NYC DATA SCIENCE ACADEMY

Control Flows, Classes and Intro to Pandas

NYC Data Science Academy

OVERVIEW

- For Loops
- While Loops
- Errors and Exceptions
- Classes
  - Attributes and Methods
  - Inheritance
- Intro to Pandas
- Reading Spreadsheet

Loops

- Loops allow the repetition of a statement or list of statements. These statements are called the *body* of the loop.
- Some variables must change in those statements; otherwise, the loop would do the exact same thing each time. Thus, the body of a loop is a *statement*, not an *expression*.
- In addition to a body, a loop has a *header* which says how often the loop should execute. There are two types of loops:
  - *for* loops: The header says exactly what will vary each time the body executes (each *iteration* of the loop), and exactly how many times it will execute (how many iterations).
  - *while* loops: The header has a condition saying when to stop; the number of iterations is not known when the loop starts.

OVERVIEW

- For Loops
  - While Loops
  - Errors and Exceptions
  - Classes
    - Attributes and Methods
    - Inheritance
  - Intro to Pandas
  - Reading Spreadsheet

For loops

- A for loop steps through each of the items in a list or tuple (technically, any iterable object):

```
for name in collection:
    statements
```

- Each time *statements* is executed - in each *iteration* of the loop - the variable *name* is bound to the next element of the list.
- *You can think of a for loop as being like a map, except that instead of calculating an expression for each element of a list, you perform an action (i.e. execute a set of statements) for each element.*

For loops for printing

- A simple example is printing the elements of a list. Since print is a statement, we can't use it in a map.

```
words = ['a', 'b', 'c', 'd', 'e']
for w in words:
    print w,     # comma suppresses the newline
a b c d e
```

- Recall that the range function generates a list of numbers:

```
for i in range(len(words)):
    print i, words[i]
0 a
1 b
2 c
3 d
4 e
```

Modifying variables in a loop

- In addition to the iteration variable taking on values in a list, you may want other variables to take on different values in each iteration. You can accomplish this by "self-assigning" to those variables. This loop sums the elements of a list:

```
primes = [2, 3, 5, 7, 11]
sum_ = 0
for p in primes:
    sum_ = sum_ + p
sum_
28
```

- The loop body (consisting of the assignment to sum) is executed 5 times. Here are the values of primes and sum at the *start* of each iteration:

sum	0	2	5	10	17
p	2	3	5	7	11

Modifying variables in a loop

- As another example, here is a different way to print a list with its elements numbered:

```
words = ['a', 'b', 'c', 'd', 'e']
i = 0
for w in words:
    print i, w
    i = i + 1
```

Exercise 1: For loops

- Print the list of prime numbers along with the running sums of those numbers:

```
primes = [2, 3, 5, 7, 11]
sum_ = 0
for ...     # fill in for loop
2 2
3 5
5 10
7 17
11 28
```

Exercise 1: For loops

- Print a list of strings with numbers determined by the lengths of the strings:

```
names = ['don', 'mike', 'vivian', 'saul']
i = 0
for ...     # fill in for loop

0 don
3 mike
7 vivian
13 saul
```

For loops to copy

- Using append, we can make a copy of a list:

```
names = ['don', 'mike', 'vivian', 'saul']
copy = []
for name in names:
   copy.append(name)
copy
['don', 'mike', 'vivian', 'saul']
```

- We can get the effect of mapping a function `f` over the list just by changing the body of the loop to: `copy.append(f(name))`.

For loops for Mapping

- You should prefer map if you have a choice, because it is more concise and more efficient. But some things are hard to do. For example, doing running sums with a map is hard. So this loop would be hard to write with map:

```
primes = [2, 3, 5, 7, 11]
prime_sums = []
sum_ = 0
for p in primes:
    sum_ = sum_ + p
    prime_sums.append(sum_)
prime_sums
[2, 5, 10, 17, 28]
```

Exercise 2: For loops

1. Write a function `map_uc(l)` that takes a list of strings and returns a list of those same strings in all upper-case. You know how to do that using map; do it this time using a for loop. You'll need to create a copy, as you did in the loop, and return that.
2. In the previous exercise, you wrote a loop that produced this output:

```
0 don
3 mike
7 vivian
13 saul
```

   For this exercise, modify that loop to put pairs of these values in a list, instead of printing them, producing:

```
[[0, 'don'], [3, 'mike'], [7, 'vivian'], [13, 'saul']]
```

OVERVIEW

- For Loops
- **While Loops**
- Errors and Exceptions
- Classes
  - Attributes and Methods
  - Inheritance
- Intro to Pandas
- Reading Spreadsheet

While loops

- While loops are used when you do not know ahead of time how many iterations you will need:
  - Sum the elements of a list up to the first zero.
  - Newton's method is used to find a zero of an equation. It works by finding values that are closer and closer to the zero, until it finds a value "close enough." But there is no way to know how many times it will have to calculate a new value to get close enough.
  - Get input from a user until the user enters 'quit'.
- With a while loop, you iterate until a given condition becomes false:

```
while condition:
    statements
```

For loops as while loops

- As a first example, this loop prints integers from 0 to 9:

```
i = 0
while i < 10:
    print i
    i = i + 1
```

- This for loop does the same thing:

```
for i in range(0, 10):
    print i
```

- In both loops, the iteration variable is i. The for loop is simpler, but the while loops allows you to do some things that you can't do with the for loop, as we will see.

For loops as while loops

- Let's go through this loop in detail:

```
i = 0
while i < 10:
    print i
    i = i + 1
```

- Before starting the loop, we set i to 0. Here are the steps of the loop:
  - Test the condition: Since i = 0 < 10, execute the body. Print 0 and increment i to 1.
  - Test the condition: Since i = 1 < 10, execute the body. Print 1 and increment i to 2.
  - Test the condition: i = 2 < 10, so print 2 and increment i to 3.
  - Continue until i = 10. Then i < 10 is false, so the loop is terminated.

While loops

- One thing we can do with while loops that is hard to do with for loops is to terminate early. This loops adds up integers starting from 1 until the sum exceeds n:

```
n = 20
i = 1
sum_ = 0
while sum_ <= n:
    sum_ = sum_ + i
    i = i + 1
sum_
21
```

While loops

- This loop is similar, but sums the numbers in a list:

```
n = 20
i = 0
sum_ = 0
while sum_ <= n:
    sum_ = sum_ + L[i]
    i = i + 1
```

- When we iterate over a list like this, we should also test that we aren't going out of bounds:

```
i = 1
sum_ = 0
while sum_ <= n and i < len(L):
    sum_ = sum_ + L[i]
    i = i + 1
```

Exercise 3: While loops

- Now we'll do similar loops, but terminate under different conditions. If we're iterating over a list, remember to check that the list index is not out of bounds.
  a. Print the elements of a numeric list, up to the first even number.
  b. Print the elements of a list of strings, up to the first string whose length exceeds 10.
  c. Sum the *even* elements of a numeric list. This loop is different in that it contains an if statement (without an else).

Break and Continue Statements

- The `break` statement immediately terminates the (for or while) loop it is in. This provides a way to terminate the loop from within the middle of the body.
- The `continue` statement terminates *the current iteration* of the loop and goes back to the header.
- This loop adds the values in a list, but ignores negative numbers, and stops if the number exceeds 100:

```
sum = 0
for x in L:
    if x < 0:
        continue
    sum = sum + x
    if sum > 100:
        break
```

OVERVIEW

- For Loops
- While Loops
- **Errors and Exceptions**
- Classes
  - Attributes and Methods
  - Inheritance
- Intro to Pandas
- Reading Spreadsheet

Exceptions

- Exceptions are a language mechanism in Python (and many other languages) for handling unexpected and undesirable situations. Typical examples are:
  - Opening a file that does not exist
  - Dividing by zero
- The exception mechanism allows a program to handle such situations gracefully, without creating a lot of extra code.

Exceptions

- The mechanisms has two parts: signal the exception; and catch the exception.
  - Signal an exception

```
raise Exception
```

  - Catch exception: try

```
try:
    commands
except Exception:
    handle exception
```

An example

- Many predefined functions, or functions you import from modules, can throw exceptions. For example, the function open below raise an error when a file indicated by `filename` does not exist.
- So we focus on handling the errors first:

```
def openfile(filename, mode):
   try:
      f = open(filename, mode)
   except:
      print 'Error:', filename, 'does not exist'
      return
   ... use f ...
```

Built-in Exceptions

- The previous except clause - with no specific exception named - catches all exceptions. However, it is best to be specific about the exceptions you to avoid responding inappropriately.
- For example, the problem of the code below is that we specify a mode that does not exist, but the error message we print out is not true -- existent.txt does exist.

```
def openfile(filename, mode):
    try:
        f = open(filename, mode)
    except:
        print 'Error:', filename, 'does not exist'

openfile('existent.txt', 'no_such_mode')
Error: existent.txt does not exist
```

Exception types

- There are many different exceptions. Here are some of the most common:
  - `Exception`: the most general exception.
  - `TypeError`: the error when you give the wrong type to a function, e.g. 3 + []
  - `ValueError`: the exception when you give a bad value (of the correct type), e.g. int('abc')
  - `IndexError`: when your list subscript is out of bounds, e.g. [][0]
  - `IOError`: when you try to open a non-existent file
- The complete list is here: docs.python.org/2/library/exceptions.html.

Built-in Exceptions

- We can see the type of an error as below:

```
def openfile(filename, mode):
    try:
        f = open(filename, mode)
    except Exception as e:
        print type(e)

openfile('nonexistent.txt', 'r')
openfile('existent.txt', 'no_such_mode')
<type 'exceptions.IOError'>
<type 'exceptions.ValueError'>
```

try statements - general form

- The general form of the try statement, and the meaning of the various parts, is:

```
try:
    statements          # start by executing these
except name:
    statements          # execute if exception "name" was raised
...                     # more named except clauses
except:
    statements          # execute if an exception was raised that
                        # is not named above
else:
    statements          # execute if no exception was raised
finally:
    statements          # execute no matter what
```

try statements - general form

- In our example:

```
def openfile(filename, mode):
    try:
        f = open(filename, mode)
    except IOError:
        print 'File doesn\'t exist in this case.'
    except ValueError:
        print 'Likely to be wrong mode in this case.'
    except:
        print 'Some other error.'
    else:
        print 'No error'
    finally:
        print 'Everybody should have this!'
```

The Attributes of an Error

- The code in the previous slide is tested in the lecture code.
- Exceptions can carry more information than just their type; they have attributes giving information specific to the error. In examples above we came up with our own error message; we may use the attributes instead. We see that the function open actually raises exception with great detail:

```
def openfile(filename, mode):
    try:
        f = open(filename, mode)
    except Exception as e:
        print 'Error: ', e.args
```

- The result of the function can be found in the lecture code.

Hands-on Session

- Please go to the **"Handling Exceptions"** in the lecture code.

OVERVIEW

- For Loops
- While Loops
- Errors and Exceptions
- **Classes**
  - Attributes and Methods
  - Inheritance
- Intro to Pandas
- Reading Spreadsheet

Classes

- *Classes* are a method of organizing code. The idea is common to virtually all programming languages designed in the past thirty years, including Ruby, Java, C++, JavaScript, Perl, Scala, etc.
- Classes are closely tied to *objects*. A class is a syntactic construct that acts as a template for objects.
  - We first write a class.
  - Then we create objects according to the template provided by that class. These are called "objects" or "instances" of the class.
- Understanding what classes are, when to use them, and how to use them can be useful. In the process, we'll learn the meaning of the term *Object-Oriented Programming*.

Objects

- *"Everything is an object"*
- In Python, every value - integer, string, list, tuple, whatever - is an object.
- By defining classes, you can in effect define your own type of data. You can even define infix operators (like +) in your class.
- You can find the class of which an object is an instance by using the type function:

```
s = set([1,2,3])
type(s)
<type 'set'>
type(3)
<type 'int'>
type({1:2})
<type 'dict'>
```

So what is an object?

- An object is a collection of values together with functions that can access those values.
  - The values have *names*, and are called *fields*, or *attributes*.
  - The functions are called *methods*.
- Together, the values represent some object and the methods are the operations you can perform on those objects.
  - For example, a ComplexNumber object would be represented by two numbers (the real and imaginary parts) and would have operations like plus and times.
  - A Library object would be represented by two lists: all the books it has, and the ones that are checked out. The operations would include check_out_book(book) and return_book(book).

What can objects represent?

- Whatever you want!
  - An object might represent a real-life thing like a person or a car or a library.
  - It might represent a mathematical object, like a complex number or a graph.
  - It might represent a computer thing, like a dictionary or list.
- The critical question in each case is: *What operations do you want to perform on the object, and what information do you need to know to carry out those operations?*
- We'll discuss some examples in the abstract on the next few slides, before doing a concrete example.

Representing complex numbers

- A complex number has the form $x+iy$, where x and y are real numbers, and i = the square root of -1.
- Complex numbers support operations like:
  - Extract real and imaginary parts: $re(x+iy) = x$, $im(x+iy) = y$
  - Addition: $x+iy + x'+iy' = (x+x') + i(y+y')$
- Represent a complex number 4.7+i0.39 by two real numbers:

Class: Complex

Representation:	r = 4.7 i = 0.39
Operations:	*re*: return the r value in rep
	im: return the i value in rep
	plus: take another Complex object, return Complex object (sums of r values and the i values)

Representing complex numbers

- Note that the definitions of the operations are the same for every object in the class.
- The Complex class is a “template” for Complex objects:
  - It says that the representation is a pair of numbers.
  - It gives the definitions of the operations.

Representing a person

- There are countless properties of persons that you might want to include in a representation, depending upon your purposes. The representation is really determined by the operations.
- Our "Person" objects will allow the following operations:
  - name() returns the person's name.
  - phone() returns the person's phone number; set_phone (phone_number) changes the phone number (this is a mutating operation).

Representing a person

- The representation of a person is pretty obvious:

Class: Person

Representation:	name = ‘Sam Spade’ phone = ‘555-123-4567)
Operations:	*name*: return name in rep
	phone: return phone in rep
	set_phone(s): change phone in rep to s

- Again, the operations are the same for each object of the class.
- This class has a mutating operation, because it is possible for a person to change his or her phone number. (A mutating operation on complex numbers doesn’t make much sense; numbers don’t change.)

Representing a dictionary

- Python has a built-in dictionary, but we'll pretend it doesn't. Again, the important thing is the operations we want our dictionary to support:
  - lookup(key) returns the value associated with a key
  - add(key, value) associates value with key (replacing whatever key might have been associated with before)
  - contains(key) says whether key has an associated value
- We want to use our new dictionary the same as the built-in dictionary (except for syntax):

```
d = Dictionary()
d.add('ny', 'albany')
d.add('nj', 'trenton')
d.lookup('ny')
albany
```

Representing a dictionary

- The representation of a dictionary is not so obvious, but we discussed a possible (very inefficient) representation in a previous class:

Class: Dictionary

Representation:	kv_pairs = list of pairs [('ny', 'albany'), ('nj', 'trenton')]
Operations:	*lookup(k)*: use filter to find *k* as the first element in a tuple in kv_pairs, and return second element
	add(k,v): add pair *(k,v)* to the beginning of kv_pairs
	contains(k): same as *lookup(k)*, but return boolean

- Again, the operations are the same for each object of the class.
- This class has a mutating operation.

Exercise 5: Representing a dictionary

- For this first exercise on classes, we won't actually create a class, but will just create the functions needed to represent a dictionary object. In the next class, we'll create a class and turn these functions into methods.
- Define the Dictionary functions:
  - `Dictionary()`: Return an empty list.
  - `add(d, k, v)`: Modify `d` (list of pairs) so that `(k, v)` becomes the first pair. This is a mutating operation. (Remember insert on lists.)
  - `lookup(d, k)`: Use filter to lookup `k`. Don't worry about if `k` is in `d`.
  - `contains(d, k)`: Use filter to determine if `k` is defined in `d`.

Exercise 5: Representing a dictionary

- These should work as above, except that they don't use object-oriented syntax:

```
d = Dictionary()
add(d, 'ny', 'albany')
add(d, 'nj', 'trenton')
lookup(d, 'ny')
'albany'
contains(d, 'nj')
True
contains(d, 'nm')
False
```

OVERVIEW

- For Loops
- While Loops
- Errors and Exceptions
- Classes
  - **Attributes and Methods**
  - Inheritance
- Intro to Pandas
- Reading Spreadsheet

Defining classes

- We'll go into details in a bit, but here is the syntax to define a simple class:

```
class classname(object):
    def __init__(self):
        initialize representation by assigning to variables

    def methodname(self, ...args...):
        define method; change representation or
        return value or both; use self.var to refer to
        variable var defined in init.
```

- In __init__, we assign the desired representation to one or more variables, so we just have to decide what their names will be.

Defining classes

- Once we have this class, we can create instances of it:

```
newobj = classname()
```

- We invoke methods using object notation:

```
newobj.methodname(...args...)
```

- Note that even though we defined the method using ordinary function definition syntax, we call it using object syntax. That is just because it is defined inside a class.

Exercise 6: A dictionary class

- We will turn out dictionary functions into a class called `Dictionary`.
- We'll get you started with the definition of `__init__`:

```
class Dictionary(object):
   def __init__(self):
      self.kv_pairs = []
```

- Define `add`, `lookup`, and `contains`. These are identical to the definitions you gave before, except: The first argument should be named `self`; the list of pairs should be referred to as `self.kv_pairs`.
- Use object notation:

```
d = Dictionary()
d.add('ny', 'albany')
etc.
```

Classes

- We're now going to go into more detail. As our example, we'll define a class Vector representing vectors in an n-dimensional space.
- We will start out with simple operations: initialize a vector with a list of numbers; calculate the length of the vector in Euclidean space.

```
vec_1 = Vector([1,2,3])
vec_1.length()
3.74165738677
```

- After that we will introduce ways to print elements, add two vectors, and other operations.

Classes: Initialization

- Here is how to initialize an object with an argument:

```
class Vector(object):
    def __init__(self, lis):
        self.coords = lis
```

- `__init__()` always takes at least one argument, `self`, that refers to the object being created. Variables of the form `self.`*name* constitute the attributes of the object, i.e. its representation. In this case, the representation is a list, `self.coords`.
- When creating an instance of the class `Vector`, as shown on the previous slide, the `__init__()` method is invoked. It initializes the `coords` attribute of that instance.

Classes: Methods

- We can add methods to classes.
- For example, if we want to calculate the length of a vector, we can add:

```
class Vector(object):
    def __init__(self, lis):
        self.coords = lis

    def length(self):
        return sum([x**2 for x in self.coords])**.5
```

- When length is invoked as "`v.length()`", the instance `v` becomes the parameter `self`. Then `self.coords` is used to refer to the `coords` attribute of the instance.

Accessing Attributes and Methods

- As noted earlier, we now can create an instance of `Vector` and access its method using dot notation. We can also look at its attribute:

```
vec_1 = Vector([1,2,3])
print vec_1.coords
print vec_1.length()
[1, 2, 3]
3.74165738677
```

- However *it is poor style to make the attributes visible outside the class definition*. The problem is this: *Users of your class - "clients" - will come to depend upon these attributes. If you want to change the representation of objects, you cannot do it because your clients' code will break.* This may not sound like a big deal, but over time it is a very serious problem.

Hiding Attributes and Methods

- Python use prefix '__' (two underscores) to hide the attributes and methods from being directly accessed outside an object.
- Now prefix L with __ and try to access it:

```
class Vector(object):
    def __init__(self, lis):
        self.__coords = lis

    def length(self):
        return sum([x**2 for x in self.__coords])**.5

V = Vector([1,2,3])
v.__coords                          # AttributeError
```

Classes: Special method names

- In Python, a class can implement certain operations that are invoked by special syntax (such as arithmetic operations or subscripting) by defining methods with special names.
- For example, the `__str__()` method is called by the `str()` built-in function and by the `print` statement to compute the string representation of an object. E.g. add this method to `Vector`:

```
    def __str__(self):
        return 'Vector' + str(self.coords)
```

then print a Vector object:

```
vec_1 = Vector([1,2,3])
print vec_1
Vector[1, 2, 3]
```

Classes: Emulating numeric types

- For list objects, '+' means to concatenate two lists. For the `Vector` class we just created, we may want to do vector addition by using the expression *u* + *v*, where *u* and *v* are instances of `Vector`.
- In python we can implement the `__add__()` method:

```
def __add__(self, other):
    return Vector(map(lambda x, y: x+y, self.coords,
                                        other.coords))
```

- Note that this method returns a new Vector object. It is very common for non-mutating operations to return new objects in this way.

Classes: Emulating numeric types

- When we add two vector objects with '+', __add__() is called:

```
u = Vector([1,2,3])
v = Vector([4,5,6])
w = u + v    # Python actually runs u.__add__(v)
print w
Vector[5, 7, 9]
```

- For more special method names: https://docs.python.org/2/reference/datamodel.html

Exercise 7: Classes

- Now our Vector class looks like this:

```
class Vector(object):
    def __init__(self, lis):
        self.coords = lis

    def length(self):
        return sum([x**2 for x in self.coords])**.5

    def __add__(self, other):
        return Vector(map(lambda x, y: x+y,
                          self.coords, other.coords))

    def __str__(self):
        return 'Vector'+str(self.coords)
```

Exercise 7: Classes

- Add two more methods to the class:
  - `__eq__(vec)`: returns *true* iff *this* vector equals `vec`.
    - `u == v` calls `u.__eq__(v)`.
  - `__mul__(vec)`: returns the dot product of this vector and `vec`.
    The dot product is defined by: (x, y, …) * (x', y', …) = xx' + yy' + …
    - `u * v` calls `u.__mul__(v)`.
- Then evaluate the following expressions (*equality* and *cos* θ):

```
u = Vector([1,1,0])
v = Vector([0,1,1])
print u == v                                  # False
print (u*v) / (u.length()*v.length())         # 0.5
```

OVERVIEW

- For Loops
- While Loops
- Errors and Exceptions
- Classes
  - Attributes and Methods
  - **Inheritance**
- Intro to Pandas
- Reading Spreadsheet

Class Inheritance

- **Inheritance** is another important feature of object-oriented programming.
- With inheritance, a class can be a "child" of another class, and inherit the attributes and methods of the class.
  - The "parent" is called the *superclass* or *base class*.
  - The "child" is the *subclass* or *derived class*.
- Every class has a superclass; this is the name given in parentheses in the class definition:

```
class Vector(object):
```

In Python, you should use `object` as the base class if you don't want to inherit from any other class.

Class Inheritance

- To illustrate, we'll start with a class called Book, representing a generic book, with a name and an author. Its only operation is `__str__`.

```
class Book(object):
    def __init__(self, name, author = None):
        self.name = name
        self.author = author

    def __str__(self):
        return '<%s> by %s' %(self.name, self.author)
```

- We can use inheritance to create classes representing specific types of books, e.g. paper books, ebooks.

Class Inheritance

- We'll create a subclass for e-books. Note that every class has to have an __init__ function.

```
class EBook(Book):
    def __init__(self, name, author = None):
        Book.__init__(self, name, author)
```

- Several things to point out here:
  - EBook is a subclass of Book, as shown in the first line.
  - __init__ has the same arguments as Book's __init__. (We'll add specialized methods soon.)
  - EBook inherits the attributes of Book. It calls Book.__init__ to initialize them.

Class Inheritance: Attributes / Methods Inheritance

- EBook inherits the attributes and methods of Book. From the definition above, EBook and Book do exactly the same things.

```
book_1 = Book('The little SAS book', None)
ebook_1 = EBook('R CookBook', None)
print book_1
print ebook_1          # inherited method from Book
<The little SAS book> by None
<R CookBook> by None
```

- *Note*: try defining EBook without calling `Book.__init__()` to see what happens.

Class Inheritance: is-a relationship

- The key point about inheritance is this: A subclass can be used wherever its superclass could be used. That's because the subclass has all the methods of the superclass, so any client using the superclass can also use the subclass.
- We say that an EBook "is a" Book, or, more generally, any object of a subclass is also an object of the superclass. The way to think about inheritance is that derived classes define *specialized* instances of the base class.

Class Inheritance: is-a relationship

- There are several operations designed to let you understand the types of objects and the subclass relationships:
  - type: This will give the actual type of an object.

```
type(book_1)
<class '__main__.Book'>
type(ebook_1)
<class '__main__.EBook'>
```

  - issubclass: Check the relationship between two classes:

```
issubclass(EBook, Book)
True
issubclass(Book, EBook)
False
```

Class Inheritance: is-a relationship

- isinstance: Checks if an object is an object of a class *or of any of its subclasses*.

```
print isinstance(ebook_1, Book)
True
print isinstance(ebook_1, EBook)
True
print isinstance(book_1, Book)
True
print isinstance(book_1, EBook)
False
```

The first line above is the most interesting: `ebook_1` is considered an instance of `Book` even though it is actually an `EBook` object.

- Note that subclasses can have subclasses. `isinstance(`*object*`, `*class*`)` will return true as long as *object* is in any descendant of *class*.

Class Inheritance

- Derived classes can have their own attributes. We can add a new attribute - format, which can be 'pdf', 'kindle', etc. - to EBook:

```
class EBook(Book):
    def __init__(self, name, fmt, author = None):
        Book.__init__(self, name, author)
        self.fmt = fmt

    def get_fmt(self):
        return self.fmt

ebook_2 = EBook('R CookBook', 'pdf')
print ebook_2.get_fmt()
pdf
```

Class Inheritance: Method overriding

- Book already provides an `__str__()` method, but if we want EBook to do something different - say, to print the format as well - we can rewrite the `__str__()` method. This is called *method overriding*.

```
class EBook(Book):
    def __init__(self, name, fmt, author = None):
        Book.__init__(self, name, author)
        self.fmt = fmt
    def __str__(self):      # override __str__() method
        return Book.__str__(self)+', format: '+self.fmt
```

- When we call `__str__()`, it invokes the method from EBook:

```
ebook_3 = EBook('R CookBook', 'pdf')
print ebook_3
<R CookBook> by None, format: pdf
```

Mutating operations on objects

- Objects can be changed (mutated) simply by assigning to their attributes. Here we allow for the title of a book to be changed:

```
class Book(object):
    def __init__(self, name, author = None):
        self.name = name
        self.author = author
    def __str__(self):
        return '<%s> by %s' %(self.name, self.author)
    def rename(self, newname):
        self.name = newname
```

Mutating operations on objects

- The result is:

```
book_1 = Book('The little SAS book', None)
print "The name of book_1 is originally %s" % book_1
book_1.rename('The SAS book')
print "The name of book_1 is now %s" % book_1
The name of book_1 is originally <The little SAS book> by None
The name of book_1 is now <The SAS book> by None
```

Exercise 8: Inheritance

- Add a new attribute and two methods to `EBook`:
  a. `size` is the number of bytes in the `EBook`. This should be added to `__init__` as an argument with default 0, and should be included in the string representation.
  b. `get_size()` returns the size.
  c. compress() divides the size in half.

OVERVIEW

- For Loops
- While Loops
- Errors and Exceptions
- Classes
  - Attributes and Methods
  - Inheritance
- **Intro to Pandas**
- Reading Spreadsheet

Introduction to Pandas

- Pandas is a popular package defining several new data types, plus a variety of convenience functions for data manipulation. The most important data type is `DataFrame`, which is inspired by the type of same name in R, a programming language popular among statisticians and data scientists. We first import the module by the keyword import.

```
import pandas as pd
```

Introduction to Pandas

- We then create a data frame with the function DataFrame from pandas. Note that
  - The name of the function called follows the module name. This is why we need as keyword to use abbreviation.
  - Data are recorded in a **nested list**. Each inner list represents a row.
  - The **column names** can be specified within the `DataFrame` function.
  - Pandas automatically increments from 0 to the number of rows (minus 1) in a data frame as the **index**.

```
nested_lst = [[1,2,3,4,'hello'],
              [5,6,7,8,'world'],[9,10,11,12,'foo']]
my_df = pd.DataFrame(nested_lst, columns =
                     ['a','b','c','d','message'])
```

IO of Data Frame

- Of course, a data frame is used for data manipulation and analysis. After we process the data, we often want to save result back to a csv file. We may do that with the to_csv function and its default setting.
  - Note that the index would be written into the first row of the csv file.

```
my_df.to_csv('my_df.csv')
```

IO of Data Frame

- Having the index in a column of a csv file can cause trouble, because pandas assumes whatever it receives is data. The index in the csv file becomes an unnamed column in the data frame, and pandas then assign again an index to the data frame.

```
pd.read_csv('my_df.csv')
```

	Unnamed: 0	a	b	c	d	message
0	0	1	2	3	4	hello
1	1	5	6	7	8	world
2	2	9	10	11	12	foo

IO of Data Frame

- To avoid that, we can "tell" pandas which column in csv is actually the index (by integer).

```
pd.read_csv('my_df.csv', index_col=0)
```

	a	b	c	d	message
0	1	2	3	4	hello
1	5	6	7	8	world
2	9	10	11	12	foo

IO of Data Frame

- Actually when writing a data frame into a csv file, index doesn't need to be included:

```
my_df.to_csv('my_df.csv', index=False)
```

- We may also apply:
  - different `sep` argument.
  - argument `header = None`
  - different `skiprows` arguments
  - `read_table` function

  to take care of different situations. Examples are in the lecture code.

Index and Column Names

- We have shown how index and columns can be dealt with when importing the data. We may also do it after data is loaded. This can be done by changing the **index** or the **column names**. They can be accessed by:
  - `df.index`
  - `df.column`
  - `df.shape`
- Turn to the lecture code for more examples.

Selection and Filtering

- We often want to focus on a small portion of the data frame. We first load the the famous iris data set into a data frame. This data set consists of 5 columns:
  - The first four columns are numeric features of the iris flowers, sepal length, sepal width, petal length and petal width.
  - The last column is the species of each observation, including setosa, versicolor and virginica.
  - We assign the data frame to the variable iris.

```
iris = pd.read_csv('iris.csv')
```

Selection and Filtering

- Methods below are widely used:
  - The methods `head()` (`tail()`) returns the first (last) 5 rows.
  - `iloc` are used for selection by integer index.
  - `loc` are used for selection by index names or column names.
  - The symbol ":" can be used for selecting multiple rows or columns.
  - Fancy indexing is selecting by conditions.
- Turn to the lecture code for examples.

OVERVIEW

- For Loops
- While Loops
- Errors and Exceptions
- Classes
  - Attributes and Methods
  - Inheritance
- Intro to Pandas
- **Reading Spreadsheet**

Reading spreadsheets

- CSV is often used for sharing spreadsheets. But it can only include *values*, not formulas or charts. We discuss processing xlsx files.
- Module openpyxl allows you to read, write, and process xlsx files:
  - Read and write values and formulas in cells.
  - Handle multi-worksheet files.
  - Format cells, and more.
- openpyxl is not a standard module; you need to install it.
- Packages for xls files exist, but we will not cover them.
- *Factoid*: xlsx files are actually zip files containing XML files, we could process xlsx files with XML parsing instead of openpyxl.

Using openpyxl

- We will discuss how to read an xlsx file and extract parts of it.
  - openpyxl can also modify and save xlsx files, but we will not discuss that.
- Full documentation is at: openpyxl.readthedocs.org
- You need to install openpyxl and load it. It has several submodules; you will load some of those, depending upon what you want to do.
- Here is how to load an xlsx file and grab the active workspace:

```
import openpyxl
wb = openpyxl.load_workbook(filename = 'file.xlsx')
ws = wb.active
```

This load formulas and not values; to load values, add `data_only=True`.

Using openpyxl

- openpyxl provides operations to extract rows and cells, and to get cell contents:
  - ws.rows() - All the rows of ws as a tuple of tuples
  - ws['*cell-address*'] - The Cell object at the given location.
  - ws['*cell-address*':'*cell-address*'] - The the cells in this range, as an iterator. Apply list() to get a list of tuples of Cell values.
  - cell.type - Type of the cell
  - cell.value - Contents. Can be a formula, if workbook was opened with data_only = False (the default).
    - There is no way to evaluate a formula in openpyxl. Once you read the workbook with formulas or values, that's it.

Example: House prices

- As an example, this is houses.xlsx:

City	Price	SqFt	PerSqFt
Cleveland	100000	1500	$66.67
Cincinnati	125000	1600	$78.13
Akron	98000	1200	$81.67
Dayton	220000	3400	$64.71

- The last column is a formula, but we'll read it as values:

```
wb = openpyxl.load_workbook(filename = 'houses.xlsx',
                            data_only = True)
ws = wb.active
```

Example: House prices

- Here are examples of the openpyxl operations:

```
ws.rows
((<Cell Sheet1.A1>, <Cell Sheet1.B1>, <Cell Sheet1.C1>, <Cell Sheet1.D1>),
   ...,
 (<Cell Sheet1.A5>, <Cell Sheet1.B5>, <Cell Sheet1.C5>, <Cell Sheet1.D5>))
map(lambda c: (type(c.value), c.value), ws.rows[2])
[(unicode, u'Cincinnati'), (long, 125000L), (long, 1600L), (float, 78.125)]
ws['b3'].value
125000L
rng = ws['a1':'b3']
list(rng)
[(<Cell Sheet1.A1>, <Cell Sheet1.B1>),
 (<Cell Sheet1.A2>, <Cell Sheet1.B2>),
 (<Cell Sheet1.A3>, <Cell Sheet1.B3>)]
```

Example: Users query prices by city

- This code lets a user enter a city and find the corresponding house price:

```
wb = openpyxl.load_workbook(filename = 'houses.xlsx',
                            data_only=True)
ws = wb.active
prices = map(lambda r: (r[0].value, r[1].value), ws.rows)
d = dict(prices)
while True:
    city = raw_input('Enter a city: ')
    if city == '': break
    print 'Price: %d' % d[city]
```

Exercise 9: Selection and Filtering

- Use both loc and iloc to select the 5th row.

Exercise 9 Solution

```
iris.loc[5,:]
or
iris.iloc[5,:]
```

Exercise 10: Selection and Filtering

- Select the third to the seventh rows from the iris data set, with both loc and iloc method.
- Select the sepal width and the species columns from the iris data set.

Exercise 10: Solution

- ```
  iris.loc[2:6,:]
  or
  iris.iloc[2:7,:]
  ```

- ```
  iris.loc[:,['Sepal.Width','Species']]
  or
  iris.iloc[:,[1,4]]
  ```

Sorting

- Sorting is one of the most common tasks. We might want to sort the data frame according to the values, the index or the column names. There are of course multiple ways to achieve these, we introduce one method for each in the lecture code.

Exercise 11: Sorting

- Import the iris data frame again:

```
iris = pd.read_csv('iris.csv')
```

- Select the sub data frame whose species are all virginica.
- How many rows have species virginica?
- Sort this sub data frame by petal length.
- What is the 20% percentile of the petal length among all virginica?
  Hint: Let's say 20% percentile is the observation that is greater than exactly 20% of the observations.

Exercise 11: Solution

- Import the iris data frame again:

```
iris = pd.read_csv('iris.csv')
```

-
```
virginica = iris.loc[iris['Species']=='virginica',:]
```
-
```
virginica.shape[0]
```
-
```
Virginica = virginica.sort_values(by='Petal.Length')
```
-
```
virginica['Petal.Length'].iloc[11]
```

Data Manipulation

- We introduce basic data manipulations including:
  - dropping an old column
  - inserting a new column
  - deriving a new column from old ones
- Examples are in the lecture code.