Building Programs with Python

For our Python introduction we're going to pretend to be a researcher named Harold Bridge (user id `hbridge`) who is studying inflammation in patients who have been given a new treatment for arthritis.
Use Mercurial to grab the files from Bitbucket and put them in an `hbridge` directory in your SWC workspace:

```
$ cd
$ cd Desktop/swc
$ hg clone https://bitbucket.org/douglatornell/swc-hbridge-files hbridge
```

You can copy and paste the `hg clone` command from the Etherpad.

Launching IPython Notebook

There are several ways that we can use Python. We're going to start with tool called IPython Notebook that runs in the browser.
In a shell window enter these commands:

```
$ cd
$ cd Desktop/swc/
$ ipython notebook
```

The shell window is now running a local web server for you. Don't close it. You will need to open another shell window to do other command line things.
Your browser should open to an "IP[y]: Notebook" page showing a list of driectories. If it doesn't, Open a new tab and enter the address `127.0.0.1:8888`.

Analyzing Patient Data Part 1

1. Explain what a library is, and what libraries are used for.
2. Load a Python library and use the things it contains.
3. Read tabular data from a file into a program.
4. Assign values to variables.
5. Select individual values and subsections from data.

- import numpy
- numpy.loadtxt(fname= delimiter=)
- weight_kg = 55
- print
- weight_lb = 2.2 * weight_kg
- type(data)
- data.shape
- data[0,0], data[0:1,0:1]
- data[0:10:2,1]
- data[:3,36:]

Analyzing Patient Data Part 2

6. Perform operations on arrays of data.
7. Display simple graphs.

- data.mean()
- data.std()
- data.mean(axis=0)
- %matplotlib inline
- from matplotlib import pyplot
- pyplot.imshow(data)
- pyplot.show()
- pyplot.plot(ave_inflammation)
- import matplotlib import pyplot as plt
- plt.subplot(1,3,1)
- plt.ylabel('average')
- plt.show()

Exercise

Create a single plot showing 1) the mean for each day and 2) the mean + 1 standard deviation for each day and the 3) the mean - 1 standard deviation for each day.

Creating Functions Part 1

1. Define a function that takes parameters.
2. Return a value from a function.
3. Test and debug a function.

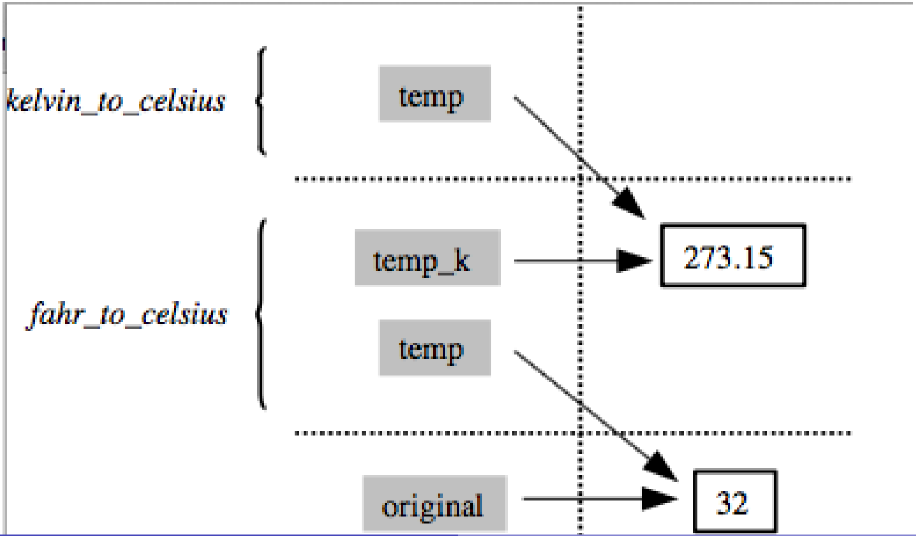
-
  ```
  def fahr_to_kelvin(temp):
       return ((temp - 32) * (5/9)) + 273.15
  ```
- from __future__ import division

-
  ```
  def kelvin_to_celsius(temp):
      return temp - 273.15
  ```

-
  ```
  def fahr_to_celsius(temp):
      temp_k = fahr_to_kelvin(temp)
      result = kelvin_to_celsius(temp_k)
      return result
  ```

Creating Functions Part 2

4. Explain the scope of a variable and the idea of encapsulation.



Creating Functions Part 3

3. Test and debug a function.
4. Explain why we should divide programs into small, single-purpose functions.

-
  ```
  def centre(data, desired):
      return (data - data.mean()) + desired
  ```
- z = numpy.zeros((2,2))
- print centre(z, 3)
- print data.std() - centred.std()
- "'centre(data, desired): return a new array containing the original data centered around the desired value."'
- help(centre)

Exercise

Write a function called analyze that takes a filename as a parameter and displays the three graphs produced in the previous lesson, i.e., analyze('inflammation-01.csv') should produce the graphs already shown, while analyze('inflammation-02.csv') should produce corresponding graphs for the second data set. Be sure to give your function a docstring. Hint: a function can just "do" something. It doesn't necessarily need to return anything.

Creating Functions Part 4

6. Set default values for function parameters.

- def center(data, desired = 0):

-
  ```
  def display(a=1, b=2, c=3):
      print 'a:', a, 'b:', b, 'c:', c

  print 'no parameters:'
  display()
  print 'one parameter:'
  display(55)
  print 'two parameters:'
  display(55, 66)
  ```
- help(numpy.loadtxt)

Analyzing Multiple Data Sets: Part 1

You should have a working function analyze. If not, its at the top of `https://douglatornell.github.io/2014-09-25-ubc/novice/python/03-loop.html`

1. Explain what a for loop does.
2. Correctly write for loops to repeat simple calculations.

-
  ```
  def print_characters(element):
      print element[0]
      print element[1]
      print element[2]
  ```
- print_characters('tin')
- print_characters('hg')

-
  ```
  def print_characters(element):
      for char in element:
          print char
  ```

Analyzing Multiple Data Sets: Part 2

3. Trace changes to a loop variable as the loop runs.
4. Trace changes to other variables as they are updated by a for loop.

-
  ```
  length = 0
  for vowel in 'aeiou':
      length = length + 1
  print 'There are', length, 'vowels'
  ```

-
  ```
  letter = 'z'
  for letter in 'abc':
      print letter
  print 'after the loop, letter is', letter
  ```
- len('aeiou')

Analyzing Multiple Data Sets: Part 3

5. Explain what a list is.
6. Create and index lists of simple values.

- odds = [1,3,5,7]
- odds[0], odds[-1]
- for number in odds:
- names = ['Newton', 'Darwig', 'Turing']
- names[1] = 'Darwin'
- name = 'Darwig'
- name[5] = 'n'
- odds.append[9]
- del odds[0]
- odds.reverse()

Exercise

Write a function called total that calculates the sum of the values in a list. (Python has a built-in function called sum that does this for you. Please don't use it for this exercise.)

Analyzing Multiple Data Sets: Part 4

7. Use a library function to get a list of filenames that match a simple wildcard pattern.
8. Use a for loop to process multiple files.

- import glob
- print glob.glob('*.ipynb')
-
```
filenames = glob.glob('*.csv')
filenames = filenames[0:3]
for f in filenames:
    print f
    analyze(f)
```

Conditionals – Making Choices

1. Write conditional statements including `if`, `elif`, and `else` branches.
2. Correctly evaluate expressions containing `and` and `or`.
3. Correctly write and interpret code containing nested loops and conditionals.

- `numpy.empty()`
- `numpy.empty_like()`
- `+=, -=, *=, /=`
- `enumerate`
- `if ... elif ... else`
- `==, !=, <, <=, >, >=`
- `and, or, not`
- non-printing characters; e.g. `\n`
- Line continuations in code

Python Modules and Command-line Programs

1. Create a Python module containing functions that can be ‘import‘-ed into notebooks and other modules.
2. Use the values of command-line arguments in a program.
3. Read data from standard input in a program so that it can be used in a pipeline.

- `%%writefile`
- `!cat`
- Module & function docstrings
- `reload()`
- `sys.argv`

- `$ python myfile.py`
- `if __name__ == "__main__":`
- `sys.stdin`
- `argparse`