

Elements Of Data Science - F2021

Week 2: Python Intro/Review and Numpy

9/20/2021

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
 - Computation on NumPy Arrays: Universal Functions
 - Aggregations: Min, Max, and Everything In Between
 - Computation on Arrays: Broadcasting
 - Comparisons, Masks, and Boolean Logic
 - Fancy Indexing
 - Sorting Arrays
 - Structured Data: NumPy's Structured Arrays

TODOs

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

TODOs

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

Getting Changes from Git

```
$ cd [to your cloned repository directory, eg: ~/proj/eods-f21]  
$ git pull
```

Questions?

TODAY

- Tools Review
- Getting "Help" Documentation
- Python (Review?)
- Numpy
- Pandas

Tools Review

- Starting Jupyter
- Notebooks and virtual environments

Getting "Help" Documentation in Python

Getting "Help" Documentation in Python

```
In [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]: 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?)

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

Dynamic Typing

- 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"`
- **None** (null): `None`
- also `long`, `complex`, `bytes`, etc.

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  
print("done looping")
```

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

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.
```

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

- can assign defaults

Function Arguments

- can assign defaults

```
In [6]: def increment(x, amount=1):  
        """Increment a value, default by 1."""  
        return x+amount  
  
        increment(2)
```

```
Out[6]: 3
```

Function Arguments

- can assign defaults

```
In [6]: def increment(x, amount=1):  
        """Increment a value, default by 1."""  
        return x+amount  
  
        increment(2)
```

Out[6]: 3

```
In [7]: increment(2, amount=2)
```

Out[7]: 4

Function Arguments Cont.

Function Arguments Cont.

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

Function Arguments Cont.

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

```
In [8]: def subtract(x,y):  
        return x-y  
  
        subtract(3,1)
```

```
Out[8]: 2
```

Function Arguments Cont.

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

```
In [8]: def subtract(x,y):  
        return x-y  
  
        subtract(3,1)
```

```
Out[8]: 2
```

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

Function Arguments Cont.

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

```
In [8]: def subtract(x,y):  
        return x-y  
  
        subtract(3,1)
```

Out[8]: 2

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

```
In [9]: def proportion(numer,denom,precision=2):  
        return round(numer/denom,precision)  
  
        proportion(2,precision=2,denom=3)
```

Out[9]: 0.67

String Formatting

String Formatting

```
In [10]: x = 3.1415  
         'the value of x is ' + str(x)
```

```
Out[10]: 'the value of x is 3.1415'
```

String Formatting

```
In [10]: x = 3.1415  
  
         'the value of x is ' + str(x)
```

```
Out[10]: 'the value of x is 3.1415'
```

```
In [11]: 'the value of x is %0.2f' % x
```

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

String Formatting

```
In [10]: x = 3.1415  
  
         'the value of x is ' + str(x)
```

```
Out[10]: 'the value of x is 3.1415'
```

```
In [11]: 'the value of x is %0.2f' % x
```

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

```
In [12]: 'the value of x is {:.10f}'.format(x)
```

```
Out[12]: 'the value of x is 3.1415000000'
```

String Formatting

```
In [10]: x = 3.1415  
  
         'the value of x is ' + str(x)
```

```
Out[10]: 'the value of x is 3.1415'
```

```
In [11]: 'the value of x is %0.2f' % x
```

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

```
In [12]: 'the value of x is {:.10f}'.format(x)
```

```
Out[12]: 'the value of x is 3.1415000000'
```

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

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

String Formatting

```
In [10]: x = 3.1415  
  
         'the value of x is ' + str(x)
```

```
Out[10]: 'the value of x is 3.1415'
```

```
In [11]: 'the value of x is %0.2f' % x
```

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

```
In [12]: 'the value of x is {:.10f}'.format(x)
```

```
Out[12]: 'the value of x is 3.1415000000'
```

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

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

- often print variable values for debugging

String Formatting

```
In [10]: x = 3.1415  
  
         'the value of x is ' + str(x)
```

```
Out[10]: 'the value of x is 3.1415'
```

```
In [11]: 'the value of x is %0.2f' % x
```

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

```
In [12]: 'the value of x is {:.10f}'.format(x)
```

```
Out[12]: 'the value of x is 3.1415000000'
```

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

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

- often print variable values for debugging

```
In [14]: f'x = {x:0.2f}'
```

```
Out[14]: 'x = 3.14'
```

String Formatting

```
In [10]: x = 3.1415  
  
        'the value of x is ' + str(x)
```

Out[10]: 'the value of x is 3.1415'

```
In [11]: 'the value of x is %0.2f' % x
```

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

```
In [12]: 'the value of x is {:.10f}'.format(x)
```

Out[12]: 'the value of x is 3.1415000000'

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

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

- often print variable values for debugging

```
In [14]: f'x = {x:0.2f}'
```

Out[14]: 'x = 3.14'

```
In [15]: f'{x = :0.2f}' # new in 3.8
```

Out[15]: 'x = 3.14'

String Formatting Cont.

String Formatting Cont.

```
In [16]: """This is a multiline string.  
The value of x is {}.""".format(x)
```

```
Out[16]: 'This is a multiline string.\nThe value of x is 3.1415.'
```

String Formatting Cont.

```
In [16]: """This is a multiline string.  
The value of x is {}.""".format(x)
```

```
Out[16]: 'This is a multiline string.\nThe value of x is 3.1415.'
```

```
In [17]: 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 [16]: """This is a multiline string.  
The value of x is {}.""".format(x)
```

```
Out[16]: 'This is a multiline string.\nThe value of x is 3.1415.'
```

```
In [17]: 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 [16]: """This is a multiline string.  
The value of x is {}.""".format(x)
```

```
Out[16]: 'This is a multiline string.\nThe value of x is 3.1415.'
```

```
In [17]: 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

```
In [18]: x='apple'  
f'the plural of {x:>10s} is {x+"s"}'
```

```
Out[18]: 'the plural of          apple is apples'
```

String Formatting Cont.

```
In [16]: """This is a multiline string.  
The value of x is {}.""".format(x)
```

```
Out[16]: 'This is a multiline string.\nThe value of x is 3.1415.'
```

```
In [17]: 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

```
In [18]: x='apple'  
f'the plural of {x:>10s} is {x+"s"}'
```

```
Out[18]: 'the plural of      apple is apples'
```

```
In [19]: x = 3  
f'the square of {x:10d} is {x**2}'
```

```
Out[19]: 'the square of      3 is 9'
```

String Formatting Cont.

```
In [16]: """This is a multiline string.  
The value of x is {}.""".format(x)
```

```
Out[16]: 'This is a multiline string.\nThe value of x is 3.1415.'
```

```
In [17]: 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

```
In [18]: x='apple'  
f'the plural of {x:>10s} is {x+"s"}'
```

```
Out[18]: 'the plural of          apple is apples'
```

```
In [19]: x = 3  
f'the square of {x:10d} is {x**2}'
```

```
Out[19]: 'the square of          3 is 9'
```

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

Python Data Types Continued: **list**

Python Data Types Continued: **list**

```
In [20]: # elements of a python list do not all have to be of the same type  
x = [42, 'e', 2.0]  
x
```

```
Out[20]: [42, 'e', 2.0]
```


Python Data Types Continued: **list**

```
In [20]: # elements of a python list do not all have to be of the same type  
x = [42, 'e', 2.0]  
x
```

```
Out[20]: [42, 'e', 2.0]
```

```
In [21]: x[0] # indexing
```

```
Out[21]: 42
```

Python Data Types Continued: **list**

```
In [20]: # elements of a python list do not all have to be of the same type  
x = [42, 'e', 2.0]  
x
```

```
Out[20]: [42, 'e', 2.0]
```

```
In [21]: x[0] # indexing
```

```
Out[21]: 42
```

```
In [22]: x[-3] # reverse indexing
```

```
Out[22]: 42
```

Python Data Types Continued: **list**

```
In [20]: # elements of a python list do not all have to be of the same type
x = [42, 'e', 2.0]
x
```

```
Out[20]: [42, 'e', 2.0]
```

```
In [21]: x[0] # indexing
```

```
Out[21]: 42
```

```
In [22]: x[-3] # reverse indexing
```

```
Out[22]: 42
```

```
In [23]: x[2] = 4 # assignment
x
```

```
Out[23]: [42, 'e', 4]
```

Python Data Types Continued: list

```
In [20]: # elements of a python list do not all have to be of the same type
x = [42, 'e', 2.0]
x
```

```
Out[20]: [42, 'e', 2.0]
```

```
In [21]: x[0] # indexing
```

```
Out[21]: 42
```

```
In [22]: x[-3] # reverse indexing
```

```
Out[22]: 42
```

```
In [23]: x[2] = 4 # assignment
x
```

```
Out[23]: [42, 'e', 4]
```

```
In [24]: x.append('a') # add a value to list
x
```

```
Out[24]: [42, 'e', 4, 'a']
```

Python Data Types Continued: **list**

```
In [20]: # elements of a python list do not all have to be of the same type  
x = [42, 'e', 2.0]  
x
```

```
Out[20]: [42, 'e', 2.0]
```

```
In [21]: x[0] # indexing
```

```
Out[21]: 42
```

```
In [22]: x[-3] # reverse indexing
```

```
Out[22]: 42
```

```
In [23]: x[2] = 4 # assignment  
x
```

```
Out[23]: [42, 'e', 4]
```

```
In [24]: x.append('a') # add a value to list  
x
```

```
Out[24]: [42, 'e', 4, 'a']
```

```
In [25]: value_at_1 = x.pop(1) # remove/delete at index  
x
```

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

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

```
In [26]: x = {'b':[2,1], 'a':1, 'c':4}
          # or x = dict(b=2,a=1,c=4)
          x # NOTE: order is not guaranteed!
```

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

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

```
In [26]: x = {'b':[2,1], 'a':1, 'c':4}
          # or x = dict(b=2,a=1,c=4)
          x # NOTE: order is not guaranteed!
```

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

```
In [27]: # index into dictionary using key
          x['b']
```

```
Out[27]: [2, 1]
```


Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

```
In [26]: x = {'b':[2,1], 'a':1, 'c':4}
         # or x = dict(b=2,a=1,c=4)
         x # NOTE: order is not guaranteed!
```

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

```
In [27]: # index into dictionary using key
         x['b']
```

```
Out[27]: [2, 1]
```

```
In [28]: # assign a value to a (new or existing) key
         x['d'] = 3
         x
```

```
Out[28]: {'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}
```

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

```
In [26]: x = {'b':[2,1], 'a':1, 'c':4}
# or x = dict(b=2,a=1,c=4)
x # NOTE: order is not guaranteed!
```

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

```
In [27]: # index into dictionary using key
x['b']
```

```
Out[27]: [2, 1]
```

```
In [28]: # assign a value to a (new or existing) key
x['d'] = 3
x
```

```
Out[28]: {'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}
```

```
In [29]: # 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[29]: 3
```

Python Data Types Continued: `dict` (Dictionary)

- Stores key:value pairs

```
In [26]: x = {'b':[2,1], 'a':1, 'c':4}
# or x = dict(b=2,a=1,c=4)
x # NOTE: order is not guaranteed!
```

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

```
In [27]: # index into dictionary using key
x['b']
```

```
Out[27]: [2, 1]
```

```
In [28]: # assign a value to a (new or existing) key
x['d'] = 3
x
```

```
Out[28]: {'b': [2, 1], 'a': 1, 'c': 4, 'd': 3}
```

```
In [29]: # 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[29]: 3
```

```
In [30]: x
```

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

Python Data Types Continued: `dict` Cont.

Python Data Types Continued: **dict** Cont.

```
In [31]: # using the same dictionary  
x
```

```
Out[31]: {'b': [2, 1], 'a': 1, 'c': 4}
```

Python Data Types Continued: **dict** Cont.

```
In [31]: # using the same dictionary  
x
```

```
Out[31]: {'b': [2, 1], 'a': 1, 'c': 4}
```

```
In [32]: # get a set of keys  
x.keys()
```

```
Out[32]: dict_keys(['b', 'a', 'c'])
```

Python Data Types Continued: **dict** Cont.

```
In [31]: # using the same dictionary  
x
```

```
Out[31]: {'b': [2, 1], 'a': 1, 'c': 4}
```

```
In [32]: # get a set of keys  
x.keys()
```

```
Out[32]: dict_keys(['b', 'a', 'c'])
```

```
In [33]: # get a set of values  
x.values()
```

```
Out[33]: dict_values([[2, 1], 1, 4])
```

Python Data Types Continued: `dict` Cont.

```
In [31]: # using the same dictionary  
x
```

```
Out[31]: {'b': [2, 1], 'a': 1, 'c': 4}
```

```
In [32]: # get a set of keys  
x.keys()
```

```
Out[32]: dict_keys(['b', 'a', 'c'])
```

```
In [33]: # get a set of values  
x.values()
```

```
Out[33]: dict_values([[2, 1], 1, 4])
```

```
In [34]: # get a set of (key,value) tuples  
x.items()
```

```
Out[34]: dict_items([('b', [2, 1]), ('a', 1), ('c', 4)])
```


Python Data Types Continued: dict Cont.

```
In [31]: # using the same dictionary  
x
```

```
Out[31]: {'b': [2, 1], 'a': 1, 'c': 4}
```

```
In [32]: # get a set of keys  
x.keys()
```

```
Out[32]: dict_keys(['b', 'a', 'c'])
```

```
In [33]: # get a set of values  
x.values()
```

```
Out[33]: dict_values([[2, 1], 1, 4])
```

```
In [34]: # get a set of (key,value) tuples  
x.items()
```

```
Out[34]: dict_items([('b', [2, 1]), ('a', 1), ('c', 4)])
```

```
In [35]: # get a list of (key,value) pairs  
list(x.items())
```

```
Out[35]: [('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 [36]: x = (2, 'e', 3, 4)
         x
```

```
Out[36]: (2, 'e', 3, 4)
```

Python Data Types Continued: tuple

- like a list, but immutable

```
In [36]: x = (2, 'e', 3, 4)
         x
```

```
Out[36]: (2, 'e', 3, 4)
```

```
In [37]: x[0] # indexing
```

```
Out[37]: 2
```

Python Data Types Continued: tuple

- like a list, but immutable

```
In [36]: x = (2, 'e', 3, 4)
         x
```

```
Out[36]: (2, 'e', 3, 4)
```

```
In [37]: x[0] # indexing
```

```
Out[37]: 2
```

```
In [38]: x[0] = 3 # assignment? Nope, error: immutable`
```

```
-----
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_5904/3594152884.py in <module>
----> 1 x[0] = 3 # assignment? Nope, error: immutable`

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

Python Data Types Continued: `set`

Python Data Types Continued: set

```
In [39]: x = {2, 'e', 'e'} # or set([2, 'e', 'e'])  
x
```

```
Out[39]: {2, 'e'}
```

Python Data Types Continued: set

```
In [39]: x = {2, 'e', 'e'} # or set([2, 'e', 'e'])  
x
```

```
Out[39]: {2, 'e'}
```

```
In [40]: x.add(1) # insert  
x
```

```
Out[40]: {1, 2, 'e'}
```


Python Data Types Continued: set

```
In [39]: x = {2, 'e', 'e'} # or set([2, 'e', 'e'])  
x
```

```
Out[39]: {2, 'e'}
```

```
In [40]: x.add(1) # insert  
x
```

```
Out[40]: {1, 2, 'e'}
```

```
In [41]: x.remove('e') # remove/delete  
x
```

```
Out[41]: {1, 2}
```

Python Data Types Continued: set

```
In [39]: x = {2, 'e', 'e'} # or set([2, 'e', 'e'])  
x
```

```
Out[39]: {2, 'e'}
```

```
In [40]: x.add(1) # insert  
x
```

```
Out[40]: {1, 2, 'e'}
```

```
In [41]: x.remove('e') # remove/delete  
x
```

```
Out[41]: {1, 2}
```

```
In [42]: x.intersection({2, 3})
```

```
Out[42]: {2}
```

Python Data Types Continued: set

```
In [39]: x = {2, 'e', 'e'} # or set([2, 'e', 'e'])  
x
```

```
Out[39]: {2, 'e'}
```

```
In [40]: x.add(1) # insert  
x
```

```
Out[40]: {1, 2, 'e'}
```

```
In [41]: x.remove('e') # remove/delete  
x
```

```
Out[41]: {1, 2}
```

```
In [42]: x.intersection({2, 3})
```

```
Out[42]: {2}
```

```
In [43]: x.difference({2, 3})
```

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

Python Data Types Continued: set

```
In [39]: x = {2, 'e', 'e'} # or set([2, 'e', 'e'])  
x
```

```
Out[39]: {2, 'e'}
```

```
In [40]: x.add(1) # insert  
x
```

```
Out[40]: {1, 2, 'e'}
```

```
In [41]: x.remove('e') # remove/delete  
x
```

```
Out[41]: {1, 2}
```

```
In [42]: x.intersection({2, 3})
```

```
Out[42]: {2}
```

```
In [43]: x.difference({2, 3})
```

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

```
In [44]: x[0] # cannot index into a set
```

```
-----  
TypeError                                Traceback (most recent call last)  
/tmp/ipykernel_5904/3791443751.py in <module>  
----> 1 x[0] # cannot index into a set  
  
TypeError: 'set' object is not subscriptable
```

Determining Length with `len`

Determining Length with `len`

```
In [45]: len([1, 2, 3])
```

```
Out[45]: 3
```

Determining Length with `len`

```
In [45]: len([1, 2, 3])
```

```
Out[45]: 3
```

```
In [46]: len({'a':1, 'b':2, 'c':3})
```

```
Out[46]: 3
```

Determining Length with `len`

```
In [45]: len([1,2,3])
```

```
Out[45]: 3
```

```
In [46]: len({'a':1, 'b':2, 'c':3})
```

```
Out[46]: 3
```

```
In [47]: len('apple')
```

```
Out[47]: 5
```

```
In [48]: len(True)
```

```
-----  
TypeError                                Traceback (most recent call last)  
/tmp/ipykernel_5904/1402046054.py in <module>  
----> 1 len(True)
```

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


Exceptions

Exceptions

```
In [49]: 'a' + 2
```

```
-----  
TypeError                                 Traceback (most recent call last)  
/tmp/ipykernel_5904/2408715624.py in <module>  
----> 1 'a' + 2  
  
TypeError: can only concatenate str (not "int") to str
```

Exceptions

```
In [49]: 'a' + 2
```

```
-----  
TypeError                                 Traceback (most recent call last)  
/tmp/ipykernel_5904/2408715624.py in <module>  
----> 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 [50]: 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 [50]: 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 [51]: 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 [52]: 3 == 3
```

```
Out[52]: True
```

Comparison Operators

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

```
In [52]: 3 == 3
```

```
Out[52]: True
```

```
In [53]: 3 != 4
```

```
Out[53]: True
```

Comparison Operators

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

```
In [52]: 3 == 3
```

```
Out[52]: True
```

```
In [53]: 3 != 4
```

```
Out[53]: True
```

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

Comparison Operators

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

```
In [52]: 3 == 3
```

```
Out[52]: True
```

```
In [53]: 3 != 4
```

```
Out[53]: True
```

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

```
In [54]: 3 < 4
```

```
Out[54]: True
```

Logical Operators

Logical Operators

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

Logical Operators

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

```
In [55]: ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

```
Out[55]: True
```

Logical Operators

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

```
In [55]: ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

```
Out[55]: True
```

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

Logical Operators

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

```
In [55]: ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

```
Out[55]: True
```

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

```
In [56]: any([0, 0, 1])
```

```
Out[56]: True
```

Logical Operators

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

```
In [55]: ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

```
Out[55]: True
```

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

```
In [56]: any([0, 0, 1])
```

```
Out[56]: True
```

- `all()`: all elements are true

Logical Operators

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

```
In [55]: ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

```
Out[55]: True
```

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

```
In [56]: any([0,0,1])
```

```
Out[56]: True
```

- `all()`: all elements are true

```
In [57]: all([0,0,1])
```

```
Out[57]: False
```

Logical Operators

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

```
In [55]: ( (3 > 5) or ((3 < 4) and (5 > 4)) ) and not (3 == 5)
```

```
Out[55]: True
```

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

```
In [56]: any([0,0,1])
```

```
Out[56]: True
```

- `all()`: all elements are true

```
In [57]: all([0,0,1])
```

```
Out[57]: 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 [58]: 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 [58]: assert 2+2 == 4
```

```
In [59]: assert 1 == 0
```

```
-----  
AssertionError                                Traceback (most recent call last)  
/tmp/ipykernel_5904/351952524.py in <module>  
----> 1 assert 1 == 0  
  
AssertionError:
```

Assert

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

```
In [58]: assert 2+2 == 4
```

```
In [59]: assert 1 == 0
```

```
-----  
AssertionError                                Traceback (most recent call last)  
/tmp/ipykernel_5904/351952524.py in <module>  
----> 1 assert 1 == 0  
  
AssertionError:
```

```
In [60]: # can add an error message  
assert 1 == 0, "1 does not equal 0"
```

```
-----  
AssertionError                                Traceback (most recent call last)  
/tmp/ipykernel_5904/134746644.py in <module>  
      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 [61]: 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 [61]: 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 [61]: 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 [62]: 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 [63]: a = []  
         for x in [0,1,2]:  
             a.append(x)  
         a
```

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

More Control Flow: **for** and **while**

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

```
In [63]: a = []  
         for x in [0,1,2]:  
             a.append(x)  
         a
```

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

- **while** something is true

More Control Flow: **for** and **while**

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

```
In [63]: a = []  
         for x in [0,1,2]:  
             a.append(x)  
         a
```

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

- **while** something is true

```
In [64]: x = 0  
         while x < 3:  
             x += 1  
         x
```

```
Out[64]: 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 [65]: 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 [65]: 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 [65]: x = 0
while True:
    x += 1
    if x == 3:
        print(x)
        break
```

3

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

```
In [66]: for x in range(3):
        if x == 1:
            continue
        print(x)
```

0

2

Generate a Range of Numbers: `range`

Generate a Range of Numbers: `range`

```
In [67]: # create list of integers from 0 up to but not including 4  
a = []  
for x in range(4):  
    a.append(x)  
a
```

```
Out[67]: [0, 1, 2, 3]
```

Generate a Range of Numbers: `range`

```
In [67]: # create list of integers from 0 up to but not including 4
a = []
for x in range(4):
    a.append(x)
a
```

```
Out[67]: [0, 1, 2, 3]
```

```
In [68]: list(range(4))
```

```
Out[68]: [0, 1, 2, 3]
```

Generate a Range of Numbers: `range`

```
In [67]: # create list of integers from 0 up to but not including 4
a = []
for x in range(4):
    a.append(x)
a
```

```
Out[67]: [0, 1, 2, 3]
```

```
In [68]: list(range(4))
```

```
Out[68]: [0, 1, 2, 3]
```

```
In [69]: list(range(3,5)) # with a start and end+1
```

```
Out[69]: [3, 4]
```

Generate a Range of Numbers: `range`

```
In [67]: # create list of integers from 0 up to but not including 4  
a = []  
for x in range(4):  
    a.append(x)  
a
```

```
Out[67]: [0, 1, 2, 3]
```

```
In [68]: list(range(4))
```

```
Out[68]: [0, 1, 2, 3]
```

```
In [69]: list(range(3,5)) # with a start and end+1
```

```
Out[69]: [3, 4]
```

```
In [70]: list(range(0,10,2)) # with start, end+1 and step-size
```

```
Out[70]: [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 [71]: for i,x in enumerate(['a','b','c']):  
         print(i,x)
```

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

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

```
In [71]: for i,x in enumerate(['a','b','c']):  
         print(i,x)
```

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

```
In [72]: list(enumerate(['a','b','c']))
```

```
Out[72]: [(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 [73]: x = [4,1,2,3]
         x.sort()
         assert x == [1,2,3,4]
```

Sorting

Two ways to sort a list:

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

```
In [73]: x = [4,1,2,3]
         x.sort()
         assert x == [1,2,3,4]
```

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

Sorting

Two ways to sort a list:

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

```
In [73]: x = [4,1,2,3]
         x.sort()
         assert x == [1,2,3,4]
```

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

```
In [74]: 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 [75]: assert sorted([1,2,3,4], reverse=True) == [4,3,2,1]
```

Sorting Cont.

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

```
In [75]: 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 [75]: 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 [76]: # 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 [77]: # 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[77]: [True, False, True, False]
```


List Comprehensions

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

```
In [77]: # 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[77]: [True, False, True, False]
```

```
In [78]: [x%2 == 0 for x in range(0,4)] # using a list comprehension
```

```
Out[78]: [True, False, True, False]
```

List Comprehensions

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

```
In [77]: # 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[77]: [True, False, True, False]
```

```
In [78]: [x%2 == 0 for x in range(0,4)] # using a list comprehension
```

```
Out[78]: [True, False, True, False]
```

```
In [79]: # 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[79]: [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 [80]: pairs = [(1, 'e'), (2, 'f'), (3, 'g')]
```

Dictionary Comprehension

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

```
In [80]: pairs = [(1, 'e'), (2, 'f'), (3, 'g')]
```

```
In [81]: dict(pairs)
```

```
Out[81]: {1: 'e', 2: 'f', 3: 'g'}
```

Dictionary Comprehension

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

```
In [80]: pairs = [(1, 'e'), (2, 'f'), (3, 'g')]
```

```
In [81]: dict(pairs)
```

```
Out[81]: {1: 'e', 2: 'f', 3: 'g'}
```

```
In [82]: # modify value and only include odd keys  
{key: 'value_'+str(val) for key, val in pairs if key%2 == 1}
```

```
Out[82]: {1: 'value_e', 3: 'value_g'}
```

Object Oriented Programming

Object Oriented Programming

```
In [85]: 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 Programming

```
In [85]: 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 [86]: 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 [87]: import math  
         math.sqrt(2)
```

```
Out[87]: 1.4142135623730951
```

Importing Modules

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

```
In [87]: import math  
         math.sqrt(2)
```

```
Out[87]: 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 [87]: import math  
math.sqrt(2)
```

```
Out[87]: 1.4142135623730951
```

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

```
In [88]: from math import sqrt, floor  
print(sqrt(2))  
print(floor(sqrt(2)))
```

```
1.4142135623730951  
1
```

Importing Modules Cont.

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

Importing Modules Cont.

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

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

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


Importing Modules Cont.

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

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

```
Out[89]: 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 [90]: from collections import Counter, defaultdict, OrderedDict
```

collections Module

```
In [90]: from collections import Counter, defaultdict, OrderedDict
```

- `Counter` : useful for counting hashable objects
- `defaultdict` : create dictionaries without checking keys
- `OrderedDict` : key,value pairs returned in order added
- others : <https://docs.python.org/3.7/library/collections.html>

collections Module: **Counter**

collections Module: Counter

```
In [91]: c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])  
c
```

```
Out[91]: Counter({'red': 2, 'blue': 3, 'green': 1})
```

collections Module: Counter

```
In [91]: c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])  
c
```

```
Out[91]: Counter({'red': 2, 'blue': 3, 'green': 1})
```

```
In [92]: c = Counter()  
for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:  
    c[word] += 1
```

collections Module: Counter

```
In [91]: c = Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])  
c
```

```
Out[91]: Counter({'red': 2, 'blue': 3, 'green': 1})
```

```
In [92]: c = Counter()  
for word in ['red', 'blue', 'red', 'green', 'blue', 'blue']:  
    c[word] += 1
```

```
In [93]: c.most_common()
```

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


collections Module Cont. : **defaultdict**

collections Module Cont. : defaultdict

```
In [94]: %xmode Minimal  
# reduce the amount printed when an exception is thrown
```

Exception reporting mode: Minimal

collections Module Cont. : defaultdict

```
In [94]: %xmode Minimal
# reduce the amount printed when an exception is thrown
```

Exception reporting mode: Minimal

```
In [95]: # 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 [94]: %xmode Minimal
# reduce the amount printed when an exception is thrown
```

Exception reporting mode: Minimal

```
In [95]: # 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 [96]: d = {}
for word in colors:
    if len(word) in d:
        d[len(word)].append(word)
    else:
        d[len(word)] = [word]
d
```

```
Out[96]: {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}
```

collections Module Cont : defaultdict

```
In [94]: %xmode Minimal
# reduce the amount printed when an exception is thrown
```

Exception reporting mode: Minimal

```
In [95]: # 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 [96]: d = {}
for word in colors:
    if len(word) in d:
        d[len(word)].append(word)
    else:
        d[len(word)] = [word]
d
```

Out[96]: {3: ['red'], 4: ['blue', 'gold'], 6: ['purple', 'orange']}

```
In [97]: d = defaultdict(list)
for word in colors:
    d[len(word)].append(word)
d
```

Out[97]: 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 [98]: 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 [98]: with open('tmp_context_example.txt', 'w') as f:  
         f.write('test')
```

```
In [99]: # 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 [98]: with open('tmp_context_example.txt', 'w') as f:  
         f.write('test')
```

```
In [99]: # instead of  
f = open('tmp_context_example.txt', 'w')  
f.write('test')  
f.close() # this is easy to forget to do
```

```
In [100]: # 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 [101]: x = 5  
          x = 'five'
```

Python Dynamic Typing

```
In [101]: x = 5  
          x = 'five'
```

- Note: still *strongly* typed

Python Dynamic Typing

```
In [101]: x = 5  
          x = 'five'
```

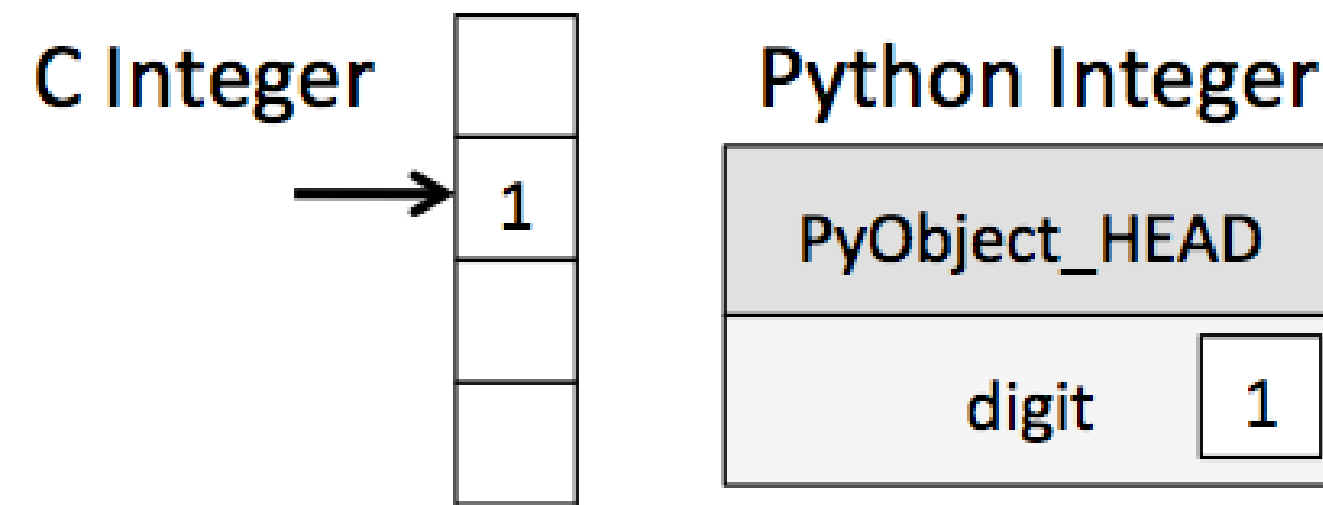
- Note: still *strongly* typed

```
In [102]: x,y = 5, 'five'  
          x+y
```

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

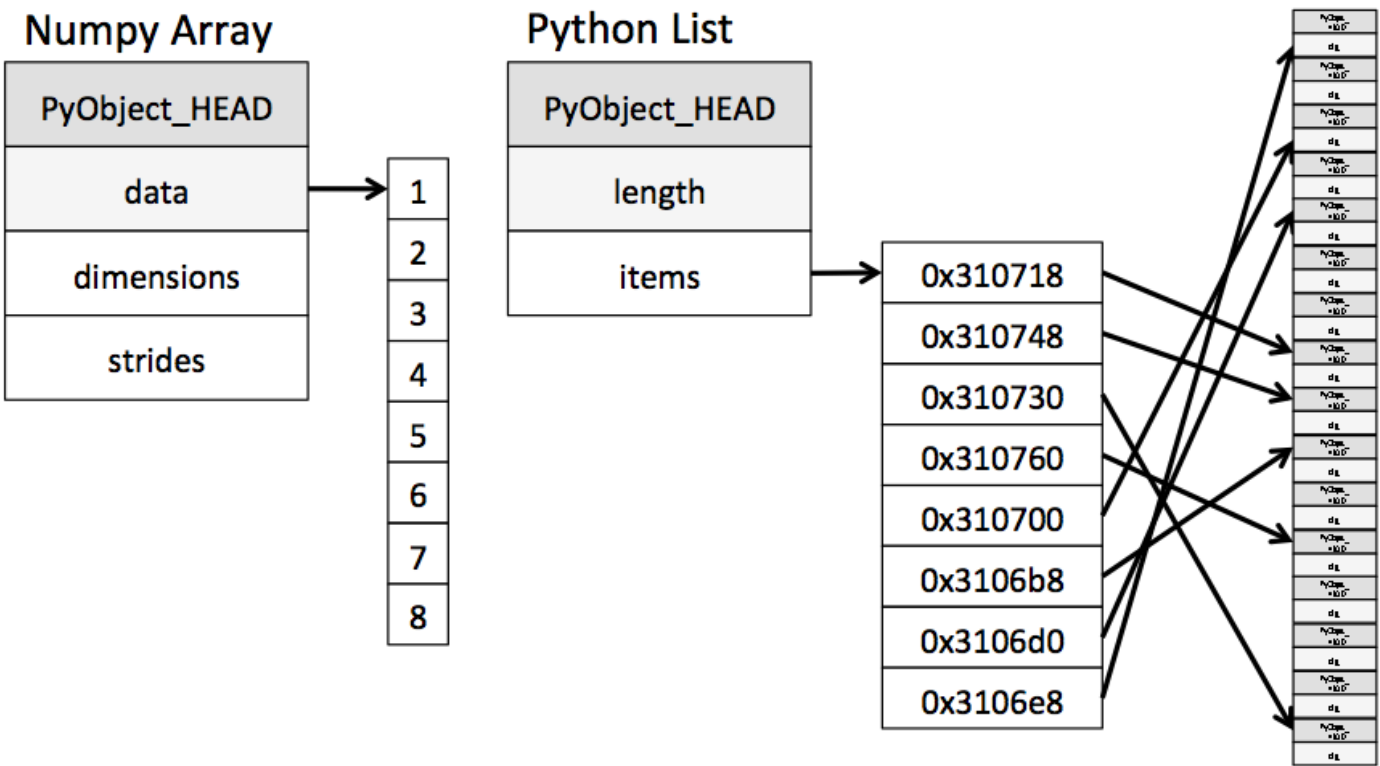
Python Dynamic Typing

Python Dynamic Typing



NumPy Array vs Python List

NumPy Array vs Python List



Importing NumPy

Importing NumPy

Often imported as alias `np`

Importing NumPy

Often imported as alias `np`

```
In [103]: import numpy as np  
          np.random.randint(10, size=5)
```

```
Out[103]: array([9, 5, 3, 7, 6])
```

NumPy Datatypes

NumPy Datatypes

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

NumPy Arrays

NumPy Arrays

```
In [104]: x = np.array([1,2,3])  
x
```

```
Out[104]: array([1, 2, 3])
```

NumPy Arrays

```
In [104]: x = np.array([1,2,3])  
x
```

```
Out[104]: array([1, 2, 3])
```

```
In [105]: type(x)
```

```
Out[105]: numpy.ndarray
```


NumPy Arrays

```
In [104]: x = np.array([1,2,3])  
x
```

```
Out[104]: array([1, 2, 3])
```

```
In [105]: type(x)
```

```
Out[105]: numpy.ndarray
```

```
In [106]: # use dtype to show the datatype of the array  
x.dtype
```

```
Out[106]: dtype('int64')
```

NumPy Arrays

```
In [104]: x = np.array([1,2,3])  
x
```

```
Out[104]: array([1, 2, 3])
```

```
In [105]: type(x)
```

```
Out[105]: numpy.ndarray
```

```
In [106]: # use dtype to show the datatype of the array  
x.dtype
```

```
Out[106]: dtype('int64')
```

```
In [107]: # np arrays can only contain one datatype and default to the most flexible type  
x = np.array([1, 'two', 3])  
x
```

```
Out[107]: array(['1', 'two', '3'], dtype='<U21')
```

NumPy Arrays

```
In [104]: x = np.array([1,2,3])  
x
```

```
Out[104]: array([1, 2, 3])
```

```
In [105]: type(x)
```

```
Out[105]: numpy.ndarray
```

```
In [106]: # use dtype to show the datatype of the array  
x.dtype
```

```
Out[106]: dtype('int64')
```

```
In [107]: # np arrays can only contain one datatype and default to the most flexible type  
x = np.array([1,'two',3])  
x
```

```
Out[107]: array(['1', 'two', '3'], dtype='<U21')
```

```
In [108]: x.dtype
```

```
Out[108]: dtype('<U21')
```

NumPy Arrays

```
In [104]: x = np.array([1,2,3])  
x
```

```
Out[104]: array([1, 2, 3])
```

```
In [105]: type(x)
```

```
Out[105]: numpy.ndarray
```

```
In [106]: # use dtype to show the datatype of the array  
x.dtype
```

```
Out[106]: dtype('int64')
```

```
In [107]: # np arrays can only contain one datatype and default to the most flexible type  
x = np.array([1,'two',3])  
x
```

```
Out[107]: array(['1', 'two', '3'], dtype='<U21')
```

```
In [108]: x.dtype
```

```
Out[108]: dtype('<U21')
```

```
In [109]: # many different ways to create numpy arrays  
np.ones(5,dtype=float)
```

```
Out[109]: 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 [110]: x = np.arange(1,6)  
x
```

```
Out[110]: array([1, 2, 3, 4, 5])
```

NumPy Array Indexing

- For single indices, works the same as list

```
In [110]: x = np.arange(1,6)  
x
```

```
Out[110]: array([1, 2, 3, 4, 5])
```

```
In [111]: x[0], x[-1], x[-2]
```

```
Out[111]: (1, 5, 4)
```


NumPy Array Slicing

NumPy Array Slicing

```
In [112]: x = np.arange(5) # note that in numpy it's arange instead of range
          x
```

```
Out[112]: array([0, 1, 2, 3, 4])
```

NumPy Array Slicing

```
In [112]: x = np.arange(5) # note that in numpy it's arange instead of range
x
```

```
Out[112]: array([0, 1, 2, 3, 4])
```

```
In [113]: # return first two items, start:end (exclusive)
x[0:2]
```

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

NumPy Array Slicing

```
In [112]: x = np.arange(5) # note that in numpy it's arange instead of range
x
```

```
Out[112]: array([0, 1, 2, 3, 4])
```

```
In [113]: # return first two items, start:end (exclusive)
x[0:2]
```

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

```
In [114]: # missing start implies position 0
x[:2]
```

```
Out[114]: array([0, 1])
```

NumPy Array Slicing

```
In [112]: x = np.arange(5) # note that in numpy it's arange instead of range
x
```

```
Out[112]: array([0, 1, 2, 3, 4])
```

```
In [113]: # return first two items, start:end (exclusive)
x[0:2]
```

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

```
In [114]: # missing start implies position 0
x[:2]
```

```
Out[114]: array([0, 1])
```

```
In [115]: # missing end implies length of array
x[2:]
```

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

NumPy Array Slicing

```
In [112]: x = np.arange(5) # note that in numpy it's arange instead of range
x
```

```
Out[112]: array([0, 1, 2, 3, 4])
```

```
In [113]: # return first two items, start:end (exclusive)
x[0:2]
```

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

```
In [114]: # missing start implies position 0
x[:2]
```

```
Out[114]: array([0, 1])
```

```
In [115]: # missing end implies length of array
x[2:]
```

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

```
In [116]: # return last two items
x[-2:]
```

```
Out[116]: array([3, 4])
```

NumPy Array Slicing with Steps

NumPy Array Slicing with Steps

```
In [117]: x
```

```
Out[117]: array([0, 1, 2, 3, 4])
```


NumPy Array Slicing with Steps

```
In [117]: x
```

```
Out[117]: array([0, 1, 2, 3, 4])
```

```
In [118]: # return every other item from position 1 to 4 exclusive  
# start:end:step_size  
x[1:4:2]
```

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

Reverse array with step-size of -1

Reverse array with step-size of -1

```
In [119]: x
```

```
Out[119]: array([0, 1, 2, 3, 4])
```

Reverse array with step-size of -1

```
In [119]: x
```

```
Out[119]: array([0, 1, 2, 3, 4])
```

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

```
Out[120]: 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 [121]: x = np.arange(5,10)
          x
```

```
Out[121]: array([5, 6, 7, 8, 9])
```

NumPy Fancy Indexing

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

```
In [121]: x = np.arange(5,10)
          x
```

```
Out[121]: array([5, 6, 7, 8, 9])
```

```
In [122]: x[[0,3]]
```

```
Out[122]: array([5, 8])
```


NumPy Fancy Indexing

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

```
In [121]: x = np.arange(5,10)  
x
```

```
Out[121]: array([5, 6, 7, 8, 9])
```

```
In [122]: x[[0,3]]
```

```
Out[122]: array([5, 8])
```

```
In [123]: x[[0,2,-1]]
```

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

Boolean Indexing using a Boolean Mask

Boolean Indexing using a Boolean Mask

In [124]:

```
x
```

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

Boolean Indexing using a Boolean Mask

In [124]:

```
x
```

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

In [125]:

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

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

Boolean Indexing using a Boolean Mask

In [124]:

```
x
```

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

In [125]:

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

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

In [126]:

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

Out[126]: array([6, 8])

Boolean Indexing using a Boolean Mask

In [124]:

```
x
```

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

In [125]:

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

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

In [126]:

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

Out[126]: array([6, 8])

In [127]:

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

Out[127]: array([7, 8, 9])

Boolean Indexing And Bitwise Operators

Boolean Indexing And Bitwise Operators

```
In [128]:
```

```
x
```

```
Out[128]: array([5, 6, 7, 8, 9])
```


Boolean Indexing And Bitwise Operators

In [128]:

```
x
```

Out[128]: array([5, 6, 7, 8, 9])

In [129]:

```
(x%2 == 0)
```

Out[129]: array([False, True, False, True, False])

Boolean Indexing And Bitwise Operators

```
In [128]: x
```

```
Out[128]: array([5, 6, 7, 8, 9])
```

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

```
Out[129]: array([False,  True, False,  True, False])
```

```
In [130]: (x > 6)
```

```
Out[130]: array([False, False,  True,  True,  True])
```

Boolean Indexing And Bitwise Operators

```
In [128]: x
```

```
Out[128]: array([5, 6, 7, 8, 9])
```

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

```
Out[129]: array([False,  True, False,  True, False])
```

```
In [130]: (x > 6)
```

```
Out[130]: array([False, False,  True,  True,  True])
```

```
In [131]: # Which values are divisible by 2 AND greater than 6?  
# 'and' expects both elements be boolean, not arrays of booleans!  
(x%2 == 0) and (x > 6)
```

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

Boolean Indexing And Bitwise Operators

```
In [128]: x
```

```
Out[128]: array([5, 6, 7, 8, 9])
```

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

```
Out[129]: array([False,  True, False,  True, False])
```

```
In [130]: (x > 6)
```

```
Out[130]: array([False, False,  True,  True,  True])
```

```
In [131]: # Which values are divisible by 2 AND greater than 6?  
# 'and' expects both elements be boolean, not arrays of booleans!  
(x%2 == 0) and (x > 6)
```

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

```
In [132]: # & compares each element pairwise  
(x%2 == 0) & (x > 6)
```

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

Boolean Indexing And Bitwise Operators

```
In [128]: x
```

```
Out[128]: array([5, 6, 7, 8, 9])
```

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

```
Out[129]: array([False,  True, False,  True, False])
```

```
In [130]: (x > 6)
```

```
Out[130]: array([False, False,  True,  True,  True])
```

```
In [131]: # Which values are divisible by 2 AND greater than 6?  
# 'and' expects both elements be boolean, not arrays of booleans!  
(x%2 == 0) and (x > 6)
```

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

```
In [132]: # & compares each element pairwise  
(x%2 == 0) & (x > 6)
```

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

```
In [133]: x[(x%2 == 0) & (x > 6)]
```

```
Out[133]: array([8])
```

Boolean Indexing And Bitwise Operators

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [134]: # Which values are even AND greater than 6?
          x[(x%2 == 0) & (x > 6)]
```

```
Out[134]: array([8])
```


Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [134]: # Which values are even AND greater than 6?  
x[(x%2 == 0) & (x > 6)]
```

```
Out[134]: array([8])
```

- or : `|` (pipe)

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [134]: # Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

```
Out[134]: array([8])
```

- or : `|` (pipe)

```
In [135]: # which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

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

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [134]: # Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

```
Out[134]: array([8])
```

- or : `|` (pipe)

```
In [135]: # which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

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

- not : `~` (tilde)

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [134]: # Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

```
Out[134]: array([8])
```

- or : `|` (pipe)

```
In [135]: # which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

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

- not : `~` (tilde)

```
In [136]: # which values are NOT (even OR greater than 6)
x[~( (x%2 == 0) | (x > 6) )]
```

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

Boolean Indexing And Bitwise Operators

- and : `&` (ampersand)

```
In [134]: # Which values are even AND greater than 6?
x[(x%2 == 0) & (x > 6)]
```

```
Out[134]: array([8])
```

- or : `|` (pipe)

```
In [135]: # which values are even OR greater than 6?
x[(x%2 == 0) | (x > 6)]
```

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

- not : `~` (tilde)

```
In [136]: # which values are NOT (even OR greater than 6)
x[~( (x%2 == 0) | (x > 6) )]
```

```
Out[136]: 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 [137]: x = [[1,2,3],[4,5,6]] # list of lists  
x
```

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

Multidimensional Lists

```
In [137]: x = [[1,2,3],[4,5,6]] # list of lists  
x
```

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

```
In [138]: # return first row  
x[0]
```

```
Out[138]: [1, 2, 3]
```

Multidimensional Lists

```
In [137]: x = [[1,2,3],[4,5,6]] # list of lists  
x
```

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

```
In [138]: # return first row  
x[0]
```

```
Out[138]: [1, 2, 3]
```

```
In [139]: # return first row, second column  
x[0][1]
```

```
Out[139]: 2
```

Multidimensional Lists

```
In [137]: x = [[1,2,3],[4,5,6]] # list of lists  
x
```

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

```
In [138]: # return first row  
x[0]
```

```
Out[138]: [1, 2, 3]
```

```
In [139]: # return first row, second column  
x[0][1]
```

```
Out[139]: 2
```

```
In [140]: # return second column?  
[row[1] for row in x]
```

```
Out[140]: [2, 5]
```

NumPy Multidimensional Arrays

NumPy Multidimensional Arrays

```
In [141]: x = np.array([[1,2,3],[4,5,6]])  
x
```

```
Out[141]: array([[1, 2, 3],  
                [4, 5, 6]])
```

NumPy Multidimensional Arrays

```
In [141]: x = np.array([[1,2,3],[4,5,6]])  
x
```

```
Out[141]: array([[1, 2, 3],  
                [4, 5, 6]])
```

```
In [142]: x[0,1] # first row, second column
```

```
Out[142]: 2
```

NumPy Multidimensional Arrays

```
In [141]: x = np.array([[1,2,3],[4,5,6]])  
x
```

```
Out[141]: array([[1, 2, 3],  
                [4, 5, 6]])
```

```
In [142]: x[0,1] # first row, second column
```

```
Out[142]: 2
```

```
In [143]: x[0,0:3] # first row
```

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


NumPy Multidimensional Arrays

```
In [141]: x = np.array([[1,2,3],[4,5,6]])  
x
```

```
Out[141]: array([[1, 2, 3],  
                [4, 5, 6]])
```

```
In [142]: x[0,1] # first row, second column
```

```
Out[142]: 2
```

```
In [143]: x[0,0:3] # first row
```

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

```
In [144]: x[0,:] # first row (first to last column)
```

```
Out[144]: array([1, 2, 3])
```

NumPy Multidimensional Arrays

```
In [141]: x = np.array([[1,2,3],[4,5,6]])  
x
```

```
Out[141]: array([[1, 2, 3],  
                [4, 5, 6]])
```

```
In [142]: x[0,1] # first row, second column
```

```
Out[142]: 2
```

```
In [143]: x[0,0:3] # first row
```

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

```
In [144]: x[0,:] # first row (first to last column)
```

```
Out[144]: array([1, 2, 3])
```

```
In [145]: x[:,1] # second column (first to last row)
```

```
Out[145]: array([2, 5])
```

NumPy Array Attributes

NumPy Array Attributes

```
In [146]: x = np.array([[1,2,3],[4,5,6]])
```

NumPy Array Attributes

```
In [146]: x = np.array([[1,2,3],[4,5,6]])
```

```
In [147]: x.ndim # number of dimensions
```

```
Out[147]: 2
```

NumPy Array Attributes

```
In [146]: x = np.array([[1,2,3],[4,5,6]])
```

```
In [147]: x.ndim # number of dimensions
```

```
Out[147]: 2
```

```
In [148]: x.shape # shape in each dimension
```

```
Out[148]: (2, 3)
```

NumPy Array Attributes

```
In [146]: x = np.array([[1,2,3],[4,5,6]])
```

```
In [147]: x.ndim # number of dimensions
```

```
Out[147]: 2
```

```
In [148]: x.shape # shape in each dimension
```

```
Out[148]: (2, 3)
```

```
In [149]: x.size # total number of elements
```

```
Out[149]: 6
```

NumPy Operations (UFuncs)

NumPy Operations (UFuncs)

```
In [150]: x = [1, 2, 3]  
          y = [4, 5, 6]
```

NumPy Operations (UFuncs)

```
In [150]: x = [1,2,3]  
          y = [4,5,6]
```

```
In [151]: x+y
```

```
Out[151]: [1, 2, 3, 4, 5, 6]
```

NumPy Operations (UFuncs)

```
In [150]: x = [1,2,3]  
         y = [4,5,6]
```

```
In [151]: x+y
```

```
Out[151]: [1, 2, 3, 4, 5, 6]
```

```
In [152]: x = np.array([1,2,3])  
         y = np.array([4,5,6])
```

NumPy Operations (UFuncs)

```
In [150]: x = [1,2,3]  
          y = [4,5,6]
```

```
In [151]: x+y
```

```
Out[151]: [1, 2, 3, 4, 5, 6]
```

```
In [152]: x = np.array([1,2,3])  
          y = np.array([4,5,6])
```

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

```
Out[153]: array([5, 7, 9])
```

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

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [154]: # square every element in a list  
x = [1,2,3]
```

```
In [155]: 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 [154]: # square every element in a list  
x = [1, 2, 3]
```

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

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

```
In [156]: [y**2 for y in x]
```

```
Out[156]: [1, 4, 9]
```

NumPy Broadcasting

Allows for vectorized computation on arrays of different sizes

```
In [154]: # square every element in a list  
x = [1,2,3]
```

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

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

```
In [156]: [y**2 for y in x]
```

```
Out[156]: [1, 4, 9]
```

```
In [157]: # 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 [154]: # square every element in a list  
x = [1,2,3]
```

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

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

```
In [156]: [y**2 for y in x]
```

```
Out[156]: [1, 4, 9]
```

```
In [157]: # square every element in a numpy array  
x = np.array([1,2,3])
```

```
In [158]: x**2
```

```
Out[158]: array([1, 4, 9])
```

NumPy **random** Submodule

NumPy **random** Submodule

Provides many random sampling functions

NumPy **random** Submodule

Provides many random sampling functions

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

Questions?