Elements Of Data Science - F2020

Week 2: Python Intro/Review, Numpy and Pandas

9/20/2020

TODOs

- **Review** PDSH Chapter 2
- **Read** PDSH Chapter 3
- **Skim** PDSH Chapter 4

- Complete Week 2 Quiz

TODOs

- Ch 2. Introduction to NumPy
  - Understanding Data Types in Python
  - The Basics of NumPy Arrays
  - Computation on NumPy Arrays: Universal Functions
  - Aggregations: Min, Max, and Everything In Between
  - Computation on Arrays: Broadcasting
  - Comparisons, Masks, and Boolean Logic
  - Fancy Indexing
  - Sorting Arrays
  - Structured Data: NumPy's Structured Arrays

TODOs

- Ch 3. Data Manipulation with Pandas
  - Introducing Pandas Objects
  - Data Indexing and Selection
  - Operating on Data in Pandas
  - Handling Missing Data
  - Hierarchical Indexing
  - Combining Datasets: Concat and Append
  - Combining Datasets: Merge and Join
  - Aggregation and Grouping
  - Pivot Tables
  - Vectorized String Operations
  - Working with Time Series
  - High-Performance Pandas: eval() and query()

TODOs

- Ch 4. Visualization with Matplotlib
  - Simple Line Plots
  - Simple Scatter Plots
  - Visualizing Errors
  - Density and Contour Plots
  - Histograms, Binnings, and Density
  - Customizing Plot Legends
  - Customizing Colorbars
  - Multiple Subplots
  - Text and Annotation
  - Customizing Ticks
  - Customizing Matplotlib: Configurations and Stylesheets
  - Three-Dimensional Plotting in Matplotlib
  - Geographic Data with Basemap
  - Visualization with Seaborn

Getting Changes from Git

```
$ cd [to_repository]
$ git pull
```

Questions?

TODAY

- Python Intro/Review
- Numpy
- Pandas

Python (Review?)

- Dynamic Typing
- Whitespace Formatting
- Basic Data Types
- Functions
- String Formatting
- Exceptions and Try-Except
- Truthiness
- Comparisons and Logical Operators
- Control Flow
- Assert
- Sorting
- List/Dict Comprehensions
- Importing Modules
- collections Module
- Object Oriented Programming

Dynamic Typing

In [1]:
```
x = 3
x = 3.14
x = 'apple'
x
```

Out[1]:
```
'apple'
```

In [2]:
```
# determine the current variable type
type(x)
```

Out[2]:
```
str
```

Basic Python Data Types

- **int** : `42`
- **float** : `4.2`, `4e2`
- **boolean (bool)** : `True`, `False`
- **string (str)** : `'num 42'`, `"num 42"`
- **none/null**: `None`
- also `long`, `complex`, `bytes`, etc.

Whitespace Formatting

- Instead of braces or brackets to delimit blocks, whitespace

```
# The pound sign marks the start of a comment. Python itself
# ignores the comments, but they're helpful for anyone reading the code.
for i in [1, 2, 3, 4, 5]:
    print(i)                        # first line in "for i" block
    for j in [1, 2, 3, 4, 5]:
        print(j)                    # first line in "for j" block
        print(i + j)                # last line in "for j" block
    print(i)                        # last line in "for i" block
print("done looping")
```

- 4 space indentations are conventional
- Style Guide : PEP 8 (https://www.python.org/dev/peps/pep-0008/)

Functions

In [3]:
```
def add_two(x):
    """Adds 2 to the number passed in."""
    return x+2


add_two(2)
```

Out[3]:
```
4
```

In [4]:
```
help(add_two)
```

```
Help on function add_two in module __main__:

add_two(x)
    Adds 2 to the number passed in.
```

Also try `add_two?` in jupyter

In [5]:
```
# add_two? # show docstring
```

In [6]:
```
# add_two?? # show code as well
```

Function Arguments

- can assign defaults

In [7]:
```
def increment(x,amount=1):
    """Increment a value, default by 1."""
    return x+amount

increment(2)
```

Out[7]: 3

In [8]:
```
increment(2,amount=2)
```

Out[8]: 4

Function Arguments Cont.

- **positional arguments** must be entered in order

In [9]:
```
def subtract(x,y):
    return x-y

subtract(3,1)
```

Out[9]: 2

- **keyword arguments** must follow positional
- can be called in any order

In [10]:
```
def proportion(numer,denom,precision=2):
    return round(numer/denom,precision)

proportion(2,precision=2,denom=3)
```

Out[10]: 0.67

String Formatting

In [11]:
```
x = 3.1415

'the value of x is ' + str(x)
```

Out[11]: 'the value of x is 3.1415'

In [12]:
```
'the value of x is %0.2f' % x
```

Out[12]: 'the value of x is 3.14'

In [13]:
```
'the value of x is {:0.2f}'.format(x)
```

Out[13]: 'the value of x is 3.14'

In [14]:
```
f'the value of x is {x:0.2f}'
```

Out[14]: 'the value of x is 3.14'

- often print variable values for debugging

In [15]:
```
f'{x:0.2f}'
```

Out[15]: '3.14'

In [16]:
```
f'{x=:0.2f}' # new in 3.8
```

Out[16]: 'x=3.14'

String Formatting Cont.

In [17]:
```
"""This is a multiline comment.
The value of x is {}.""".format(x)
```

Out[17]:
```
'This is a multiline comment.\nThe value of x is 3.1415.'
```

In [18]:
```
print("""This is a multiline comment.
The value of x is {}.""".format(x))
```

```
This is a multiline comment.
The value of x is 3.1415.
```

- common specifiers: `%s strings, %d integers, %f floats`

In [19]:
```
x='apple'
f'the plural of {x:10s} is {x+"s"}'
```

Out[19]:
```
'the plural of apple      is apples'
```

In [20]:
```
x = 3
f'the square of {x:10d} is {x**2}'
```

Out[20]:
```
'the square of          3 is 9'
```

- to learn more https://realpython.com/python-string-formatting/

Python Data Types Continued: `list`

In [21]:
```
x = [42,'e',2.0]
x
```

Out[21]: [42, 'e', 2.0]

In [22]:
```
x[0] # indexing
```

Out[22]: 42

In [23]:
```
x[-1] # reverse indexing
```

Out[23]: 2.0

In [24]:
```
x[0] = 4 # assignment
x[0]
```

Out[24]: 4

In [25]:
```
x.append('a') # add to
x
```

Out[25]: [4, 'e', 2.0, 'a']

In [26]:
```
x.pop(1) # remove/delete
```

Out[26]: 'e'

In [27]:
```
x
```

Out[27]: [4, 2.0, 'a']

Python Data Types Continued: Dictionary / `dict`

In [28]:
```
x = {'b':2,'a':1,'c':4} # or x = dict(b=2,a=1,c=4)
x # NOTE: order is not guaranteed!
```

Out[28]:
```
{'b': 2, 'a': 1, 'c': 4}
```

In [29]:
```
x['b'] # indexing`
```

Out[29]:
```
2
```

In [30]:
```
x['d'] = 3 # assignment`
x
```

Out[30]:
```
{'b': 2, 'a': 1, 'c': 4, 'd': 3}
```

In [31]:
```
x.pop('d',None) # remove/delete
```

Out[31]:
```
3
```

In [32]:
```
x
```

Out[32]:
```
{'b': 2, 'a': 1, 'c': 4}
```

Python Data Types Continued: Dictionary / `dict` Cont.

```
In [33]: x

Out[33]: {'b': 2, 'a': 1, 'c': 4}

In [34]: x.keys()

Out[34]: dict_keys(['b', 'a', 'c'])

In [35]: x.values()

Out[35]: dict_values([2, 1, 4])

In [36]: x.items()

Out[36]: dict_items([('b', 2), ('a', 1), ('c', 4)])

In [37]: list(x.items())

Out[37]: [('b', 2), ('a', 1), ('c', 4)]
```

Python Data Types Continued: `tuple`

- like a list, but **immutable**

In [38]:
```
x = (2,'e')
x
```

Out[38]:
```
(2, 'e')
```

In [39]:
```
x[0] # indexing
```

Out[39]:
```
2
```

In [40]:
```
x[0] = 3 # assignment? Nope, error: immutable`
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-8666b70a2e1c> in <module>
----> 1 x[0] = 3 # assignment? Nope, error: immutable`

TypeError: 'tuple' object does not support item assignment
```

Python Data Types Continued: `set`

In [41]:
```
x = {2,'e','e'} # or set([2,'e','e'])
x
```

Out[41]:
```
{2, 'e'}
```

In [42]:
```
x.add(1) # insert
x
```

Out[42]:
```
{1, 2, 'e'}
```

In [43]:
```
x.remove('e') # remove/delete
x
```

Out[43]:
```
{1, 2}
```

In [44]:
```
x.intersection({2,3})
```

Out[44]:
```
{2}
```

In [45]:
```
x.difference({2,3})
```

Out[45]:
```
{1}
```

In [46]:
```
x[0] # cannot index into a set
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-46-06a9d9e044b1> in <module>
----> 1 x[0] # cannot index into a set

TypeError: 'set' object is not subscriptable
```

Determining Length with `len`

In [47]:
```
len([1,2,3])
```

Out[47]: 3

In [48]:
```
len({'a':1,'b':2,'c':3})
```

Out[48]: 3

In [49]:
```
len('apple')
```

Out[49]: 5

In [50]:
```
len(True)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-50-8c3be77e04f4> in <module>
----> 1 len(True)

TypeError: object of type 'bool' has no len()
```

Exceptions

```
In [51]: 'a' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-51-63196ba7153f> in <module>
----> 1 'a' + 2

TypeError: can only concatenate str (not "int") to str
```

Other common exceptions:

- SyntaxError
- IndentationError
- ValueError
- TypeError
- IndexError
- KeyError
- and many more https://docs.python.org/3/library/exceptions.html

Catching Exceptions with `try-except`

In [52]:
```
try:
    'a' + 2
except TypeError as e:
    print(f"We did this on purpose, and here's what's wrong:\n{e}")
```

```
We did this on purpose, and here's what's wrong:
can only concatenate str (not "int") to str
```

In [53]:
```
try:
    set([1,2,3])[0]
except SyntaxError as e:
    print(f"Print this if there's a syntax error")
except Exception as e:
    print(f"Print this for any other error")
```

```
Print this for any other error
```

Truthiness

- boolean: `True`, `False`

- These all translate to `False`:
  - `None`
  - `[]`
  - `{}`
  - `''`
  - `set()`
  - `0`
  - `0.0`

Comparison Operators

- equality: `==`
- inequality: `!=`

In [54]:
```
3 == 3
```

Out[54]: True

In [55]:
```
3 != 4
```

Out[55]: True

- less than: `<`
- greater than: `>`
- '(less than/greater than) or equal to: `<=` , `>=`

In [56]:
```
3 < 4
```

Out[56]: True

Logical Operators

- logical operators: `and`, `or`, `not`

In [57]:
```
( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

Out[57]: True

- `any()`: at least one element is true

In [58]:
```
any([0,0,1])
```

Out[58]: True

- `all()`: all elements are true

In [59]:
```
all([0,0,1])
```

Out[59]: False

- **bitwise operators** (we'll see these in numpy and pandas) : `&` (and), `|` (or), `~` (not)

Assert

Assert

- use `assert` to test anything we know should be true
- simple unit test
- raises exception when assertion is false, otherwise nothing

Assert

- use `assert` to test anything we know should be true
- simple unit test
- raises exception when assertion is false, otherwise nothing

In [60]:
```
assert increment(2,2) == 4
```

Assert

- use `assert` to test anything we know should be true
- simple unit test
- raises exception when assertion is false, otherwise nothing

In [60]:
```
assert increment(2,2) == 4
```

In [61]:
```
assert 1 == 0
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-61-e99f91a18d62> in <module>
----> 1 assert 1 == 0

AssertionError:
```

Assert

- use `assert` to test anything we know should be true
- simple unit test
- raises exception when assertion is false, otherwise nothing

In [60]:
```
assert increment(2,2) == 4
```

In [61]:
```
assert 1 == 0
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-61-e99f91a18d62> in <module>
----> 1 assert 1 == 0

AssertionError:
```

In [62]:
```
assert 1 == 0, "1 does not equal 0"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
<ipython-input-62-6f246726711b> in <module>
----> 1 assert 1 == 0, "1 does not equal 0"

AssertionError: 1 does not equal 0
```

Control Flow: `if`:`elif`:`else`

- `if` then `elif` then `else`

In [63]:
```
x = 3
if x > 0:
    print('x > 0')
elif x < 0:
    print('x < 0')
else:
    print('x == 0')
```

```
x > 0
```

- single line `if` then `else`

In [64]:
```
print("x < 0") if (x < 0) else print("x >= 0")
```

```
x >= 0
```

More Control Flow: `for` and `while`

- `for` loop

In [65]:
```
a = []
for x in [0,1,2]: # for variable in some_iterable
    a.append(x)
a
```

Out[65]:
```
[0, 1, 2]
```

- `while` loop

In [66]:
```
x = 0
while x < 3:
    x += 1
x
```

Out[66]:
```
3
```

More Control Flow: `break` and `continue`

- `break` : break out of current loop

In [67]:
```
x = 0
while True:
    x += 1
    if x == 3:
        break
x
```

Out[67]:
```
3
```

- `continue` : continue immediately to next iteration of loop

In [68]:
```
for x in range(3):
    if x == 1:
        continue
    print(x)
```

```
0
2
```

Generate a Range of Numbers: `range`

In [69]:
```
# create list of integers from 0 up to but not including 4
a = []
for x in range(4):
    a.append(x)
a
```

Out[69]:
```
[0, 1, 2, 3]
```

In [70]:
```
list(range(4))
```

Out[70]:
```
[0, 1, 2, 3]
```

In [71]:
```
list(range(3,5)) # with a start and end+1
```

Out[71]:
```
[3, 4]
```

In [72]:
```
list(range(0,10,2)) # with start, end+1 and step-size
```

Out[72]:
```
[0, 2, 4, 6, 8]
```

Keep track of list index or for-loop iteration: `enumerate`

In [73]:
```
for i,x in enumerate(['a','b','c']):
    print(i,x)
```

```
0 a
1 b
2 c
```

In [74]:
```
list(enumerate(['a','b','c']))
```

Out[74]:
```
[(0, 'a'), (1, 'b'), (2, 'c')]
```

Sorting

Two ways to sort a list:

1. by changing the list itself: `list.sort()`

In [75]:
```
x = [4,1,2,3]
x.sort()
assert x == [1,2,3,4]
```

2. without changing the list: `sorted()`

In [76]:
```
x = [4,1,2,3]
y = sorted(x)
assert x == [4,1,2,3]
assert y == [1,2,3,4]
```

Sorting Cont.

- To sort descending, use `reverse=True`:

In [77]:
```
assert sorted([1,2,3,4], reverse=True) == [4,3,2,1]
```

- Pass a `lambda` function to 'key' to specify what to sort by:

In [78]:
```
d = {'a':3,'b':5,'c':1}

s = sorted(d.items(), key=lambda x: x[1])

assert s == [('c', 1), ('a', 3), ('b', 5)]
```

List Comprehensions

- Like a single line for loop over a list or other iterable

In [79]:
```
# which numbers between 0 and 3 inclusive are divisible by 2?
is_even = []
for x in range(0,4):
    is_even.append(x%2 == 0)
is_even
```

Out[79]:
```
[True, False, True, False]
```

In [80]:
```
[x%2 == 0 for x in range(0,4)] # using a list comprehension
```

Out[80]:
```
[True, False, True, False]
```

In [81]:
```
# what are the indices of the vowels in 'apple'?
vowels = ['a','e','i','o','u']
[i for i,x in enumerate('apple') if x in vowels]
```

Out[81]:
```
[0, 4]
```

Dictionary Comprehension

- list comprehension but for key,value pairs
- can add logic to dictionary creation

In [82]:
```
pairs = [(1,'e'),(2,'f'),(3,'g')]
```

In [83]:
```
dict(pairs)
```

Out[83]:
```
{1: 'e', 2: 'f', 3: 'g'}
```

In [84]:
```
# modify value and only include odd keys
{key:'value_'+val for key,val in pairs if key%2 == 1}
```

Out[84]:
```
{1: 'value_e', 3: 'value_g'}
```

Object Oriented Programming

In [87]:
```python
class MyClass:
    """A descriptive docstring."""

    # constructor
    def __init__(self,myvalue = 0):  # what happens when created
        # attributes
        self.myvalue = myvalue

    def __repr__(self):   # what gets printed out (string repr.)
        return f'MyClass(myvalue={self.myvalue})'

    # any other methods
    def get_value(self):
        """Return the value in myvalue."""
        return self.myvalue
```

In [88]:
```python
x = MyClass(100) # instantiate object

assert x.get_value() == 100 # use object method
```

Importing Modules

- Want to calculate square root?

In [89]:
```
import math
math.sqrt(2)
```

Out[89]:
```
1.4142135623730951
```

- Want to import a submodule or function?

In [90]:
```
from math import sqrt,floor
print(sqrt(2))
print(floor(sqrt(2)))
```

```
1.4142135623730951
1
```

Importing Modules Cont.

- Want to import a module using an alias?

In [91]:
```
import math as m
m.sqrt(2)
```

Out[91]:
```
1.4142135623730951
```

- Don't do!

```
from math import *
# what if there is a math.print() function?
# what happens when we then call print()?
```

`collections` Module

In [92]:
```
from collections import Counter, defaultdict, OrderedDict
```

- `Counter` : useful for counting hashable objects
- `defaultdict` : create dictionaries without checking keys
- `OrderedDict` : key,value pairs returned in order added

- others : https://docs.python.org/3.7/library/collections.html

`collections` Module: `Counter`

In [93]:
```
c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
c
```

Out[93]:
```
Counter({'red': 2, 'blue': 3, 'green': 1})
```

In [94]:
```
c = Counter()
for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
    c[word] += 1
```

In [95]:
```
c.most_common()
```

Out[95]:
```
[('blue', 3), ('red', 2), ('green', 1)]
```

`collections` Module Cont. : `defaultdict`

In [97]:
```
# create mapping from length of word to list of words
colors = ['red', 'blue', 'purple', 'gold', 'orange']
d = {}
for word in colors:
    d[len(word)].append(word)
```

```
KeyError: 3
```

In [98]:
```
d = {}
for word in colors:
    if len(word) in d:
        d[len(word)].append(word)
    else:
        d[len(word)] = [word]
d
```

Out[98]:
```
{3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}
```

In [99]:
```
d = defaultdict(list)

for word in colors:
     d[len(word)].append(word)
d
```

Out[99]:
```
defaultdict(list, {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']})
```

Contexts

- a context is like applying a scope with helper functions
- For example: open and write to a file

In [100]:
```
with open('tmp_context_example.txt','w') as f:
    f.write('test')
```

In [101]:
```
# instead of
f = open('tmp_context_example.txt','w')
f.write('test')
f.close() # this is easy to forget to do
```

In [102]:
```
# remove the example file we just created
%rm tmp_context_example.txt
```

Questions?

Working with Data

Want to:

- transform and select data quickly (numpy)
- manipulate datasets: load, save, group, join, etc. (pandas)
- keep things organized (pandas)

Intro to NumPy

Provides (from numpy.org):

- a powerful N-dimensional array object
- sophisticated (broadcasting) functions
- linear algebra and random number capabilities
- (Fourier transform, tools for integrating C/C++ and Fortran code, etc.)

Python Dynamic Typing

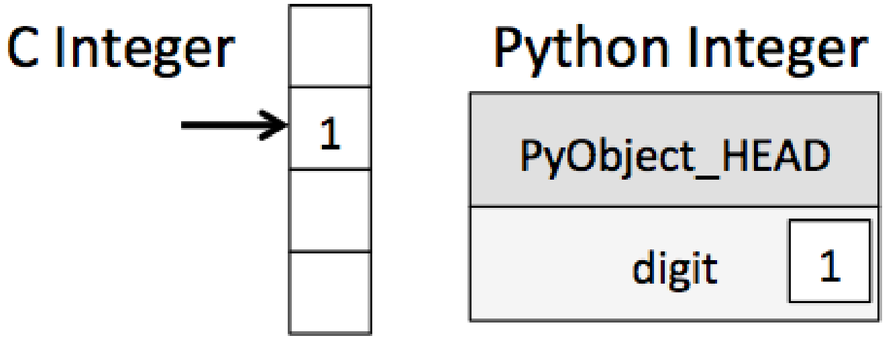
In [103]:
```
x = 5
x = 'five'
```

- Note: still *strongly* typed

In [104]:
```
x,y = 5,'five'
x+y
```

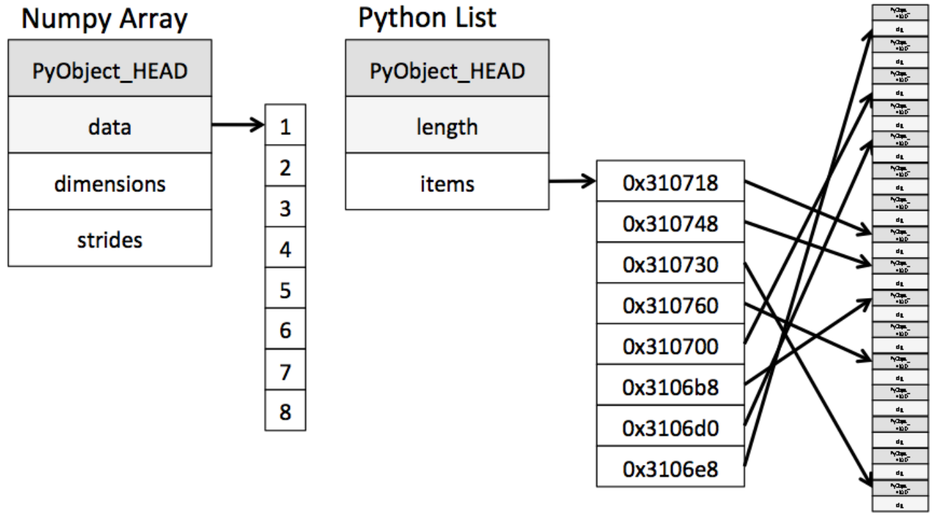
```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Python Dynamic Typing



PDHS Chap 2.

NumPy Array vs Python List



PDHS Chap 2.

Importing NumPy

Often imported as alias `np`

In [105]:
```
import numpy as np

np.random.randint(10,size=5)
```

Out[105]:
```
array([3, 5, 4, 3, 0])
```

NumPy Datatypes

```
bool_       Boolean (True or False) stored as a byte
int_        Default integer type (same as C long; normally either int64 or int32)
intc        Identical to C int (normally int32 or int64)
intp        Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8        Byte (-128 to 127)
int16       Integer (-32768 to 32767)
int32       Integer (-2147483648 to 2147483647)
int64       Integer (-9223372036854775808 to 9223372036854775807)
uint8       Unsigned integer (0 to 255)
uint16      Unsigned integer (0 to 65535)
uint32      Unsigned integer (0 to 4294967295)
uint64      Unsigned integer (0 to 18446744073709551615)
float_      Shorthand for float64.
float16     Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32     Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
float64     Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_    Shorthand for complex128.
complex64   Complex number, represented by two 32-bit floats
complex128  Complex number, represented by two 64-bit floats
```

NumPy Arrays

In [106]:
```
x = np.array([1,2,3])
x
```

Out[106]:
```
array([1, 2, 3])
```

In [107]:
```
# use dtype to show the datatype of the array
x.dtype
```

Out[107]:
```
dtype('int64')
```

In [108]:
```
# np arrays can only contain one datatype
x = np.array([1,'two',3])
x
```

Out[108]:
```
array(['1', 'two', '3'], dtype='<U21')
```

In [109]:
```
x.dtype
```

Out[109]:
```
dtype('<U21')
```

In [203]:
```
# many different ways to create numpy arrays
np.ones(5,dtype=float)
```

Out[203]:
```
array([1., 1., 1., 1., 1.])
```

NumPy Array Indexing

- For single indices, works the same as list

In [113]:
```
x = np.arange(1,6)
x
```

Out[113]:
```
array([1, 2, 3, 4, 5])
```

In [114]:
```
x[0], x[-1], x[-2]
```

Out[114]:
```
(1, 5, 4)
```

NumPy Array Slicing

In [115]:
```
x = np.arange(5)
x
```

Out[115]:
```
array([0, 1, 2, 3, 4])
```

In [116]:
```
# return first two items, start:end (exclusive)
x[0:2]
```

Out[116]:
```
array([0, 1])
```

In [117]:
```
# missing start implies position 0
x[:2]
```

Out[117]:
```
array([0, 1])
```

In [118]:
```
# missing end implies length of array
x[2:]
```

Out[118]:
```
array([2, 3, 4])
```

In [119]:
```
# return last two items
x[-2:]
```

Out[119]:
```
array([3, 4])
```

NumPy Array Slicing with Steps

In [120]:
```
x
```

Out[120]:
```
array([0, 1, 2, 3, 4])
```

In [121]:
```
# return every other item from position 1 to 4 exclusive
# start:end:step_size
x[1:4:2]
```

Out[121]:
```
array([1, 3])
```

Reverse array with step-size of -1

In [122]:
```
x[::-1]
```

Out[122]:
```
array([4, 3, 2, 1, 0])
```

NumPy Fancy Indexing

- Accessing multiple, non-consective indices at once using a list

In [123]:
```
x = np.arange(5,10)
x
```

Out[123]:
```
array([5, 6, 7, 8, 9])
```

In [124]:
```
x[[0,3]]
```

Out[124]:
```
array([5, 8])
```

In [125]:
```
x[[0,2,-1]]
```

Out[125]:
```
array([5, 7, 9])
```

Boolean Indexing

In [126]:
```
x
```

Out[126]:
```
array([5, 6, 7, 8, 9])
```

In [127]:
```
# Which indices have a value divisible by 2?
# mod operator % returns remainder of division
x%2 == 0
```

Out[127]:
```
array([False,  True, False,  True, False])
```

In [128]:
```
# Which values are divisible by 2?
x[x%2 == 0]
```

Out[128]:
```
array([6, 8])
```

In [129]:
```
# Which values are greater than 6?
x[x > 6]
```

Out[129]:
```
array([7, 8, 9])
```

Boolean Indexing And Bitwise Operators

In [130]:
```
x
```

Out[130]:
```
array([5, 6, 7, 8, 9])
```

In [131]:
```
(x%2 == 0)
```

Out[131]:
```
array([False,  True, False,  True, False])
```

In [132]:
```
(x > 6)
```

Out[132]:
```
array([False, False,  True,  True,  True])
```

In [133]:
```
# Which values are divisible by 2 AND greater than 6?
# 'and' expexts both elements be boolean, not arrays of booleans!
(x%2 == 0) and (x > 6)
```

```
ValueError: The truth value of an array with more than one element is ambiguous. Use a.any() or a.all()
```

In [134]:
```
# & compares each element pairwise
(x%2 == 0) & (x > 6)
```

Out[134]:
```
array([False, False, False,  True, False])
```

In [135]:
```
x[(x%2 == 0) & (x > 6)]
```

Out[135]:
```
array([8])
```

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

In [136]:
```
# Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

Out[136]:
```
array([8])
```

- or : `|` (pipe)

In [137]:
```
# which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

Out[137]:
```
array([6, 7, 8, 9])
```

- not : `~` (tilde)

In [138]:
```
# which values are NOT (even OR greater than 6)
x[~( (x%2 == 0) | (x > 6) )]
```

Out[138]:
```
array([5])
```

- see PDHS for more info

Indexing Review

- standard array indexing (including reverse/negative)
- slicing (start,end,step-size)
- fancy indexing (list/array of indices)
- boolean indexing (list/array of booleans)

Multidimensional Lists

In [139]:
```
x = [[1,2,3],[4,5,6]] # list of lists
x
```

Out[139]:
```
[[1, 2, 3], [4, 5, 6]]
```

In [140]:
```
# return first row
x[0]
```

Out[140]:
```
[1, 2, 3]
```

In [141]:
```
# return first row, second column
x[0][1]
```

Out[141]:
```
2
```

In [142]:
```
# return second column?
[row[1] for row in x]
```

Out[142]:
```
[2, 5]
```

NumPy Multidimensional Arrays

In [143]:
```
x = np.array([[1,2,3],[4,5,6]])
x
```

Out[143]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [144]:
```
x[0,1] # first row, first column
```

Out[144]:
```
2
```

In [145]:
```
x[0,0:3] # first row
```

Out[145]:
```
array([1, 2, 3])
```

In [146]:
```
x[0,:] # first row (first to last column)
```

Out[146]:
```
array([1, 2, 3])
```

In [147]:
```
x[:,0] # second column (first to last row)
```

Out[147]:
```
array([1, 4])
```

NumPy Array Attributes

In [148]:
```
x = np.array([[1,2,3],[4,5,6]])
```

In [149]:
```
x.ndim # number of dimensions
```

Out[149]: 2

In [150]:
```
x.shape # shape in each dimension
```

Out[150]: (2, 3)

In [151]:
```
x.size # total number of elements
```

Out[151]: 6

NumPy Operations (UFuncs)

In [152]:
```
x = [1,2,3]
y = [4,5,6]
```

In [153]:
```
x+y
```

Out[153]:
```
[1, 2, 3, 4, 5, 6]
```

In [154]:
```
x = np.array([1,2,3])
y = np.array([4,5,6])
```

In [155]:
```
x+y
```

Out[155]:
```
array([5, 7, 9])
```

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

In [156]:
```
# square every element in a list
x = [1,2,3]
```

In [157]:
```
x**2
```

```
TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

In [158]:
```
[y**2 for y in x]
```

Out[158]:
```
[1, 4, 9]
```

In [159]:
```
# square every element in a numpy array
x = np.array([1,2,3])
```

In [160]:
```
x**2
```

Out[160]:
```
array([1, 4, 9])
```

NumPy `random` Submodule

Provides many random sampling functions

```
from numpy.random import ...
```

Intro to Pandas

Pandas is an open source, BSD-licensed library providing:

- **high-performance, easy-to-use data structures** and
- **data analysis tools**

In [161]:
```
# usually imported as pd
import pandas as pd
```

- **Series**: 1D array with a flexible index
- **Dataframe**: 2D matrix with flexible index and column names

Pandas Series

- 1D array of data (any numpy datatype) plus an associated **index** array

In [162]:
```
s = pd.Series(np.random.rand(4))
s
```

Out[162]:
```
0    0.215178
1    0.625398
2    0.521590
3    0.580152
dtype: float64
```

In [163]:
```
# return the values of the series
s.values
```

Out[163]:
```
array([0.21517773, 0.62539808, 0.5215901 , 0.58015187])
```

In [164]:
```
# return the index of the series
s.index
```

Out[164]:
```
RangeIndex(start=0, stop=4, step=1)
```

Pandas Series Cont.

- index is very flexible (integers, strings, ...)

In [165]:
```
# create Series from array and index of strings
s = pd.Series(np.random.rand(4),index=['a','b','c','d'])
s
```

Out[165]:
```
a    0.573676
b    0.902324
c    0.482111
d    0.817792
dtype: float64
```

In [166]:
```
s['a']
```

Out[166]:
```
0.5736759330744037
```

In [167]:
```
s[['c','d']]
```

Out[167]:
```
c    0.482111
d    0.817792
dtype: float64
```

Pandas Series Cont.

In [168]:
```
# create series from a dictionary
s = pd.Series({'a':1,'b':2,'c':3,'d':4})
s
```

Out[168]:
```
a    1
b    2
c    3
d    4
dtype: int64
```

In [169]:
```
s[s.index[-2:]]
```

Out[169]:
```
c    3
d    4
dtype: int64
```

Pandas DataFrame

- tabular datastructure
- each column a single datatype
- contains both row and column indices
- single column == Series

Pandas DataFrame Cont.

In [170]:
```
df = pd.DataFrame({'Year':[2017,2018,2018,2019],
                   'Class_Name':['A','A','B','A'],
                   'Measure1':[2.1,3.0,2.4,1.9]
                  })
```

In [171]:
```
df
```

Out[171]:

	Year	Class_Name	Measure1
0	2017	A	2.1
1	2018	A	3.0
2	2018	B	2.4
3	2019	A	1.9

Pandas DataFrame Cont.

In [172]:
```
data = [[2017,'A',2.1],
        [2018,'A',3.0],
        [2018,'B',2.4],
        [2019,'A',1.9]]
```

In [173]:
```
df = pd.DataFrame(data,
                  columns=['Year','Class_Name','Measure1'],
                  index=['001','002','003','004'])
df.shape
```

Out[173]: (4, 3)

In [174]:
```
df
```

Out[174]:

	Year	Class_Name	Measure1
001	2017	A	2.1
002	2018	A	3.0
003	2018	B	2.4
004	2019	A	1.9

Pandas Attributes

- Get shape of DataFrame : `shape`

```
In [175]: df.shape # rows, columns

Out[175]: (4, 3)
```

- Get index values : `index`

```
In [176]: df.index

Out[176]: Index(['001', '002', '003', '004'], dtype='object')
```

- Get column values : `columns`

```
In [177]: df.columns

Out[177]: Index(['Year', 'Class_Name', 'Measure1'], dtype='object')
```

Pandas Indexing/Selection

Select by label:

- `.loc[]`

In [178]:
```
df.loc['001']
```

Out[178]:
```
Year          2017
Class_Name       A
Measure1       2.1
Name: 001, dtype: object
```

In [179]:
```
df.loc['001','Measure1']
```

Out[179]:
```
2.1
```

Pandas Indexing/Selection Cont.

Select by position:

- `.iloc[]`

In [180]:
```
df.iloc[0]
```

Out[180]:
```
Year          2017
Class_Name       A
Measure1       2.1
Name: 001, dtype: object
```

In [181]:
```
df.iloc[0,2]
```

Out[181]:
```
2.1
```

Pandas Indexing/Selection Cont.

Selecting multiple rows/columns: use list (fancy indexing)

In [182]:
```
df.loc[['002','004']]
```

Out[182]:

	Year	Class_Name	Measure1
002	2018	A	3.0
004	2019	A	1.9

In [183]:
```
df.loc[['002','004'],['Year','Measure1']]
```

Out[183]:

	Year	Measure1
002	2018	3.0
004	2019	1.9

Pandas Slicing

In [184]:
```
# Get last two rows
df.iloc[-2:]
```

Out[184]:

	Year	Class_Name	Measure1
003	2018	B	2.4
004	2019	A	1.9

In [185]:
```
# Get first two rows and first two columns
df.iloc[:2,:2]
```

Out[185]:

	Year	Class_Name
001	2017	A
002	2018	A

NOTE: `.iloc` is **exclusive** (start:end+1)

Pandas Slicing Cont.

Can also slice using labels:

In [186]:
```
df.loc['002':'004']
```

Out[186]:

	Year	Class_Name	Measure1
002	2018	A	3.0
003	2018	B	2.4
004	2019	A	1.9

In [187]:
```
df.loc['002':'004',:'Class_Name']
```

Out[187]:

	Year	Class_Name
002	2018	A
003	2018	B
004	2019	A

NOTE: `.loc` is **inclusive**

Pandas Slicing Cont.

How to indicate all rows or all columns? `:`

In [188]:
```
df.loc[:,'Measure1']
```

Out[188]:
```
001    2.1
002    3.0
003    2.4
004    1.9
Name: Measure1, dtype: float64
```

In [189]:
```
df.iloc[2:,:]
```

Out[189]:

	Year	Class_Name	Measure1
003	2018	B	2.4
004	2019	A	1.9

Pandas Indexing Cont.

Shortcut for indexing:

In [190]:
```
df['Class_Name']
```

Out[190]:
```
001    A
002    A
003    B
004    A
Name: Class_Name, dtype: object
```

In [191]:
```
# can use dot notation if there is no space in label
df.Class_Name
```

Out[191]:
```
001    A
002    A
003    B
004    A
Name: Class_Name, dtype: object
```

Panda Selection Chaining

Get 'Year' and 'Measure1' for first 3 rows:

In [192]:
```
df.iloc[:3].loc[:,['Year','Measure1']]
```

Out[192]:

	Year	Measure1
001	2017	2.1
002	2018	3.0
003	2018	2.4

For records '001' and '003' get last two columns

In [193]:
```
df.loc[['001','003']].iloc[:,-2:]
```

Out[193]:

	Class_Name	Measure1
001	A	2.1
003	B	2.4

Panda Selection Chaining Cont.

For record '001' get last two columns?:

In [194]:
```
df.loc['001'].iloc[:,-2:]
```

```
IndexingError: Too many indexers
```

In [195]:
```
df.loc['001'].iloc[-2:]
```

Out[195]:
```
Class_Name      A
Measure1      2.1
Name: 001, dtype: object
```

Pandas `head` and `tail`

Get a quick view of the first or last rows in a DataFrame

In [196]:
```
df.head() # first 5 rows by default
```

Out[196]:

	Year	Class_Name	Measure1
001	2017	A	2.1
002	2018	A	3.0
003	2018	B	2.4
004	2019	A	1.9

In [197]:
```
df.tail(2) # only print 2 rows
```

Out[197]:

	Year	Class_Name	Measure1
003	2018	B	2.4
004	2019	A	1.9

Pandas Boolean Mask

In [205]:
```
# Which rows have Class_Name of 'A'?
df.Class_Name == 'A'
```

Out[205]:
```
001     True
002     True
003    False
004     True
Name: Class_Name, dtype: bool
```

In [206]:
```
# Get all data for rows with with Class_Name 'A'
df.loc[df.Class_Name == 'A']
```

Out[206]:

	Year	Class_Name	Measure1
001	2017	A	2.1
002	2018	A	3.0
004	2019	A	1.9

In [207]:
```
# Get Measure1 for all records for Class_Name 'A'
df.loc[df.Class_Name == 'A','Measure1']
```

Out[207]:
```
001    2.1
002    3.0
004    1.9
Name: Measure1, dtype: float64
```

Pandas Boolean Mask Cont.

Get all records for class 'A' before 2019

In [201]:
```
df.loc[(df.Class_Name == 'A') & (df.Year < 2019)]
```

Out[201]:

	Year	Class_Name	Measure1
001	2017	A	2.1
002	2018	A	3.0

Get all records in a set of years:

In [202]:
```
df.loc[df.Year.isin([2017,2019])]
```

Out[202]:

	Year	Class_Name	Measure1
001	2017	A	2.1
004	2019	A	1.9

Pandas Selection Review

Pandas Selection Review

Questions?