CME 193: Introduction to Scientific Python

Winter 2017

Lecture 4: File I/O and Object Oriented Programming

Blake Jennings

`stanford.edu/~bmj/cme193`

Contents

- File I/O

- Classes

File I/O

How to read from and write to disk.

Much of the code you write will require input files for data and you will want to save output to a file as well.

The file object

- Interaction with the file system is pretty straightforward in Python.
- Done using *file objects*
- We can instantiate a file object using `open` or `file`

Opening a file

```
f = open(filename, option)
```

- filename: path and filename
- option:

'r'	read file
'w'	write to file
'a'	append to file

We need to close a file after we are done: `f.close()`

with open() as f

Very useful way to open, read/write and close file:

```
with open(’data/text_file.txt’, ’r’) as f:
    print f.read()
```

Python takes care of safely opening/closing file for you - just do operations within the indented block.

This method is actually recommended for many file i/o operations.

Reading files

read() Read entire file (or first n characters, if supplied)

readline() Reads a single line per call

readlines() Returns a list with lines (splits at newline)

Another fast option to read a file

```
with open(’f.txt’, ’r’) as f:
    for line in f:
        print line
```

Reading files

read() Read entire file (or first n characters, if supplied)

readline() Reads a single line per call

readlines() Returns a list with lines (splits at newline)

Another fast option to read a file

```
with open('f.txt', 'r') as f:
    for line in f:
        print line
```

Writing to file

Use `write()` to write to a file

```
with open(filename, ’w’) as f:
    f.write("Hello, {}!\n".format(name))
```

More writing examples

```
# write elements of list to file
with open(filename, ’w’) as f:
    for x in xs:
        f.write(’{}\n’.format(x))

# write elements of dictionary to file
with open(filename, ’w’) as f:
    for k, v in d.iteritems():
        f.write(’{}: {}\n’.format(k, v))
```

Exercise

Suppose you are given a text file where each line has a student's name and a bunch of grades for the student.

```
Jabrill 94 98 100 88 92
Jourdan 92 99 82 83 95
Ryan 77 80 94 87 92
```

Write a program that opens this file and writes a new file that on each line has the students' names as well as the average assignment score.

Exercise

```
with open(’grades.txt’, ’r’) as fin:
  with open(’grade_avg.txt’, ’a’) as fout:
    for line in fin:
      student = line.split()
      grades = [float(student[i]) for i in range(1,len(student))]
      avg = sum(grades)/(len(student)-1)
      fout.write(’{} {:.2f}\n’.format(student[0],avg))
```

File buffering

When writing to disk, the writes are buffered and periodically actually written to disk - Python takes care of this for us.

Everything goes to disk when file is closed.

`flush()` method for files allows us to manually push buffered writes to disk.

Contents

- File I/O

- Classes

Defining our own objects

So far, we have seen many objects in the course that come standard with Python.

- Integers
- Strings
- Lists
- Dictionaries
- etc

But often one wants to build (much) more complicated structures.

Consider 'building' a house in Python

Suppose you have a program that needs to store all information about houses. How are we storing all information about this house?

- A house might be a list with two elements, one for rooms, one for construction information
- house = [{bathroom: ..., kitchen: ...}, [brick, wood, ...]]
- For the rooms we might again want to know about what's in the room, what it's made off
- So bathroom = [materials, bathtub, sink], where materials is a list

We get a terribly nested structure, impossible to handle!

Consider ‘building’ a house in Python

Suppose you have a program that needs to store all information about houses. How are we storing all information about this house?

- A house might be a list with two elements, one for rooms, one for construction information
- house = [{bathroom: ..., kitchen: ...}, [brick, wood, ...]]
- For the rooms we might again want to know about what’s in the room, what it’s made off
- So bathroom = [materials, bathtub, sink], where materials is a list

We get a terribly nested structure, impossible to handle!

Consider ‘building’ a house in Python

Suppose you have a program that needs to store all information about houses. How are we storing all information about this house?

- A house might be a list with two elements, one for rooms, one for construction information
- house = [{bathroom: ..., kitchen: ...}, [brick, wood, ...]]
- For the rooms we might again want to know about what’s in the room, what it’s made off
- So bathroom = [materials, bathtub, sink], where materials is a list

We get a terribly nested structure, impossible to handle!

Consider 'building' a house in Python

Suppose you have a program that needs to store all information about houses. How are we storing all information about this house?

- A house might be a list with two elements, one for rooms, one for construction information
- house = [{bathroom: ..., kitchen: ...}, [brick, wood, ...]]
- For the rooms we might again want to know about what's in the room, what it's made off
- So bathroom = [materials, bathtub, sink], where materials is a list

We get a terribly nested structure, impossible to handle!

Procedural Programming

The previous example is what we've done before: procedural programming.

We relied on built in objects and data structures (ints, floats, lists, dictionaries, etc.) as well as our own procedures (functions, control flow statements, etc.) to write our desired programs.

As we see, sometimes not the easiest to use this technique.

Object Oriented Programming

Construct our own objects

- House
- Room
- etc

- Structure in familiar form (abstraction)
- Much easier to understand
- Code becomes very reusable

Object Oriented Programming

Construct our own objects

- House
- Room
- etc

- Structure in familiar form (abstraction)
- Much easier to understand
- Code becomes very reusable

Object Oriented Programming

Express computation in terms of objects, which are instances of classes

Class Blueprint (only one)

Object Instance (many)

Classes have attributes

- instance variables
- functions are called methods

```
a = 'HElLo, WoRld!'
a_lc = a.lower()
print a_lc
```

Object Oriented Programming

Express computation in terms of objects, which are instances of classes

Class Blueprint (only one)

Object Instance (many)

Classes have attributes

- instance variables
- functions are called methods

```
a = 'HElLo, WoRld!'
a_lc = a.lower()
print a_lc
```

Object Oriented Programming

Express computation in terms of objects, which are instances of classes

Class Blueprint (only one)

Object Instance (many)

Classes have attributes

- instance variables
- functions are called methods

```
a = 'HElLo, WoRld!'
a_lc = a.lower()
print a_lc
```

Python's way

In languages such as C++ and Java: data protection with private and public attributes and methods.

Not in Python: only basics such as inheritance.

Don't abuse power: works well in practice and leads to simple code.

More on encapsulation later!

Simplest example

```
# define class:
class Leaf:
    pass

# instantiate object
leaf = Leaf()

print leaf
# <__main__.Leaf instance at 0x10049df80>
```

Initializing an object

Define how a class is instantiated by defining the `__init__` *method*.

Initializing an object

The init or *constructor method*.

```
class Leaf:
    def __init__(self, color):
        self.color = color  # (default) public attribute

redleaf = Leaf('red')
blueleaf = Leaf('blue')

print redleaf.color
# red
```

Note how we *access* object *attributes*. We will have more on public/private attributes and encapsulation next time.

Self

The `self` parameter seems strange at first sight.

It refers to the the object (instance) itself.

Hence `self.color = color` sets the color of the object `self.color` equal to the variable `color`.

Another example

Classes have *methods* (similar to functions)

```
class Stock():
    def __init__(self, name, symbol, prices=[]):
        self.name = name
        self.symbol = symbol
        self.prices = prices

    def high_price(self):
        if len(self.prices) == 0:
            return ’MISSING PRICES’
        return max(self.prices)

apple = Stock(’Apple’, ’APPL’, [500.43, 570.60])
print apple.high_price()
```

Recall: *list.append()* or *dict.items()*. These are simply class methods!

Another example

Classes have *methods* (similar to functions)

```
class Stock():
    def __init__(self, name, symbol, prices=[]):
        self.name = name
        self.symbol = symbol
        self.prices = prices

    def high_price(self):
        if len(self.prices) == 0:
            return 'MISSING PRICES'
        return max(self.prices)

apple = Stock('Apple', 'APPL', [500.43, 570.60])
print apple.high_price()
```

Recall: *list.append()* or *dict.items()*. These are simply class methods!

Class attributes

```
class Leaf:
    n_leafs = 0  # class attribute: shared

    def __init__(self, color):
        self.color = color  # object attribute
        Leaf.n_leafs += 1

redleaf = Leaf(’red’)
blueleaf = Leaf(’blue’)

print redleaf.color
# red
print Leaf.n_leafs
# 2
```

Class attributes are shared among all objects of that class.

Attributes are by default public!

```
class Leaf:
  def __init__(self, color):
    self.color = color

  def getColor(self):
    return self.color

blueleaf = Leaf(’blue’)
color = blueleaf.getColor()
print color # blue
color = ’red’
print blueleaf.getColor() # ?
blueleaf.color = ’red’
print blueleaf.getColor() # ?
```

What do you think is returned by the last two print statements?

Access attributes by returning them with methods

What happens with this example?

```
class Student:
  def __init__(self, name, classes=[]):
    self.name = name
    self.classes = classes

  def getClasses(self):
    return self.classes

  def addClass(self, c):
    self.classes.append(c)

s = Student('Jabrill')
s.addClass('CME 193')
classes = s.getClasses()
print classes # ['CME 193']
classes[0] = 'CME 195'
print s.getClasses() # ?
```

Access attributes by returning them with methods

How would you fix the last example?

```
class Student:
  def __init__(self, name, classes=[]):
    self.name = name
    self.classes = classes

  def getClasses(self):
    return list(self.classes)

  def addClass(self, c):
    self.classes.append(c)

s = Student('Jabrill')
s.addClass('CME 193')
classes = s.getClasses()
print classes # ['CME 193']
classes[0] = 'CME 195'
print s.getClasses() # ?
```

Access attributes by returning them with methods

How would you fix the last example?

```
class Student:
  def __init__(self, name, classes=[]):
    self.name = name
    self.classes = classes

  def getClasses(self):
    return list(self.classes)

  def addClass(self, c):
    self.classes.append(c)

s = Student(’Jabrill’)
s.addClass(’CME 193’)
classes = s.getClasses()
print classes # [’CME 193’]
classes[0] = ’CME 195’
print s.getClasses() # ?
```

“Private” attributes

Python does have a way to declare attributes as private.

```
class Leaf:
  def __init__(self, color):
    self.__color = color

  def getColor(self):
    return self.__color

blueleaf = Leaf('blue')
color = blueleaf.getColor()
print color # blue
color = 'red'
print blueleaf.getColor() # ?
print blueleaf.__color
# AttributeError: Leaf instance has
# no attribute '__color'
```

“Private” attributes

However, there is still a way for us to access private attributes outside of the class methods.

```
class Leaf:
  def __init__(self, color):
    self.__color = color

  def getColor(self):
    return self.__color

blueleaf = Leaf(’blue’)
color = blueleaf.getColor()
print color # blue
color = ’red’
print blueleaf.getColor() # ?
print blueleaf._Leaf__color
# ?
```

Public vs private attributes

As we've seen, no real way to fully protect our class attributes.

Bad practice to access object attributes directly.

Can use "private" attributes to make it harder/more work to access.

Function override

We can override built in methods to define how our objects behave with Python operators/functions.

Some methods to override

- `__init__`(self,...): Constructor
- `__repr__`(self): Represent the object (machine)
- `__cmp__`(self, other): Compare self and other
- `__add__`(self, other): Add self and other
- Many more!

Example

```
class Student:
  def __init__(self, name, classes=[]):
    self.name = name
    self.classes = classes

  def getClasses(self):
    return self.classes

  def getName(self):
    return self.name

s = Student(’Jabrill’, [’CME 193’, ’CME 195’])
print s
# <__main__.Student instance at 0x7f68f0c39d88>
print s.getName() # Jabrill
print s.getClasses() # [’CME 193’, ’CME 195’]
```

Example

```
class Student:
  def __init__(self, name, classes=[]):
    self.name = name
    self.classes = classes

  def getClasses(self):
    return self.classes

  def getName(self):
    return self.name

  def __repr__(self):
    string = ’%s:’ % self.name
    string += ’ %s’ % self.classes
    return string

s = Student(’Jabrill’, [’CME 193’, ’CME 195’])
print s
# Jabrill: [’CME 193’, ’CME 195’]
```

Operator overloading

This ability to define how objects behave with basic operators allows us to define interactions between objects.

One of the most powerful aspects of object oriented programming.

Example

Implementing Rational numbers

```
class Rational:
    pass
```

Setup

What information should the class hold?

- Numerator
- Denominator

Setup

What information should the class hold?

- Numerator
- Denominator

Init

Implement the `__init__` method

```
class Rational:
    def __init__(self, p, q=1):
        self.p = p
        self.q = q
```

Init

Implement the `__init__` method

```python
class Rational:
    def __init__(self, p, q=1):
        self.p = p
        self.q = q
```

Issues

Issues?

```
class Rational:
    def __init__(self, p, q=1):
        self.p = p
        self.q = q
```

Ignore the division by 0 for now, more on that later.

Issues

Issues?

```python
class Rational:
    def __init__(self, p, q=1):
        self.p = p
        self.q = q
```

Ignore the division by 0 for now, more on that later.

Greatest common divisor

$\frac{10}{20}$ and $\frac{1}{2}$ are the same rational. We'd like to for the objects to be the same. Let's store our rational numbers in fully reduced form.

Implement a `gcd(a, b)` function that computes the greatest common divisor of a and b.

```python
def gcd(a, b):
    if b == 0:
        return a
    else:
        return gcd(b, a%b)
```

If interested: Verify Euclidean Algorithm

Greatest common divisor

```python
class Rational:
    def __init__(self, p, q=1):
        g = gcd(p, q)
        self.p = p / g
        self.q = q / g
```

Can put the gcd function in the same file as your class outside of the class definition (easiest to put above class definition) or import file with the function.

Representing your class: Operator overloading

Implement `__repr__` or `__str__` early to `print`

Useful for debugging

Operator overloading: adding two Rationals

Add Rationals just like Ints and Doubles?

`Rational(10,2) + Rational(4,3)`

To use +, we implement the `__add__` method

```
class Rational:
    # ...
    def __add__(self, other):
        p = self.p * other.q + other.p * self.q
        q = self.q * other.q
        return Rational(p, q)
    # ...
```

Does this change any of self or other?

Operator overloading: Comparing

`__cmp__` compares objects

- If `self` is smaller than `other`, return a negative value
- If `self` and `other` are equal, return 0
- If `self` is larger than `other`, return a positive value

You will implement this function and others in the exercise for today.

Class hierarchy through inheritance

It can be useful (especially in larger projects) to have a hierarchy of classes.

Example

- Animal
  - Bird
    - Hawk
    - Seagull
    - ...
  - Pet
    - Dog
    - Cat
    - ...
  - ...

Inheritance

Can have one class inherit attributes from another class.

Original class is called base class or parent class.

New class is called derived class or child class.

Child classes will usually redefine or add new attributes.

Inheritance

Suppose we first define an abstract class

```
class Animal:
    def __init__(self, n_legs, color):
        self.n_legs = n_legs
        self.color = color

    def make_noise(self):
        print 'noise'
```

Inheritance

We can define sub classes and inherit from another class.

```
class Dog(Animal):
    def __init__(self, color, name):
        Animal.__init__(self, 4, color)
        self.name = name
    def make_noise(self):
        print self.name + ’: ’ + ’woof’

bird = Animal(2, ’white’)
bird.make_noise()
# noise
brutus = Dog(’black’, ’Brutus’)
brutus.make_noise()
# Brutus: woof
shelly = Dog(’white’, ’Shelly’)
shelly.make_noise()
# Shelly: woof
```

Final notes

Many more things you can do with OOP in Python, this is just an intro.

Many sources for further information.

Can document classes just as you can functions. Good practice to do that.