DEPAUL UNIVERSITY
COLLEGE OF COMPUTING AND DIGITAL MEDIA

CSC 243 NOTES
Fall 2020: Amber Settle

Monday, November 9, 2020

Announcements

- Questions about the seventh assignment?
- The eighth assignment is due Thursday, November 12th – questions?

Programmer-defined classes

Classes provide a way for the application programmer to define new types.

An exercise

To review the basics of classes, we'll do an exercise.

Exercise 1: Develop a new class called `Animal` that abstracts animals and supports three methods:

a) `setSpecies()`: Takes a species (a string) as a parameter and sets the species of the animal object to it
b) `setLanguage()`: Takes a language (a string) as a parameter and sets the language of the animal object to it.
c) `speak()`: Take no parameters and returns a message from the animal shown as below.

An example of how the `Animal` class would be used is provided below:

```
>>> snoopy = Animal()
>>> snoopy.setSpecies('dog')
>>> snoopy.setLanguage('bark')
>>> snoopy.speak()
'I am a dog and I bark'
```

Writing constructors

The method `__init__` is a class method that is automatically called when a new instance object is being constructed.

Exercise: Write a default constructor for the `Account` class that sets the balance to 0.

The following is an example of how it would get used:

```
>>> acct = Account()
>>> acct
```

```
Account(0)
```

In order to create an `Account` object with a non-zero initial balance, we have to use both the constructor and the set method:

```
>>> acct = Account()
>>> acct
Account(0)
>>> acct.set(100)
>>> acct
Account(100)
>>>
```

It would be a lot more convenient to create an `Account` object using a parameter to the constructor:

```
>>> acct = Account(100)
```

How could we do that?

Exercise: Change the default constructor in the `Account` definition so that it takes a parameter.

Now what happens when we try to do the following?

```
>>> acct = Account()
```

How can we fix this?

Exercise: Make the constructor work for both a parameter and no parameters.

An exercise

To review the basics of classes, we'll continue a previous exercise.

Exercise 2: Add a constructor to the `Animal` class that takes two parameters, representing the species and language of the animal.

```
>>> a1 = Animal('rabbit', 'squeak')
```

Exercise 3: Modify the constructor of the `Animal` class so that it can take two parameters (species, language), one parameter (species only), or no parameters. If a parameter is missing the string 'default' will be used for the missing value (or pick your favorite animal and language as the default).

```
>>> a1 = Animal('rabbit', 'squeak')
>>> a2 = Animal('turtle')
>>> a3 = Animal(lang = 'peep')
```

Exercise 4: Add definitions for `__str__` and `__repr__` to the Animal class. **Bonus**: Make the string method call `speak()`. Write code to test the new methods.

```
>>> a1 = Animal('rabbit', 'squeak')
>>> a1
Animal(rabbit, squeak)
>>> print(a1)
I am a rabbit and I squeak
>>>
```

The solutions for all of them are in the file **animal.py**.

Inheritance

As we discussed, in Python a class is typically defined as inheriting from another class.

However, the only examples we've seen so far are defined as subclasses of the built-in class object, which has no attributes. Therefore `Account` does not inherit anything from it.

In general though, it is useful to define the new class as inheriting the attributes of another class for the purposes of code re-use.

We will first consider classes that inherit attributes from their **superclass**.

For example, the class `Savings` below is defined to inherit from the already defined class `Account`:

```
class Savings(Account):
    'a savings account'

    def setRate(self, value):
        'set the yearly interest rate'
        self.rate = value

    def addInterest(self):
        'add one month of interest to the balance'
        self.balance += self.balance*(self.rate/12)


    def get(self):
        'prints the value of the balance'
        display = float(self.balance)
        print('balance = ${:.2f}\nrate =
{}%'.format(display, self.rate))
```

This class is found in **BankAccountSecond.py** file.

The superclass is listed in parentheses in the class header. Classes inherit attributes from all "ancestor" classes.

The class `Savings` is said to be a **subclass** of the class `Account`. The class `Account` is said to be a **superclass** of the class `Savings`.

The class `Savings` inherits all the methods of `Account`. It has additional attributes, namely the methods `setRate()` and `addInterest()` and the variable self.rate.

However, the method `get()` has been redefined in `Savings`, meaning that it is **overloaded**.

```
>>> acct = Savings()
>>> acct.setRate(0.003)
>>> acct.get()
balance = $100.00
rate = 0.003%
>>> acct2 = Account()
>>> acct2.get()
0.0
>>>
```

Because of the added variable `self.rate`, the constructor for the `Account` class does not suffice for the `Savings` class.

Why? Consider the following:

```
>>> acct = Savings()
>>> acct.get()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    acct.get()
  File
"C:/Users/asettle/Documents/Courses/Csc
242/Notes/Week
2/Programs/BankAccountSecondSol.py", line
38, in get
    print('balance = ${:.2f}\nrate =
{}%'.format(display, self.rate))
AttributeError: 'Savings' object has no
attribute 'rate'
>>>
```

What happened? And how can we fix it?

Exercise: Add a constructor to the `Savings` class.

Note the duplicated code in the `Account` and `Savings` constructors. Eliminate the duplicated code by making the `Savings` constructor call the `Account` constructor.

Instances inherit attributes from all accessible classes.

Draw a picture of `Savings` and its relationship to `Account`. It will be similar to the drawing on page 280.

When an object o of a class c invokes a method m, the Python interpreter will search for method m among the attributes of c.

If it is not there, it will search for it in the superclass of c and then further up the ancestor tree of classes.

Exercise: Develop a subclass `Bird` of the class `Animal` that supports one new method called `fly()`. The method `fly()` takes one parameter `n`, and returns the string "I am flying n feet!"

```
>>> b1 = Bird()
>>> b1.fly(10)
'I am flying 10 feet!'
>>>
```

Writing constructors

The method `__init__` is a class method that is automatically called when a new instance object is being constructed.

Exercise: Add a constructor to the `Bird` class that takes one parameter representing the language of the bird.

```
>>> b1 = Bird('tweet')
```

Exercise: Modify the constructor for the Bird class so that it allows no parameters or one parameter representing the language. If no parameters are provided the language for the bird should be 'default'. Then make it call the constructor for the Animal class.

```
>>> b1 = Bird('tweet')
>>> b2 = Bird()
```

The solutions for all of them are in the file **animal.py**.

String representations

There are two overloaded operators that return **the string representation of the object**: __repr__() and __str__().

The operator **__repr__()** is supposed to return the canonical string representation of the object.

The operator **__str__()** returns an informal, ideally very readable, string representation of the object.

Does the __repr__ and __str__ method for the Account class work for the Savings class? Why or why not? If not, how would we fix it?

Exercise: Add a definition for __repr__ to the Bird class that overload the one obtained from the parent class. Use the __str__ method as is.

```
>>> b1 = Bird('tweet')
>>> b1
Bird(tweet)
>>> print(b1)
I am a bird and I tweet
>>>
```

The solutions are in **animal.py**.

Operator overloading

By overloading common operators such as ==, +, *, split(), len(), etc. we can define class behavior that is more familiar.

The following is an extended version of the Account class that has two operators: > and +

```
class Account(object):
    'a bank account class'

    def set(self, value):
        'set the balance to value'
        self.balance = value

    def get(self):
        'return the current balance on the account'
        return self.balance

…

    def __gt__(self, other):
        if self.balance > other.balance:
            return True
        else:
            return False

    def __add__(self, other):
        return Account(self.balance + other.balance)
```

You would use the operators as follows:

```
>>> acct1 = Account(100)
>>> acct2 = Account(50)
>>> acct3 = acct1 + acct2
>>> type(acct3)
<class '__main__.Account'>
>>> acct3.get()
150
```

```
>>> acct1.get()
100
>>> acct2.get()
50
>>> acct1 > acct2
True
>>> acct1 > acct3
False
>>>
```

Exercises

Exercise 1: Add an equality method to the Animal class. Two Animals are equal if their species and languages are the same.

```
>>> a1 = Animal('dog', 'woof')
>>> a2 = Animal('dog', 'yipyipyip')
>>> a3 = Animal('cat', 'woof')
>>> a4 = Animal('dog', 'woof')
>>> a1 == a2
False
>>> a2 == a3
False
>>> a3 == a4
False
>>> a1 == a4
True
>>>
```

Exercise 2: Add a concatenation operator to the `Animal` class that creates hybrids of the animals being concatenated.

See below for an example:

```
>>> annika = Animal('cat', '___meow___')
>>> annika
Animal(cat, ___meow___)
>>> bonnie = Animal('dog', 'woof')
>>> bonnie
Animal(dog, woof)
>>> clyde = annika + bonnie
>>> clyde
Animal(cat/dog, ___meow___/woof)
>>>
```

See the solution in **animal.py**.

The W3

The next topic we'll cover in this class is how to manipulate files on the World Wide Web (WWW or W3).

The W3 is a **system of linked documents stored on servers** on the Intranet.

A **web browser** is typically used to:

1. View the web pages that may contain text, images, videos, and other multimedia
2. Navigate between web pages by using hyperlinks, by which pages are typically accessed

The crucial people/institutions/technologies:

- People
  - Doug Englebart
  - Tim Berners Lee
  - Robert Cailliau
  - Marc Andreesen
- Institutions
  - CERN
  - Netscape
- Technologies
  - HyperText
  - URL (uniform resource locator)
  - HTML (hypertext markup language)
  - HTTP (hypertext transfer protocol)

Uniform Resource Locators (URLs)

In order to unambiguously identify and access particular resources on the W3, each resource must have a unique identifier.

A **uniform resource identifier** (URI) is a string of characters used to identify a resource on the Internet.

A URI can be classified as a **locator** (URL) or a **name** (URN) or both.

A URN functions like a person's name whereas a URL is like a person's street address. So the URN is the item's identity, and the URL is the method for finding it.

A URL is a URI that, in addition to identifying a resource, specifies the **means of acting upon or obtaining the representation**. It does this one of two ways (although the two can be combined):

1. Through a description of the primary access mechanism
2. Through a network "location"

From the W3C (http://www.w3.org/Addressing/):

> Uniform Resource Identifiers (URIs aka URLs) are short strings that identify resources in the web: documents, images, downloadable files, services, electronic mailboxes, and other resources.

They make resources available under a variety of naming schemes and access methods such as HTTP, FTP, and Internet mail addressable in the same simple way. They reduce the tedium of "log into this server, then issue this magic command …" down to a single click.

Here is a typical URL dissected:

```
http://facweb.cdm.depaul.edu/asettle/csc243/web/test.html
\___/ \_____________________/\_____________________________/
  |            |                         |
protocol     host                      path
```

Here is a more general (and fabricated) example:

```
http://user:pass@example.com:992/animal/bird?species=seagull#wings
 \___/\_______/\_________/\___/\_________/\_____________/\____/
   |       |         |      |       |            |           |
protocol  login     hosts   port   path        query  anchor/fragment
```

Hypertext Markup Language (HTML)

From the W3C (http://www.w3.org/TR/html401/intro/intro.html#h-2.2):

> To publish information for global distribution, one needs a universally understood language, a kind of publishing mother tongue that all computers may potentially understand.
>
> The publishing language used by the World Wide Web is HTML (from HyperText Markup Language).

HTML gives authors the means to:

- Publish online documents with headings, text, tables, lists, photos, etc.
- Retrieve online information via hypertext links, at the click of a button
- Design forms for conducting transactions with remote services, for use in searching for information, making reservations, ordering products, etc.
- Include spreadsheets, video clips, sound clips, and other applications directly in their documents.

Some examples:

- http://facweb.cdm.depaul.edu/asettle/csc243/web/test.html
- http://facweb.cs.depaul.edu/asettle/csc243/web/thekids.html
- http://facweb.cdm.depaul.edu/asettle/csc243/web/test2.html

An HTML primer

A file containing an HTML document should be named with an .html or .htm extension. Any such document placed on a web server will appear on the W3.

The following is our first example, also found in the file **test.html**:

```
<html>
<head>
  <title>Test page</title>
</head>

<body>
<h2> Hello World </h2>

<p>
A paragraph. The first line breaks here<br>, but that's OK.
</p>

<h1>Bigger heading</h1>

<p>
The second para<br>graph.
</p>
</body>
</html>
```

Note that every declaration has an opening tag (`<html>`) and a closing tag (`</html>`).

The same format applies for most of the tags in HTML.

HTML document structure

Every HTML document consists of two basic parts: the “head” and the “body”.

The head starts with the `<head>` tag and ends with the closing `</head>` tag.
The body starts with the `<body>` tag and ends with the closing `</body>` tag.

Both the head and the body must be enclosed in an html tag:
`<html>…</html>`

Thus a typical HTML document has the following basic structure:

```
<html>
<head>
…
</head>

<body>
…
</body>
</html>
```

The above example illustrates an important feature of HTML tags, called nesting.

> If you open tag A and then open tag B, you must close tag B before closing tag A.

Head

The **head** is where you place the title of your document and possibly other features (like references to style sheets, event handlers for dynamic content, etc.).

The title tag usually contains the site's name which is displayed in the browser's title bar and tabs.

```
<title>Test page</title>
```

Body

The **body** of the HTML document contains all content that is displayed in a browser including text, images, lists, tables, and more.

Text in HTML documents is usually enclosed in paragraph tags, and new lines break using the break tag.

```
<p>
A paragraph.  The first line breaks here<br>, but that's
OK.
</p>
<p>
The second para<br>graph.
</p>
```

Headlines

There are **six types of headline tags**, or headings.

> Headings provide a shortcut for creating larger text, as well as providing a logical hierarchy for ordering content.
>
> While you can simulate the headline effect by playing with the text's weight and size, but correct HTML is also about semantically ordering text. Using headings provides a way to do that.

The headings are in decreasing size and weight, meaning that h1 is larger than h6.

> See **test.html** for an example.

Lists

Often a page has to include bulleted or numbered lists of various things.

In HTML there are multiple ways to construct ordered and unordered lists of items.

Ordered lists

The tags for **ordered lists** in HTML are `<ol>` and `</ol>`. To create **list items** within an ordered list you use the tags `<li>` and `</li>`.

The default for ordered lists is to use **Arabic numerals starting at 1**.

For example, the following:

```
<ol>
<li>Item 1</li>
<li>Item 2</li>
</ol>
```

Would be rendered as:

1. Item 1
2. Item 2

You can change the default numbering using the **type attribute**.

For example, the following:

```
<ol type ="A">
<li>Item 1</li>
<li>Item 2</li>
<li>Item 3</li>
</ol>
```

Would be rendered as:

A. Item 1
B. Item 2
C. Item 3

Unordered lists

To create an unordered (bulleted) list, you need to use the `<ul>` and `</ul>` tags.

For example, the following:

```
<ul>
<li>Item 1</li>
<li>Item 2</li>
</ul>
```

Would render as follows:

- Item 1
- Item 2

You can also use the **type attribute** to change the bullets that display.

For example, the following:

```
<ul type = "square">
<li>Item 1</li>
<li>Item 2</li>
</ul>
```

Would render as follows:

- Item 1
- Item 2

Images

To insert an image into an HTML page, you need to use the `<img>` tag.

In the `<img>` tag the **src attribute** is where you specify the location and file name of the image that you want to display.

For example, the following would place a file called jojo.jpg located in the same directory as the HTML file into the page.

```
<img src="jojo.jpg">
```

See the full page here:
http://facweb.cdm.depaul.edu/asettle/csc243/web/thekids.html

Hyperlinks

HTML uses the anchor (<a> and) tag to create a link to another document.

An anchor can point to any resource on the Web, including an HTML page, an image, a sound file, a movie, etc.

The syntax for creating an anchor looks like:

```
<a href =
"http://facweb.cdm.depaul.edu/asettle/csc243/test.html">Test
Page</a>
```

This particular anchor uses what is called an **absolute URL**. This is the URL that you could type into a browser window in order to move to the page. (Note that it must include the http:// although some browsers don't require that information).

You can also create anchors using a different kind of reference:

```
<a href="test4.html">Go to test 4!</a>
```

This is what is called a **relative URL**. See the full example in **test2.html**.

It gives a URL relative to the current page. So the absolute URL for this would include http://, the server on which the HTML page in which it's embedded is located, and the path on the server to reach the directory in which the HTML file is located.

The anchor tag is also used to create links that can **send a mail message**, provided that the browser accessing the web page is connected to software that allows for the creation of e-mail messages:

```
<a href="mailto:asettle@cdm.depaul.edu"> Click here to e-
mail Amber</a>
```

See the full example in **test2.html** and another example in **test3.html**.

Web clients and the urllib module

Users use browsers to access web pages on the W3. Any program may act as a client and access and download web pages.

To do this in Python, you would use the **urllib.request** module.

It contains functions and classes that are used to open URLs in a way similar to how files are opened.

A Python program can thus be an HTTP client and request and receive resources on the web.

In particular, the **urllib.request** method **urlopen**() function is similar to the built-in function open(), but accepts universal resource locators (URLs) instead of file names:

```
>>> from urllib.request import urlopen
>>> response = urlopen('http://www.google.com')
>>> type(response)
<class 'http.client.HTTPResponse'>
>>>
```

A file-like object is returned. It supports the methods **read**(), **getheaders**(), **geturl**() and many others.

For example:

```
>>> response.getheaders()
[('Date', 'Mon, 07 May 2012 19:48:04 GMT'), ('Expires', '-
1'), ('Cache-Control', 'private, max-age=0'), ('Content-\
…
 'SAMEORIGIN'), ('Connection', 'close')]
>>> html = response.read()
>>> type(html)
```

```
<class 'bytes'>
>>> response.geturl()
'http://www.google.com'
>>> html = html.decode()
>>> type(html)
<class 'str'>
>>> html.count('google')
87
```

Another useful method in the **urllib.request** module is **urlretrieve(url, filename)**.

It copies a network object denoted by a URL to a local file filename.

```
>>> from urllib.request import urlretrieve
>>> urlretrieve('http://www.google.com',
'google.txt')
('google.txt', <http.client.HTTPMessage object at
0x02478830>)
```

In the file **url.py** is a program that opens the URL for a specified page and returns a string that corresponds to the contents.

Let's step through what that program is doing.

Parsing HTML files

The goal of this portion of the class is to learn to mine web pages for information.

To do that we need tools to automatically parse HTML web pages. The Python module **html.parser** provides classes that make it easy to parse HTML files.

The class **HTMLParser** from this module parses HTML files as follows:

1. The HTMLParser class is instantiated without arguments.
2. An HTMLParser instance is fed HTML data and calls handler functions when tags begin and end.

For example:

```
>>> url =
'http://facweb.cdm.depaul.edu/asettle/csc243/web/test
.html'
>>> from urllib.request import urlopen
>>> content = urlopen(url).read().decode()
>>> parser = HTMLParser()
>>> parser.feed(content)
>>>
```

Why did nothing happen?

The handler methods of the HTMLParser class are meant to be overridden by the user to provide a desired behavior.

Until we write definitions for the methods, those methods won't have any behaviors.

Some of the **handler methods** we can override include:

- **handle_starttag(tag, attrs)**: Start tag handler
- **handle_endtag(tag)**: End tag handler
- **handle_data(data)**: Arbitrary text data handler

For example, **handle_starttag**() would be invoked when an opening tag (i.e. something of the form <**tag attrs**> is encountered.

Note that the attributes are contained in a list (attrs) where each attribute in the list is represented by a tuple storing the name and value of the attribute.

handle_endtag() would be invoked when a closing tag (i.e. <**/tag**>) is encountered.

For an example see the **htmlParser.py** file.

In that file a subclass of the HTMLParser class is created that **overrides the handle_starttag() and handle_endtag() methods** so that they print information about the fact that the tag was encountered.

An example of using it looks like:

```
>>> url =
'http://facweb.cdm.depaul.edu/asettle/csc243/we
b/test.html'
>>> from urllib.request import urlopen
>>> content = urlopen(url).read().decode()
>>> parser = MyHTMLParser()
>>> parser.feed(content)
Encountered a html start tag
Encountered a head start tag
Encountered a title start tag
Encountered a title end tag
Encountered a head end tag
Encountered a body start tag
Encountered a h2 start tag
Encountered a h2 end tag
Encountered a p start tag
Encountered a br start tag
Encountered a p end tag
Encountered a h1 start tag
Encountered a h1 end tag
Encountered a p start tag
Encountered a br start tag
Encountered a p end tag
```

```
Encountered a body end tag
Encountered a html end tag
>>>
```

Recall what the **test.html** file looks like.

Exercises

To understand how to override the functions of the HTMLParser class, let's work through a series of exercises.

Exercise 1: Write a parser class **LinksParser** that prints to the screen the links found in the web page provided to it.

For example, it would produce this:

```
>>>
testParser('http://facweb.cdm.depaul.edu/asettle/csc2
43/web/test1.html')
http://www.cnn.com
test3.html
mailto:somePERSON@university.ca
>>>
testParser('http://facweb.cdm.depaul.edu/asettle/csc2
43/web/test2.html')
mailto:asettle@cdm.depaul.edu
test4.html
>>>
testParser('http://facweb.cdm.depaul.edu/asettle/csc2
43/web/test3.html')
mailto:nobody@xyz.com
test4.html
http://www.uchicago.edu
```