Welcome to
Python 2
Session #2

Michael Purcaro & The GSBS Bootstrappers

February 2014

michael.purcaro@umassmed.edu

Extended Exercise 1

Goal: count how many signal peaks are present in processed ENCODE ChIP-seq data on chromosome 7

url:
http://bib3.umassmed.edu/~purcarom/Python2/Lecture1/ENCFF002COQ.narrowPeak

File format: one peak per line

genome.ucsc.edu/FAQ/FAQformat.html#format12

Answer hint: between 2000 and 3000

Extended Exercise 2

Modify code from Extended Exercise 1 to compute what percentage of chromosome 7 (assume hg19) is covered by peaks.

Length of chr7 in hg19: 159138663

(Length of HG19 chromosomes in hg19.chrom.sizes in bioinfo.umassmed.edu/bootstrappers/bootstrappers-courses/python2/lecture1/)

Answer hint: <5%

Building Blocks: Functions

- Functions help divide code up into smaller chunks
  - easier to understand
  - easier to test individual function
  - encourages code reuse
  - can hide details not pertinent to higher-level understanding of code
  - helps introduces structure to code
    - in some ways analogous to English paragraphs
      - i.e. more difficult to read a paper or novel if there were no paragraph breaks or sentences: all the words just run together

Building Blocks: Functions

```
def functionName(parameters):
    code (no more than a "screenful",
                    i.e. approx. <40 lines)
    optionally, return something


def chrNumAsString(num):
    return "chr" + str(num)
```
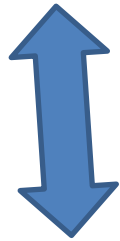
Same function! Just different "parameter" name!

```
def chrNumAsString(chrNum):
    return "chr" + str(chrNum)
```

Building Blocks: Functions

```python
def averageOfList(v):
    sumTotal = 0
    for e in v:
        sumTotal += e
    return sumTotal / float(len(v))


print averageOfList([1,1,2,3,5,8,13,21])
print averageOfList([100, 120, 110])


myList = [12,123,123,123,32,523,453,45,4567,578]
print averageOfList(myList)
```
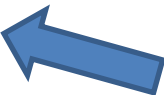
Easy code reuse!

Download a file if size changed

```python
def get_file_if_size_diff(url, path):
    fn = url.split('/')[-1]
    out_fnp = os.path.join(d, fn)
    net_file_size = int(urllib.urlopen(url).info()['Content-Length'])
    if os.path.exists(out_fnp):
        fn_size = os.path.getsize(out_fnp)
        if fn_size == net_file_size:
            print "skipping download of", fn
            return out_fnp
        else:
            print "files sizes differed:"
            print "\t", "on disk:", fn_size
            print "\t", "from net:", net_file_size
    print "retrieving", fn
    urllib.urlretrieve(url, out_fnp)
    return out_fnp
```

Get the filesize from the web server

downloads url into file named out_fnp

Exercise 3

- Convert your code from Extended Exercises 1 and 2 into functions!
- Hints:
  - Move code from these exercises into functions
  - Make functions that do the following:
    - Download a url to a certain file
      - `def downloadFile(url, fileNameAndPath)`
      - (use urllib.URLopener().retrieve(url, fileNameAndPath)…..)
    - Return a list of peaks for a given chromosome
      - `def getPeaks(fileNameAndPath, chromosomeNum):`
    - Compute the percentage of a given chromosome covered by the peaks
      - `def computePercentageChromosomeCovered(listOfPeaks, chromosomeLength):`

Building Blocks: dictionary

- Easy way to associate one thing (called a *key*) to another (called the *value*)

- Example: simple daily calendar

```
dailyCalendar = {}          ⬅ Curly braces for empty dictionary!
dailyCalendar = {key1: value1, key2 : value2,…}
dailyCalendar = { "9AM" : "PI Meeting",
                  "11AM" : "conference ",
                  "4PM" : "book club"
                }

print dailyCalendar["4PM"]
dailyCalendar["4PM"] = "PI meeting"
print dailyCalendar["4PM"]
print dailyCalendar["3PM"]
```

Building Blocks: dictionary

```
dailyCalendar = { "9AM" : "PI Meeting",
                  "11AM" : "conference ",
                  "4PM" : "book club"
                  }


print dailyCalendar["4PM"]
```
book club
```
dailyCalendar["4PM"] = "PI meeting"
print dailyCalendar["4PM"]
```
PI meeting

Building Blocks: dictionary

```
dailyCalendar = { "9AM" : "PI Meeting",
                  "11AM" : "conference ",
                  "4PM" : "book club"
                  }


print dailyCalendar["3PM"]
```

<messy error!>
KeyError: '3PM'

Building Blocks: dictionary

```
dailyCalendar = { "9AM" : "PI Meeting",
                  "11AM" : "conference ",
                  "4PM" : "book club"
                  }


if "3PM" in dailyCalendar:
    print dailyCalendar["3PM"]
```

Building Blocks: dictionary

```
dailyCalendar = { "9AM" : "PI Meeting",
                  "11AM" : "conference ",
                  "4PM" : "book club"
                }

for time, obligation in dailyCalendar.iteritems():
    print "at", time, ":\t", obligation
```

Building Blocks: dictionary

```
dailyCalendar = { "9AM" : "PI Meeting",
                  "11AM" : "conference ",
                  "4PM" : "book club"
                  }


for time, obligation in dailyCalendar.iteritems():
    print "at", time, ":\t", obligation
at 9AM : PI Meeting
at 4PM : PI meeting
at 11AM : conference call
```

Building Blocks: dictionary

- Dictionary can hold wide variety of data

```
d = {}
d[1] = ["a", "b", ["q", "r", "s"]]
d[2] = 123
d[3] = set([1,1,2,3,5,8,13])

print d[2]
123
```

Building Blocks: set

- Keeps only unique items

```
nums = set()

nums.add(1)
nums.add(1)
nums.add(2)
nums.add(3)
nums.add(5)
nums.add(8)
nums.add(8)

print nums

print 10 in nums
```

Building Blocks: set

```
nums = set()

nums.add(1)
nums.add(1)
nums.add(2)
nums.add(3)
nums.add(5)
nums.add(8)
nums.add(8)

print nums
set([8, 1, 2, 3, 5])

print 10 in nums
False
```

Building Blocks: defaultdict

- Like a normal dictionary, but supplies a default value if the key is not in the dictionary

```
from collections import defaultdict

dictionaryOfIntegers = defaultdict(int)
dictionaryOfFloats = defaultdict(float)

dailyCalendar = defaultdict(str)
dailyCalendar["9AM"] = "PI Meeting"
dailyCalendar["11AM"] = "conference call"
dailyCalendar["4PM"] = "book club"

print dailyCalendar["4PM"]
print dailyCalendar["3PM"]
```

Building Blocks: defaultdict

- Like a normal dictionary, but supplies a default value if the key is not in the dictionary

```
from collections import defaultdict

dailyCalendar = defaultdict(str)
dailyCalendar["9AM"] = "PI Meeting"
dailyCalendar["11AM"] = "conference call"
dailyCalendar["4PM"] = "book club"

print dailyCalendar["4PM"]
book club
print dailyCalendar["3PM"]
```

No error → returned an empty string!

Extended Exercise 3

- Goal: Count how many DNase-seq peaks overlap ChIP-seq peaks on chromosome 7

- ChIP-seq data url (same as before):
  http://bib3.umassmed.edu/~purcarom/Python2/Lecture1/ENCFF002COQ.narrowPeak

- DNase-seq data url:
  http://bib3.umassmed.edu/~purcarom/Python2/Lecture2/ENCFF001VZW.narrowPeak

- Hint: add the positions of every ChIP-seq peak on chromosome 7 into a set

- Hint: check if any position in a DNase-seq peak on chromosome 7 is in the ChIP-seq set

- Answer hint: between 1000 and 2000

```
import os, sys

chipFnp ="ENCFF002COQ.narrowPeak"

chipseqPeaks = set()

with open(chipFnp) as f:
    for line in f:
        toks = line.split()
        chr = toks[0]
        if "chr7" != chr:
            continue
        for p in xrange(int(toks[1]), int(toks[2])):
            chipseqPeaks.add(p)
```

```python
cellLineDnaseFn = "ENCFF001VZW.narrowPeak"

count = 0

with open(cellLineDnaseFn) as f:
    for line in f:
        toks = line.split()
        chr = toks[0]
        if "chr7" != chr:
            continue
        for p in xrange(int(toks[1]), int(toks[2])):
            if p in chipseqPeaks:
                count += 1
                break

print count
```

Building Blocks: Classes

- Strings, dictionary, defaultdict, set are all examples of classes

- In an ideal world, a class contains
  - data (numbers, strings, lists, other classes, etc.)
  - the functions allowed to manipulate the data

Class examples

- s = "Hello World!"

- print s.startswith("He")

- print s.split()

Functions in the string class

v = [1, 1, 2, 3, 5, 8, 13, 21]
print v.count(1)
print v.index(21)

Functions in the list class

24

Building Blocks: Classes

- Strings, dictionary, defaultdict, set are all examples of classes
- In an ideal world, a class contains
  - data (numbers, strings, lists, other classes, etc.)
  - the functions allowed to manipulate the data
- Provides further mechanisms to give code structure
  - Allows data and concepts to be encapsulated (i.e. hidden or only occur in one place)

Building Blocks: Classes

```python
class ExampleKlass():
    def __init__(self, strData):
        self.data = strData

    def printData(self):
        print self.data

ek = ExampleKlass("hi!")
ek.printData()
```

Building Blocks: Classes

```
class ExampleMyString():
    def __init__(self, s):
        self.string = s

    def starts(self, ss):
        return self.string.startswith(ss)
```

self used INTERNALLY in class only!

```
s = ExampleMyString("hi!")
print s.starts("h")
```

Classes and Objects

- In object-oriented programming, a "class" is the code that defines
  - the class variables
  - the functions allowed to operate on those class variables
  - No memory taken up
  - *Like the abstract concept of a book*
- An "object" is an instance of a class
  - Memory will be allocated for the variables in the object
  - *Like an actual book*

Building Blocks: Classes

- Classes have special functions (called constructors or initializers) that perform certain operations (like setting up data structures, etc.) when the object is first created

```
class Book():

    def __init__(self, title, author):
        self.title = title
        self.author = author


b = Book("Programming", "Joe Smith")
print b.title, b.author
```

2 underscores

Building Blocks: Classes

- Python classes have a "`self`" variable that understands how to access data in the class

```python
class Experiment():
    def __init__(self, expID):
        self.expID = expID
        self.listJSON = EncodeMetadataHelpers.expListingJSON(expID)
        self.assay = self.listJSON["assay_term_name"]
        self.status = self.listJSON["status"]
        self.url = BASE_URL + "experiments/" + self.expID
        self.jsonUrl = self.url + "/?format=json"
        self.expJSON = None
        self.expFiles = None

    def lab(self):
        t = self.listJSON["lab"]["title"]
        if "Michael Snyder, Stanford" == t: return "Snyder"
        if "John Stamatoyannopoulos, UW" == t: return "Stam"
        if "Bradley Bernstein, Broad" == t: return "Bernstein"
        if "Richard Myers, HAIB" == t: return "Myers"
        return t
```

Class Class Exercise

Make a "Paths" class that encapsulates the path manipulations we performed earlier, and allows the user to specify the lecture number

```
homeFolder = os.path.abspath(os.path.expanduser("~"))
desktopFolder = os.path.join(homeFolder, "Desktop")
python2folder = os.path.join(desktopFolder, "python_2")
lecture1folder = os.path.join(python2folder, "lecture_1")
print "today's lecture folder location will be:", lecture1folder
mkdir_p(lecture1folder)
```

Don't have the lecture number always be 1

- Hint:
  ```
  class Paths:
      def __init__(self, lectureNumber):
          <code from above>
  ```

- Have a helper function that takes a fileName and returns the full path to that file
  - i.e. `return os.path.join(self.lecture1folder, fileName)`

Class Class Exercise

```
homeFolder = os.path.abspath(os.path.expanduser("~"))
python2folder = os.path.join(homeFolder, "python_2")
lecture1folder = os.path.join(python2folder, "lecture_1")
print "today's lecture folder location will be:", lecture1folder
mkdir_p(lecture1folder)


class Paths():
    def __init__(self, lectureNumber):
        self.homeFolder = os.path.abspath(os.path.expanduser("~"))
        self.desktopFolder = os.path.join(self.homeFolder, "Desktop")
        self.python2folder = os.path.join(self.desktopFolder, "python_2")
        lecture = "lecture_" + str(lectureNumber)
        self.lectureFolder = os.path.join(self.python2folder, lecture)
        print "today's lecture folder location will be:", self.lectureFolder
        mkdir_p(self.lectureFolder)


paths = Paths(1)
```

Why use classes?

- What data needed to describe a book?

title = ""

author_first_name = ""

author_last_name = ""

year_published = ""

num_pages = ""

Why use classes?

- What data needed to describe a book?

```
title = ""
author_1_first_name = ""
author_1_last_name = ""
author_2_first_name = ""
author_2_last_name = ""
year_published = ""
num_pages = ""
```

Why use classes?

- What data needed to describe a book?

```
title = ""
author_1_first_name = ""
author_1_last_name = ""
author_2_first_name = ""
author_2_last_name = ""
...
author_n_first_name = ""
author_n_last_name = ""
year_published = ""
num_pages = ""
front_cover_picture = ""
back_cover_picture = ""
language = ""
edition_number = ""
```

Book class composition

```python
class Author:
    def __init__(self):
        self.first_name = ""
        ...


class Edition:
    def __init__(self):
        self.title = ""
        self.authors = [Author(...), ]
        self.isHardcover = True
        ...


class Book:
    def __init__(self):
        self.editions = [Edition(...),]
```

Book class composition

```
books = []
b = Book()
edition1 = Edition(…)
b.editions.append(edition1)
books.append(b)
```

Extended Exercise 4

- Rework your code from the previous Extended Exercises into a ChipseqData class, similar to

```
class ChipseqData:
    def __init__(self, paths, url):
```

- The class initializer should only download the chipseq data if needed

  - What are the pros/cons of downloading the data during class initialization?

- The class should have a function to return the number of peaks found in a given chromosome

- The class should also have a function to compute the percentage of a given chromosome covered by peaks

- Would a Peak class (that understood how to parse each line of the narrow peak) be useful? Try it!

Homework #1

- Work through these problems from
  http://rosalind.info/problems/list-view/

- DNA Counting DNA Nucleotides
- RNA Transcribing DNA into RNA
- REVC Complementing a Strand of DNA
- GC Computing GC Content
- HAMM Counting Point Mutations
- SPLC RNA Splicing
- PROT Translating RNA into Protein
- SUBS Finding a Motif in DNA
- PRTM Calculating Protein Mass
- REVP Locating Restriction Sites

Due: February 10, 2015