Basic data structures - start with Series then build up to DataFrames

Pandas quick start guide for Series

- A **Series** is a 1D array that can hold any type of data (numeric types, non-numeric, Python objects and so forth).
  - Each entry is **labeled** with an index that is used to keep track of what each entry is, and can be used to lookup the value corresponding to each index during analysis (remember keys in dictionaries? similar idea)
  - These labels are fixed - they will always index the same value unless you explicitly break that link.
  - The list of labels that forms the index can either be declared upon series creation or, by default, it will range from 0 to len(data)-1.
    - If you're going to use Pandas to organize your data, specifying usable and informative labels is a good idea because that's one of the main advantages of organizing your data in this manner

Warning. Pandas will allow you to specify non-unique labels. This can be ok for operations that don't rely on indexing by label. However, operations that do rely on unique labels for indexing may lead to unexpected problems so in general its good practice to use unique labels!

Import Pandas and random

In [ ]:
```
# import a generic pandas object and also a few specific functions that we'll use
import pandas as pd
import random as random
```

Create a series of data stored in a list, and then make a set of index labels

In [ ]:
```
# For this simulation, lets have 12 subjects, and some data
# generated psuedo-randomly from a uniform distribution
N = 12

# generate N random numbers
data=[]
for i in range(N):
  data.append(random.randint(1,10))

# have a look
print(data)
```

Make a list of subject names for use as index labels

In [ ]:
```
label_prefix = 'Sub'
index=[]
for n in range(0,N):
    index.append(label_prefix+str(n))

# print our list of index labels
print('Index labels: ', index, '\n')
```

Then make our Pandas Series by passing in our data array and our index labels

In [ ]:
```
s = pd.Series(data, index=index)
print(s)
```

Note that each subject is now a field in the series and can be used to retrieve the corresponding value...there are a few ways to do this

- can access by number
- can access by field
- can access by index label

In [ ]:
```
print(s[0])
```

In [ ]:
```
# access by field
print(s.Sub11)
```

In [ ]:
```
# access by index label
print(s['Sub0'])
```

Can also use labels to check for membership or to index over labels

In [ ]:
```
# check for membership
print('Sub11' in s)
```

iterate over index labels

In [ ]:
```
for i in s.index:
  print(i)
```

iterate over values...

In [ ]:
```
for v in s.values:
  print(v)
```

In [ ]:
```
# can also get to the values more directly like this:
for d in s:
  print(d)
```

Cover a few other optional (but important) parameters of the pd.Series call

- dtype - default is to infer the data type (int32, float64, str, etc) based on the values in data
  - However, can also explicitly declare the data
  - This can be good if you want to, for example, re-cast the data to save space or to make types compatible
  - But this may also have important negative consequences if not done thoughtfully!

Example: change from int to str

- Note that the dtype of series 's' is now an 'object'. This is the Pandas version of a Python 'str'

Make up some data and corresponding labels to play with

In [ ]:
```
data = [10, 23, 88, 43, 29]
labels = [0,1,2,3,4]
```

In [ ]:
```
# make a series with the data array from above, but this time make it a str
# instead of the inferred int64 type
s = pd.Series(data, index=labels, dtype='str')

# we're now
# all set to do a bunch of str operations without having to deal with
# recasting each time we interact with the values in s
print(s[0]=='10')
```

Re-make our series as int64 before moving on because we'll want them to be ints for the next several cells.

In [ ]:
```
s = pd.Series(data, index=labels, dtype='int64')
```

Slicing a Pandas series

- start, stop, step notation from lists...

In [ ]:
```
# first 3 values - notice that you get the label along with the values
print(s[:3])
```

In [ ]:
```
s[2:-1]    # 3rd entry to len(s)-1
```

In [ ]:
```
# reverse, etc
s[::-1]
```

In [ ]:
```
# every other, etc
s[::2]
```

Another example using more advanced slicing...

- this is super handy when cleaning data to exclude outliers!

In [ ]:
```
s[s>=20]    #all entries greater than or equal to 30
```

Find values within a range

In [ ]:
```
s[(s>=20) & (s<=45)]
```

There is also the 'between' method to find values within a range

- the 'between' method will return True/False depending on whether each entry falls in between the bounds.
- can then use that index to find values within a range!

In [ ]:
```
ind = s.between(23, 45, inclusive=False)
s[ind]
```

Series objects have many built in operations

list of attributes and methods

In [ ]:
```
# shouldn't need to re-run, but make sure that you've got int64 data here (and
# not str)
s = pd.Series(data, index=labels, dtype='int64')
```

In [ ]:
```
# attributes
print('Data Type: ', s.dtype)
```

In [ ]:
```
# basic methods
print('Mean: ', s.mean(), ' Std:', s.std(), 'Max: ', s.max())
```

In [ ]:
```
# numerical derivative
print('Diff: ', s.diff())
```

Find the mean of all values that fall within a range...

- can also apply other methods to compute std, etc after filtering

In [ ]:
```
s[(s>=10) & (s<=45)].mean()
```