Elements Of Data Science - F2024

Week 2: Python Intro/Review and Numpy

9/15/2024

TODOs

- **Review** Selections from PDSH Chapter 2

- **Read** Selections from PDSH Chapter 3

- **Skim** Selections from PDSH Chapter 4

- Complete Week 2 Quiz (Due 1 week from today, Sep 22 @11:59pm)

Announcements

Office hours

- Monday 6pm-6:50pm, location: DSI Conference room
- Skype, by appointment only, Tuesdays 6pm+. (link to be sent once OH requested)

TODOs

- Ch 2. Introduction to NumPy
  - Understanding Data Types in Python
  - The Basics of NumPy Arrays
  - Skim: Computation on NumPy Arrays: Universal Functions
  - Aggregations: Min, Max, and Everything In Between
  - Skim: 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
    - Skim: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

1. `cd` to the cloned class repostory

2. `git pull`

example:

```
$ cd ~/proj/eods-f24
$ git pull
```

Questions?

TODAY

- Tools Review

- Getting "Help" Documentation

- Python (Review?)

- Numpy

Tools Review

- Starting Jupyter
- Notebooks, Kernels and Virtual Environments

Getting "Help" Documentation in Python

Getting "Help" Documentation in Python

```
In [1]:   1  help(print)
```

```
Help on built-in function print in module builtins:

print(...)
    print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

    Prints the values to a stream, or to sys.stdout by default.
    Optional keyword arguments:
    file:  a file-like object (stream); defaults to the current sys.stdout.
    sep:   string inserted between values, default a space.
    end:   string appended after the last value, default a newline.
    flush: whether to forcibly flush the stream.
```

Getting "Help" Documentation in Python

```
In [1]:  1  help(print)

         Help on built-in function print in module builtins:

         print(...)
             print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

             Prints the values to a stream, or to sys.stdout by default.
             Optional keyword arguments:
             file:  a file-like object (stream); defaults to the current sys.stdout.
             sep:   string inserted between values, default a space.
             end:   string appended after the last value, default a newline.
             flush: whether to forcibly flush the stream.
```

Also, in ipython/jupyter:

```
print?               # show docstring
print??              # show code as well
print([SHIFT+TAB]    # get help in a popup
```

Python (Review?)

- Whitespace Formatting
- Dynamic Typing
- 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

Whitespace Formatting

- Instead of braces or brackets to delimit blocks, use whitespace

```python
# 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
```

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

Dynamic Typing

- don't need to specify type at variable creation (though they'll get one at runtime)

Dynamic Typing

- don't need to specify type at variable creation (though they'll get one at runtime)

```
In [2]:   1  x = 3
          2  x = 3.14
          3  x = 'apple'
          4  x

Out[2]:  'apple'
```

Dynamic Typing

- don't need to specify type at variable creation (though they'll get one at runtime)

```
In [2]:  1  x = 3
         2  x = 3.14
         3  x = 'apple'
         4  x
```

Out[2]: 'apple'

```
In [3]:  1  # to determine the current variable type
         2  type(x)
```

Out[3]: str

Basic Python Data Types

- **int** (integer): `42`

- **float** : `4.2`, `4e2`

- **bool** (boolean): `True`, `False`

- **str** (string) : `'num 42'`, `"num 42"`,
  `"""multi-line string"""`

- **None** (null): `None`

- also `long`, `complex`, `bytes`, etc.

Functions

Functions

```
In [4]:  1  def add_two(x):
         2      """Adds 2 to the number passed in."""
         3      return x+2
         4
         5
         6  add_two(2)

Out[4]:  4
```

Functions

In [4]:
```python
1  def add_two(x):
2      """Adds 2 to the number passed in."""
3      return x+2
4
5
6  add_two(2)
```

Out[4]: 4

In [5]:
```python
1  help(add_two)
```

```
Help on function add_two in module __main__:

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

Functions

```
In [4]:   1  def add_two(x):
          2      """Adds 2 to the number passed in."""
          3      return x+2
          4
          5
          6  add_two(2)
```

Out[4]: 4

```
In [5]:   1  help(add_two)
```

```
Help on function add_two in module __main__:

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

Reminder, also in ipython/jupyter:

- `add_two?`   `# show docstring`
- `add_two??`   `# show code as well`
- `add_two([SHIFT+TAB]` `# get help in a popup`

Function Arguments

Function Arguments

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

Function Arguments

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

```
In [6]:  1  def subtract(x,y):
         2      return x-y
         3
         4  subtract(3,1)

Out[6]: 2
```

Function Arguments

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

```
In [6]:   1  def subtract(x,y):
          2      return x-y
          3
          4  subtract(3,1)

Out[6]: 2
```

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

Function Arguments

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

```
In [6]:   1  def subtract(x,y):
          2      return x-y
          3
          4  subtract(3,1)
```

```
Out[6]:  2
```

- **keyword arguments** must follow positional

- can be called in any order

```
In [7]:   1  def proportion(numer,denom,precision=2):
          2      return round(numer/denom,precision)
          3
          4  proportion(2,precision=2,denom=3)
```

```
Out[7]:  0.67
```

String Formatting

String Formatting

```
In [8]:  1  x = 3.1415
         2
         3  'the value of x is ' + str(x)

Out[8]:  'the value of x is 3.1415'
```

String Formatting

```
In [8]: 1  x = 3.1415
        2
        3  'the value of x is ' + str(x)
```

Out[8]: 'the value of x is 3.1415'

```
In [9]: 1  'the value of x is %0.2f' % x
```

Out[9]: 'the value of x is 3.14'

String Formatting

```
In [8]:   1  x = 3.1415
          2
          3  'the value of x is ' + str(x)
```

Out[8]: 'the value of x is 3.1415'

```
In [9]:   1  'the value of x is %0.2f' % x
```

Out[9]: 'the value of x is 3.14'

```
In [10]:  1  'the value of x is {:0.10f} and not {}'.format(x, 10)
```

Out[10]: 'the value of x is 3.1415000000 and not 10'

String Formatting

```
In [8]:  1  x = 3.1415
         2
         3  'the value of x is ' + str(x)
```

Out[8]: 'the value of x is 3.1415'

```
In [9]:  1  'the value of x is %0.2f' % x
```

Out[9]: 'the value of x is 3.14'

```
In [10]:  1  'the value of x is {:0.10f} and not {}'.format(x, 10)
```

Out[10]: 'the value of x is 3.1415000000 and not 10'

```
In [11]:  1  f'the value of x is {x:0.2f}'
          2  # note: f-string is a literal string, prefixed with 'f', which contains expressions inside braces.
```

Out[11]: 'the value of x is 3.14'

String Formatting

```
In [8]:  1  x = 3.1415
         2
         3  'the value of x is ' + str(x)
```

Out[8]: 'the value of x is 3.1415'

```
In [9]:  1  'the value of x is %0.2f' % x
```

Out[9]: 'the value of x is 3.14'

```
In [10]:  1  'the value of x is {:0.10f} and not {}'.format(x, 10)
```

Out[10]: 'the value of x is 3.1415000000 and not 10'

```
In [11]:  1  f'the value of x is {x:0.2f}'
          2  # note: f-string is a literal string, prefixed with 'f', which contains expressions inside braces.
```

Out[11]: 'the value of x is 3.14'

- often want to print variable values for debugging

String Formatting

```
In [8]:  1  x = 3.1415
         2
         3  'the value of x is ' + str(x)
```

Out[8]: 'the value of x is 3.1415'

```
In [9]:  1  'the value of x is %0.2f' % x
```

Out[9]: 'the value of x is 3.14'

```
In [10]:  1  'the value of x is {:0.10f} and not {}'.format(x, 10)
```

Out[10]: 'the value of x is 3.1415000000 and not 10'

```
In [11]:  1  f'the value of x is {x:0.2f}'
          2  # note: f-string is a literal string, prefixed with 'f', which contains expressions inside braces.
```

Out[11]: 'the value of x is 3.14'

- often want to print variable values for debugging

```
In [12]:  1  f'x = {x:0.2f}'
```

Out[12]: 'x = 3.14'

String Formatting

```
In [8]:   1  x = 3.1415
          2
          3  'the value of x is ' + str(x)
```

Out[8]:   'the value of x is 3.1415'

```
In [9]:   1  'the value of x is %0.2f' % x
```

Out[9]:   'the value of x is 3.14'

```
In [10]:  1  'the value of x is {:0.10f} and not {}'.format(x, 10)
```

Out[10]: 'the value of x is 3.1415000000 and not 10'

```
In [11]:  1  f'the value of x is {x:0.2f}'
          2  # note: f-string is a literal string, prefixed with 'f', which contains expressions inside braces.
```

Out[11]: 'the value of x is 3.14'

- often want to print variable values for debugging

```
In [12]:  1  f'x = {x:0.2f}'
```

Out[12]: 'x = 3.14'

```
In [13]:  1  f'{x = :0.2f}'  # new in 3.8
```

Out[13]: 'x = 3.14'

String Formatting Cont.

String Formatting Cont.

```
In [14]:  1 """This is a multiline string.
          2 The value of x is {}.""".format(x)

Out[14]: 'This is a multiline string.\nThe value of x is 3.1415.'
```

String Formatting Cont.

In [14]:
```python
"""This is a multiline string.
The value of x is {}.""".format(x)
```

Out[14]: 'This is a multiline string.\nThe value of x is 3.1415.'

In [15]:
```python
print("""This is a multiline string.
The value of x is {}.""".format(x))
```

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

String Formatting Cont.

```
In [14]:   1  """This is a multiline string.
           2  The value of x is {}.""".format(x)
```

Out[14]: 'This is a multiline string.\nThe value of x is 3.1415.'

```
In [15]:   1  print("""This is a multiline string.
           2  The value of x is {}.""".format(x))
```

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

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

Python Data Types Continued: `list`

Python Data Types Continued: `list`

```
In [22]:  1  # elements of a python list do not all have to be of the same type
          2  x = [42,'e',2.0]
          3  x
```

Out[22]: [42, 'e', 2.0]

Python Data Types Continued: `list`

```
In [22]:  1  # elements of a python list do not all have to be of the same type
          2  x = [42,'e',2.0]
          3  x
```

Out[22]: `[42, 'e', 2.0]`

```
In [23]:  1  x[0] # indexing
```

Out[23]: `42`

Python Data Types Continued: `list`

In [22]:
```python
1  # elements of a python list do not all have to be of the same type
2  x = [42,'e',2.0]
3  x
```

Out[22]: [42, 'e', 2.0]

In [23]:
```python
1  x[0] # indexing
```

Out[23]: 42

In [24]:
```python
1  x[-1] # reverse indexing
```

Out[24]: 2.0

Python Data Types Continued: `list`

```
In [22]:  1  # elements of a python list do not all have to be of the same type
          2  x = [42,'e',2.0]
          3  x
```

Out[22]: [42, 'e', 2.0]

```
In [23]:  1  x[0] # indexing
```

Out[23]: 42

```
In [24]:  1  x[-1] # reverse indexing
```

Out[24]: 2.0

```
In [25]:  1  x[2] = 4 # assignment
          2  x
```

Out[25]: [42, 'e', 4]

Python Data Types Continued: `list`

```
In [22]:  1  # elements of a python list do not all have to be of the same type
          2  x = [42,'e',2.0]
          3  x
```

Out[22]: [42, 'e', 2.0]

```
In [23]:  1  x[0] # indexing
```

Out[23]: 42

```
In [24]:  1  x[-1] # reverse indexing
```

Out[24]: 2.0

```
In [25]:  1  x[2] = 4 # assignment
          2  x
```

Out[25]: [42, 'e', 4]

```
In [26]:  1  x.append('a') # add a value to list
          2  x
```

Out[26]: [42, 'e', 4, 'a']

Python Data Types Continued: `list`

In [22]:
```python
1  # elements of a python list do not all have to be of the same type
2  x = [42,'e',2.0]
3  x
```

Out[22]: `[42, 'e', 2.0]`

In [23]:
```python
1  x[0] # indexing
```

Out[23]: `42`

In [24]:
```python
1  x[-1] # reverse indexing
```

Out[24]: `2.0`

In [25]:
```python
1  x[2] = 4 # assignment
2  x
```

Out[25]: `[42, 'e', 4]`

In [26]:
```python
1  x.append('a') # add a value to list
2  x
```

Out[26]: `[42, 'e', 4, 'a']`

In [27]:
```python
1  value_at_1 = x.pop(1) # remove/delete at index
2  x
```

Out[27]: `[42, 4, 'a']`

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

```
In [28]:   1  x = {'b':[2,1],'a':1,'c':4}
           2  # or x = dict(b=2,a=1,c=4)
           3  x

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

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

```
In [28]: 1 x = {'b':[2,1],'a':1,'c':4}
         2 # or x = dict(b=2,a=1,c=4)
         3 x
```

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

```
In [29]: 1 # index into dictionary using key
         2 x['b']
```

Out[29]: [2, 1]

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

In [28]:
```python
1  x = {'b':[2,1],'a':1,'c':4}
2  # or x = dict(b=2,a=1,c=4)
3  x
```

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

In [29]:
```python
1  # index into dictionary using key
2  x['b']
```

Out[29]: [2, 1]

In [30]:
```python
1  # assign a value to a (new or existing) key
2  x['d'] = 3
3  x
```

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

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

In [28]:
```python
1  x = {'b':[2,1],'a':1,'c':4}
2  # or x = dict(b=2,a=1,c=4)
3  x
```

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

In [29]:
```python
1  # index into dictionary using key
2  x['b']
```

Out[29]: `[2, 1]`

In [30]:
```python
1  # assign a value to a (new or existing) key
2  x['d'] = 3
3  x
```

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

In [31]:
```python
1  # remove/delete
2  # can specify a return a value if key does not exist (here it's None), otherwise throws an exception
3  x.pop('eods', None)
```

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

```
In [28]:  1  x = {'b':[2,1],'a':1,'c':4}
          2  # or x = dict(b=2,a=1,c=4)
          3  x
```

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

```
In [29]:  1  # index into dictionary using key
          2  x['b']
```

Out[29]: [2, 1]

```
In [30]:  1  # assign a value to a (new or existing) key
          2  x['d'] = 3
          3  x
```

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

```
In [31]:  1  # remove/delete
          2  # can specify a return a value if key does not exist (here it's None), otherwise throws an exception
          3  x.pop('eods', None)
```

```
In [32]:  1  x
```

Out[32]: {'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

In [28]:
```python
1  x = {'b':[2,1],'a':1,'c':4}
2  # or x = dict(b=2,a=1,c=4)
3  x
```

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

In [29]:
```python
1  # index into dictionary using key
2  x['b']
```

Out[29]: [2, 1]

In [30]:
```python
1  # assign a value to a (new or existing) key
2  x['d'] = 3
3  x
```

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

In [31]:
```python
1  # remove/delete
2  # can specify a return a value if key does not exist (here it's None), otherwise throws an exception
3  x.pop('eods', None)
```

In [32]:
```python
1  x
```

Out[32]: {'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}

In [33]:
```python
1  x.pop('d', None)
```

Python Data Types Continued: `dict` Cont.

Python Data Types Continued: `dict` Cont.

```python
In [34]:  1  # using the same dictionary
          2  x
```

Out[34]: {'b': [2, 1], 'a': 1, 'c': 4}

Python Data Types Continued: `dict` Cont.

```
In [34]:  1  # using the same dictionary
          2  x
```

Out[34]: {'b': [2, 1], 'a': 1, 'c': 4}

```
In [35]:  1  # get a set of keys
          2  x.keys()
```

Out[35]: dict_keys(['b', 'a', 'c'])

Python Data Types Continued: `dict` Cont.

```
In [34]:  1  # using the same dictionary
          2  x
```

Out[34]: {'b': [2, 1], 'a': 1, 'c': 4}

```
In [35]:  1  # get a set of keys
          2  x.keys()
```

Out[35]: dict_keys(['b', 'a', 'c'])

```
In [36]:  1  # get a set of values
          2  x.values()
```

Out[36]: dict_values([[2, 1], 1, 4])

Python Data Types Continued: `dict` Cont.

```
In [34]:   1  # using the same dictionary
           2  x
```

Out[34]: {'b': [2, 1], 'a': 1, 'c': 4}

```
In [35]:   1  # get a set of keys
           2  x.keys()
```

Out[35]: dict_keys(['b', 'a', 'c'])

```
In [36]:   1  # get a set of values
           2  x.values()
```

Out[36]: dict_values([[2, 1], 1, 4])

```
In [37]:   1  # get a set of (key,value) tuples
           2  x.items()
```

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

Python Data Types Continued: `dict` Cont.

```
In [34]:   1  # using the same dictionary
           2  x
```

Out[34]: {'b': [2, 1], 'a': 1, 'c': 4}

```
In [35]:   1  # get a set of keys
           2  x.keys()
```

Out[35]: dict_keys(['b', 'a', 'c'])

```
In [36]:   1  # get a set of values
           2  x.values()
```

Out[36]: dict_values([[2, 1], 1, 4])

```
In [37]:   1  # get a set of (key,value) tuples
           2  x.items()
```

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

```
In [38]:   1  # get a list of (key,value) pairs
           2  list(x.items())
```

Out[38]: [('b', [2, 1]), ('a', 1), ('c', 4)]

Python Data Types Continued: `tuple`

- like a list, but **immutable**

Python Data Types Continued: `tuple`

- like a list, but **immutable**

```
In [39]:  1  x = (2,'e',3,4)
          2  x

Out[39]:  (2, 'e', 3, 4)
```

Python Data Types Continued: `tuple`

- like a list, but **immutable**

```
In [39]: 1 x = (2,'e',3,4)
         2 x

Out[39]: (2, 'e', 3, 4)

In [40]: 1 x[0] # indexing

Out[40]: 2
```

Python Data Types Continued: `tuple`

- like a list, but **immutable**

```
In [39]:  1  x = (2,'e',3,4)
          2  x

Out[39]: (2, 'e', 3, 4)
```

```
In [40]:  1  x[0] # indexing

Out[40]: 2
```

```
In [41]:  1  x[0] = 3 # assignment? Nope, error: immutable`

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[41], line 1
----> 1 x[0] = 3

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

```
In [42]:  1  a = (1,2,[10,12])
```

```
In [43]:  1  a[2][0] =15
```

```
In [44]:  1  a[2].append(10)
```

```
In [45]:  1  a

Out[45]: (1, 2, [15, 12, 10])
```

Python Data Types Continued: `set`

```
In [46]:  1  lst = [1,1,1,1,1,1,5,5,'Jack']
          2  lst = list(set(lst))
```

```
In [47]:  1  lst
```

Out[47]: [1, 5, 'Jack']

Python Data Types Continued: `set`

```
In [46]:  1  lst = [1,1,1,1,1,1,5,5,'Jack']
          2  lst = list(set(lst))
```

```
In [47]:  1  lst
```

Out[47]: `[1, 5, 'Jack']`

```
In [48]:  1  x = {2,'e','e'} # or set([2,'e','e'])
          2  x
```

Out[48]: `{2, 'e'}`

Python Data Types Continued: `set`

```
In [46]:  1  lst = [1,1,1,1,1,1,5,5,'Jack']
          2  lst = list(set(lst))
```

```
In [47]:  1  lst
```

Out[47]: [1, 5, 'Jack']

```
In [48]:  1  x = {2,'e','e'} # or set([2,'e','e'])
          2  x
```

Out[48]: {2, 'e'}

```
In [49]:  1  x.add(1) # insert
          2  x
```

Out[49]: {1, 2, 'e'}

Python Data Types Continued: `set`

```
In [46]:   1  lst = [1,1,1,1,1,1,5,5,'Jack']
           2  lst = list(set(lst))
```

```
In [47]:   1  lst
```

Out[47]: [1, 5, 'Jack']

```
In [48]:   1  x = {2,'e','e'} # or set([2,'e','e'])
           2  x
```

Out[48]: {2, 'e'}

```
In [49]:   1  x.add(1) # insert
           2  x
```

Out[49]: {1, 2, 'e'}

```
In [50]:   1  x.remove('e') # remove/delete
           2  x
```

Out[50]: {1, 2}

Python Data Types Continued: `set`

```
In [46]:  1  lst = [1,1,1,1,1,1,5,5,'Jack']
          2  lst = list(set(lst))
```

```
In [47]:  1  lst
```

Out[47]: [1, 5, 'Jack']

```
In [48]:  1  x = {2,'e','e'} # or set([2,'e','e'])
          2  x
```

Out[48]: {2, 'e'}

```
In [49]:  1  x.add(1) # insert
          2  x
```

Out[49]: {1, 2, 'e'}

```
In [50]:  1  x.remove('e') # remove/delete
          2  x
```

Out[50]: {1, 2}

```
In [51]:  1  x.intersection({2,3})
```

Out[51]: {2}

Python Data Types Continued: `set`

```
In [46]:  1  lst = [1,1,1,1,1,1,5,5,'Jack']
          2  lst = list(set(lst))
```

```
In [47]:  1  lst
```

Out[47]: [1, 5, 'Jack']

```
In [48]:  1  x = {2,'e','e'} # or set([2,'e','e'])
          2  x
```

Out[48]: {2, 'e'}

```
In [49]:  1  x.add(1) # insert
          2  x
```

Out[49]: {1, 2, 'e'}

```
In [50]:  1  x.remove('e') # remove/delete
          2  x
```

Out[50]: {1, 2}

```
In [51]:  1  x.intersection({2,3})
```

Out[51]: {2}

```
In [52]:  1  x.difference({2,3})
```

Out[52]: {1}

Python Data Types Continued: `set`

```
In [46]:  1  lst = [1,1,1,1,1,1,5,5,'Jack']
          2  lst = list(set(lst))
```

```
In [47]:  1  lst
```

Out[47]: [1, 5, 'Jack']

```
In [48]:  1  x = {2,'e','e'} # or set([2,'e','e'])
          2  x
```

Out[48]: {2, 'e'}

```
In [49]:  1  x.add(1) # insert
          2  x
```

Out[49]: {1, 2, 'e'}

```
In [50]:  1  x.remove('e') # remove/delete
          2  x
```

Out[50]: {1, 2}

```
In [51]:  1  x.intersection({2,3})
```

Out[51]: {2}

```
In [52]:  1  x.difference({2,3})
```

Out[52]: {1}

```
In [53]:  1  x[0] # cannot index into a set
```

Determining Length with `len`

Determining Length with `len`

```
In [54]:  1  len([1,2,3])

Out[54]:  3
```

Determining Length with `len`

```
In [54]:  1  len([1,2,3])

Out[54]:  3
```

```
In [55]:  1  len({'a':1,'b':2,'c':3})

Out[55]:  3
```

Determining Length with `len`

```
In [54]:  1  len([1,2,3])

Out[54]:  3
```

```
In [55]:  1  len({'a':1,'b':2,'c':3})

Out[55]:  3
```

```
In [56]:  1  len('apple')

Out[56]:  5
```

Exceptions

Exceptions

```
In [57]:  1 'a' + 2

          ---------------------------------------------------------------------------
          TypeError                                 Traceback (most recent call last)
          Cell In[57], line 1
          ----> 1 'a' + 2

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

Exceptions

```
In [57]:  1 'a' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[57], line 1
----> 1 'a' + 2

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

Common exceptions:

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

Catching Exceptions with `try`-`except`

Catching Exceptions with `try-except`

```python
In [58]:  1  try:
          2      'a' + 2
          3  except TypeError as e:
          4      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
```

Catching Exceptions with `try-except`

```
In [58]:  1  try:
          2      'a' + 2
          3  except TypeError as e:
          4      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 [59]:  1  try:
          2      set([1,2,3])[0]
          3  except SyntaxError as e:
          4      print(f"Print this if there's a syntax error")
          5  except Exception as e:
          6      print(f"Print this for any other error")
```

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

Truthiness

Truthiness

- boolean: `True`, `False`

- These all translate to `False`:

  - `None`
  - `[]` (empty list)
  - `{}` (empty dictionary)
  - `''` (empty string)
  - `set()`
  - `0`
  - `0.0`

Comparison Operators

Comparison Operators

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

Comparison Operators

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

```
In [60]:  1  3 == 3

Out[60]:  True
```

Comparison Operators

- equality: ==
- inequality: !=

```
In [60]:  1  3 == 3
```

Out[60]: True

```
In [61]:  1  3 != 4
```

Out[61]: True

Comparison Operators

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

```
In [60]:   1  3 == 3
```

```
Out[60]:  True
```

```
In [61]:   1  3 != 4
```

```
Out[61]:  True
```

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

Comparison Operators

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

```
In [60]:  1  3 == 3
```

Out[60]: True

```
In [61]:  1  3 != 4
```

Out[61]: True

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

```
In [62]:  1  3 > 4
```

Out[62]: False

Logical Operators

Logical Operators

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

Logical Operators

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

```
In [63]:  1  ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)

Out[63]:  True
```

Logical Operators

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

```
In [63]:  1  ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)

Out[63]:  True
```

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

Logical Operators

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

```
In [63]:   1   ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)

Out[63]:   True
```

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

```
In [64]:   1   any([0,0,0])

Out[64]:   False
```

Logical Operators

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

```
In [63]:  1  ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)

Out[63]:  True
```

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

```
In [64]:  1  any([0,0,0])

Out[64]:  False
```

- `all()`: all elements are true

Logical Operators

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

```
In [63]:  1  ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

```
Out[63]:  True
```

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

```
In [64]:  1  any([0,0,0])
```

```
Out[64]:  False
```

- `all()`: all elements are true

```
In [65]:  1  all([True,1,False])
```

```
Out[65]:  False
```

Logical Operators

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

```
In [63]:    1  ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)

Out[63]:  True
```

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

```
In [64]:    1  any([0,0,0])

Out[64]:  False
```

- `all()`: all elements are true

```
In [65]:    1  all([True,1,False])

Out[65]:  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 [66]:   1  assert 2+2 != 4


           ---------------------------------------------------------------------
           AssertionError                                Traceback (most recent call last)
           Cell In[66], line 1
           ----> 1 assert 2+2 != 4

           AssertionError:
```

Assert

- use `assert` to test anything we know should be true

- simple unit test

- raises exception when assertion is false, otherwise nothing

```
In [66]:  1  assert 2+2 != 4
```

```
---------------------------------------------------------------------
AssertionError                              Traceback (most recent call last)
Cell In[66], line 1
----> 1 assert 2+2 != 4

AssertionError:
```

```
In [67]:  1  assert 1 == 0, 'if this is wrong what you want to display'
```

```
---------------------------------------------------------------------
AssertionError                              Traceback (most recent call last)
Cell In[67], line 1
----> 1 assert 1 == 0, 'if this is wrong what you want to display'

AssertionError: if this is wrong what you want to display
```

Assert

- use `assert` to test anything we know should be true

- simple unit test

- raises exception when assertion is false, otherwise nothing

```
In [66]:    1  assert 2+2 != 4
```

```
---------------------------------------------------------------------
AssertionError                          Traceback (most recent call last)
Cell In[66], line 1
----> 1 assert 2+2 != 4

AssertionError:
```

```
In [67]:    1  assert 1 == 0, 'if this is wrong what you want to display'
```

```
---------------------------------------------------------------------
AssertionError                          Traceback (most recent call last)
Cell In[67], line 1
----> 1 assert 1 == 0, 'if this is wrong what you want to display'

AssertionError: if this is wrong what you want to display
```

```
In [68]:    1  # can add an error message
            2  assert 1 == 0, "1 does not equal 0"
```

```
---------------------------------------------------------------------
AssertionError                          Traceback (most recent call last)
Cell In[68], line 2
```

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

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

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

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

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

```
In [69]:  1  x = 3
          2  if x > 0:
          3      print('x > 0')
          4  elif x < 0:
          5      print('x < 0')
          6  else:
          7      print('x == 0')

          x > 0
```

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

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

```
In [69]:  1  x = 3
          2  if x > 0:
          3      print('x > 0')
          4  elif x < 0:
          5      print('x < 0')
          6  else:
          7      print('x == 0')

          x > 0
```

- single-line `if` then `else`

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

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

```
In [69]:  1  x = 3
          2  if x > 0:
          3      print('x > 0')
          4  elif x < 0:
          5      print('x < 0')
          6  else:
          7      print('x == 0')
```

```
x > 0
```

- single-line `if` then `else`

```
In [70]:  1  print("x < 0") if (x < 0) else print("x >= 0")
```

```
x >= 0
```

More Control Flow: `for` and `while`

More Control Flow: `for` and `while`

- `for` each element of an iterable: do something

More Control Flow: `for` and `while`

- `for` each element of an iterable: do something

```
In [71]:  1  a = []
          2  for x in [0,1,2]:
          3      a.append(x)
          4  a

Out[71]:  [0, 1, 2]
```

More Control Flow: `for` and `while`

- `for` each element of an iterable: do something

```
In [71]:  1  a = []
          2  for x in [0,1,2]:
          3      a.append(x)
          4  a

Out[71]:  [0, 1, 2]
```

- `while` something is true

More Control Flow: `for` and `while`

- `for` each element of an iterable: do something

```
In [71]:  1  a = []
          2  for x in [0,1,2]:
          3      a.append(x)
          4  a
```

Out[71]: [0, 1, 2]

- `while` something is true

```
In [72]:  1  x = 0
          2  while x < 3:
          3      x += 1
          4  x
```

Out[72]: 3

More Control Flow: `break` and `continue`

More Control Flow: `break` and `continue`

- `break` : break out of current loop

More Control Flow: `break` and `continue`

- `break` : break out of current loop

```
In [73]:  1  x = 0
          2  while True:
          3      x += 1
          4      if x == 10:
          5          print(x)
          6          break

         10
```

More Control Flow: **break** and **continue**

- **break** : break out of current loop

```
In [73]:   1  x = 0
           2  while True:
           3      x += 1
           4      if x == 10:
           5          print(x)
           6          break
```

```
10
```

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

```
In [74]:   1  for name in ['John', 'Ben', 'Julia']:
           2      if name == 'Ben':
           3          continue
           4      print(name)
```

```
John
Julia
```

More Control Flow: `break` and `continue`

- `break` : break out of current loop

```
In [73]:  1  x = 0
          2  while True:
          3      x += 1
          4      if x == 10:
          5          print(x)
          6          break
```

```
10
```

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

```
In [74]:  1  for name in ['John', 'Ben', 'Julia']:
          2      if name == 'Ben':
          3          continue
          4      print(name)
```

```
John
Julia
```

```
In [75]:  1  for x in range(5):
          2      if x == 1:
          3          continue
          4      print(x)
```

```
0
2
3
```

Generate a Range of Numbers: `range`

Generate a Range of Numbers: `range`

```
In [76]:    1  # create list of integers from 0 up to but not including 4
            2  a = []
            3  for x in range(4):
            4      a.append(x)
            5  a

Out[76]:  [0, 1, 2, 3]
```

```
In [77]:    1  range(4)

Out[77]:  range(0, 4)
```

Generate a Range of Numbers: `range`

```
In [76]:  1  # create list of integers from 0 up to but not including 4
          2  a = []
          3  for x in range(4):
          4      a.append(x)
          5  a
```

Out[76]: [0, 1, 2, 3]

```
In [77]:  1  range(4)
```

Out[77]: range(0, 4)

```
In [78]:  1  list(range(4))
```

Out[78]: [0, 1, 2, 3]

Generate a Range of Numbers: `range`

In [76]:
```python
# create list of integers from 0 up to but not including 4
a = []
for x in range(4):
    a.append(x)
a
```

Out[76]: `[0, 1, 2, 3]`

In [77]:
```python
range(4)
```

Out[77]: `range(0, 4)`

In [78]:
```python
list(range(4))
```

Out[78]: `[0, 1, 2, 3]`

In [79]:
```python
list(range(3,5)) # with a start and end+1
```

Out[79]: `[3, 4]`

Generate a Range of Numbers: `range`

In [76]:
```python
1  # create list of integers from 0 up to but not including 4
2  a = []
3  for x in range(4):
4      a.append(x)
5  a
```

Out[76]: [0, 1, 2, 3]

In [77]:
```python
1  range(4)
```

Out[77]: range(0, 4)

In [78]:
```python
1  list(range(4))
```

Out[78]: [0, 1, 2, 3]

In [79]:
```python
1  list(range(3,5)) # with a start and end+1
```

Out[79]: [3, 4]

In [80]:
```python
1  list(range(0,10,2)) # with start, end+1 and step-size
```

Out[80]: [0, 2, 4, 6, 8]

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

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

```
In [81]:   1  for i,x in enumerate(['a','b','c']):
           2      print(i,x)

0 a
1 b
2 c
```

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

```
In [81]:  1 for i,x in enumerate(['a','b','c']):
          2     print(i,x)
```

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

```
In [82]:  1 list(enumerate(['a','b','c']))
```

```
Out[82]: [(0, 'a'), (1, 'b'), (2, 'c')]
```

Sorting

Sorting

Two ways to sort a list:

Sorting

Two ways to sort a list:

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

Sorting

Two ways to sort a list:

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

```
In [83]:  1  x = [4,1,2,3]
          2  x.sort()
          3  assert x == [1,2,3,4], 'Not same lists'
```

Sorting

Two ways to sort a list:

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

```
In [83]:  1  x = [4,1,2,3]
          2  x.sort()
          3  assert x == [1,2,3,4], 'Not same lists'
```

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

Sorting

Two ways to sort a list:

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

```
In [83]:  1  x = [4,1,2,3]
          2  x.sort()
          3  assert x == [1,2,3,4], 'Not same lists'
```

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

```
In [84]:  1  x = [4,1,2,3]
          2  y = sorted(x)
          3  assert x == [4,1,2,3]
          4  assert y == [1,2,3,4]
```

Sorting Cont.

Sorting Cont.

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

Sorting Cont.

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

```python
In [85]:   1  assert sorted([1,2,3,4], reverse=True) == [4,3,2,1]
```

Sorting Cont.

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

```
In [85]:  1  assert sorted([1,2,3,4], reverse=True) == [4,3,2,1]
```

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

Sorting Cont.

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

```
In [85]:  1  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 [86]:  1  # for example, to sort a dictionary by value
          2  d = {'a':3,'b':5,'c':1}
          3
          4  # recall that .items() returns a set of key,value tuples
          5  s = sorted(d.items(), key=lambda x: x[1])
          6
          7  assert s == [('c', 1), ('a', 3), ('b', 5)]
```

```
In [87]:  1  sorted(d.items(), key=lambda x: x[0])
```

```
Out[87]:  [('a', 3), ('b', 5), ('c', 1)]
```

List Comprehensions

List Comprehensions

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

List Comprehensions

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

```
In [88]:  1 # which integers between 0 and 3 inclusive are divisible by 2?
          2 is_even = []
          3 for x in range(0,4):
          4     is_even.append(x%2 == 0)
          5 is_even

Out[88]: [True, False, True, False]
```

List Comprehensions

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

```
In [88]:  1  # which integers between 0 and 3 inclusive are divisible by 2?
          2  is_even = []
          3  for x in range(0,4):
          4      is_even.append(x%2 == 0)
          5  is_even
```

Out[88]: [True, False, True, False]

```
In [89]:  1  [x%2 == 0 for x in range(0,4)] # using a list comprehension
```

Out[89]: [True, False, True, False]

List Comprehensions

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

In [88]:
```python
1  # which integers between 0 and 3 inclusive are divisible by 2?
2  is_even = []
3  for x in range(0,4):
4      is_even.append(x%2 == 0)
5  is_even
```

Out[88]: [True, False, True, False]

In [89]:
```python
1  [x%2 == 0 for x in range(0,4)] # using a list comprehension
```

Out[89]: [True, False, True, False]

In [90]:
```python
1  # what are the indices of the vowels in 'apple'?
2  vowels = ['a','e','i','o','u']
3  [i for i,x in enumerate('apple') if x not in vowels]
```

Out[90]: [1, 2, 3]

Dictionary Comprehension

Dictionary Comprehension

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

Dictionary Comprehension

- list comprehension but for (key,value) pairs

- can add logic to dictionary creation

```
In [91]:   1  pairs = [(1,'e'),(2,'f'),(3,'g')]
```

Dictionary Comprehension

- list comprehension but for (key,value) pairs

- can add logic to dictionary creation

```
In [91]:  1  pairs = [(1,'e'),(2,'f'),(3,'g')]
```

```
In [92]:  1  dict(pairs)
```

Out[92]: {1: 'e', 2: 'f', 3: 'g'}

Dictionary Comprehension

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

```
In [91]:   1  pairs = [(1,'e'),(2,'f'),(3,'g')]
```

```
In [92]:   1  dict(pairs)
```

Out[92]: {1: 'e', 2: 'f', 3: 'g'}

```
In [93]:   1  # modify value and only include odd keys
           2  {key:'value_'+ val for key,val in pairs if key%2 == 1}
```

Out[93]: {1: 'value_e', 3: 'value_g'}

Object Oriented

Object Oriented

```python
In [96]:  1  class MyClass:
          2      """A descriptive docstring."""
          3
          4      # constructor
          5      def __init__(self,myvalue = 0):  # what happens when created
          6          # attributes
          7          self.myvalue = myvalue
          8
          9      def __repr__(self):    # what gets printed out (string repr.)
         10          return f'MyClass(myvalue={self.myvalue})'
         11
         12      # any other methods
         13
         14      def get_value(self):
         15          """Return the value in myvalue."""
         16          return self.myvalue
```

Object Oriented

```python
In [96]:
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
```

```python
In [97]:
x = MyClass(100)            # instantiate object

assert x.myvalue == 100     # access object attribute

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

Importing Modules

- Want to import a module/library? Use `import`

Importing Modules

- Want to import a module/library? Use `import`

```
In [98]:  1  import math
          2
          3  math.sqrt(2)

Out[98]:  1.4142135623730951
```

Importing Modules

- Want to import a module/library? Use `import`

```
In [98]:   1  import math
           2
           3  math.sqrt(2)

Out[98]:   1.4142135623730951
```

- Want to import a submodule or function from a module? Use `from`

Importing Modules

- Want to import a module/library? Use `import`

```
In [98]:  1  import math
          2
          3  math.sqrt(2)
```

Out[98]: 1.4142135623730951

- Want to import a submodule or function from a module? Use `from`

```
In [99]:  1  from math import sqrt,floor
          2
          3  print(sqrt(2))
          4  print(floor(sqrt(2)))
```

1.4142135623730951
1

Importing Modules Cont.

- Want to import a module using an alias? Use 'as'

Importing Modules Cont.

- Want to import a module using an alias? Use 'as'

```
In [100]:   1  import math as m
            2  m.sqrt(2)

Out[100]:   1.4142135623730951
```

Importing Modules Cont.

- Want to import a module using an alias? Use 'as'

```
In [100]:   1  import math as m
            2  m.sqrt(2)

Out[100]:   1.4142135623730951
```

- Don't do: `import *`

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

`collections` Module

`collections` Module

```
In [101]:   1  from collections import Counter, defaultdict
```

`collections` Module

```
In [101]:   1  from collections import Counter, defaultdict
```

- `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`

collections Module: **Counter**

```
In [102]:   1  c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
            2  c

Out[102]:  Counter({'red': 2, 'blue': 3, 'green': 1})
```

collections Module: Counter

```
In [102]:  1  c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
           2  c
```

Out[102]: Counter({'red': 2, 'blue': 3, 'green': 1})

```
In [103]:  1  c = Counter()
           2  for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
           3      c[word] += 1
```

collections Module: **Counter**

```python
c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
c
```

Counter({'red': 2, 'blue': 3, 'green': 1})

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

```python
c.most_common()
```

[('blue', 3), ('red', 2), ('green', 1)]

`collections` Module Cont.: `defaultdict`

`collections` Module Cont.: `defaultdict`

```
In [105]:   1  %xmode Minimal
            2  # reduce the amount printed when an exception is thrown
```

Exception reporting mode: Minimal

collections Module Cont.: **defaultdict**

```
1  %xmode Minimal
2  # reduce the amount printed when an exception is thrown
```

Exception reporting mode: Minimal

In [106]:
```
1  # create mapping from length of word to list of words
2  colors = ['red', 'blue', 'purple', 'gold', 'orange']
3  d = {}
4  for word in colors:
5      d[len(word)].append(word)
```

KeyError: 3

collections Module Cont. : **defaultdict**

In [105]:
```
1 %xmode Minimal
2 # reduce the amount printed when an exception is thrown
```

Exception reporting mode: Minimal

In [106]:
```
1 # create mapping from length of word to list of words
2 colors = ['red', 'blue', 'purple', 'gold', 'orange']
3 d = {}
4 for word in colors:
5     d[len(word)].append(word)
```

KeyError: 3

In [107]:
```
1 d = {}
2 for word in colors:
3     if len(word) in d:
4         d[len(word)].append(word)
5     else:
6         d[len(word)] = [word]
7 d
```

Out[107]: {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}

collections Module Cont. : **defaultdict**

```
1  %xmode Minimal
2  # reduce the amount printed when an exception is thrown
```

```
Exception reporting mode: Minimal
```

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

```
KeyError: 3
```

```
1  d = {}
2  for word in colors:
3      if len(word) in d:
4          d[len(word)].append(word)
5      else:
6          d[len(word)] = [word]
7  d
```

Out[107]: {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}

```
1  d = defaultdict(list)
2  for word in colors:
3      d[len(word)].append(word)
4  d
```

Out[108]: defaultdict(list, {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']})

collections Module Cont.: **defaultdict**

In [105]:
```
1  %xmode Minimal
2  # reduce the amount printed when an exception is thrown
```

Exception reporting mode: Minimal

In [106]:
```
1  # create mapping from length of word to list of words
2  colors = ['red', 'blue', 'purple', 'gold', 'orange']
3  d = {}
4  for word in colors:
5      d[len(word)].append(word)
```

KeyError: 3

In [107]:
```
1  d = {}
2  for word in colors:
3      if len(word) in d:
4          d[len(word)].append(word)
5      else:
6          d[len(word)] = [word]
7  d
```

Out[107]: {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}

In [108]:
```
1  d = defaultdict(list)
2  for word in colors:
3      d[len(word)].append(word)
4  d
```

Out[108]: defaultdict(list, {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']})

Contexts

Contexts

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

Contexts

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

```
In [110]:  1  with open('tmp_context_example.txt','w') as f:
           2      f.write('test')
```

Contexts

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

```
In [110]:   1  with open('tmp_context_example.txt','w') as f:
            2      f.write('test')
```

```
In [111]:   1  # instead of
            2  f = open('tmp_context_example.txt','w')
            3  f.write('test')
            4  f.close() # this is easy to forget to do
```

Contexts

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

In [110]:
```python
1  with open('tmp_context_example.txt','w') as f:
2      f.write('test')
```

In [111]:
```python
1  # instead of
2  f = open('tmp_context_example.txt','w')
3  f.write('test')
4  f.close() # this is easy to forget to do
```

In [112]:
```python
1  # remove the example file we just created
2  %rm tmp_context_example.txt
```

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

Questions?

Working with Data

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

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

Python Dynamic Typing

```
In [113]:   1  x = 5
            2  x = 'five'
```

Python Dynamic Typing

```
In [113]:   1  x = 5
            2  x = 'five'
```

- Note: still *strongly* typed

- Python is both a strongly typed and a dynamically typed language. Strong typing means that variables do have a type and that the type matters when performing operations on a variable. Dynamic typing means that the type of the variable is determined only during runtime.

Python Dynamic Typing

```
In [113]:   1  x = 5
            2  x = 'five'
```

- Note: still *strongly* typed

- Python is both a strongly typed and a dynamically typed language. Strong typing means that variables do have a type and that the type matters when performing operations on a variable. Dynamic typing means that the type of the variable is determined only during runtime.

```
In [114]:   1  x,y = 5,'five'
            2  x+y

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

Python Dynamic Typing

Python Dynamic Typing
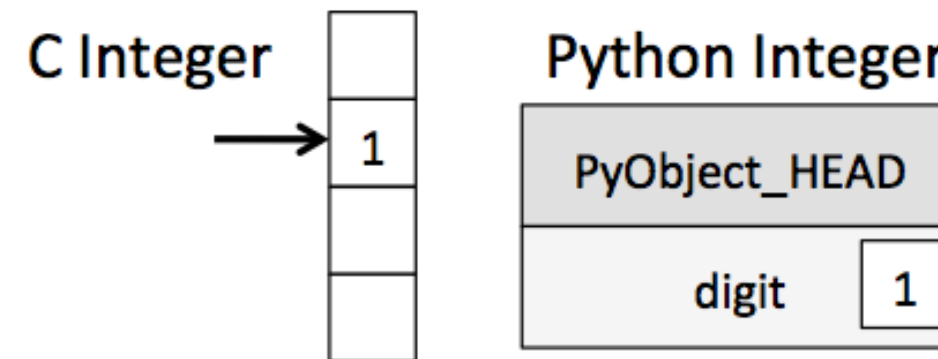
[PDHS Chap 2.](https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html)

- A C integer is essentially a label for a position in memory whose bytes encode an integer value.
- A Python integer is a pointer to a position in memory containing all the Python object information, including the bytes that contain the integer value.
  - **ob_refcnt**, a reference count that helps Python silently handle memory allocation and deallocation
  - **ob_type**, which encodes the type of the variable
  - **ob_size**, which specifies the size
  - **ob_digit**, which contains the actual integer value that we expect the Python variable to

NumPy Array vs Python List

- The NumPy array essentially contains a single pointer to one contiguous block of data.

- The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw earlier.

- The advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.

NumPy Array vs Python List

- The NumPy array essentially contains a single pointer to one contiguous block of data.

- The Python list, on the other hand, contains a pointer to a block of pointers, each of which in turn points to a full Python object like the Python integer we saw earlier.

- The advantage of the list is flexibility: because each list element is a full structure containing both data and type information, the list can be filled with data of any desired type. Fixed-type NumPy-style arrays lack this flexibility, but are much more efficient for storing and manipulating data.
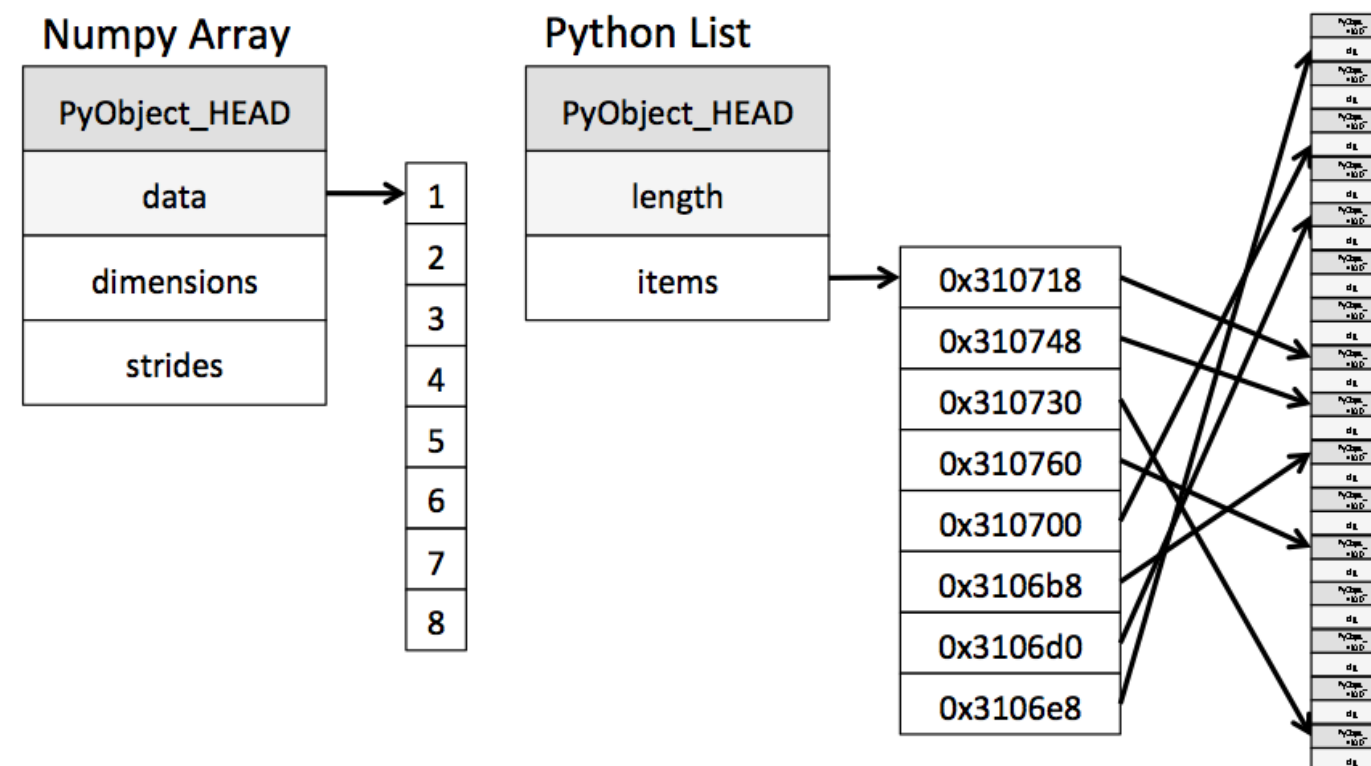
Importing NumPy

Importing NumPy

Often imported as alias `np`

```
In [115]:  1  a = 10
```

```
In [116]:  1  a??
```

Importing NumPy

Often imported as alias `np`

```
In [115]:  1  a = 10
```

```
In [116]:  1  a??
```

```
In [117]:  1  import numpy as np
           2
           3  np.random.randint(10,size=5)
```

Out[117]: array([0, 3, 8, 0, 9])

NumPy Datatypes

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

NumPy Arrays

```
In [118]:  1  x = np.array([1,2,3])
           2  x

Out[118]:  array([1, 2, 3])
```

NumPy Arrays

```
In [118]:  1  x = np.array([1,2,3])
           2  x
```

Out[118]: array([1, 2, 3])

```
In [119]:  1  type(x)
```

Out[119]: numpy.ndarray

NumPy Arrays

```
In [118]:   1  x = np.array([1,2,3])
            2  x
```

Out[118]: array([1, 2, 3])

```
In [119]:   1  type(x)
```

Out[119]: numpy.ndarray

```
In [120]:   1  # use dtype to show the datatype of the array
            2  x.dtype
```

Out[120]: dtype('int64')

NumPy Arrays

In [118]:
```python
1  x = np.array([1,2,3])
2  x
```

Out[118]: array([1, 2, 3])

In [119]:
```python
1  type(x)
```

Out[119]: numpy.ndarray

In [120]:
```python
1  # use dtype to show the datatype of the array
2  x.dtype
```

Out[120]: dtype('int64')

In [121]:
```python
1  # np arrays can only contain one datatype and default to the most flexible type
2  x = np.array([1,'two',3])
3  x
```

Out[121]: array(['1', 'two', '3'], dtype='<U21')

NumPy Arrays

In [118]:
```python
1 x = np.array([1,2,3])
2 x
```

Out[118]: array([1, 2, 3])

In [119]:
```python
1 type(x)
```

Out[119]: numpy.ndarray

In [120]:
```python
1 # use dtype to show the datatype of the array
2 x.dtype
```

Out[120]: dtype('int64')

In [121]:
```python
1 # np arrays can only contain one datatype and default to the most flexible type
2 x = np.array([1,'two',3])
3 x
```

Out[121]: array(['1', 'two', '3'], dtype='<U21')

In [122]:
```python
1 x.dtype
```

Out[122]: dtype('<U21')

NumPy Arrays

```
In [118]:   1  x = np.array([1,2,3])
            2  x
```

Out[118]: array([1, 2, 3])

```
In [119]:   1  type(x)
```

Out[119]: numpy.ndarray

```
In [120]:   1  # use dtype to show the datatype of the array
            2  x.dtype
```

Out[120]: dtype('int64')

```
In [121]:   1  # np arrays can only contain one datatype and default to the most flexible type
            2  x = np.array([1,'two',3])
            3  x
```

Out[121]: array(['1', 'two', '3'], dtype='<U21')

```
In [122]:   1  x.dtype
```

Out[122]: dtype('<U21')

```
In [123]:   1  # many different ways to create numpy arrays
            2  np.zeros(5,dtype=float)
```

Out[123]: array([0., 0., 0., 0., 0.])

NumPy Array Indexing

NumPy Array Indexing

- For single indices, works the same as list

NumPy Array Indexing

- For single indices, works the same as list

```
In [124]:  1  x = np.arange(1,6)
           2  x

Out[124]:  array([1, 2, 3, 4, 5])
```

NumPy Array Indexing

- For single indices, works the same as list

```
In [124]:    1  x = np.arange(1,6)
             2  x

Out[124]: array([1, 2, 3, 4, 5])

In [125]:    1  x[0], x[-1], x[-2], x[-4]

Out[125]: (1, 5, 4, 2)
```

NumPy Array Slicing

NumPy Array Slicing

```
1 x = np.arange(5) # note that in numpy it's arange instead of range
2 x
```

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

NumPy Array Slicing

```
In [126]:   1  x = np.arange(5) # note that in numpy it's arange instead of range
            2  x
```

Out[126]: array([0, 1, 2, 3, 4])

```
In [127]:   1  # return first two items, start:end (exclusive)
            2  x[0:2]
```

Out[127]: array([0, 1])

NumPy Array Slicing

```
In [126]:  1  x = np.arange(5) # note that in numpy it's arange instead of range
           2  x
```

Out[126]: array([0, 1, 2, 3, 4])

```
In [127]:  1  # return first two items, start:end (exclusive)
           2  x[0:2]
```

Out[127]: array([0, 1])

```
In [128]:  1  # missing start implies position 0
           2  x[:4]
```

Out[128]: array([0, 1, 2, 3])

NumPy Array Slicing

```
In [126]:  1  x = np.arange(5)  # note that in numpy it's arange instead of range
           2  x
```

Out[126]: array([0, 1, 2, 3, 4])

```
In [127]:  1  # return first two items, start:end (exclusive)
           2  x[0:2]
```

Out[127]: array([0, 1])

```
In [128]:  1  # missing start implies position 0
           2  x[:4]
```

Out[128]: array([0, 1, 2, 3])

```
In [129]:  1  # missing end implies length of array
           2  x[2:]
```

Out[129]: array([2, 3, 4])

NumPy Array Slicing

```
In [126]:  1  x = np.arange(5) # note that in numpy it's arange instead of range
           2  x
```

Out[126]: array([0, 1, 2, 3, 4])

```
In [127]:  1  # return first two items, start:end (exclusive)
           2  x[0:2]
```

Out[127]: array([0, 1])

```
In [128]:  1  # missing start implies position 0
           2  x[:4]
```

Out[128]: array([0, 1, 2, 3])

```
In [129]:  1  # missing end implies length of array
           2  x[2:]
```

Out[129]: array([2, 3, 4])

```
In [130]:  1  # return last two items
           2  x[-3:]
```

Out[130]: array([2, 3, 4])

NumPy Array Slicing with Steps

NumPy Array Slicing with Steps

```
In [131]:  1 x
Out[131]: array([0, 1, 2, 3, 4])
```

NumPy Array Slicing with Steps

```
In [131]:  1 x
```

Out[131]: array([0, 1, 2, 3, 4])

```
In [132]:  1 # return every other item from position 1 to 4 exclusive
           2 # start:end:step_size
           3 x[1:4:1]
```

Out[132]: array([1, 2, 3])

Reverse array with step-size of -1

Reverse array with step-size of -1

```
In [133]:  1 x

Out[133]:  array([0, 1, 2, 3, 4])
```

Reverse array with step-size of -1

```
In [133]:   1  x
```

Out[133]: array([0, 1, 2, 3, 4])

```
In [134]:   1  x[::-1]
```

Out[134]: array([4, 3, 2, 1, 0])

NumPy Fancy Indexing

NumPy Fancy Indexing

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

NumPy Fancy Indexing

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

```
In [135]:  1  x = np.arange(5,10)
           2  x

Out[135]:  array([5, 6, 7, 8, 9])
```

NumPy Fancy Indexing

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

```
In [135]:   1  x = np.arange(5,10)
            2  x

Out[135]:   array([5, 6, 7, 8, 9])
```

```
In [136]:   1  x[[0,3]]

Out[136]:   array([5, 8])
```

NumPy Fancy Indexing

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

```
In [135]:  1  x = np.arange(5,10)
           2  x
```

Out[135]: array([5, 6, 7, 8, 9])

```
In [136]:  1  x[[0,3]]
```

Out[136]: array([5, 8])

```
In [137]:  1  x[[0,2,-1]]
           2
```

Out[137]: array([5, 7, 9])

Boolean Indexing using a Boolean Mask

Boolean Indexing using a Boolean Mask

```
In [138]:  1  x
```

Out[138]: array([5, 6, 7, 8, 9])

Boolean Indexing using a Boolean Mask

```
In [138]:  1  x
```

Out[138]: array([5, 6, 7, 8, 9])

```
In [139]:  1  # Which indices have a value divisible by 2?
           2  # mod operator % returns remainder of division
           3  x%2 == 0
```

Out[139]: array([False, True, False, True, False])

Boolean Indexing using a Boolean Mask

```
In [138]:  1  x
```

Out[138]: array([5, 6, 7, 8, 9])

```
In [139]:  1  # Which indices have a value divisible by 2?
           2  # mod operator % returns remainder of division
           3  x%2 == 0
```

Out[139]: array([False, True, False, True, False])

```
In [140]:  1  # Which values are divisible by 2?
           2  x[x%2 == 0]
```

Out[140]: array([6, 8])

Boolean Indexing using a Boolean Mask

```
In [138]:  1  x
```

Out[138]: array([5, 6, 7, 8, 9])

```
In [139]:  1  # Which indices have a value divisible by 2?
           2  # mod operator % returns remainder of division
           3  x%2 == 0
```

Out[139]: array([False, True, False, True, False])

```
In [140]:  1  # Which values are divisible by 2?
           2  x[x%2 == 0]
```

Out[140]: array([6, 8])

```
In [141]:  1  # Which values are greater than 6?
           2  x[x >= 2]
```

Out[141]: array([5, 6, 7, 8, 9])

Boolean Indexing And Bitwise Operators

Boolean Indexing And Bitwise Operators

```
In [142]:   1 x
```

Out[142]: array([5, 6, 7, 8, 9])

Boolean Indexing And Bitwise Operators

```
In [142]:  1  x
```

Out[142]: array([5, 6, 7, 8, 9])

```
In [143]:  1  (x%2 == 0)
```

Out[143]: array([False, True, False, True, False])

Boolean Indexing And Bitwise Operators

```
In [142]:  1  x
```

Out[142]: array([5, 6, 7, 8, 9])

```
In [143]:  1  (x%2 == 0)
```

Out[143]: array([False, True, False, True, False])

```
In [144]:  1  (x > 6)
```

Out[144]: array([False, False, True, True, True])

Boolean Indexing And Bitwise Operators

```
In [142]:   1  x
```

```
Out[142]:  array([5, 6, 7, 8, 9])
```

```
In [143]:   1  (x%2 == 0)
```

```
Out[143]:  array([False,  True, False,  True, False])
```

```
In [144]:   1  (x > 6)
```

```
Out[144]:  array([False, False,  True,  True,  True])
```

```
In [145]:   1  # Which values are divisible by 2 AND greater than 6?
            2  # 'and' expexts both elements to be boolean, not arrays of booleans!
            3  (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()
```

Boolean Indexing And Bitwise Operators

```
In [142]:  1  x
```

```
Out[142]:  array([5, 6, 7, 8, 9])
```

```
In [143]:  1  (x%2 == 0)
```

```
Out[143]:  array([False,  True, False,  True, False])
```

```
In [144]:  1  (x > 6)
```

```
Out[144]:  array([False, False,  True,  True,  True])
```

```
In [145]:  1  # Which values are divisible by 2 AND greater than 6?
           2  # 'and' expexts both elements to be boolean, not arrays of booleans!
           3  (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 [146]:  1  # & compares each element pairwise
           2  (x%2 == 0) & (x > 6)
```

```
Out[146]:  array([False, False, False,  True, False])
```

```
In [147]:  1  x[np.array([False, False, False,  True, False])]
```

```
Out[147]:  array([8])
```

Boolean Indexing And Bitwise Operators

```
In [142]:  1  x
```

```
Out[142]:  array([5, 6, 7, 8, 9])
```

```
In [143]:  1  (x%2 == 0)
```

```
Out[143]:  array([False,  True, False,  True, False])
```

```
In [144]:  1  (x > 6)
```

```
Out[144]:  array([False, False,  True,  True,  True])
```

```
In [145]:  1  # Which values are divisible by 2 AND greater than 6?
           2  # 'and' expexts both elements to be boolean, not arrays of booleans!
           3  (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 [146]:  1  # & compares each element pairwise
           2  (x%2 == 0) & (x > 6)
```

```
Out[146]:  array([False, False, False,  True, False])
```

```
In [147]:  1  x[np.array([False, False, False,  True, False])]
```

```
Out[147]:  array([8])
```

```
In [148]:  1  x[(x%2 == 0) & (x > 6)]
```

```
Out[148]:  array([8])
```

Boolean Indexing And Bitwise Operators

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [149]:  1  # Which values are even AND greater than 6?
           2  x[(x%2 == 0) & (x > 6)]

Out[149]:  array([8])
```

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [149]:   1  # Which values are even AND greater than 6?
            2  x[(x%2 == 0) & (x > 6)]

Out[149]:  array([8])
```

- or : `|` (pipe)

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [149]:   1  # Which values are even AND greater than 6?
            2  x[(x%2 == 0) & (x > 6)]
```

Out[149]: array([8])

- or : `|` (pipe)

```
In [150]:   1  # which values are even OR greater than 6?
            2  x[(x%2 == 0) | (x > 6)]
```

Out[150]: array([6, 7, 8, 9])

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [149]:    1  # Which values are even AND greater than 6?
             2  x[(x%2 == 0) & (x > 6)]
```

```
Out[149]: array([8])
```

- or : `|` (pipe)

```
In [150]:    1  # which values are even OR greater than 6?
             2  x[(x%2 == 0) | (x > 6)]
```

```
Out[150]: array([6, 7, 8, 9])
```

- not : `~` (tilde)

```
In [151]:    1  ( (x%2 == 0) | (x > 6) )
```

```
Out[151]: array([False,  True,  True,  True,  True])
```

```
In [152]:    1  ~( (x%2 == 0) | (x > 6) )
```

```
Out[152]: array([ True, False, False, False, False])
```

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [149]:   1  # Which values are even AND greater than 6?
            2  x[(x%2 == 0) & (x > 6)]
```

Out[149]: array([8])

- or : `|` (pipe)

```
In [150]:   1  # which values are even OR greater than 6?
            2  x[(x%2 == 0) | (x > 6)]
```

Out[150]: array([6, 7, 8, 9])

- not : `~` (tilde)

```
In [151]:   1  ( (x%2 == 0) | (x > 6) )
```

Out[151]: array([False, True, True, True, True])

```
In [152]:   1  ~( (x%2 == 0) | (x > 6) )
```

Out[152]: array([True, False, False, False, False])

```
In [153]:   1  # which values are NOT (even OR greater than 6)
            2  x[~( (x%2 == 0) | (x > 6) )]
```

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [149]:  1  # Which values are even AND greater than 6?
           2  x[(x%2 == 0) & (x > 6)]
```

```
Out[149]: array([8])
```

- or : `|` (pipe)

```
In [150]:  1  # which values are even OR greater than 6?
           2  x[(x%2 == 0) | (x > 6)]
```

```
Out[150]: array([6, 7, 8, 9])
```

- not : `~` (tilde)

```
In [151]:  1  ( (x%2 == 0) | (x > 6) )
```

```
Out[151]: array([False,  True,  True,  True,  True])
```

```
In [152]:  1  ~( (x%2 == 0) | (x > 6) )
```

```
Out[152]: array([ True, False, False, False, False])
```

```
In [153]:  1  # which values are NOT (even OR greater than 6)
           2  x[~( (x%2 == 0) | (x > 6) )]
```

Indexing Review

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

Multidimensional Lists

```
In [154]:   1  x = [[1,2,3],[4,5,6]] # list of lists
            2  x
```

Out[154]: `[[1, 2, 3], [4, 5, 6]]`

Multidimensional Lists

```
In [154]:  1  x = [[1,2,3],[4,5,6]] # list of lists
           2  x
```

Out[154]: [[1, 2, 3], [4, 5, 6]]

```
In [155]:  1  # return first row
           2  x[0]
```

Out[155]: [1, 2, 3]

Multidimensional Lists

In [154]:
```
1  x = [[1,2,3],[4,5,6]] # list of lists
2  x
```

Out[154]: [[1, 2, 3], [4, 5, 6]]

In [155]:
```
1  # return first row
2  x[0]
```

Out[155]: [1, 2, 3]

In [156]:
```
1  # return first row, second column
2  x[1][2]
```

Out[156]: 6

Multidimensional Lists

In [154]:
```python
1  x = [[1,2,3],[4,5,6]] # list of lists
2  x
```

Out[154]: [[1, 2, 3], [4, 5, 6]]

In [155]:
```python
1  # return first row
2  x[0]
```

Out[155]: [1, 2, 3]

In [156]:
```python
1  # return first row, second column
2  x[1][2]
```

Out[156]: 6

In [157]:
```python
1  # return second column?
2  [row[1] for row in x]
```

Out[157]: [2, 5]

NumPy Multidimensional Arrays

NumPy Multidimensional Arrays

```
In [158]:   1  x = np.array([[1,2,3],[4,5,6]])
            2  x

Out[158]: array([[1, 2, 3],
                 [4, 5, 6]])
```

NumPy Multidimensional Arrays

In [158]:
```
1 x = np.array([[1,2,3],[4,5,6]])
2 x
```

Out[158]: array([[1, 2, 3],
                  [4, 5, 6]])

In [159]:
```
1 x[0,1] # first row, second column
```

Out[159]: 2

NumPy Multidimensional Arrays

```
In [158]:  1  x = np.array([[1,2,3],[4,5,6]])
           2  x
```

```
Out[158]:  array([[1, 2, 3],
                  [4, 5, 6]])
```

```
In [159]:  1  x[0,1] # first row, second column
```

```
Out[159]:  2
```

```
In [160]:  1  x[:,1] # first row
```

```
Out[160]:  array([2, 5])
```

NumPy Multidimensional Arrays

```
In [158]:    1  x = np.array([[1,2,3],[4,5,6]])
             2  x

Out[158]: array([[1, 2, 3],
                 [4, 5, 6]])

In [159]:    1  x[0,1] # first row, second column

Out[159]: 2

In [160]:    1  x[:,1] # first row

Out[160]: array([2, 5])

In [161]:    1  x[0,:] # first row (first to last column)

Out[161]: array([1, 2, 3])
```

NumPy Multidimensional Arrays

```
In [158]:    1  x = np.array([[1,2,3],[4,5,6]])
             2  x

Out[158]: array([[1, 2, 3],
                 [4, 5, 6]])

In [159]:    1  x[0,1] # first row, second column

Out[159]: 2

In [160]:    1  x[:,1] # first row

Out[160]: array([2, 5])

In [161]:    1  x[0,:] # first row (first to last column)

Out[161]: array([1, 2, 3])

In [162]:    1  x[:,1] # second column (first to last row)

Out[162]: array([2, 5])
```

NumPy Array Attributes

NumPy Array Attributes

```
In [163]:   1   x = np.array([[1,2,3],[4,5,6]])
```

NumPy Array Attributes

```
In [163]:  1  x = np.array([[1,2,3],[4,5,6]])
```

```
In [164]:  1  x.ndim # number of dimensions
```

```
Out[164]:  2
```

NumPy Array Attributes

```
In [163]:   1  x = np.array([[1,2,3],[4,5,6]])
```

```
In [164]:   1  x.ndim # number of dimensions
```

Out[164]: 2

```
In [165]:   1  x.shape # shape in each dimension
```

Out[165]: (2, 3)

NumPy Array Attributes

```
In [163]:  1  x = np.array([[1,2,3],[4,5,6]])
```

```
In [164]:  1  x.ndim # number of dimensions
```

Out[164]: 2

```
In [165]:  1  x.shape # shape in each dimension
```

Out[165]: (2, 3)

```
In [166]:  1  x.size # total number of elements
```

Out[166]: 6

NumPy Operations (UFuncs)

NumPy Operations (UFuncs)

```
In [167]:   1  x = [1,2,3]
            2  y = [4,5,6]
```

NumPy Operations (UFuncs)

```
In [167]:  1  x = [1,2,3]
           2  y = [4,5,6]
```

```
In [168]:  1  x+y
```

Out[168]: [1, 2, 3, 4, 5, 6]

NumPy Operations (UFuncs)

```
In [167]:   1  x = [1,2,3]
            2  y = [4,5,6]
```

```
In [168]:   1  x+y
```

```
Out[168]:  [1, 2, 3, 4, 5, 6]
```

```
In [169]:   1  x = np.array([1,2,3])
            2  y = np.array([4,5,6])
```

NumPy Operations (UFuncs)

```
In [167]:  1  x = [1,2,3]
           2  y = [4,5,6]
```

```
In [168]:  1  x+y
```

Out[168]: [1, 2, 3, 4, 5, 6]

```
In [169]:  1  x = np.array([1,2,3])
           2  y = np.array([4,5,6])
```

```
In [170]:  1  x+y
```

Out[170]: array([5, 7, 9])

NumPy Operations (UFuncs)

```
In [167]:  1  x = [1,2,3]
           2  y = [4,5,6]
```

```
In [168]:  1  x+y
```

```
Out[168]:  [1, 2, 3, 4, 5, 6]
```

```
In [169]:  1  x = np.array([1,2,3])
           2  y = np.array([4,5,6])
```

```
In [170]:  1  x+y
```

```
Out[170]:  array([5, 7, 9])
```

```
In [171]:  1  %time sum(range(0,int(1e8)))
```

```
CPU times: user 1.38 s, sys: 5.57 ms, total: 1.39 s
Wall time: 1.38 s
```

```
Out[171]:  4999999950000000
```

NumPy Operations (UFuncs)

```
In [167]:  1  x = [1,2,3]
           2  y = [4,5,6]
```

```
In [168]:  1  x+y
```

Out[168]: [1, 2, 3, 4, 5, 6]

```
In [169]:  1  x = np.array([1,2,3])
           2  y = np.array([4,5,6])
```

```
In [170]:  1  x+y
```

Out[170]: array([5, 7, 9])

```
In [171]:  1  %time sum(range(0,int(1e8)))
```

```
CPU times: user 1.38 s, sys: 5.57 ms, total: 1.39 s
Wall time: 1.38 s
```

Out[171]: 4999999950000000

```
In [172]:  1  %time np.arange(0,int(1e8)).sum()
```

```
CPU times: user 289 ms, sys: 278 ms, total: 568 ms
Wall time: 591 ms
```

Out[172]: 4999999950000000

NumPy Broadcasting

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [173]:    1  # square every element in a list
             2  x = [1,2,3]
```

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [173]:  1  # square every element in a list
           2  x = [1,2,3]
```

```
In [174]:  1  x**2
```

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

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [173]:   1  # square every element in a list
            2  x = [1,2,3]
```

```
In [174]:   1  x**2
```

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

```
In [175]:   1  # square every element in a numpy array
            2  x = np.array([1,2,3])
```

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [173]:   1  # square every element in a list
            2  x = [1,2,3]
```

```
In [174]:   1  x**2
```

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

```
In [175]:   1  # square every element in a numpy array
            2  x = np.array([1,2,3])
```

```
In [176]:   1  x**2
```

Out[176]: array([1, 4, 9])

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [173]:   1  # square every element in a list
            2  x = [1,2,3]
```

```
In [174]:   1  x**2
```

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

```
In [175]:   1  # square every element in a numpy array
            2  x = np.array([1,2,3])
```

```
In [176]:   1  x**2
```

Out[176]: array([1, 4, 9])

```
In [177]:   1  a = np.array([1.0, 2.0, 3.0])
            2  b = 2.0
            3  a * b
```

Out[177]: array([2., 4., 6.])

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [173]:   1  # square every element in a list
            2  x = [1,2,3]
```

```
In [174]:   1  x**2
```

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

```
In [175]:   1  # square every element in a numpy array
            2  x = np.array([1,2,3])
```

```
In [176]:   1  x**2
```

Out[176]: array([1, 4, 9])

```
In [177]:   1  a = np.array([1.0, 2.0, 3.0])
            2  b = 2.0
            3  a * b
```

Out[177]: array([2., 4., 6.])

is equivalent to

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [173]:  1  # square every element in a list
           2  x = [1,2,3]
```

```
In [174]:  1  x**2
```

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

```
In [175]:  1  # square every element in a numpy array
           2  x = np.array([1,2,3])
```

```
In [176]:  1  x**2
```

Out[176]: array([1, 4, 9])

```
In [177]:  1  a = np.array([1.0, 2.0, 3.0])
           2  b = 2.0
           3  a * b
```

Out[177]: array([2., 4., 6.])

is equivalent to

NumPy `random` Submodule

NumPy `random` Submodule

Provides many random sampling functions

```
In [179]:  1  from numpy.random import seed, randn
```

```
In [180]:  1  seed(1991)
```

```
In [181]:  1  randn(10)
```

```
Out[181]: array([-0.94343179,  0.04928867, -1.28887945, -0.12919796,  0.73604408,
                  -0.27182505, -0.22028639, -0.2412858 ,  0.78027846, -0.95460321])
```

NumPy `random` Submodule

Provides many random sampling functions

```
In [179]:   1  from numpy.random import seed, randn
```

```
In [180]:   1  seed(1991)
```

```
In [181]:   1  randn(10)
```

```
Out[181]:  array([-0.94343179,  0.04928867, -1.28887945, -0.12919796,  0.73604408,
                   -0.27182505, -0.22028639, -0.2412858 ,  0.78027846, -0.95460321])
```

`from numpy.random import ...`

- `rand` : random floats

- `randint` : random integers

- `randn` : standard normal distribution

- `permutation` : random permutation

Questions?