects that were ever created
- unnamed (anonymous)
- reference count

Objects are deleted by a built-in **garbage collector** when their reference count goes to zero

You have no control over object deletion (only over the references).

What we call "variables" in other languages we should think of as "references" instead in Python

References are how we access objects

- scoped in *functions* (local namespace) and *modules* (global namespace)
- **but not** in nested code blocks
  - which is commonly the case in other languages

Read more here: Idiomatic Python (http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#other-languages-have-variables)

Containers: `lists`

In [5]:
```
# creation (words, numbers)
meta = ['foo', 'bar', 'zip', 'zap']
nums = [4, 7, 8, 7, 5]
```

In [144]:
```
meta
```

Out[144]:
```
['foo', 'bar', 'zip', 'zap']
```

In [145]:
```
nums
```

Out[145]:
```
[4, 7, 8, 7, 5]
```

In [129]:
```
# lookup: ZERO INDEXED
meta[0]
```

Out[129]:
```
'foo'
```

In [130]:
```
nums[3]
```

Out[130]:
```
7
```

In [131]:
```
# len
len(meta)
```

Out[131]:
```
4
```

In [132]:
```
len(nums)
```

Out[132]:
```
5
```

In [6]:
```
# change entry
meta[2]
```

Out[6]:
```
'zip'
```

In [7]:
```
meta[2] = 'blort'
```

In [8]:
```
meta
```

Out[8]:
```
['foo', 'bar', 'blort', 'zap']
```

In [134]:
```
# print: STATEMENT NOT FUNCTION
print "I've been using Python for", 12, "years"
```

```
I've been using Python for 12 years
```

In [136]:
```
# looping
for m in meta:
    print "Found word", w
```

```
Found word foo
Found word bar
Found word zip
Found word zap
```

In [137]:
```
for n in nums:
    print "Found number", n, "whose square is", n**2
```

```
Found number 4 whose square is 16
Found number 7 whose square is 49
Found number 8 whose square is 64
Found number 7 whose square is 49
Found number 5 whose square is 25
```

In [138]:
```
# sum
sum(nums)
```

Out[138]: 31

In [139]:
```
# max/min
```

In [140]:
```
max(nums)
```

Out[140]: 8

In [141]:
```
min(nums)
```

Out[141]: 4

In [2]:
```
# what is "in" the list?
stuff = [3, 8, 'ping', 'pong', ['a', 'b', 'c'], 42]
```

In [143]:
```
stuff
```

Out[143]: [3, 8, 'ping', 'pong', ['a', 'b', 'c'], 42]

In [1]:
```
# the list just contains references
```

In [3]:
```
# make a prediction:
len(stuff)
```

Out[3]: 6

In [9]:
```
# aliases
alt = meta
```

In [10]:
```
alt
```

Out[10]: ['foo', 'bar', 'blort', 'zap']

In [11]:
```
alt[1]
```

Out[11]: 'bar'

In [12]:
```
alt[1] = 'wibble'
```

In [13]:
```
alt
```

Out[13]: ['foo', 'wibble', 'blort', 'zap']

In [14]:
```
meta
```

Out[14]:
```
['foo', 'wibble', 'blort', 'zap']
```

In [15]:
```
alt is meta # two references to the same object
```

Out[15]:
```
True
```

Functions

In [16]:
```
# average function (def, parameters, colon, white space, return)
def average(vals):
    total = sum(vals)
    avg = total/float(len(vals))
    return avg
```

In [17]:
```
average(nums)
```

Out[17]:
```
6.2000000000000002
```

In []:
```
# duck typing
```

In [20]:
```
# exceptions
average(meta)
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call last
)
<ipython-input-20-093e0619e416> in <module>()
      1 # exceptions
----> 2 average(meta)

<ipython-input-16-81120b8cb208> in average(vals)
      1 # average function (def, parameters, colon, white space, return)
      2 def average(vals):
----> 3     total = sum(vals)
      4     avg = total/float(len(vals))
      5     return avg

TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

In [21]:
```
# sorted
nums
```

Out[21]:
```
[4, 7, 8, 7, 5]
```

In [22]:
```
sorted(nums) # returns a new list with sorted contents
```

Out[22]:
```
[4, 5, 7, 7, 8]
```

In [23]:
```
nums
```

Out[23]:
```
[4, 7, 8, 7, 5]
```

In [24]:
```
sorted(meta)
```

Out[24]:
```
['blort', 'foo', 'wibble', 'zap']
```

Exercise 1

You are going to write a "median" function that will return the middle value from a list of numbers. Use the `average` function as a template.

Given the input:

```
[3, 7, 2, 9, 6]
```

The function should return `6`.

Before you start, indicate to your neighbor your expected chance of success on a scale of 1 to 10. If your chance of success is greater than your neighbors, check in with them after a few minutes to see if they need help.

Hints:

- Remember that white space is significant in Python!
- The function, internally, will need to create a sorted copy of the input list of numbers
- Dividing the length of the list by 2 will give you a "good enough" measure of the middle of the list
- Remember "duck typing": expect that the right kind of data will be passed to your function

Bonus:

- Try out your function on a list of words

In [25]:
```
def median(entries):
    ordered = sorted(entries)
    middle = ordered[len(entries)/2]
    return middle
```

In [26]:
```
median(nums)
```

Out[26]:
```
7
```

In [27]:
```
median(meta)
```

Out[27]:
```
'wibble'
```

In []:
```
# help (dir, len)
```

In [28]:
```
help(dir)
```

```
Help on built-in function dir in module __builtin__:

dir(...)
    dir([object]) -> list of strings

    If called without an argument, return the names in the current scope.
    Else, return an alphabetized list of names comprising (some of) the at
tributes
    of the given object, and of attributes reachable from it.
    If the object supplies a method named __dir__, it will be used; otherw
ise
    the default dir() logic is used and returns:
      for a module object: the module's attributes.
      for a class object:  its attributes, and recursively the attributes
        of its bases.
      for any other object: its attributes, its class's attributes, and
        recursively the attributes of its class's base classes.
```

In [29]:
```
help(len)
```

```
Help on built-in function len in module __builtin__:

len(...)
    len(object) -> integer

    Return the number of items of a sequence or collection.
```

In [30]:
```
# docstring
def median(entries):
    " return the middle entry from an iterable based on its sorted order
"
    ordered = sorted(entries)
    middle = ordered[len(entries)/2]
    return middle
```

In [31]:
```
median
```

Out[31]:
```
<function __main__.median>
```

In [32]:
```
help(median)
```

```
Help on function median in module __main__:

median(entries)
    return the middle entry from an iterable based on its sorted order
```

In [33]:
```
median(nums)
```

Out[33]:
```
7
```

In [34]:
```
nums
```

Out[34]:
```
[4, 7, 8, 7, 5]
```

In [35]:
```
# in place sort (and OO nature of Python)
nums.sort()
```

In [36]:
```
nums
```

Out[36]: [4, 5, 7, 7, 8]

In [37]:
```
meta
```

Out[37]: ['foo', 'wibble', 'blort', 'zap']

In [38]:
```
meta.sort()
```

In [39]:
```
meta
```

Out[39]: ['blort', 'foo', 'wibble', 'zap']

In [40]:
```
# add entry
meta.append('zip')
```

In [41]:
```
meta
```

Out[41]: ['blort', 'foo', 'wibble', 'zap', 'zip']

In [42]:
```
nums.append(12)
```

In [43]:
```
nums
```

Out[43]: [4, 5, 7, 7, 8, 12]

In [44]:
```
# grow list with another (for loop, method)
extra = ['aaa', 'bbb', 'ccc']
for e in extra:
    meta.append(e)
```

In [45]:
```
meta
```

Out[45]: ['blort', 'foo', 'wibble', 'zap', 'zip', 'aaa', 'bbb', 'ccc']

In [46]:
```
# reset
meta = ['foo', 'bar', 'zip', 'zap']
```

In [47]:
```
meta
```

Out[47]: ['foo', 'bar', 'zip', 'zap']

In [48]:
```
meta.append(extra)
```

In [49]:
```
meta
```

Out[49]: ['foo', 'bar', 'zip', 'zap', ['aaa', 'bbb', 'ccc']]

In [50]:
```
# OOPS! That wasn't what we wanted.
```

In [52]:
```
meta.pop() # remove and return the the last object in the list
```

Out[52]:
```
['aaa', 'bbb', 'ccc']
```

In [53]:
```
meta
```

Out[53]:
```
['foo', 'bar', 'zip', 'zap']
```

In [54]:
```
meta.extend(extra)
```

In [55]:
```
meta
```

Out[55]:
```
['foo', 'bar', 'zip', 'zap', 'aaa', 'bbb', 'ccc']
```

In [56]:
```
# remove entry (a few different ways)
meta[2]
```

Out[56]:
```
'zip'
```

In [57]:
```
del meta[2]
```

In [58]:
```
meta
```

Out[58]:
```
['foo', 'bar', 'zap', 'aaa', 'bbb', 'ccc']
```

In [59]:
```
meta.pop()
```

Out[59]:
```
'ccc'
```

In [60]:
```
meta
```

Out[60]:
```
['foo', 'bar', 'zap', 'aaa', 'bbb']
```

In [61]:
```
meta.remove('bar')
```

In [62]:
```
meta
```

Out[62]:
```
['foo', 'zap', 'aaa', 'bbb']
```

In [63]:
```
# membership
'zap' in meta
```

Out[63]:
```
True
```

In [64]:
```
'bar' in meta
```

Out[64]:
```
False
```

In [67]:
```
# slicing
skip = range(55, 150, 12) # start, end, step
```

In [68]:
```
skip
```

Out[68]:
```
[55, 67, 79, 91, 103, 115, 127, 139]
```

In [69]:
```
skip[3]
```

Out[69]: 91

In [71]:
```
skip[3:6] # NOTICE: half-open interval: end index is not included
```

Out[71]: [91, 103, 115]

In [72]:
```
# negatives
fruit = 'apples oranges peaches pears grapes'.split()
```

In [73]:
```
fruit
```

Out[73]: ['apples', 'oranges', 'peaches', 'pears', 'grapes']

In [74]:
```
fruit[-1]
```

Out[74]: 'grapes'

In [75]:
```
fruit[-2]
```

Out[75]: 'pears'

In [76]:
```
fruit[-4:-1] # same rules: end index is not included
```

Out[76]: ['oranges', 'peaches', 'pears']

In [77]:
```
# defaults
fruit[0:3]
```

Out[77]: ['apples', 'oranges', 'peaches']

In [78]:
```
fruit[:3]
```

Out[78]: ['apples', 'oranges', 'peaches']

In [79]:
```
fruit[3:6]
```

Out[79]: ['pears', 'grapes']

In [80]:
```
fruit[3:]
```

Out[80]: ['pears', 'grapes']

In [81]:
```
# strides
fruit[1:6:2]
```

Out[81]: ['oranges', 'pears']

In [85]:
```
skip
```

Out[85]: [55, 67, 79, 91, 103, 115, 127, 139]

In [84]:
```
skip[2:8:2]
```

Out[84]: [79, 103, 127]

In []:
```
# slice delete
```

In [86]:
```
skip[2:4]
```

Out[86]:
```
[79, 91]
```

In [87]:
```
del skip[2:4]
```

In [88]:
```
skip
```

Out[88]:
```
[55, 67, 103, 115, 127, 139]
```

In [89]:
```
skip[2:4]
```

Out[89]:
```
[103, 115]
```

In [90]:
```
# slice update
skip[2:4] = range(10, 30, 5)
```

In [91]:
```
skip
```

Out[91]:
```
[55, 67, 10, 15, 20, 25, 127, 139]
```

In [94]:
```
# list math
[10, 20] + [10, 20]
```

Out[94]:
```
[10, 20, 10, 20]
```

In [95]:
```
[10, 20] * 3
```

Out[95]:
```
[10, 20, 10, 20, 10, 20]
```

In [97]:
```
# conditionals: long words, short and long words
meta = 'foo bar ping pong zip zap blort wibble'.split()
```

In [98]:
```
meta
```

Out[98]:
```
['foo', 'bar', 'ping', 'pong', 'zip', 'zap', 'blort', 'wibble']
```

In [99]:
```
for m in meta:
    print 'checking', m
    if len(m) >= 4:
        print "\tfound long word", m, "length", len(m)
```

```
checking foo
checking bar
checking ping
        found long word ping length 4
checking pong
        found long word pong length 4
checking zip
checking zap
checking blort
        found long word blort length 5
checking wibble
        found long word wibble length 6
```

In [100]:
```
for m in meta:
    print 'checking', m
    if len(m) >= 4:
        print "\tfound long word", m, "length", len(m)
    elif len(m) < 4:
        print "\ttoo short!", m
```

```
checking foo
        too short! foo
checking bar
        too short! bar
checking ping
        found long word ping length 4
checking pong
        found long word pong length 4
checking zip
        too short! zip
checking zap
        too short! zap
checking blort
        found long word blort length 5
checking wibble
        found long word wibble length 6
```

In [101]:
```
# long words function
def findlong(words):
    " return a list of the long words in a list of words "
    result = []
    for w in words:
        if len(w) >= 4:
            result.append(w)
    return result
```

In [102]:
```
findlong(meta)
```

Out[102]:
```
['ping', 'pong', 'blort', 'wibble']
```

In [104]:
```
fruit
```

Out[104]:
```
['apples', 'oranges', 'peaches', 'pears', 'grapes']
```

In [105]:
```
findlong(fruit)
```

Out[105]:
```
['apples', 'oranges', 'peaches', 'pears', 'grapes']
```

In [106]:
```
meta
```

Out[106]:
```
['foo', 'bar', 'ping', 'pong', 'zip', 'zap', 'blort', 'wibble']
```

In [107]:
```
result # locally scoped *reference*
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-107-a5b1e83cd027> in <module>()
----> 1 result

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

In [112]:
```
# default parameters
def findlong(words, cutoff=4):
    " return list of words at least `cutoff` in length (default 4) "
    result = []
    for w in words:
        if len(w) >= cutoff:
            result.append(w)
    return result
```

In [113]:
```
findlong(meta)
```

Out[113]:
```
['ping', 'pong', 'blort', 'wibble']
```

In [114]:
```
findlong(fruit)
```

Out[114]:
```
['apples', 'oranges', 'peaches', 'pears', 'grapes']
```

In [115]:
```
findlong(fruit, 7)
```

Out[115]:
```
['oranges', 'peaches']
```

In [116]:
```
# named arguments
```

In [117]:
```
findlong(meta, cutoff=5)
```

Out[117]:
```
['blort', 'wibble']
```

In [119]:
```
# multi-value return
def shortlong(words, split=4):
    """ split words into 'short' and 'long'
        where long is at least `split` in length """
    short = []
    long  = []
    for w in words:
        if len(w) >= split:
            long.append(w)
        else:
            short.append(w)
    return short, long
```

In [120]:
```
result = shortlong(meta)
```

In [121]:
```
result
```

Out[121]:
```
(['foo', 'bar', 'zip', 'zap'], ['ping', 'pong', 'blort', 'wibble'])
```

In [122]:
```
len(result)
```

Out[122]:
```
2
```

In [123]:
```
type(result)
```

Out[123]:
```
tuple
```

In [124]:
```
a = result[0]
b = result[1]
```

```
In [125]: a
```

```
Out[125]: ['foo', 'bar', 'zip', 'zap']
```

```
In [126]: b
```

```
Out[126]: ['ping', 'pong', 'blort', 'wibble']
```

```
In [130]: # or use tuple unpacking
          a, b = shortlong(meta, 4)
```

```
In [131]: a
```

```
Out[131]: ['foo', 'bar', 'zip', 'zap']
```

```
In [132]: b
```

```
Out[132]: ['ping', 'pong', 'blort', 'wibble']
```

Scripts

- syntactically valid Python code in a file
- execute with `python filename` from command line
- your IDE may give you some short-cuts

NOTE: bare expressions and `print`

Bare expressions such as:

```
7 + 5
meta[3]
average(vals)
```

only display a representation of the result in the console when using the **interactive** interpreter.

Such "bare expressions" will produce no output in a program (although they will be executed).

Use `print` statements in your program if you want output to appear.

```
In []: # script
```

```
In [133]: # complex
          s = 3 + 2j
          t = 3 - 2j
```

```
In [134]: s + t
```

```
Out[134]: (6+0j)
```

```
In [135]: s * t
```

```
Out[135]: (13+0j)
```

In [136]:
```
type(s)
```

Out[136]:
```
complex
```

In [ ]:
```
# we have min/max/sum: other math? e.g. sin?
```

In [140]:
```
cos(3.14/2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-140-b84c71399323> in <module>()
----> 1 cos(3.14/2)

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

In [ ]:
```
# import
```

In [141]:
```
import math
```

In [142]:
```
cos(3.14/2)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-142-b84c71399323> in <module>()
----> 1 cos(3.14/2)

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

In [143]:
```
math.cos(3.14/2)
```

Out[143]:
```
0.0007963267107332633
```

Python Standard Library

- python.org standard library reference documentation (https://docs.python.org/2.7/library/index.html)
- Python Standard Library offers about 300 amazing modules:
  - included with every Python distribution ("*batteries included*")
  - well documented
  - stable APIs (7-12 years)
  - great performance (highly optimized -- implemented in C if necessary)
  - widely used (field tested: problems will surface fast)
  - code reviewed
  - great test coverage (~100%)
- A killer feature of Python
  - so why wouldn't you use them?
  - sing it from the mountain tops!
- Check the Standard Library **first**:
  - before writing it yourself
  - before looking in the Cheeseshop
- Python Tutorial provides a nice overview in the last few chapters
  - skimmable
  - includes examples

In [144]:
```
# from/as targeted import: random.randint
from math import sin
sin(3.14/2)
```

Out[144]:
```
0.9999996829318346
```

In [145]:
```
from random import randint
```

In [148]:
```
randint(1, 6)
```

Out[148]:
```
4
```

In [149]:
```
# sys (version, version_info, executable, path)
import sys
```

In [150]:
```
sys.version
```

Out[150]:
```
'2.7.8 |Anaconda 2.1.0 (x86_64)| (default, Aug 21 2014, 15:21:46) \n[GCC
4.2.1 (Apple Inc. build 5577)]'
```

In [151]:
```
sys.version_info
```

Out[151]:
```
sys.version_info(major=2, minor=7, micro=8, releaselevel='final', serial=
0)
```

In [152]:
```
sys.executable
```

Out[152]:
```
'/Users/ijstokes/anaconda/bin/python'
```

In [153]:
```
sys.path
```

Out[153]:
```
['',
 '/Users/ijstokes/anaconda/lib/python27.zip',
 '/Users/ijstokes/anaconda/lib/python2.7',
 '/Users/ijstokes/anaconda/lib/python2.7/plat-darwin',
 '/Users/ijstokes/anaconda/lib/python2.7/plat-mac',
 '/Users/ijstokes/anaconda/lib/python2.7/plat-mac/lib-scriptpackages',
 '/Users/ijstokes/anaconda/lib/python2.7/lib-tk',
 '/Users/ijstokes/anaconda/lib/python2.7/lib-old',
 '/Users/ijstokes/anaconda/lib/python2.7/lib-dynload',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/PIL',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/Sphinx-1.2.3-py2.7
.egg',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/cisco-web',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/cisco-net',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/cisco-test',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/runipy-0.1.1-py2.7
.egg',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/setuptools-5.8-py2
.7.egg',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/wx-3.0-osx_cocoa',
 '/Users/ijstokes/anaconda/lib/python2.7/site-packages/IPython/extensions
']
```

In [154]:
```
# raw_input
raw_input('What is your name? ')
```

```
What is your name? Ian
```

Out[154]:
```
'Ian'
```

In [155]:
```
sys.argv # not useful here
```

Out[155]:
```
['-c',
 '-f',
 '/Users/ijstokes/.ipython/profile_default/security/kernel-40341c6f-d4be-
45cd-ba2b-bad34e3d0f18.json',
 '--pylab=inline',
 "--IPKernelApp.parent_appname='ipython-notebook'",
 '--profile-dir',
 '/Users/ijstokes/.ipython/profile_default',
 '--parent=1']
```

In []:
```
# program: long words (internal cutoff, CLI cutoff)
```

In []:
```
# pyflakes
```

Exercise 2

Write a single file program that uses `sys.argv` to read in a sequence of numbers and return the average. Name the file `calc-average`

Executing this from the command line:

```
$ python calc-average 3 12 6
```

should return the result `7`

As before, estimate your likelihood of success and check-in with your neighbor. Be prepared to help others complete this.

Hints:

- Add generous debugging output via the `print` statement
- Remember `sys.argv` is a list of strings, and Python does not transmogrify "number-like-strings" into numbers for you -- you will need to do this yourself with a loop and a call to `int` or `float`
- Remember that `sys.argv[0]` is the program name: you want to skip this entry by judicious slicing

Bonus:

- Return a usage message if no CLI arguments are provided
- Check out argparse (http://docs.python.org/2/library/argparse.html) for a better way to handle CLI arguments
- If you know what a *shbang* line is, try to set this up for the program

```
In []: # map
```

```
In []: # list comprehension
```

```
In []: # exception handling (on type conversion)
```

Great Python Tools

- IPython (http://ipython.org)
  - command line: `ipython`
  - web view: `ipython notebook`
  - Matlab-like mode: `ipython notebook --pylab=inline`
    - pre-imports ~1000 functions
    - inline graphics
  - great for exploratory/interactive work
  - not for writing programs
- Spyder (https://code.google.com/p/spyderlib/) Python IDE
  - included with Anaconda
- PyCharm (https://www.jetbrains.com/pycharm/) Python IDE
  - free community edition
- PyDev (http://pydev.org/) Eclipse IDE plugin for Python development
- Sublime Text 3 (http://www.sublimetext.com/) cross-platform programmer's editor
- Git (http://git-scm.com/)

In []:
```
# ipython CLI
```

In []:
```
# ipython notebook --pylab=inline # matlab-like environment
```

Reserved Words

- 30 reserved words (aka keywords)
  - http://docs.python.org/2.7/reference/lexical_analysis.html#keywords (http://docs.python.org/2.7/reference/lexical_analysis.html#keywords)
  - *logic:* `and, not, or`
  - *namespaces:* `import, from, as, del, global`
  - *object creation:* `class, def, lambda`
  - *functions:* `return, print, yield`
  - *looping:* `while, for, break, continue`
  - *conditional:* `if, else, elif`
  - *exeptions:* `try, except, finally, raise`
  - *misc:* `pass, assert, with, exec, in, is`
- Interpreter starts up knowing (almost) *nothing* but language syntax

In []:
```
# Is "dir" a reserved word?
```

In []:
```
# objects and references
```

In []:
```
# namespaces: locals, globals, builtin
```

__builtin__

- The Python Language defines a special module called ***builtin*** that is part of the Standard Library
- It contains *functions*, *exceptions*, and *classes* that are very common:
  - 3 special entries: *None*, *True*, *False* (keywords in Python 3)
  - 10 core types
    - *int, long, float, bool, complex, str, list, dict, tuple, set*
  - 20 supporting types
    - *file, xrange, object, ...*
  - 40 exceptions (upper camel case, mostly ending in *Error* or *Warning*)
  - 50 functions
    - Math: *abs min max pow round sum divmod*
    - Logic: *all any apply map filter reduce*
    - Check: *callable isinstance issubclass*
    - Convert: *bin chr hex cmp coerce oct ord unichr*
    - Introspect: *dir id vars locals globals hasattr getattr setattr delattr compile eval execfile intern hash repr*
    - File: *open*
    - Iterable: *len range zip iter next sorted*
    - Other: *format reload*
- Any reference lookup that doesn't find the reference in the *local* namespace (first) or the *global* (which means *module*) namespace (second) will check the `__builtin__` modules namespace (third)
- CPython automatically provides a reference to the `__builtin__` module in every *global* namespace but gives it the name `__builtins__`
  - under normal use, you never need to use this module reference
- If the *local* or *global* namespace has a reference that is found in `__builtin__` then the `__builtin__` reference will be masked

In []:
```
# sorter.py and word-sorter (incl. module docstring)
```

Exercise 3

Create a module `mystats` in a file `mystats.py`, and in it put just the functions `average` and `median`. Include *docstrings* for your module and functions.

Now try importing it, check the doc string (`help(mystats)`), and try using the functions:

```
>>> import mystats
>>> help(mystats)
>>> help(mystats.average)
>>> vals = [4, 8, 2, 9, 12]
>>> mystats.average(vals)
7
>>> mystats.median(vals)
8
```

Now create a new program `calc-average2` that uses this module to get the functions.

Work with your neighbor for mutual encouragement.

Hints:

- Include copious debugging (`print` statements)
- These files all need to be in the same directory where your interactive interpreter is running

```
In []: # __name__ check
```

```
In []: # sys.path, $PYTHONPATH
```

```
In []: # module meta-data: name, doc, file: dunder attributes
```

Mutability

Mutability of an object is a property of the object based on its type

- some objects are mutable (e.g. lists, dictionaries, modules, your own instance objects)
- some objects are not (e.g. numbers, strings, tuples)

In Python we **never** need to ask the question "is variable the object or just a reference to the object?" because:

- all **things** are **anonymous**, **autonomous** objects
- we **only** ever access objects via **references**

You can be forgiven for getting confused! Most other languages don't work this way (or at least don't *appear* to work this way)

- assignment statements don't update an object with a new value
- they reassign the reference to a new object
- methods on an object may modify the object's internal state
  - if that is what they are supposed to do
  - if the object is mutable

In []:
```
# 3, 7, foo, bar
```

In []:
```
# upper: new list of upper case version words
```

In []:
```
# upper: update in place
```

In []:
```
# filter: new list of long words
```

In []:
```
# filter: update in place to remove short words
```

Dictionaries

- it would not be too much to say that the Python language is built on dictionaries
- most highly tuned data structure in the language
- use them fearlessly
- in other languages, variously known as a *HashMap* or *Associative Array*

In []:
```
# creation: ian, maggie, hilary
```

In []:
```
# read
```

In []:
```
# update entry
```

In []:
```
# add (color)
```

In []:
```
# KeyError
```

In []:
```
# membership
```

In []:
```
# update from address dict
```

In []:
```
# iteration
```

In []:
```
# vitals function: height, weight, eyes, hair
```

Exercise 4

Create a dictionary to describe books. Think about the different fields it could have. Create 3 different dictionaries for made-up (or real) books.

Write a single parameter function `fix()` that takes your dictionary object (i.e. with the field names you chose) and peforms the following:

- Change the book title entry to all upper case using the `.upper()` method on a string
- Change the book author entry to title case using the `.title()` method on a string
- Print out "too long" if the book is over 400 pages
- Print out "too short" if the book is under 100 pages

Try out your `fix()` function on your book dictionaries

Put your books in a list then iterate over that list and call `fix()` on each one in turn

Bonus

- make `short` and `long` parameters to `fix()`
- give those parameters defaults

Classes

In []:
```
# 2D point with lists
```

In []:
```
# distance function
```

In []:
```
# tuple version
```

In []:
```
# dictionary version
```

In []:
```
# empty Point class
```

In []:
```
# class version (function with two Point instances)
```

In []:
```
# angle function: atan(y/x)*180/pi
```

In []:
```
# angle method (calling from class)
```

In []:
```
# angle method (calling from instance)
```

In []:
```
# create_point function
```

In []:
```
# create_point method
```

In []:
```
# __init__
```

In []:
```
# self
```

In []:
```
# distance: function to method
```

In []:
```
# __repr__
```

In []:
```
# __str__
```

In []:
```
# Line class (2 points)
```

In []:
```
# length method
```

In []:
```
# Line class (list of points)
```

In []:
```
# __getitem__
```

In []:
```
# __len__
```

In []:
```
# __contains__
```

There are lots of *dunder methods* that you can implement to get special behavior:

- Magic Method Summary (http://www.rafekettler.com/magicmethods.html)

Exercise 5

Create a `Square` class that describes a square based on 2 `Point` instances for the bottom left and top right corners.

Add an `.area()` method that returns the area of the square

Add a `.circum()` method that returns the circumference

Bonus: Create a `Traingle` class based on 3 `Point` instances. Unless you are a geometry whiz, skip the `.area()` method.

Inheritance

In []:
```
# Shape (with "points", str, repr)
```

In []:
```
# UnitsLine (with "units")
```

In []:
```
# ColorQuad (new-style, with "color")
# super: super(ColorQuad, self).__init__(...)
```

FileIO

In []:
```
# read (shopping.txt)
```

In []:
```
# seek
```

In []:
```
# read all
```

In []:
```
# close
```

In []:
```
# with (context manager)
```

In []:
```
# but usually, iteration and line-by-line reading
# remember, it is all text!
```

In []:
```
# read (numbers.dat)
```

In []:
```
# generate list, return average
```

In []:
```
# read (points.dat)
```

In []:
```
# convert to Point objects
```

In []:
```
# create Line object from collection of points
```

In []:
```
# write line length to file
```

Python + Numbers + Pictures =

(numpy, matplotlib, pandas)

- save your work
- exit out of Python, IPython, Spyder, etc. etc.
  - shut it all down from the command line
- start up *IPython Notebook* in *pylab* mode with inline graphics:

```
$ ipython notebook --pylab=inline
```

- this is the easiest way to explore graphics in Python for this workshop

Attribution:

- Weather data from Environment Canada (http://climate.weather.gc.ca/)
- Python examples adapted from Julia Evans Pandas Cookbook (https://github.com/jvns/pandas-cookbook)

In []:
```
jan = []
with open('jan_2012.txt') as fh:
    for line in fh:
        entry = float(line.strip())
        jan.append(entry)
```

In []:
```
plot(jan)
```

In []:
```
figure()
hold(True)
title('mid-day temperatures in Montreal, January 2012')
ylabel('temp (C)')
xlabel('day')
plot(jan)
show()
```

In []:
```
from datetime import datetime
dt_conv = lambda dt: datetime.strptime(dt, '%m/%d/%y %H:%M')
weather = genfromtxt('weather_2012.csv',
    converters={0:dt_conv}, delimiter=',', skip_header=1,
    dtype=[
        ('dt',     'datetime64[h]'),
        ('temp',   '|f8'),
        ('dew',    '|f8'),
        ('humid',  '|f8'),
        ('wind',   '|f8'),
        ('vis',    '|f8'),
        ('pres',   '|f8'),
        ('cond',   'a20')
    ])
```

In []:
```
from collections import Counter
```

In []:
```
cond = Counter(weather['cond'])
```

In []:
```
cond.most_common(10)
```

Phew! `numpy` is amazing and gives great (Matlab-like) performance for array/matrix data, but can be a handful sometimes.

`pandas` to the rescue: Provides a wrapper around numpy `ndarray` data structures to make them similar to R's very versatile *Data Frame*

In []:
```
import pandas as pd

pd.read_csv('weather_2012.csv')
```

In []:
```
pd.read_csv('weather_2012.csv', index_col='Date/Time')
```

In []:
```
weather = pd.read_csv('weather_2012.csv', index_col='Date/Time')
```

In []:
```
figure(figsize=(15,6))
hold(True)
weather['Temp (C)'].plot()
weather['Wind Spd (km/h)'].plot(secondary_y=True)
```

In []:
```

```