Elements Of Data Science - F2023

Week 2: Python Intro/Review and Numpy

9/18/2023

TODOs

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

- Complete Week 2 Quiz

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-f23
$ 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

```
# 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]:
```
x = 3
x = 3.14
x = 'apple'
x
```

Out[2]: 'apple'

Dynamic Typing

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

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

Out[2]: 'apple'

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


add_two(2)
```

Out[4]: 4

Functions

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


add_two(2)
```

Out[4]: 4

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

```
Help on function add_two in module __main__:

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

Functions

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


add_two(2)
```

Out[4]: 4

In [5]:
```
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]:
```
def subtract(x,y):
    return x-y

subtract(3,1)
```

Out[6]: 2

Function Arguments

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

In [6]:
```
def subtract(x,y):
    return x-y

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

subtract(3,1)
```

Out[6]: 2

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

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

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]:
```
x = 3.1415

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

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

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

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

In [10]:
```
'the value of x is {:0.10f}'.format(x)
```

Out[10]: 'the value of x is 3.1415000000'

String Formatting

In [8]:
```
x = 3.1415

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

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

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

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

In [10]:
```
'the value of x is {:0.10f}'.format(x)
```

Out[10]: 'the value of x is 3.1415000000'

In [11]:
```
f'the value of x is {x:0.2f}'
# 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]:
```
x = 3.1415

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

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

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

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

In [10]:
```
'the value of x is {:0.10f}'.format(x)
```

Out[10]: 'the value of x is 3.1415000000'

In [11]:
```
f'the value of x is {x:0.2f}'
# 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]:
```
x = 3.1415

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

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

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

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

In [10]:
```
'the value of x is {:0.10f}'.format(x)
```

Out[10]: 'the value of x is 3.1415000000'

In [11]:
```
f'the value of x is {x:0.2f}'
# 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]:
```
f'x = {x:0.2f}'
```

Out[12]: 'x = 3.14'

String Formatting

In [8]:
```
x = 3.1415

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

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

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

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

In [10]:
```
'the value of x is {:0.10f}'.format(x)
```

Out[10]: 'the value of x is 3.1415000000'

In [11]:
```
f'the value of x is {x:0.2f}'
# 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]:
```
f'x = {x:0.2f}'
```

Out[12]: 'x = 3.14'

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

Out[13]: 'x = 3.14'

String Formatting Cont.

String Formatting Cont.

In [14]:
```
"""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.'
```

String Formatting Cont.

In [14]:
```
"""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]:
```
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]:
```
"""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]:
```
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.
```

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

String Formatting Cont.

In [14]:
```
"""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]:
```
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.
```

- common specifiers: `%s strings, %d integers, %f floats`
- to learn more https://realpython.com/python-string-formatting/

Python Data Types Continued: `list`

Python Data Types Continued: `list`

In [16]:
```
# elements of a python list do not all have to be of the same type
x = [42,'e',2.0]
x
```

Out[16]:
```
[42, 'e', 2.0]
```

Python Data Types Continued: `list`

In [16]:
```
# elements of a python list do not all have to be of the same type
x = [42,'e',2.0]
x
```

Out[16]:
```
[42, 'e', 2.0]
```

In [17]:
```
x[0] # indexing
```

Out[17]:
```
42
```

Python Data Types Continued: `list`

In [16]:
```
# elements of a python list do not all have to be of the same type
x = [42,'e',2.0]
x
```

Out[16]:
```
[42, 'e', 2.0]
```

In [17]:
```
x[0] # indexing
```

Out[17]:
```
42
```

In [18]:
```
x[-3] # reverse indexing
```

Out[18]:
```
42
```

Python Data Types Continued: `list`

In [16]:
```
# elements of a python list do not all have to be of the same type
x = [42,'e',2.0]
x
```

Out[16]: [42, 'e', 2.0]

In [17]:
```
x[0] # indexing
```

Out[17]: 42

In [18]:
```
x[-3] # reverse indexing
```

Out[18]: 42

In [19]:
```
x[2] = 4 # assignment
x
```

Out[19]: [42, 'e', 4]

Python Data Types Continued: `list`

In [16]:
```
# elements of a python list do not all have to be of the same type
x = [42,'e',2.0]
x
```

Out[16]:
```
[42, 'e', 2.0]
```

In [17]:
```
x[0] # indexing
```

Out[17]:
```
42
```

In [18]:
```
x[-3] # reverse indexing
```

Out[18]:
```
42
```

In [19]:
```
x[2] = 4 # assignment
x
```

Out[19]:
```
[42, 'e', 4]
```

In [20]:
```
x.append('a') # add a value to list
x
```

Out[20]:
```
[42, 'e', 4, 'a']
```

Python Data Types Continued: `list`

In [16]:
```
# elements of a python list do not all have to be of the same type
x = [42,'e',2.0]
x
```

Out[16]: [42, 'e', 2.0]

In [17]:
```
x[0] # indexing
```

Out[17]: 42

In [18]:
```
x[-3] # reverse indexing
```

Out[18]: 42

In [19]:
```
x[2] = 4 # assignment
x
```

Out[19]: [42, 'e', 4]

In [20]:
```
x.append('a') # add a value to list
x
```

Out[20]: [42, 'e', 4, 'a']

In [21]:
```
value_at_1 = x.pop(1) # remove/delete at index
x
```

Out[21]: [42, 4, 'a']

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

In [22]:
```
x = {'b':[2,1],'a':1,'c':4}
# or x = dict(b=2,a=1,c=4)
x
```

Out[22]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

In [22]:
```
x = {'b':[2,1],'a':1,'c':4}
# or x = dict(b=2,a=1,c=4)
x
```

Out[22]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

In [23]:
```
# index into dictionary using key
x['b']
```

Out[23]:
```
[2, 1]
```

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

In [22]:
```
x = {'b':[2,1],'a':1,'c':4}
# or x = dict(b=2,a=1,c=4)
x
```

Out[22]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

In [23]:
```
# index into dictionary using key
x['b']
```

Out[23]:
```
[2, 1]
```

In [24]:
```
# assign a value to a (new or existing) key
x['d'] = 3
x
```

Out[24]:
```
{'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}
```

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

In [22]:
```
x = {'b':[2,1],'a':1,'c':4}
# or x = dict(b=2,a=1,c=4)
x
```

Out[22]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

In [23]:
```
# index into dictionary using key
x['b']
```

Out[23]:
```
[2, 1]
```

In [24]:
```
# assign a value to a (new or existing) key
x['d'] = 3
x
```

Out[24]:
```
{'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}
```

In [25]:
```
# remove/delete
# can specify a return a value if key does not exist (here it's None), otherwise throws an exception
x.pop('d',None)
```

Out[25]:
```
3
```

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

In [22]:
```
x = {'b':[2,1],'a':1,'c':4}
# or x = dict(b=2,a=1,c=4)
x
```

Out[22]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

In [23]:
```
# index into dictionary using key
x['b']
```

Out[23]:
```
[2, 1]
```

In [24]:
```
# assign a value to a (new or existing) key
x['d'] = 3
x
```

Out[24]:
```
{'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}
```

In [25]:
```
# remove/delete
# can specify a return a value if key does not exist (here it's None), otherwise throws an exception
x.pop('d',None)
```

Out[25]:
```
3
```

In [26]:
```
x
```

Out[26]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

Python Data Types Continued: `dict` Cont.

Python Data Types Continued: `dict` Cont.

In [27]:
```
# using the same dictionary
x
```

Out[27]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

Python Data Types Continued: `dict` Cont.

In [27]:
```
# using the same dictionary
x
```

Out[27]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

In [28]:
```
# get a set of keys
x.keys()
```

Out[28]:
```
dict_keys(['b', 'a', 'c'])
```

Python Data Types Continued: `dict` Cont.

In [27]:
```
# using the same dictionary
x
```

Out[27]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

In [28]:
```
# get a set of keys
x.keys()
```

Out[28]:
```
dict_keys(['b', 'a', 'c'])
```

In [29]:
```
# get a set of values
x.values()
```

Out[29]:
```
dict_values([[2, 1], 1, 4])
```

Python Data Types Continued: `dict` Cont.

In [27]:
```
# using the same dictionary
x
```

Out[27]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

In [28]:
```
# get a set of keys
x.keys()
```

Out[28]:
```
dict_keys(['b', 'a', 'c'])
```

In [29]:
```
# get a set of values
x.values()
```

Out[29]:
```
dict_values([[2, 1], 1, 4])
```

In [30]:
```
# get a set of (key,value) tuples
x.items()
```

Out[30]:
```
dict_items([('b', [2, 1]), ('a', 1), ('c', 4)])
```

Python Data Types Continued: `dict` Cont.

In [27]:
```
# using the same dictionary
x
```

Out[27]:
```
{'b': [2, 1], 'a': 1, 'c': 4}
```

In [28]:
```
# get a set of keys
x.keys()
```

Out[28]:
```
dict_keys(['b', 'a', 'c'])
```

In [29]:
```
# get a set of values
x.values()
```

Out[29]:
```
dict_values([[2, 1], 1, 4])
```

In [30]:
```
# get a set of (key,value) tuples
x.items()
```

Out[30]:
```
dict_items([('b', [2, 1]), ('a', 1), ('c', 4)])
```

In [31]:
```
# get a list of (key,value) pairs
list(x.items())
```

Out[31]:
```
[('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 [32]:
```
1 x = (2,'e',3,4)
2 x
```

Out[32]:
```
(2, 'e', 3, 4)
```

Python Data Types Continued: `tuple`

- like a list, but **immutable**

In [32]:
```
1 x = (2,'e',3,4)
2 x
```

Out[32]: (2, 'e', 3, 4)

In [33]:
```
1 x[0] # indexing
```

Out[33]: 2

Python Data Types Continued: `tuple`

- like a list, but **immutable**

In [32]:
```
1 x = (2,'e',3,4)
2 x
```

Out[32]:
```
(2, 'e', 3, 4)
```

In [33]:
```
1 x[0] # indexing
```

Out[33]:
```
2
```

In [34]:
```
1 x[0] = 3 # assignment? Nope, error: immutable`
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[34], line 1
----> 1 x[0] = 3

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

Python Data Types Continued: `set`

Python Data Types Continued: `set`

In [35]:
```
1 x = {2,'e','e'} # or set([2,'e','e'])
2 x
```

Out[35]:
```
{2, 'e'}
```

Python Data Types Continued: `set`

In [35]:
```
x = {2,'e','e'} # or set([2,'e','e'])
x
```

Out[35]: {2, 'e'}

In [36]:
```
x.add(1) # insert
x
```

Out[36]: {1, 2, 'e'}

Python Data Types Continued: `set`

In [35]:
```
x = {2,'e','e'} # or set([2,'e','e'])
x
```

Out[35]: {2, 'e'}

In [36]:
```
x.add(1) # insert
x
```

Out[36]: {1, 2, 'e'}

In [37]:
```
x.remove('e') # remove/delete
x
```

Out[37]: {1, 2}

Python Data Types Continued: `set`

In [35]:
```
x = {2,'e','e'} # or set([2,'e','e'])
x
```

Out[35]: {2, 'e'}

In [36]:
```
x.add(1) # insert
x
```

Out[36]: {1, 2, 'e'}

In [37]:
```
x.remove('e') # remove/delete
x
```

Out[37]: {1, 2}

In [38]:
```
x.intersection({2,3})
```

Out[38]: {2}

Python Data Types Continued: `set`

In [35]:
```
x = {2,'e','e'} # or set([2,'e','e'])
x
```

Out[35]: {2, 'e'}

In [36]:
```
x.add(1) # insert
x
```

Out[36]: {1, 2, 'e'}

In [37]:
```
x.remove('e') # remove/delete
x
```

Out[37]: {1, 2}

In [38]:
```
x.intersection({2,3})
```

Out[38]: {2}

In [39]:
```
x.difference({2,3})
```

Out[39]: {1}

Python Data Types Continued: `set`

In [35]:
```
x = {2,'e','e'} # or set([2,'e','e'])
x
```

Out[35]:
```
{2, 'e'}
```

In [36]:
```
x.add(1) # insert
x
```

Out[36]:
```
{1, 2, 'e'}
```

In [37]:
```
x.remove('e') # remove/delete
x
```

Out[37]:
```
{1, 2}
```

In [38]:
```
x.intersection({2,3})
```

Out[38]:
```
{2}
```

In [39]:
```
x.difference({2,3})
```

Out[39]:
```
{1}
```

In [40]:
```
x[0] # cannot index into a set
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[40], line 1
----> 1 x[0]

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

Determining Length with `len`

Determining Length with `len`

In [41]:
```
1 len([1,2,3])
```

Out[41]: 3

Determining Length with `len`

In [41]:
```
1 len([1,2,3])
```

Out[41]: 3

In [42]:
```
1 len({'a':1,'b':2,'c':3})
```

Out[42]: 3

Determining Length with `len`

In [41]:
```
1 len([1,2,3])
```

Out[41]: 3

In [42]:
```
1 len({'a':1,'b':2,'c':3})
```

Out[42]: 3

In [43]:
```
1 len('apple')
```

Out[43]: 5

Exceptions

Exceptions

In [44]:
```
1 'a' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[44], line 1
----> 1 'a' + 2

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

Exceptions

```
In [44]: 1 'a' + 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[44], 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`

In [45]:

```
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
```

Catching Exceptions with `try-except`

In [45]:
```
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 [46]:
```
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

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 [47]:
```
1 3 == 3
```

Out[47]: True

Comparison Operators

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

In [47]:
```
1 3 == 3
```

Out[47]: True

In [48]:
```
1 3 != 4
```

Out[48]: True

Comparison Operators

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

```
In [47]:
1 3 == 3
```

```
Out[47]: True
```

```
In [48]:
1 3 != 4
```

```
Out[48]: True
```

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

Comparison Operators

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

In [47]:
```
1 3 == 3
```

Out[47]: True

In [48]:
```
1 3 != 4
```

Out[48]: True

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

In [49]:
```
1 3 < 4
```

Out[49]: True

Logical Operators

Logical Operators

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

Logical Operators

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

In [50]:
```
( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

Out[50]: True

Logical Operators

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

In [50]:
```
1 ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

Out[50]: True

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

Logical Operators

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

In [50]:
```
( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

Out[50]: True

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

In [51]:
```
any([0,0,1])
```

Out[51]: True

Logical Operators

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

In [50]:
```
1 ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

Out[50]: True

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

In [51]:
```
1 any([0,0,1])
```

Out[51]: True

- `all()`: all elements are true

Logical Operators

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

In [50]:
```
1 ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

Out[50]: True

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

In [51]:
```
1 any([0,0,1])
```

Out[51]: True

- `all()`: all elements are true

In [52]:
```
1 all([0,0,1])
```

Out[52]: False

Logical Operators

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

In [50]:
```
( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

Out[50]: True

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

In [51]:
```
any([0,0,1])
```

Out[51]: True

- `all()`: all elements are true

In [52]:
```
all([0,0,1])
```

Out[52]: 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 [53]:
```
1 assert 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 [53]:
```
1 assert 2+2 == 4
```

In [54]:
```
1 assert 1 == 0
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[54], line 1
----> 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 [53]:
```
1 assert 2+2 == 4
```

In [54]:
```
1 assert 1 == 0
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[54], line 1
----> 1 assert 1 == 0

AssertionError:
```

In [55]:
```
1 # can add an error message
2 assert 1 == 0, "1 does not equal 0"
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[55], line 2
      1 # can add an error message
----> 2 assert 1 == 0, "1 does not equal 0"

AssertionError: 1 does not equal 0
```

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 [56]:

```
x = 3
if x > 0:
    print('x > 0')
elif x < 0:
    print('x < 0')
else:
    print('x == 0')
```

```
x > 0
```

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

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

In [56]:
```
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`

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

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

In [56]:
```
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 [57]:
```
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 [58]:
```
a = []
for x in [0,1,2]:
    a.append(x)
a
```

Out[58]:
```
[0, 1, 2]
```

More Control Flow: `for` and `while`

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

In [58]:
```
a = []
for x in [0,1,2]:
    a.append(x)
a
```

Out[58]:
```
[0, 1, 2]
```

- `while` something is true

More Control Flow: `for` and `while`

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

In [58]:
```
a = []
for x in [0,1,2]:
    a.append(x)
a
```

Out[58]: [0, 1, 2]

- `while` something is true

In [59]:
```
x = 0
while x < 3:
    x += 1
x
```

Out[59]: 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 [60]:

```
x = 0
while True:
    x += 1
    if x == 3:
        print(x)
        break
```

```
3
```

More Control Flow: `break` and `continue`

- `break` : break out of current loop

In [60]:
```
x = 0
while True:
    x += 1
    if x == 3:
        print(x)
        break
```

```
3
```

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

More Control Flow: `break` and `continue`

- `break` : break out of current loop

In [60]:
```
x = 0
while True:
    x += 1
    if x == 3:
        print(x)
        break
```

```
3
```

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

In [61]:
```
for x in range(10):
    if x == 1:
        continue
    print(x)
```

```
0
2
3
4
5
6
7
8
9
```

Generate a Range of Numbers: `range`

Generate a Range of Numbers: `range`

In [62]:
```
# create list of integers from 0 up to but not including 4
a = []
for x in range(4):
    a.append(x)
a
```

Out[62]:
```
[0, 1, 2, 3]
```

Generate a Range of Numbers: `range`

In [62]:
```
# create list of integers from 0 up to but not including 4
a = []
for x in range(4):
    a.append(x)
a
```

Out[62]: [0, 1, 2, 3]

In [63]:
```
list(range(4))
```

Out[63]: [0, 1, 2, 3]

Generate a Range of Numbers: `range`

In [62]:
```
# create list of integers from 0 up to but not including 4
a = []
for x in range(4):
    a.append(x)
a
```

Out[62]: [0, 1, 2, 3]

In [63]:
```
list(range(4))
```

Out[63]: [0, 1, 2, 3]

In [64]:
```
list(range(3,5)) # with a start and end+1
```

Out[64]: [3, 4]

Generate a Range of Numbers: `range`

In [62]:
```
# create list of integers from 0 up to but not including 4
a = []
for x in range(4):
    a.append(x)
a
```

Out[62]: [0, 1, 2, 3]

In [63]:
```
list(range(4))
```

Out[63]: [0, 1, 2, 3]

In [64]:
```
list(range(3,5)) # with a start and end+1
```

Out[64]: [3, 4]

In [65]:
```
list(range(0,10,2)) # with start, end+1 and step-size
```

Out[65]: [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 [66]:
```
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 [66]:
```
for i,x in enumerate(['a','b','c']):
    print(i,x)
```

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

In [67]:
```
list(enumerate(['a','b','c']))
```

Out[67]:
```
[(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 [68]:
```
x = [4,1,2,3]
x.sort()
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 [68]:
```
x = [4,1,2,3]
x.sort()
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 [68]:
```
x = [4,1,2,3]
x.sort()
assert x == [1,2,3,4], 'Not same lists'
```

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

In [69]:
```
x = [4,1,2,3]
y = sorted(x)
assert x == [4,1,2,3]
assert y == [1,2,3,4]
```

Sorting Cont.

Sorting Cont.

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

Sorting Cont.

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

In [70]:
```
1 assert sorted([1,2,3,4], reverse=True) == [4,3,2,1]
```

Sorting Cont.

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

In [70]:
```
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 [70]:
```
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 [71]:
```
# for example, to sort a dictionary by value
d = {'a':3,'b':5,'c':1}

# recall that .items() returns a set of key,value tuples
s = sorted(d.items(), key=lambda x: x[1])

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

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 [72]:
```
# which integers 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[72]:
```
[True, False, True, False]
```

List Comprehensions

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

In [72]:
```
# which integers 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[72]:
```
[True, False, True, False]
```

In [73]:
```
[x%2 == 0 for x in range(0,4)] # using a list comprehension
```

Out[73]:
```
[True, False, True, False]
```

List Comprehensions

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

In [72]:
```
# which integers 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[72]:
```
[True, False, True, False]
```

In [73]:
```
[x%2 == 0 for x in range(0,4)] # using a list comprehension
```

Out[73]:
```
[True, False, True, False]
```

In [74]:
```
# 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[74]:
```
[0, 4]
```

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 [75]:
```
1 pairs = [(1,'e'),(2,'f'),(3,'g')]
```

Dictionary Comprehension

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

In [75]:
```
1 pairs = [(1,'e'),(2,'f'),(3,'g')]
```

In [76]:
```
1 dict(pairs)
```

Out[76]:
```
{1: 'e', 2: 'f', 3: 'g'}
```

Dictionary Comprehension

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

In [75]:
```
pairs = [(1,'e'),(2,'f'),(3,'g')]
```

In [76]:
```
dict(pairs)
```

Out[76]:
```
{1: 'e', 2: 'f', 3: 'g'}
```

In [77]:
```
# modify value and only include odd keys
{key:'value_'+str(val) for key,val in pairs if key%2 == 1}
```

Out[77]:
```
{1: 'value_e', 3: 'value_g'}
```

Object Oriented

Object Oriented

In [80]:

```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
```

Object Oriented

In [80]:
```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 [81]:
```python
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 [82]:
```
1 import math
2
3 math.sqrt(2)
```

Out[82]:
```
1.4142135623730951
```

Importing Modules

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

In [82]:
```
1 import math
2
3 math.sqrt(2)
```

Out[82]:
```
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 [82]:
```
1 import math
2
3 math.sqrt(2)
```

Out[82]:
```
1.4142135623730951
```

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

In [83]:
```
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 [84]:
```
1 import math as m
2 m.sqrt(2)
```

Out[84]:
```
1.4142135623730951
```

Importing Modules Cont.

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

In [84]:
```
1 import math as m
2 m.sqrt(2)
```

Out[84]: 1.4142135623730951

- Don't do: `import *`

```
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 [85]:
```
1 from collections import Counter, defaultdict
```

`collections` Module

In [85]:
```
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 [86]:
```
1 c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
2 c
```

Out[86]:
```
Counter({'red': 2, 'blue': 3, 'green': 1})
```

`collections` Module: `Counter`

In [86]:
```
c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
c
```

Out[86]:
```
Counter({'red': 2, 'blue': 3, 'green': 1})
```

In [87]:
```
c = Counter()
for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
    c[word] += 1
```

`collections` Module: `Counter`

In [86]:
```
c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
c
```

Out[86]:
```
Counter({'red': 2, 'blue': 3, 'green': 1})
```

In [87]:
```
c = Counter()
for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:
    c[word] += 1
```

In [88]:
```
c.most_common()
```

Out[88]:
```
[('blue', 3), ('red', 2), ('green', 1)]
```

`collections` Module Cont. : `defaultdict`

`collections` Module Cont.: `defaultdict`

In [89]:
```
%xmode Minimal
# reduce the amount printed when an exception is thrown
```

```
Exception reporting mode: Minimal
```

`collections` Module Cont.: `defaultdict`

In [89]:
```
%xmode Minimal
# reduce the amount printed when an exception is thrown
```

```
Exception reporting mode: Minimal
```

In [90]:
```
# 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
```

`collections` Module Cont.: `defaultdict`

In [89]:
```
%xmode Minimal
# reduce the amount printed when an exception is thrown
```

```
Exception reporting mode: Minimal
```

In [90]:
```
# 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 [91]:
```
d = {}
for word in colors:
    if len(word) in d:
        d[len(word)].append(word)
    else:
        d[len(word)] = [word]
d
```

Out[91]:
```
{3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}
```

`collections` Module Cont.: `defaultdict`

In [89]:
```
%xmode Minimal
# reduce the amount printed when an exception is thrown
```

```
Exception reporting mode: Minimal
```

In [90]:
```
# 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 [91]:
```
d = {}
for word in colors:
    if len(word) in d:
        d[len(word)].append(word)
    else:
        d[len(word)] = [word]
d
```

Out[91]:
```
{3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}
```

In [92]:
```
d = defaultdict(list)
for word in colors:
     d[len(word)].append(word)
d
```

Out[92]:
```
defaultdict(list, {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']})
```

`collections` Module Cont.: `defaultdict`

In [89]:
```
%xmode Minimal
# reduce the amount printed when an exception is thrown
```

```
Exception reporting mode: Minimal
```

In [90]:
```
# 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 [91]:
```
d = {}
for word in colors:
    if len(word) in d:
        d[len(word)].append(word)
    else:
        d[len(word)] = [word]
d
```

Out[91]:
```
{3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}
```

In [92]:
```
d = defaultdict(list)
for word in colors:
     d[len(word)].append(word)
d
```

Out[92]:
```
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 [94]:
```
with open('tmp_context_example.txt','w') as f:
    f.write('test')
```

Contexts

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

In [94]:
```
with open('tmp_context_example.txt','w') as f:
    f.write('test')
```

In [95]:
```
# instead of
f = open('tmp_context_example.txt','w')
f.write('test')
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 [94]:
```
with open('tmp_context_example.txt','w') as f:
    f.write('test')
```

In [95]:
```
# instead of
f = open('tmp_context_example.txt','w')
f.write('test')
f.close() # this is easy to forget to do
```

In [96]:
```
# remove the example file we just created
%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 [97]:

```
1 x = 5
2 x = 'five'
```

Python Dynamic Typing

In [97]:
```
1 x = 5
2 x = 'five'
```

- Note: still *strongly* typed

```
  -Python is both a strongly typed and a dynamically typed language. Strong typ
ng means
  that variables do have a type and that the type matters when performing opera
ions
  on a variable. Dynamic typing means that the type of the variable is determin
d only during
  runtime.
```

Python Dynamic Typing

In [97]:
```
x = 5
x = 'five'
```

- Note: still *strongly* typed

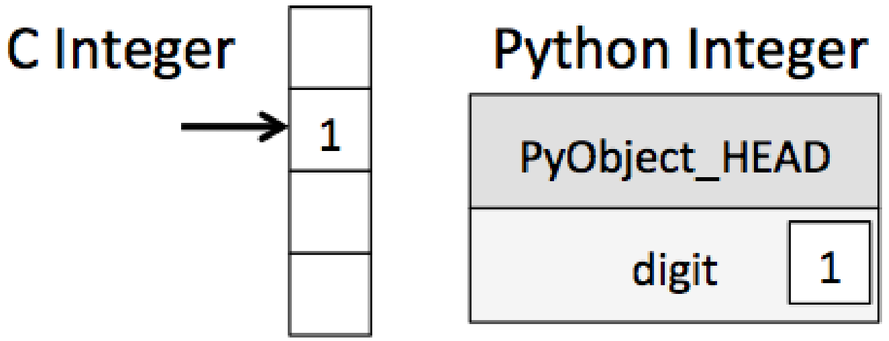
```
  -Python is both a strongly typed and a dynamically typed language. Strong typ
ng means
  that variables do have a type and that the type matters when performing opera
ions
  on a variable. Dynamic typing means that the type of the variable is determin
d only during
  runtime.
```

In [98]:
```
x,y = 5,'five'
x+y
```

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

Python Dynamic Typing

Python Dynamic Typing

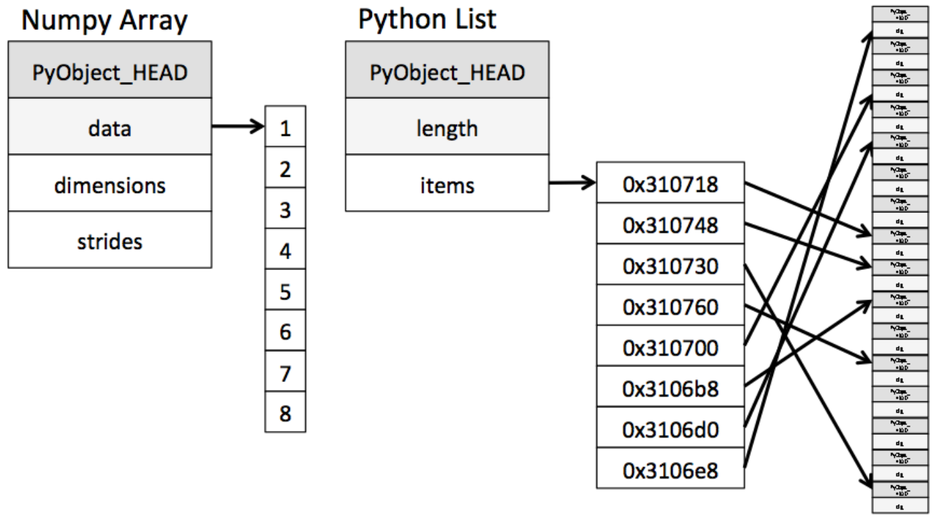


[PDHS Chap 2.](https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html)

- 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.

NumPy Array vs Python List

NumPy Array vs Python List



PDHS Chap 2.

Importing NumPy

Importing NumPy

Often imported as alias `np`

Importing NumPy

Often imported as alias `np`

In [99]:
```
1 import numpy as np
2
3 np.random.randint(10,size=5)
```

Out[99]:
```
array([5, 6, 8, 4, 2])
```

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 [100]:
```
1 x = np.array([1,2,3])
2 x
```

Out[100]:
```
array([1, 2, 3])
```

NumPy Arrays

In [100]:
```
1 x = np.array([1,2,3])
2 x
```

Out[100]:
```
array([1, 2, 3])
```

In [101]:
```
1 type(x)
```

Out[101]:
```
numpy.ndarray
```

NumPy Arrays

In [100]:
```
1 x = np.array([1,2,3])
2 x
```

Out[100]:
```
array([1, 2, 3])
```

In [101]:
```
1 type(x)
```

Out[101]:
```
numpy.ndarray
```

In [102]:
```
1 # use dtype to show the datatype of the array
2 x.dtype
```

Out[102]:
```
dtype('int64')
```

NumPy Arrays

In [100]:
```
x = np.array([1,2,3])
x
```

Out[100]:
```
array([1, 2, 3])
```

In [101]:
```
type(x)
```

Out[101]:
```
numpy.ndarray
```

In [102]:
```
# use dtype to show the datatype of the array
x.dtype
```

Out[102]:
```
dtype('int64')
```

In [103]:
```
# np arrays can only contain one datatype and default to the most flexible type
x = np.array([1,'two',3])
x
```

Out[103]:
```
array(['1', 'two', '3'], dtype='<U21')
```

NumPy Arrays

In [100]:
```
x = np.array([1,2,3])
x
```

Out[100]:
```
array([1, 2, 3])
```

In [101]:
```
type(x)
```

Out[101]:
```
numpy.ndarray
```

In [102]:
```
# use dtype to show the datatype of the array
x.dtype
```

Out[102]:
```
dtype('int64')
```

In [103]:
```
# np arrays can only contain one datatype and default to the most flexible type
x = np.array([1,'two',3])
x
```

Out[103]:
```
array(['1', 'two', '3'], dtype='<U21')
```

In [104]:
```
x.dtype
```

Out[104]:
```
dtype('<U21')
```

NumPy Arrays

In [100]:
```
x = np.array([1,2,3])
x
```

Out[100]:
```
array([1, 2, 3])
```

In [101]:
```
type(x)
```

Out[101]:
```
numpy.ndarray
```

In [102]:
```
# use dtype to show the datatype of the array
x.dtype
```

Out[102]:
```
dtype('int64')
```

In [103]:
```
# np arrays can only contain one datatype and default to the most flexible type
x = np.array([1,'two',3])
x
```

Out[103]:
```
array(['1', 'two', '3'], dtype='<U21')
```

In [104]:
```
x.dtype
```

Out[104]:
```
dtype('<U21')
```

In [105]:
```
# many different ways to create numpy arrays
np.ones(5,dtype=float)
```

Out[105]:
```
array([1., 1., 1., 1., 1.])
```

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 [106]:
```
1 x = np.arange(1,6)
2 x
```

Out[106]:
```
array([1, 2, 3, 4, 5])
```

NumPy Array Indexing

- For single indices, works the same as list

In [106]:
```
1 x = np.arange(1,6)
2 x
```

Out[106]:
```
array([1, 2, 3, 4, 5])
```

In [107]:
```
1 x[0], x[-1], x[-2], x[-4]
```

Out[107]:
```
(1, 5, 4, 2)
```

NumPy Array Slicing

NumPy Array Slicing

In [108]:
```
1 x = np.arange(5) # note that in numpy it's arange instead of range
2 x
```

Out[108]:
```
array([0, 1, 2, 3, 4])
```

NumPy Array Slicing

In [108]:
```
x = np.arange(5) # note that in numpy it's arange instead of range
x
```

Out[108]: array([0, 1, 2, 3, 4])

In [109]:
```
# return first two items, start:end (exclusive)
x[0:2]
```

Out[109]: array([0, 1])

NumPy Array Slicing

In [108]:
```
x = np.arange(5) # note that in numpy it's arange instead of range
x
```

Out[108]: array([0, 1, 2, 3, 4])

In [109]:
```
# return first two items, start:end (exclusive)
x[0:2]
```

Out[109]: array([0, 1])

In [110]:
```
# missing start implies position 0
x[:2]
```

Out[110]: array([0, 1])

NumPy Array Slicing

In [108]:
```
x = np.arange(5) # note that in numpy it's arange instead of range
x
```

Out[108]:
```
array([0, 1, 2, 3, 4])
```

In [109]:
```
# return first two items, start:end (exclusive)
x[0:2]
```

Out[109]:
```
array([0, 1])
```

In [110]:
```
# missing start implies position 0
x[:2]
```

Out[110]:
```
array([0, 1])
```

In [111]:
```
# missing end implies length of array
x[2:]
```

Out[111]:
```
array([2, 3, 4])
```

NumPy Array Slicing

In [108]:
```
x = np.arange(5) # note that in numpy it's arange instead of range
x
```

Out[108]:
```
array([0, 1, 2, 3, 4])
```

In [109]:
```
# return first two items, start:end (exclusive)
x[0:2]
```

Out[109]:
```
array([0, 1])
```

In [110]:
```
# missing start implies position 0
x[:2]
```

Out[110]:
```
array([0, 1])
```

In [111]:
```
# missing end implies length of array
x[2:]
```

Out[111]:
```
array([2, 3, 4])
```

In [112]:
```
# return last two items
x[-2:]
```

Out[112]:
```
array([3, 4])
```

NumPy Array Slicing with Steps

NumPy Array Slicing with Steps

In [113]:
```
1 x
```

Out[113]:
```
array([0, 1, 2, 3, 4])
```

NumPy Array Slicing with Steps

In [113]:
```
1 x
```

Out[113]:
```
array([0, 1, 2, 3, 4])
```

In [114]:
```
1 # return every other item from position 1 to 4 exclusive
2 # start:end:step_size
3 x[1:4:2]
```

Out[114]:
```
array([1, 3])
```

Reverse array with step-size of -1

Reverse array with step-size of -1

In [115]:
```
1 x
```

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

Reverse array with step-size of -1

In [115]:
```
1 x
```

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

In [116]:
```
1 x[::-1]
```

Out[116]:
```
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 [117]:
```
1 x = np.arange(5,10)
2 x
```

Out[117]:
```
array([5, 6, 7, 8, 9])
```

NumPy Fancy Indexing

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

In [117]:
```
1 x = np.arange(5,10)
2 x
```

Out[117]:
```
array([5, 6, 7, 8, 9])
```

In [118]:
```
1 x[[0,3]]
```

Out[118]:
```
array([5, 8])
```

NumPy Fancy Indexing

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

In [117]:
```
1 x = np.arange(5,10)
2 x
```

Out[117]:
```
array([5, 6, 7, 8, 9])
```

In [118]:
```
1 x[[0,3]]
```

Out[118]:
```
array([5, 8])
```

In [119]:
```
1 x[[0,2,-1]]
2
```

Out[119]:
```
array([5, 7, 9])
```

Boolean Indexing using a Boolean Mask

Boolean Indexing using a Boolean Mask

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

Out[120]:
```
array([5, 6, 7, 8, 9])
```

Boolean Indexing using a Boolean Mask

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

Out[120]:
```
array([5, 6, 7, 8, 9])
```

In [121]:
```
1 # Which indices have a value divisible by 2?
2 # mod operator % returns remainder of division
3 x%2 == 0
```

Out[121]:
```
array([False,  True, False,  True, False])
```

Boolean Indexing using a Boolean Mask

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

Out[120]:
```
array([5, 6, 7, 8, 9])
```

In [121]:
```
# Which indices have a value divisible by 2?
# mod operator % returns remainder of division
x%2 == 0
```

Out[121]:
```
array([False,  True, False,  True, False])
```

In [122]:
```
# Which values are divisible by 2?
x[x%2 == 0]
```

Out[122]:
```
array([6, 8])
```

Boolean Indexing using a Boolean Mask

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

Out[120]:
```
array([5, 6, 7, 8, 9])
```

In [121]:
```
# Which indices have a value divisible by 2?
# mod operator % returns remainder of division
x%2 == 0
```

Out[121]:
```
array([False,  True, False,  True, False])
```

In [122]:
```
# Which values are divisible by 2?
x[x%2 == 0]
```

Out[122]:
```
array([6, 8])
```

In [123]:
```
# Which values are greater than 6?
x[x > 6]
```

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

Boolean Indexing And Bitwise Operators

Boolean Indexing And Bitwise Operators

In [124]:
```
1 x
```

Out[124]:
```
array([5, 6, 7, 8, 9])
```

Boolean Indexing And Bitwise Operators

In [124]:
```
1 x
```

Out[124]:
```
array([5, 6, 7, 8, 9])
```

In [125]:
```
1 (x%2 == 0)
```

Out[125]:
```
array([False,  True, False,  True, False])
```

Boolean Indexing And Bitwise Operators

In [124]:
```
1 x
```

Out[124]:
```
array([5, 6, 7, 8, 9])
```

In [125]:
```
1 (x%2 == 0)
```

Out[125]:
```
array([False,  True, False,  True, False])
```

In [126]:
```
1 (x > 6)
```

Out[126]:
```
array([False, False,  True,  True,  True])
```

Boolean Indexing And Bitwise Operators

In [124]:
```
1 x
```

Out[124]:
```
array([5, 6, 7, 8, 9])
```

In [125]:
```
1 (x%2 == 0)
```

Out[125]:
```
array([False,  True, False,  True, False])
```

In [126]:
```
1 (x > 6)
```

Out[126]:
```
array([False, False,  True,  True,  True])
```

In [127]:
```
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 [124]:
```
x
```

Out[124]:
```
array([5, 6, 7, 8, 9])
```

In [125]:
```
(x%2 == 0)
```

Out[125]:
```
array([False,  True, False,  True, False])
```

In [126]:
```
(x > 6)
```

Out[126]:
```
array([False, False,  True,  True,  True])
```

In [127]:
```
# Which values are divisible by 2 AND greater than 6?
# 'and' expexts both elements to 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 [128]:
```
# & compares each element pairwise
(x%2 == 0) & (x > 6)
```

Out[128]:
```
array([False, False, False,  True, False])
```

Boolean Indexing And Bitwise Operators

In [124]:
```
x
```

Out[124]:
```
array([5, 6, 7, 8, 9])
```

In [125]:
```
(x%2 == 0)
```

Out[125]:
```
array([False,  True, False,  True, False])
```

In [126]:
```
(x > 6)
```

Out[126]:
```
array([False, False,  True,  True,  True])
```

In [127]:
```
# Which values are divisible by 2 AND greater than 6?
# 'and' expexts both elements to 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 [128]:
```
# & compares each element pairwise
(x%2 == 0) & (x > 6)
```

Out[128]:
```
array([False, False, False,  True, False])
```

In [129]:
```
x[(x%2 == 0) & (x > 6)]
```

Out[129]:
```
array([8])
```

Boolean Indexing And Bitwise Operators

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

In [130]:
```
# Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

Out[130]:
```
array([8])
```

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

In [130]:
```
# Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

Out[130]:
```
array([8])
```

- or : `|` (pipe)

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

In [130]:
```
# Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

Out[130]:
```
array([8])
```

- or : `|` (pipe)

In [131]:
```
# which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

Out[131]:
```
array([6, 7, 8, 9])
```

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

In [130]:
```
# Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

Out[130]:
```
array([8])
```

- or : `|` (pipe)

In [131]:
```
# which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

Out[131]:
```
array([6, 7, 8, 9])
```

- not : `~` (tilde)

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

In [130]:
```
# Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

Out[130]:
```
array([8])
```

- or : `|` (pipe)

In [131]:
```
# which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

Out[131]:
```
array([6, 7, 8, 9])
```

- not : `~` (tilde)

In [132]:
```
# which values are NOT (even OR greater than 6)
x[~( (x%2 == 0) | (x > 6) )]
```

Out[132]:
```
array([5])
```

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

In [130]:
```
# Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

Out[130]:
```
array([8])
```

- or : `|` (pipe)

In [131]:
```
# which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

Out[131]:
```
array([6, 7, 8, 9])
```

- not : `~` (tilde)

In [132]:
```
# which values are NOT (even OR greater than 6)
x[~( (x%2 == 0) | (x > 6) )]
```

Out[132]:
```
array([5])
```

- see PDHS for more info

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 [133]:
```
1 x = [[1,2,3],[4,5,6]] # list of lists
2 x
```

Out[133]:
```
[[1, 2, 3], [4, 5, 6]]
```

Multidimensional Lists

In [133]:
```
x = [[1,2,3],[4,5,6]] # list of lists
x
```

Out[133]:
```
[[1, 2, 3], [4, 5, 6]]
```

In [134]:
```
# return first row
x[0]
```

Out[134]:
```
[1, 2, 3]
```

Multidimensional Lists

In [133]:
```
x = [[1,2,3],[4,5,6]] # list of lists
x
```

Out[133]:
```
[[1, 2, 3], [4, 5, 6]]
```

In [134]:
```
# return first row
x[0]
```

Out[134]:
```
[1, 2, 3]
```

In [135]:
```
# return first row, second column
x[0][1]
```

Out[135]:
```
2
```

Multidimensional Lists

In [133]:
```
x = [[1,2,3],[4,5,6]] # list of lists
x
```

Out[133]:
```
[[1, 2, 3], [4, 5, 6]]
```

In [134]:
```
# return first row
x[0]
```

Out[134]:
```
[1, 2, 3]
```

In [135]:
```
# return first row, second column
x[0][1]
```

Out[135]:
```
2
```

In [136]:
```
# return second column?
[row[1] for row in x]
```

Out[136]:
```
[2, 5]
```

NumPy Multidimensional Arrays

NumPy Multidimensional Arrays

In [137]:
```
1 x = np.array([[1,2,3],[4,5,6]])
2 x
```

Out[137]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

NumPy Multidimensional Arrays

In [137]:
```
x = np.array([[1,2,3],[4,5,6]])
x
```

Out[137]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [138]:
```
x[0,1] # first row, second column
```

Out[138]:
```
2
```

NumPy Multidimensional Arrays

In [137]:
```
x = np.array([[1,2,3],[4,5,6]])
x
```

Out[137]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [138]:
```
x[0,1] # first row, second column
```

Out[138]:
```
2
```

In [139]:
```
x[0,0:3] # first row
```

Out[139]:
```
array([1, 2, 3])
```

NumPy Multidimensional Arrays

In [137]:
```
x = np.array([[1,2,3],[4,5,6]])
x
```

Out[137]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [138]:
```
x[0,1] # first row, second column
```

Out[138]:
```
2
```

In [139]:
```
x[0,0:3] # first row
```

Out[139]:
```
array([1, 2, 3])
```

In [140]:
```
x[0,:] # first row (first to last column)
```

Out[140]:
```
array([1, 2, 3])
```

NumPy Multidimensional Arrays

In [137]:
```
x = np.array([[1,2,3],[4,5,6]])
x
```

Out[137]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

In [138]:
```
x[0,1] # first row, second column
```

Out[138]:
```
2
```

In [139]:
```
x[0,0:3] # first row
```

Out[139]:
```
array([1, 2, 3])
```

In [140]:
```
x[0,:] # first row (first to last column)
```

Out[140]:
```
array([1, 2, 3])
```

In [141]:
```
x[:,1] # second column (first to last row)
```

Out[141]:
```
array([2, 5])
```

NumPy Array Attributes

NumPy Array Attributes

In [142]:
```
1 x = np.array([[1,2,3],[4,5,6]])
```

NumPy Array Attributes

In [142]:
```
1 x = np.array([[1,2,3],[4,5,6]])
```

In [143]:
```
1 x.ndim # number of dimensions
```

Out[143]: 2

NumPy Array Attributes

In [142]:
```
1 x = np.array([[1,2,3],[4,5,6]])
```

In [143]:
```
1 x.ndim # number of dimensions
```

Out[143]: 2

In [144]:
```
1 x.shape # shape in each dimension
```

Out[144]: (2, 3)

NumPy Array Attributes

In [142]:
```
1 x = np.array([[1,2,3],[4,5,6]])
```

In [143]:
```
1 x.ndim # number of dimensions
```

Out[143]: 2

In [144]:
```
1 x.shape # shape in each dimension
```

Out[144]: (2, 3)

In [145]:
```
1 x.size # total number of elements
```

Out[145]: 6

NumPy Operations (UFuncs)

NumPy Operations (UFuncs)

In [146]:
```
x = [1,2,3]
y = [4,5,6]
```

NumPy Operations (UFuncs)

In [146]:
```
1 x = [1,2,3]
2 y = [4,5,6]
```

In [147]:
```
1 x+y
```

Out[147]:
```
[1, 2, 3, 4, 5, 6]
```

NumPy Operations (UFuncs)

In [146]:
```
x = [1,2,3]
y = [4,5,6]
```

In [147]:
```
x+y
```

Out[147]:
```
[1, 2, 3, 4, 5, 6]
```

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

NumPy Operations (UFuncs)

In [146]:
```
x = [1,2,3]
y = [4,5,6]
```

In [147]:
```
x+y
```

Out[147]:
```
[1, 2, 3, 4, 5, 6]
```

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

In [149]:
```
x+y
```

Out[149]:
```
array([5, 7, 9])
```

NumPy Operations (UFuncs)

In [146]:
```
x = [1,2,3]
y = [4,5,6]
```

In [147]:
```
x+y
```

Out[147]:
```
[1, 2, 3, 4, 5, 6]
```

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

In [149]:
```
x+y
```

Out[149]:
```
array([5, 7, 9])
```

In [150]:
```
%time sum(range(0,int(1e8)))
```

```
CPU times: user 1.34 s, sys: 8.38 ms, total: 1.35 s
Wall time: 1.34 s
```

Out[150]:
```
4999999950000000
```

NumPy Operations (UFuncs)

In [146]:
```
x = [1,2,3]
y = [4,5,6]
```

In [147]:
```
x+y
```

Out[147]:
```
[1, 2, 3, 4, 5, 6]
```

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

In [149]:
```
x+y
```

Out[149]:
```
array([5, 7, 9])
```

In [150]:
```
%time sum(range(0,int(1e8)))
```

```
CPU times: user 1.34 s, sys: 8.38 ms, total: 1.35 s
Wall time: 1.34 s
```

Out[150]:
```
4999999950000000
```

In [151]:
```
%time np.arange(0,int(1e8)).sum()
```

```
CPU times: user 268 ms, sys: 263 ms, total: 530 ms
Wall time: 586 ms
```

Out[151]:
```
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 [152]:
```
# square every element in a list
x = [1,2,3]
```

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

In [152]:
```
# square every element in a list
x = [1,2,3]
```

In [153]:
```
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 [152]:
```
# square every element in a list
x = [1,2,3]
```

In [153]:
```
x**2
```

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

In [154]:
```
# square every element in a numpy array
x = np.array([1,2,3])
```

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

In [152]:
```
# square every element in a list
x = [1,2,3]
```

In [153]:
```
x**2
```

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

In [154]:
```
# square every element in a numpy array
x = np.array([1,2,3])
```

In [155]:
```
x**2
```

Out[155]:
```
array([1, 4, 9])
```

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

In [152]:
```
# square every element in a list
x = [1,2,3]
```

In [153]:
```
x**2
```

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

In [154]:
```
# square every element in a numpy array
x = np.array([1,2,3])
```

In [155]:
```
x**2
```

Out[155]:
```
array([1, 4, 9])
```

In [156]:
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

Out[156]:
```
array([2., 4., 6.])
```

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

In [152]:
```
# square every element in a list
x = [1,2,3]
```

In [153]:
```
x**2
```

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

In [154]:
```
# square every element in a numpy array
x = np.array([1,2,3])
```

In [155]:
```
x**2
```

Out[155]:
```
array([1, 4, 9])
```

In [156]:
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

Out[156]:
```
array([2., 4., 6.])
```

is equivalent to

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

In [152]:
```
# square every element in a list
x = [1,2,3]
```

In [153]:
```
x**2
```

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

In [154]:
```
# square every element in a numpy array
x = np.array([1,2,3])
```

In [155]:
```
x**2
```

Out[155]:
```
array([1, 4, 9])
```

In [156]:
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

Out[156]:
```
array([2., 4., 6.])
```

is equivalent to

NumPy `random` Submodule

NumPy `random` Submodule

Provides many random sampling functions

NumPy `random` Submodule

Provides many random sampling functions

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

- `rand` : random floats
- `randint` : random integers
- `randn` : standard normal distribution
- `permutation` : random permutation
- `normal` : Gaussian normal distribution
- `seed` : seed the random generator

Questions?