Goals for Today:

Python:

- Classes
- Reading files
- Importing

Classes

A class object is like a data structure except that it also holds **methods**

attributes: data objects associated with the class

methods: functions that are attached to classes. A method is typically logically connected with the data of the class it is attached to, and it typically has all of the class attributes within its **scope**.

Python has a lot of built-in classes:

```
a = [1,3,4,2]           # lists are implemented as a class
a.sort(reverse=True)    # .sort() is a method of the list class
print("a:"+str(a))
```

```
a:[4, 3, 2, 1]
```

- That's correct, **lists** are actually implemented as a **class** in python
- The list `[1,3,4,2]` is an attribute (or data) of the class
- `.sort()` is a method of the list class (Note: it even accepts optional arguments like a regular function).
- methods sometimes don't return anything. In the case of .sort() it has modified an attribute of the class (the list itself gets changed).

Classes

Even data objects are implemented as classes:

```
a = 'my string'
print(a.split())
print(a.split('t'))
```

```
['my', 'string']
['my s', 'ring']
```

Where `.split()` is a method of the string class with an optional argument for a delimiter.

There is a difference between the **class definition** and a specific **instance** of the class.

You can think of the **class** as the general template, and the **instances** as populating that template with specific attributes.

For example, "`string1`" and "`string2`" are both members of the string **class** (and have the same set of methods associated with them), but they have different **attributes** (i.e., the string itself).

This distinction will become obvious once we define our own classes.

Classes – Examples of Useful String Methods

Strings come with a lot of methods that make formatting easy:

```
# Example 1: .format method replaces each {} in a string with
a specified object
a = "substitute here -> {}"
print(a)
print(a.format("my string is awesome!"))
```

```
substitute here -> {}
substitute here -> my string is awesome!
```

Where `.format()` is a method for replacing instances of `{}` with other strings.

```
# Example 2: {} syntax can be used to format strings with a gi
ven precision or number of spaces
b = "{:12.6f} {:<20s} {:<10s} {:<2d}".format(20,"20chars","10c
hars",1)
print(b)
```

```
   20.000000 20chars              10chars    1
```

In the formatting specifications `{:12.6f} {:<20s} {:<10s} {:<2d}`, the letters `f`, `s` and `d`, mean `float`, `str`, and `int` respectively. The `<` means left-align. The numbers `12.6` mean the float should occupy twelve characters, with six values after the decimal (0 or spaces are automatically padded). The `:` is always required for formatted substitution.

Classes – Examples of Useful String Methods

Case-setting methods:

```
# Example 3: methods for capitalization etc.
c = "MiXeD CaSe 123"
print(c.upper())
print(c.lower())
print(c.capitalize())
print(c.title())
```

```
MIXED CASE 123
mixed case 123
Mixed case 123
Mixed Case 123
```

Combining methods:

```
# Example 4: strip and method chaining example
d = "this is a simple sentence."
print(d.strip('.te').title())
print(d.title().strip('.te'))
```

```
His Is A Simple Sentenc
This Is A Simple Sentenc
```

`.strip(chars)` removes chars from the start and end of the string.

Methods can be combined and are evaluated in order from left to right

Classes – Examples of Useful String Methods

.join() for formatting:

```
# Example 5: .join method
e = ["this","is","a","simple","sentence"]
print("".join(e))
print(" ".join(e))
print("...".join(e))
```

```
Thisisasimplesentence
this is a simple sentence
this...is...a...simple...sentence
```

`.join()` combines other strings using the string attribute as a delimiter.

Combining `.join()` with `.format()` and **list comprehensions** is powerful for formatting lists of numbers:

```
# Example 6: use join to format a list of numbers
f = [1,2,3,4]
print(" ".join([ "{:<10.6f}".format(i) for i in f ]))
```

```
1.000000   2.000000   3.000000   4.000000
```

This is useful when writing data to text files. This is also a good example to study for reinforcing several of the concepts and methods that we've covered.

Classes – Examples of Structure Methods

Lists, sets, and dictionaries have their own methods (we've seen some already):

```
# Example 1: list.index(object) example
a = [1,2,3,4,4,5]
print("a.index(4) = {}".format(a.index(4)))
print("a[a.index(4)] = {}".format(a[a.index(4)]))
```

```
a.index(4) = 3
a[a.index(4)] = 4
```

`.index(object)` returns the index of the **first** instance of object in the list.

We've already encountered the `list.pop()` method:

```
# Example 2: list.pop(index) example
b = a.pop(4)
print("a = {}".format(a))
print("b = {}".format(b))
```

```
a = [1, 2, 3, 4]
b = 4
```

Classes – Examples of Structure Methods

We've also seen a couple methods for sets:

```
# Example 3: set examples
a = {1,2}
b = {2,3}
print(a.intersection(b))
print(a.union(b))
```

```
{2}
{1,2,3}
```

`dict.keys()` returns the keys in a dictionary (in a set-like object) that can be useful for loops or evaluating membership:

```
# Example 4: dictionary.keys() examples
a = {1:"one",2:"two"}
print(a.keys())
print(1 in a.keys())
print(3 in a.keys())
```

```
dict_keys([1, 2])
True
False
```

File Objects*

Files are an example of a built-in class that we have not covered yet. They need to be mentioned here because files have special methods for reading data and this is central to almost all of the useful things we will do with Python.

The easiest way to create an instance of a file is to open one:

```
# Note: hamlet.txt needs to already exist in folder
f = open('hamlet.txt','r')
print(f)
```

```
<_io.TextIOWrapper name='hamlet.txt' mode='r' encoding='UTF-8'>
```

Here `open()` is a built-in Python function that opens files for reading (`'r'` option) or writing (`'w'` option) and returns a file object.

Our pointer `f` now points to the file object (i.e., `_io.TextIOWrapper` in the printed result is Python's way of telling you that).

Note: Classes are so ubiquitous in object-oriented programming languages that “object” and “class instance” are often used interchangeably.

file.read() and file.readlines()

For reference, here are the first lines of `hamlet.txt`:

```
ACT I
SCENE I. Elsinore. A platform before the castle.
FRANCISCO at his post. Enter to him BERNARDO
BERNARDO Who's there?
```

File objects have built-in methods for reading and closing files:

```
# Example 1: .readlines() and .close() method
f = open('hamlet.txt','r')
a = f.readlines(10) # optional argument specifies number of bytes to read
print("a: {}".format(a))
f.close()

# Example 2: .read() method
f = open('hamlet.txt','r')
b = f.read(10) # optional argument specifies number of bytes to read
print("b: {}".format(b))
```

```
a: ['ACT I\n', 'SCENE I. Elsinore. A platform before the castle.\n']
b: ACT I
SCEN
```

Example 1: `.readlines()` returns each line to a list based on the `\n` char.

Example 2: `.read()` returns the text file as a single string.

*By default these methods read the WHOLE FILE, which is **bad for large files**. The optional argument of each specifies the number of bytes to return.*

file.read() and file.readlines()

Before showing you a better way to read large files, let's look at one more illustrative example:

```
# Example 2: .read() method
f = open('hamlet.txt','r')
b = f.read(10)
print("b: {}".format(b))

# Example 3: file objects remember their place
c = f.readlines(10)
print("c: {}".format(c))
```

```
b: ACT I
SCEN
c: ['E I. Elsinore. A platform before the castle.\n']
```

The result in c is different from the previous slide. This is because we didn't close the file after the `.read(10)` call.

The file object has an attribute that keeps track of the pointer to the current position in the file (starting at the beginning when you open the file). As you read from the file, the position of this attribute changes.

File Objects are Iterable

File objects also have defined iteration rules. This allows you to loop over them line by line without reading the whole thing into memory. ***This is almost always preferred to using*** `.readlines()`:

```
# Example 4: Reading files using for loops
f = open('hamlet.txt','r')
for lines in f:
  try:
    print(lines.split()[0])
  except:
    pass
f.close()
```

```
ACT
SCENE
FRANCISCO
BERNARDO
Who's
FRANCISCO
[ … TRUNCATED OUTPUT … ]
```

The iteration rules for file objects returns each line of the file (based on new line characters) at each iteration of the `for` loop.

`try` and `except` are used here because not every line has a word.

File Objects Are Iterable

If you are like me and commonly forget to close things, then you can make it a habit to use the `with:` control statement, which automatically closes anything it calls:

```
# Example 5: Same as previous but using with statement
with open('hamlet.txt','r') as f:
  for lines in f:
      try:
        print(lines.split()[0])
      except:
        pass
```

```
ACT
SCENE
FRANCISCO
BERNARDO
Who's
FRANCISCO
[ … TRUNCATED OUTPUT … ]
```

`with:` automatically closes files after the closes `'hamlet.txt'` after the end of the control statement.

Python and common modules that we use have other specialized functions for extracting data from files, but file iteration is the most basic and flexible.

Writing Data with File Objects

Lastly, you will often want to write data to files.

File objects have the `.write()` method for this purpose, but you need to be sure that you open a file object with the correct permission:

```
# Example 6: writing data to file
a = [1,2,3,4]
with open('test.txt','w') as f:
  for i in a:
    f.write("{:<20.6f}\n".format(float(i)))
```

Contents of `test.txt` (in same directory as program*):

```
1.000000
2.000000
3.000000
4.000000
```

Be careful, opening an existing text file with open(name,'w') will overwrite its contents.

Use `open(name,'a')` when you open the file if you want to append to it.

*You can specify arbitrary locations using absolute (`/path/file`) or relative (`path/file`) paths. Note the difference in the leading forward slash (`/`).

Custom Classes

In general applications, you may find that you want to create your own classes, with certain attributes and methods beyond those that are available through other libraries or standard python distributions.

Defining a basic class with only attributes and no methods, is almost identical to defining a function:

```
class test:
  a=[1,2,3]
  b="string"
  c=1

# Create an instance of our class and print an attribute
instance = test()
print(instance.a)
```

```
[1, 2, 3]
```

The first line `class test:` indicates that we are creating a class named `test`

The indented section indicates the attributes (a,b,c) that belong to the class.

We create an **instance** of the class test and assign it to the **pointer** `instance`.

Access attributes of a class using the `.attribute` syntax, without parenthesis.

Custom Classes: `__init__`

First, that example is silly since the whole point of classes is to organize **attributes and methods** into a single object.

Second, we typically want our attributes to be unique to the instance, not hard-coded into the class.

To pass attributes at initialization, we use a special* method called `__init__`:

```
# Example 2: A class with attributes assigned at initialization.
class test:
  def __init__(self,a,b,c):
    self.a=a
    self.b=b
    self.c=c
```

• Methods are defined identically to functions, except that they are defined within a class definition (i.e., indented within the `class` keyword block).

• The first argument of all methods is usually **self**.*

• **self** is a pointer to the instance of the class. We could have used any name, but whatever we choose for the first argument in the `__init__` method is interpreted by Python as the pointer to "current instance of this class".

*We could use any name besides **self,** as long as we are consistent. but self is what is standard to use.

*Python has a whole collection of special methods that start and end with the double underscore ("dunder" methods)

Custom Classes: `__init__`

We have a few more things to say about `__init__(self,args)`:

```
# Example 2: A class with attributes assigned at initialization.
class test:
  def __init__(self,a,b,c):
    self.a=a
    self.b=b
    self.c=c
```

`__init__` is special because Python will automatically execute this method whenever you create an instance of this class. That is why you initialize the class with `class_name()` (like you are calling a function):

```
# Create an instance of our class and print its attributes
inst = test([1,2,3],"string",1)
print(inst.a)
```

```
[1, 2, 3]
```

We've defined the class to take three inputs from the user (`a`, `b`, and `c`) and assign them to the class attributes `self.a`, `self.b`, and `self.c`.

Note that `self` is an implicit argument in the __init__ method (and all others), We don't need to explicitly supply it when we initialize the object or call methods.

Custom Classes: `__init__`

If we define several instances of a class, they will have the same attributes but potentially different values for those attributes.

```
# Create two instances of our class and print their attributes
inst1 = test(1,1,1)
inst2 = test(0,0,0)
print("inst1.a = {}    inst2.a = {}".format(inst1.a,inst2.a))
print("inst1.b = {}    inst2.b = {}".format(inst1.b,inst2.b))
print("inst1.c = {}    inst2.c = {}".format(inst1.c,inst2.c))
```

```
inst1.a = 1     inst2.a = 0
inst1.b = 1     inst2.b = 0
inst1.c = 1     inst2.c = 0
```

By defining the class attributes using `__init__` the two instances of our object hold different values for `a`, `b`, and `c`.

The `__init__` method is the biggest stumbling block for people starting to study classes. In summary, it accomplishes two things. First, it gives the programmer the ability to pass inputs when an instance of the class is created. Second, it gives defines a pointer to the class instance (`self` by convention) that needs to be used when creating instance specific attributes and subsequent methods.

Custom Classes: Defining other methods

The reason that `self` is needed may become clearer when considering how you might add additional methods. Let's add a method to our class:

```
# Example 3: Create a class with an add function
class test:
  def __init__(self,a,b,c):
    self.a=a
    self.b=b
    self.c=c
  def add(self):
    return self.a + self.b + self.c

inst = test(1,1,1)
print(inst.add())
```

```
3
```

We've added a second method to our class called `add`, which returns the sum of attributes `a`, `b`, and `c`.

When we defined `add` we used the syntax `def add(self):`, meaning that we are passing the class instance as an input argument to the function.

If we had defined the method as `def add():`, then the method wouldn't have any inputs to add together.

Custom Classes: Defining other methods

`__init__` isn't the only special method within python that you might want to utilize for your class. What if we wanted to define how our class behaves in an equals comparison (i.e., the `==` logical operator):

```
# Example 4: Create a class with a custom equals comparison
class test:
  def __init__(self,a,b,c):
    self.a=a
    self.b=b
    self.c=c
  def __eq__(self,other):
    return self.c == other.c

inst1 = test(1,1,1)
inst2 = test(2,2,1)
inst3 = test(2,2,2)
print("inst1 == inst2 : {}".format(inst1 == inst2))
print("inst1 == inst2 : {}".format(inst1 == inst3))
```

```
inst1 == inst2 : True
inst1 == inst2 : False
```

In this case we have defined logical equality based on the attribute `c`.

`other` is interpreted by `__eq__` to mean the “other object” in the operator.

Importing: Use the Python Community

Python has a lot of built-in capability that isn't available automatically (it would create a lot of overhead if every time you wanted to do something you loaded every python function in existence.)

You can access these additional functions/classes/etc by using **import**:

```
import itertools
a = [1,2,3,4]
print([ i for i in itertools.combinations(a,2) ])
print([ i for i in itertools.combinations(a,3) ])
```

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

The syntax here is `import library_name`. Afterwards, functions/classes within that library can be accessed by `library_name.function()` etc.

You can also give the imported library a different name:

```
import itertools as IT
a = [1,2,3,4]
print([ i for i in IT.combinations(a,2) ])
print([ i for i in IT.combinations(a,3) ])
```

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

Importing: Use the Python Community

Common built-in libraries that you will often want to use are **math** (has common constants and trig functions), **os** (has common file and operating system operations), **random** (has common pseudorandom number generators) and **time** (has functions to return the current date, time etc.):

```
import math
print(math.sin(math.pi))
```

```
1.2246467991473532e-16
```

You can also import functions/classes individually:

```
from random import randint
print(randint(1,10))
```

```
5
```

Don't make a practice of importing the whole library without its namespace:

```
from random import *
```

This can lead to clashes if two libraries have functions with the same name.

There are many more libraries developed by specialized communities, including **scipy**, **matplotlib**, **scikit-learn**, and **keras** that we will also use in this course!