Programming and Data Science for Biology (PDSB)

Session 6
Spring 2018

Packaging: Writing Python API, CLI, and packing for distribution.

Review: Python Classes

Object oriented programming.
Organized, simple code.

Access attributes and functions from a Class object.

Many libraries are organized around a few Class objects.

```
## a simple class with an init function
class Simple:
    def __init__(self, name):
        self.name = name

## an instance of the class object
x = Simple('deren')
x.name
deren
```

Review: Packaging Python

Packages are made up of .py files that are connected by imports and __init__.py files.

```
# installable globally
pip install mypackage

# importable globally
import mypackage

# has metadata
mypackage.__version__
0.1
```

Review:
CLI vs. API

The API is the Python code, meant to be used by developers or other coders.

The CLI is meant to be run at the command line. The simplest interface for users.

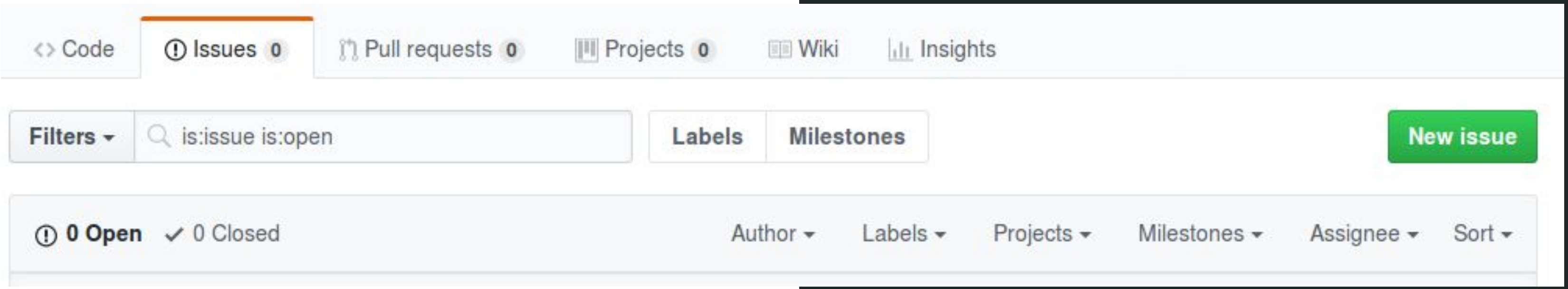
Please read instructions carefully, don’t just copy others’ assignments:
“edit the helloworld function to take at least two arguments and to execute a conditional print() statement in response to both. Be creative, but don't spend too much time on it…”

```
# Install w/ setup.py in development mode
pip install -e .

# in setupy.py build CLI script
entry_points={
     'console_scripts': ['helloworld =
helloworld.__main__:main']}

# has metadata
$ helloworld -n Deren
Hello Deren
```

Review:
Github Issues/Tickets

Let other people know about bugs in their code, or make requests for new features.

A place to comment on each other's code



Scientific Python: numpy, scipy, and pandas

```
# install the following packages and then open a jupyter notebook
$ conda install numpy scipy pandas
```

Review: Python lists

Lists are enclosed in square brackets and are a very flexible data type for storing a series of objects.

We can select elements by **indexing** or **slicing**

```
# Create a list
mylist = [‘this’, ‘is’, ‘a’, ‘list’]

# select one or multiple elements
mylist[2]
‘A’


mylist[1:]
[‘is’, ‘a’, ‘list’]
```

Review: Python lists

How do we represent a matrix or array of data in Python?

```
# One way is with lists in lists
mylist = [
     [0, 1, 2, 3],
     [4, 5, 6, 7],
     [8, 9, 10, 11],
     ]

# the result looks like this:
[[0,1,2,3],[4,5,6,7],[8,9,10,11]]
```

Review: Python lists

We can access rows of the list, or items by using sequential indexing.

```
# Matrix represented by lists in a list
mylist
[[0,1,2,3],[4,5,6,7],[8,9,10,11]]

# Select first row of data
mylist[0]
[0,1,2,3]

# Select an item from a list in list
mylist[1][0]
4
```

Review: Python lists

But what if we want to select columns of the matrix?

```
# Matrix represented by lists in a list
mylist
[[0,1,2,3],[4,5,6,7],[8,9,10,11]]

# Select columns… not so simple
[i[1] for i in mylist]
[1,5,9]
```

Review: Python lists

And what if we want to **operate** on rows or columns of a matrix?

```
# multiply all elements of matrix by 2
# … doesn’t do what we might expect
mylist * 2
[[0,1,2,3],[4,5,6,7],[8,9,10,11],
 [0,1,2,3],[4,5,6,7],[8,9,10,11]]
```

Numpy
(numerical python)

This is an excerpt from the Python Data Science Handbook by Jake VanderPlas; Jupyter notebooks are available on GitHub.

Introduction to NumPy

< More IPython Resources | Contents | Understanding Data Types in Python >

This chapter, along with chapter 3, outlines techniques for effectively loading, storing, and manipulating in-memory data in Python. The topic is very broad: datasets can come from a wide range of sources and a wide range of formats, including be collections of documents, collections of images, collections of sound clips, collections of numerical measurements, or nearly anything else. Despite this apparent heterogeneity, it will help us to think of all data fundamentally as arrays of numbers.

What is numpy

Very fast n-dimensional array objects
Create matrices of any dimension and datatype stored in memory efficiently.

Very fast methods for operating on arrays
Transform, reshape, slice, index, add, divide

Very fast library of functions for linear algebra, probability, and other operations.
random sample, eigen decomposition, map/reduce.

```
# multiply all elements of matrix by 2
# … doesn’t do what we might expect
mylist * 2
[[0,1,2,3],[4,5,6,7],[8,9,10,11],
 [0,1,2,3],[4,5,6,7],[8,9,10,11]]
```

How to: numpy

The first thing to do is to create or init an array. It can either be filled with data to begin with, or it can be empty (filled with null values like zeros).

```
# convention: import numpy and name it np
import numpy as np

# create an array
arr = np.array([1, 2, 3, 4])
array([1, 2, 3, 4])

# arrays print nicely
print(arr)
[1, 2, 3, 4]
```

How to: numpy

Multiple dimensional arrays are easy to generate. They can be made just like we did before with lists, or you can init an array using just **shape** arguments.

```
# create an array like we did with lists
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
[[1, 2, 3],
 [4, 5, 6]]

# get the dimensions of an array object
arr.shape
(2, 3)
```

How to: numpy

Multiple dimensional arrays are easy to generate. They can be made just like we did before with lists, or you can init an array using just **shape** arguments.

```
# get the dimensions of an array object
arr.shape
(2, 3)

# init an empty array of size (2,3)
arr = np.zeros((2, 3))
print(arr)
[[0, 0, 0],
 [0, 0, 0]]
```

Slicing a 2-d array

As before with lists we can select an **element** of an array, or select a **row**, but now we can also select **columns** of an array.

```
# an array created from a range of values
arr = np.array([[1, 2, 3], [4, 5, 6]])

# select element
arr[0, 0]
1

# select row
arr[0, :]
[1, 2, 3]

# select columns
arr[:, 1]
[2, 5]
```

Fancy slicing

As before with lists we can select an **element** of an array, or select a **row**, but now we can also select **columns** of an array.

```
# an array created from a range of values
arr = np.array([[1, 2, 3], [4, 5, 6]])

# get first row, last two columns
arr[0, -2:]
array([2, 3])

# get last row, then reverse its order
arr[-1][::-1]
array([6, 5, 4])
```

Fancy slicing

In a list you can index or slice, but you cannot select ordered elements with an index (although you could do so using list comprehension). But, in numpy you can do very easily.

```
# a list and an array
lst = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
arr = np.array(lst)

# get first, third and seventh index
lst[[1, 3, 7]]
TypeError: list indices must be integers..

# get first, third and seventh index
arr[[1, 3, 7]]
array([1, 3, 7])
```

Fancy slicing

Another way to get a slice from an array is using a boolean mask. This can be a very efficient way to

```
# create a boolean mask
mask = np.array([True, False, True, False])
arr = np.array([1, 2, 3, 4])

# only True elements will return
arr[mask]
array([1, 3])

# get the inverse selection
arr[np.invert(mask)]
array([2, 4])
```

Reshaping an array

A numpy array is technically called an **ndarray** object, because it can hold data in multiple dimensions. A 1-d array is like a list, a 2-d array is like a matrix or data table, a 3-d array can be thought of like a cube.

Data can be generated in a given shape, or reshaped into a different dimension later.

```
# init an array from a range of values
arr = np.arange(10)
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# reshape into a 2-d array of shape (5, 2)
arr.reshape((5, 2))
array([0, 1],
      [2, 3],
      [4, 5],
      [6, 7],
      [8, 9],
      [10, 11]])
```

Create then fill

A common workflow when working with arrays is to create an empty array of the dimension you wish and then fill it with values. This can be done with the **.zeros** function and a shape entered as a tuple.

```
# init an empty array of size (3, 5, 10)
arr = np.zeros((3, 5, 2))


# this is simply three five-by-two arrays

array([[[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]],

       [[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]],

       [[ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.],
        [ 0.,  0.]]])
```

Array attributes

An array is a Class object just like the ones that we've created in class. It has **functions** that you can call from it to operate on the array, and it has **attributes** that you can access to get info about the array.

Some very useful attributes of arrays that we will often want to know and access is its **size, shape,** and **dtype**

```
# get the array shape
arr = np.zeros((2, 50, 50))
arr.shape
(2, 50, 50)

# get the array size
arr.size
5000

# get the dtype (more on this soon)
arr.dtype
dtype('float64')
```

Numpy functions

Many operations can be accessed from an array Class object by calling <object>.<func> whereas many other functions are called directly from the numpy library with an array entered as the argument. For many functions, you do use either way.

```
# use the function sum from an array obj
arr = np.array([1, 2, 3, 4])
arr.sum()
10

# or, call sum from numpy on the array
np.sum(arr)
10
```

Numpy functions

Many operations can be accessed from an array Class object by calling <object>.<func> whereas many other functions are called directly from the numpy library with an array entered as the argument. For many functions, you can use either way.

```
# common statistical operations
arr = np.arange(20)

arr.min()         # 0
arr.max()         # 19
arr.mean()        # 9.5
arr.std()         # 5.766281297335398
arr.argmax()      # 0
arr.argmin()      # 19
```

Numpy functions

Functions can operate on **axes** of an array to operate on a row or column, or any dimension at a time.

This is again *super fast,* incredibly faster than accessing columns or rows from a list.

```
# common statistical operations, w/ axis
arr = np.array([
  [1,2,3,4],
  [5,6,7,8],
  [9,10,11,12]])

arr.sum(axis=0)     # [15, 18, 21, 24]
arr.sum(axis=0)     # [0, 26, 42]
arr.min(axis=0)     # [1, 2, 3, 4]
arr.min(axis=1)     # [1, 5, 9]
arr.mean(axis=1)    # [2.5, 6.5, 10.5]
```

Iterating over arrays

You can iterate over rows or columns, but it is often easiest to iterate over an index range and then select from the array by indexing. This works for arrays of any dimension.

But, using **.T** is an easy way to *transpose* an array (flip its dimensions).

```
for row in arr:
    print(row)
[1,2,3,4]
[5,6,7,8]

for col in arr.T:
    print(col)
[1,5]
[2,6]
[3,7]
[4,8]
```

Iterating over arrays

You can iterate over rows or columns, but it is often easiest to iterate over an index range and then select from the array by indexing. This works for arrays of any dimension.

But, using **.T** is an easy way to *transpose* an array (flip its dimensions).

```
# same as last page, but using indexing
for row in range(arr.shape[0]):
    print(arr[row, :])
[1,2,3,4]
[5,6,7,8]

for col in range(arr.shape[1]):
    print(arr[:, col])
[1,5]
[2,6]
[3,7]
[4,8]
```

Element-wise operations

As we said before, lists do not allow you to perform operations all at once to manipulate all elements, whereas in numpy you can do this.

```
# adding to arrays adds at each element
arr0 = np.array([0, 1, 2, 3])
arr1 = np.array([5, 4, 3, 2])
arr0 + arr1
[5, 5, 5 ,5]

# operations to array affect each element
arr0 * 2
[0, 2, 4, 6]
```

Join arrays

In a list you can index or slice, but you cannot select ordered elements with an index. You can only do so with list comprehension. However, In numpy you can do easily.

```
# a list and an array
d0 = [0, 1, 2, 3]
d1 = [4, 5, 6, 7]
arr = np.concatenate([arr0, arr1])
array([0,1,2,3,4,5,6,7])

# join along an axis
arr = np.stack([d0, d1], axis=0)
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
arr = np.stack([d0, d1], axis=1)
array([[0, 4],
       [1, 5],
       [2, 6],
       [3, 7]])
```

Data types (dtypes)

The dtype of an array is the format it uses to store data. As it said in your reading, all data can be thought of as values in an array, and those values are stored using **bits**, or binary digits. There are 8 bits in a **byte** (e.g., Gb of memory). Using more bits or bytes means using more memory.

The differences among dtypes are minimal for most small operations, but can matter for writing super speedy code, which we'll see later.

```
# default int dtype is int64
arr64 = np.array([0, 1, 2, 3])
arr.dtype
dtype('int64')

# for small values use int8 (1 byte)
arr8 = np.array([0, 1, 2, 3], dtype='int8')
arr.dtype
dtype('int8')

# the elements of an array are the same type
np.array([0, 1, 2])           # 'int64'
np.array([0, 1, 2, 7.3])      # 'float64'
np.array([0, 1, 2, 'cat'])    # '<U21'
```

Copy & return

There is an important difference between lists and arrays in the way that a slice or index is returned.

Lists return a **copy**, which is a separate object that can be modified.

Arrays return a **view**, which is a subset of the same object that when modified affects the original array.

```
# init a list and modify a slice of it
lst = [0, 1, 2, 3]
sub = lst[0:2]          # returns copy
sub[0] = 7
sub, lst
[7,1], [0,1,2,3])

# same in array, but both are modified!
arr = np.array([0, 1, 2, 3])
sub = arr[0:2]          # returns view
sub[0] = 7
print(sub, arr)
[7,1], [7,1,2,3]
```

Copy & return

There is an important difference between lists and arrays in the way that a slice or index is returned.

Lists return a **copy**, which is a separate object that can be modified.

Arrays return a **view**, which is a subset of the same object that when modified affects the original array.

```
# .copy() returns a copy of an array
arr = np.array([0, 1, 2, 3])
sub = arr[0:2].copy()   # returns copy
sub[0] = 7
print(sub, arr)
[7,1], [0,1,2,3]
```

numpy.random

One of your assignment notebook is all about the numpy.random library

```
# efficient functions for random sampling
np.random.choice(range(10),  4)
[0,5,2,7]

np.random.choice(list(“ACGT”)  6,
replace=True)
[‘A’,‘T’,‘C’,‘A’,‘A’,‘T’]
```

Pandas DataFrames

Pandas is another powerful library for manipulating data and doing statistics in Python. The primary Class object in pandas, the **DataFrame,** is a type of table that is similar to the most common datatype in R, and is also similar to the way we see data in excel spreadsheets.

A **DataFrame** is a ***labeled 2-d array***
A **Series** is a ***labeled 1-d array***

```
# convention: import and name pd
import pandas as pd


# the DataFrame object
arr = np.arange(20).reshape((5, 4))
means = arr.mean(axis=1)
data = pd.DataFrame({'mean': means})
```

```
   mean
0   1.5
1   5.5
2   9.5
3  13.5
4  17.5
```

Pandas Series

The Series object is the little brother to the DataFrame object in pandas. It is for small simple tables of data that include only 1 column.

It is a labeled 1-d array.

The **index** holds the data (row) labels

```
# init a Series object
s = pd.Series([1,2,3,4], index=list('abcd'))
```

```
a    1
b    2
c    3
d    4
dtype: int64
```

Pandas Series

A Series object can be sliced and indexed just like an array, but in addition to using index values it can also be accessed by index names.

```
# index by label
s['a']
1

# slice by labels
s['b':'d']
b    2
c    3
d    4
dtype: int64
```

Pandas Series

We can filter using ‘fancy indexing’

```
# filter operation returns a boolean mask
mask = s > 2
a    False
b    False
c    True
d    True
dtype: bool


# apply mask for ‘fancy indexing’
s[s>2]
c    3
d    4
dtype: int64
```

Pandas DataFrames

Two-dimensional labelled arrays

Several easy ways to create a DataFrame:

1) use Python dict to enter key:value pairs.

2) use ndarray with index names and transpose.

```
# create a DataFrame from 2 arrays
a = np.random.randint(0, 10, 5)
b = np.random.randint(0, 10, 5)
data = pd.DataFrame({'a': a, 'b':b})
```

```
   a  b
0  8  8
1  2  5
2  0  4
3  0  1
4  6  4
```

Init a DataFrame

Several easy ways to create a DataFrame:

1) use Python dict to enter key:value pairs.

2) use ndarray with index names and transpose.

```
# create a DataFrame from one 2-d array
c = np.random.randint(0, 10, (2,5))
pd.DataFrame(c, index=['a', 'b']).T
```

	a	b
0	8	8
1	2	5
2	0	4
3	0	1
4	6	4

Pandas is very friendly with missing data

Automatically fils NaN for columns with data for an index in one series but not matching in another.

```
# create a DataFrame from two Series
a = pd.Series({'a': 1, 'b':2, 'c':3})
b = pd.Series({'c': 5, 'd':8, 'e':9})
pd.DataFrame({'col1': a, 'col2': b})
```

```
    col1  col2
a   1.0   NaN
b   2.0   NaN
c   3.0   5.0
d   NaN   8.0
e   NaN   9.0
```

Operations

In general, pandas can do most of the same operations that numpy can for summarizing data in an array, such as *sum, mean, median, std,* and it can also calculate these values over an axis argument.

dataframe is returned and *displayed* in jupyter as a nice rendered HTML table.

```
# create a DataFrame from 2 arrays
a = np.random.randint(0, 10, 5)
b = np.random.randint(0, 10, 5)
data = pd.DataFrame({‘a’: a, ‘b’:b})

data.mean(axis=1)
0    8.0
1    3.5
2    2.0
3    0.5
4    5.0
dtype: float64
```

Very good for reading data files

We’ll see in the assignment notebooks that pandas has some really useful functions for reading in data tables from CSV files, or other formats.

```
# load DataFrame from CSV file
data = pd.read_csv(‘table.csv’)

# load with many args for diff formats
data = pd.read_csv(
    ‘table.csv’
    sep=‘\t’,
    header=None,
    index_col=1)
```

Load both tables and combine into one

As 'table.csv'

height	width
1	2
3	4
5	6
7	8
9	10
11	12

As 'table2.csv'

length	girth
10	100
20	200
30	300
40	400
50	500
60	600

```
# load DataFrames from CSV files
d1 = pd.read_csv(‘table.csv’)
d2 = pd.read_csv(‘table2.csv’)


# try to concatenate them
data = pd.concat([d1, d2])
```

```
   girth  height  length  width
0    NaN   1.0    NaN    2.0
1    NaN   3.0    NaN    4.0
2    NaN   5.0    NaN    6.0
3    NaN   7.0    NaN    8.0
4    NaN   9.0    NaN    10.0
5    NaN   11.0   NaN    12.0
0  100.0   NaN    10.0   NaN
1  200.0   NaN    20.0   NaN
2  300.0   NaN    30.0   NaN
3  400.0   NaN    40.0   NaN
4  500.0   NaN    50.0   NaN
5  600.0   NaN    60.0   NaN
```

Concat on axis=1

As 'table.csv'

height	width
1	2
3	4
5	6
7	8
9	10
11	12

As 'table2.csv'

length	girth
10	100
20	200
30	300
40	400
50	500
60	600

```
# load DataFrames from CSV files
d1 = pd.read_csv(‘table.csv’)
d2 = pd.read_csv(‘table2.csv’)


# concatenate along axis
data = pd.concat([d1, d2], axis=1)
   height  width  girth  length
0     1      2     100    10
1     3      4     200    20
2     5      6     300    30
3     7      8     400    40
4     9      10    500    50
5     11     12    600    60
```

Explore your data

We'll see in the assignment notebooks that pandas has some really useful functions for reading in data tables from CSV files, or other formats.

```
# load DataFrames from URL
url='http://eaton-lab.org/data/iris-\
     data-dirty.csv'
d1 = pd.read_csv(url)


# get top and bottom entries, summary
d1.head()
d1.tail()
d1.describe()
```

```
              0          1            2            3
count  150.000000  148.000000  150.000000
150.000000
mean  5.843333   3.058108   3.758667   1.198667
std   0.828066   0.434094   1.764420   0.763161
min   4.300000   2.000000   1.000000   0.100000
25%   5.100000   2.800000   1.600000   0.300000
50%   5.800000   3.000000   4.350000   1.300000
75%   6.400000   3.300000   5.100000   1.800000
max   7.900000   4.400000   6.900000   2.500000
```

Efficiency and use

Pandas is very fast and efficient, but *numpy is still much faster.* This is simply because dataframes hold much more information (labels, etc.) whereas ndarrays are barebones data.

For working with really big data use numpy. If working on small datasets with labels use pandas. Even when working with numpy use pandas to dress up your results to look pretty.

```
# create an array and DF with same data
arr = np.arange(20).reshape((5, 4))
df = pd.DataFrame(arr)

# compare speed of numpy and pandas
%timeit arr.mean(axis=0)
9.06 µs ± 1.19 µs per loop

%timeit df.mean(axis=0)
103 µs ± 4.52 µs per loop
```

Assignments

Fork and clone today's repo:
6-scientific-python

Read and execute these notebook to learn more numpy, scipy, and pandas skills:

1) nb-6.2-numpy-random.ipynb
2) nb-6.3-scipy.ipynb
3) nb-6.4-pandas-intro.ipynb

Assignment: write an importable Python package (using text-editor or jupyter) that accomplishes a set of defined tasks using numpy & pandas. Push package to your repo, and pull-request your notebook.

1) nb-6.5-assignment.ipynb

```
# write a class object to analyze DNA
import seqlib
dna = seqlib.Seqlib(10, 30)


# functions will operate on numpy arrays
stats = dna.calculate_statistics()


# results will be formatted as DataFrames
nucleotide_diversity
0    0.01
1    0.05
2    0.90
3    0.55
4    0.04
dtype: float64
```

Assignments and readings

Assignment is due Friday at 5pm

Code review is due Monday by class.

Assignment: Link to Session 6 repo
Readings: See syllabus
Collaborate: Work together in this gitter chatroom