INTRODUCTION TO PROGRAMMING
WEEK 7 - MODULES

MODULES

Another name for a python file is a module. Python modules are self-contained python files that can be imported into each other to aid in execution. The concept of a module helps to further encapsulate and organize code into its relevant pieces. Try this - create a python file called `first.py` and in the same directory, create another python file called `second.py`.

Here are the contents of `first.py`:

```
import second

print 'This is first.py speaking!'
```

Here are the contents of `second.py`:

```
print 'This is second.py speaking!'
```

Now, try running `first.py`. The output you should get is:

```
This is second.py speaking!
This is first.py speaking!
```

Import Statement

From the example above, you can see that the `import` statement brings the code from one module into another. When `first.py` executed the line `import second`, it brought in `second.py` and essentially executed it within `first.py`. While this may not seem immediately useful, it can be useful when trying to access the functions and classes of another module.

Try this - create a python file called `main.py` and in the same directory, create another python file called `converters.py`. Here are the contents of `main.py`:

```
import converters

feet = converters.YardsToFeet(12)
print str(feet) + ' feet'
```

Here are the contents of `converters.py`:

```
def YardsToFeet(yards):
  feet = yards * 3
```

```
  return feet
```

Now, run `main.py`. The output you should get is:

```
36 feet
```

The `YardsToFeet` function in `converters.py` was successfully imported into the `main.py` module and used to convert a value.

Dot Operator

In the examples above, we saw the use of our old friend, the Dot Operator. Again, the Dot Operator specifies that the property or function right of the Dot belongs to the object left of the Dot.

```
converters.YardsToFeet(12)
```

In `main.py`, we imported the module `converters.py` and this module object was (implicitly) assigned to the variable `converters`. Now that we imported `converters`, we can use its functions in `main.py` without any problem.

Importing and using classes from modules

Now create a file called `vehicles.py` and inside `vehicles.py`, add the following:

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

  def GetNumDoors(self):
    return '4'

  def GetHorsePower(self):
    return 'probably between 100 - 150'
```

Now inside `main.py`, add the following code:

```
import vehicles

my_sedan = vehicles.Sedan('brown bruiser', 'brown')
print my_sedan.GetNumDoors() # '4'
```

When you run `main.py`, you see that it behaves similar to if the class definition of `Sedan` were located in `main.py` itself. The same syntax can be used for inheritance as well. Alter the code in `main.py` to this:

```
import vehicles

class Honda(vehicles.Sedan):
  def GetHorsePower(self):
    return 'probably between 120 and 160'

my_honda = Honda('rice rocket', 'green')
print my_honda.GetNumDoors() # '4'
print my_honda.GetHorsePower() # 'probably between 120 and 160'
```

In this case, we are using `vehicles.py`'s `Sedan` class as the superclass to the `Honda` class.

Renaming imported modules

Sometimes a module name is too long or uses a word that you would rather use for something else. In these cases, use the **`as`** operator to assign the module import to a variable.

```
import converters as cv

feet = cv.YardsToFeet(12)
print str(feet) + ' feet'
```

Importing modules from sub-directories

While importing modules is useful for encapsulation, oftentimes we want to group related modules together. For instance, in one of our previous example, what if we created a module called `cars.py` and put the classes `Sedan` and `Honda` in it, while also creating a module called `motorcycles.py` and putting two classes, `Harley` and `Kawasaki` in there? Of course we would want to relate cars and motorcycles, as they are both Vehicles. The best way to do this would be to create a directory named Vehicles and to put both `cars.py` and `motorcycles.py` in it.

To import modules from subdirectories, we have to follow 3 steps:

1. Create a directory with an empty `__init__.py` file in it
2. Add relevant modules to directory
3. Import the modules with a relative path (explained below)

- Let's try this. Please do the following:

1) Create a `main.py` file
1) In the same directory as `main.py`, create a directory called **utilities**
2) Inside the utilities directory, create an empty python file named **__init__.py**.
3) Inside the utilities directory, create a python file called **`converters.py`**
4) Inside **`converters.py`** provide the following content:

```
# converters.py

def YardsToFeet(yards):
  return (yards * 3)
```

Now, if we wanted to call the `YardsToFeet()` function from `main.py` (one directory level up), how would we do that? There are a number of solutions, below are the possible contents of main.py (all of these work):

1) Use the **`from`** clause to specify the directory from which the module is being imported:

```
from utilities import converter

print converter.YardsToFeet(8)
```

2) Use the **`from`** clause to specify the directory, and **rename** the imported module as something different:

```
from utilities import converter as cv

print cv.YardsToFeet(8)
```

3) Specify the **relative path** of the module, using the **dot-operator** to represent the directory structure. (Please don't use this convention...it can be verbose)

```
import utilities.converter

print utilities.converter.YardsToFeet(8)
```

4) Specify the **relative path** of the module, using the **dot-operator** to represent the directory structure, and **rename** the imported module as something different:

```
import utilities.converter as cv

print cv.YardsToFeet(8)
```

__NAME__ ATTRIBUTE AND MAIN FUNCTION

Up to now, we have just been putting the executable parts of a python script un-encapsulated in the body of the document. While this works just fine, it is not ideal, especially when working with multi-module scripts. Remember our example from the beginning of the chapter?

Running `first.py` prints:

```
This is second.py speaking!
This is first.py speaking!
```

The loose, executable code in `second.py` was executed in the body of `first.py`. While this may work in some cases, in other cases we might want to import something from `second.py` without executing its program body. Often with multi-module development, we want a module to be **importable** while also **independently executable** but we don't want these two features to mix.

The __name__ attribute

Every python module has a `__name__` attribute that takes on the value '**`__main__`**' when the module is run by itself (and not imported). Alter the contents of `first.py` to this:

```
if __name__ == '__main__':
  print 'This is first.py speaking!'
```

Alter the contents of second.py to this:

```
if __name__ == '__main__':
  print 'This is second.py speaking!'
else:
  print 'This is second.py speaking after being imported!'
```

Try running `first.py` and `second.py` - What is the output?

Running `first.py`:

```
This is second.py speaking after being imported!
This is first.py speaking!
```

Running `second.py`:

```
This is second.py speaking!
```

By using the `__name__` attribute we can define a module's behavior when its run by itself and its behavior when it is imported. This is an important separation for us.

The main() method

From the section above, we can see that by using the statement:

```
if __name__ == '__main__':
```

We can put all the independently executable code (non-imported) of a module under this if statement. However, proper python convention stipulates that we should put that code in a `main()` function of the module instead of listing it all under the `if __name__` statement.

So as a part of proper python convention, from here on out - we will incorporate the `__name__` attribute into every module and also define all of a module's independently executable code inside a **`main()`** function instead of leaving it just laying strewn about the body of the document.

If we need to write a script that prints "hello world!", we would have done this in the past:

```
print 'hello world'
```

Now, the program should look like this:

```
def main():
 print 'hello world!'

if __name__ == '__main__':
 main()
```

Please incorporate this format into all your python scripts/modules from now on. It is a good habit that allows for portable multi-module development, testing of python scripts, and is a part of the Google Python Style Guide.

If we wanted to build some primitive test cases for `converters.py` that execute only when the module is run by itself we could do this:

```
# converters.py

def YardsToFeet(yards):
  return yards * 3

def main():
  if YardsToFeet(12) != 36:
    print 'YardsToFeet failed at 12'
  else:
    print 'YardsToFeet passed at 12'

if __name__ == '__main__':
  main()
```

From here on out, all the executable code in your python modules need to be in a `main()` function called from `__name__ == '__main__'` as shown above. This applies even if the module isn't to be imported anywhere else.

DEFAULT PYTHON MODULES

In addition to the modules you create and import yourself, Python provides many default modules that you can utilize for useful functions. You can find a full list of them here:

http://docs.python.org/library/index.html

Here are some that we will cover in class:

- **`csv`** - for reading/parsing csv files
- **`datetime`** - for creating date/time objects that can be compared or validated
- **`os`** - finding and manipulating things in the file structure
- **`random`** - generating random numbers

The reason that these modules need to be imported to be used is to reduce run-time. If every time a python script had to load, it also had to load 10000 lines of modules to handle random numbers, dates, etc. then that would really slow the script down.

This week, we'll be looking into 2 modules: `random` and `datetime`.

RANDOM

The random module allows you to generate random numbers which is extremely useful in creating unique ID numbers or experimental situations when a user needs to be randomly put into 1 of 2 or more groups. Let's take a look at some of the module's functions.

randint()

The `randint()` function takes 2 parameters, a starting integer (a) and an end integer (b) and generates a random integer in between these two (`a <= N =< b`). The implementation is like this:

```
import random

my_int = random.randint(1, 10) # random int from 1 to 10
```

random()

The `random()` function generates a random floating point number between 0 and 1 (e.g. .3967230013). This is a great one for generating percentages.

```
import random

my_float = random.random() # random float from 0 to 1
```

What's a floating point number? How do you work with them? See my write up here.

choice()

The `choice()` function allows you to randomly choose an element out of a list.

```
import random
```

```
names = ['albert', 'satish', 'saurabh']
winner = random.choice(names)

print winner + ' just won 1 million rupeees!'
```

DATETIME

The `datetime` module is a great module for working with dates. This is a very rich module and we'll only be taking a look at its most basic classes and functions.

date objects

To work with dates, you want to first create a date object. The date object constructor **takes 3 integer parameters**, a year, a month, and a day like so:

```
datetime.date(<year>, <month>, <day>)
```

Once created, the date object has year, month, and day properties that echo back that date's year, month, and date properties. Here's an example:

```
import datetime

birthday = datetime.date(2000, 12, 20)
print birthday.year # 2000
print birthday.month # 12
print birthday.day # 20
```

today()

The most commonly used method of date objects is today() which returns a date object for today (this is taken from the computer that you're running your script off of).

```
today = datetime.date.today()

print today.year # 2011
print today.month # 2
print today.day # 23
```

weekday()

One common operation with `date` objects is to get the specific day of the week (Monday - Sunday) that the date falls on. To do that just use the `weekday()` method.

The `weekday()` method returns an integer between 0 and 6 where Monday is 0 and Sunday is 6.

```
import datetime

class_date = datetime.date(2011, 2, 23)
print class_date.weekday() # 2
```

Adding/Subtracting time to dates

Often, you'll want to add or subtract time to/from a date and do operations on the resulting `date` object. To do this, you'll need to create an object called a `timedelta` which represents a change in time.

The `timedelta` object's constructor takes 7 optional parameters - weeks, days, hours, minutes, seconds, microseconds, and milliseconds. For our purposes, we will be worried mostly with days and weeks. Here is some sample code:

```
import datetime

thirty_days = datetime.timedelta(days=30)
my_date = datetime.date(2011, 2, 23)
new_date = my_date + thirty_days

print new_date.year # 2011
print new_date.month # 3
print new_date.day # 25
print new_date.weekday() # 4
```

Similarly, creating a `timedelta` object also could have been done with any of the other optional parameters:

```
ten_hours = datetime.timedelta(hours=10)
three_weeks = datetime.timedelta(weeks=3)
```

getting the difference between 2 days

Another common operation is to get the time that elapsed between 2 `dates`. To do this, simply subtract the two `date` objects and you will get a `timedelta` object which you can manipulate.

```
import datetime

date1 = datetime.date(2011, 5, 13)
date2 = datetime.date(1982, 11, 24)
```

```
date_diff = date1 - date2

print date_diff.days # 10397
```

comparing dates/timedeltas

To compare date objects and timedelta objects, just use the regular inequality operators that we have been using (>, <, =, !=, etc.)

```
import datetime

date1 = datetime.date(2011, 5, 13)
date2 = datetime.date(1982, 11, 24)

date_diff = date1 - date2
two_decades = datetime.timedelta(weeks=(52*10*2))

if date1 > date2:
  print 'date1 is later than date2!'

if date_diff > two_decades:
  print 'more than 2 decades passed btw date1 and date2'
```

datetime objects

All the same functionality (and more!) is also available with `datetime` objects which allow you to create date objects with more information (including the hour, minute, and second) of an event.

The constructor for `datetime` objects take the form:

```
datetime.datetime(<year>, <month>, <day>, <hour>, <minute>, <second>)
```

Example:

```
import datetime

birth_time = datetime.datetime(1986, 4, 29, 14, 15, 0)

print birth_time.weekday() # 1
print birth_time.hour # 14
print birth_time.minute # 15
print birth_time.second # 0
```

now()

A handy function of the datetime module is the now() function which generates a datetime object for right now (this is taken from the computer you are running python off of):

```
import datetime

this_moment = datetime.datetime.now()
print this_moment.hour
```

Creating datetime objects from strings

One way to create date or datetime objects is to take a string and do a split on the substring that separates the year, month, and day (‘/’ in the case of ‘4/28/2008’ and ‘-’ in the case of ‘1996-7-24’). However, the datetime object has a handy function called strptime (string parse time).

To use strptime, you will need 2 things - a string to parse and a string pattern for how to parse it.

The pattern for parsing can look a little confusing at first, but here are the critical template values:

%m - month (1 - 12)
%d - day of the month (1 - 31)
%Y - year
%H - hour (0 - 23)
%M - minute (0 - 59)
%S - second (0 - 59)

Using the template values, you construct a string pattern that will match the particular date string you want to create a `datetime` object out of. For instance:

```
import datetime

birthday = datetime.datetime.strptime(‘2/23/2006’, ‘%m/%d/%Y’)
print birthday.weekday() # 3

registration_time = datetime.datetime.strptime(‘1/1/1995 5:01:04’,
                                               ‘%m/%d/%Y %H:%M:%S’)
print registration_time.weekday() # 6

dash_date = datetime.datetime.strptime(‘2000-03-15’, ‘%Y-%m-%d’)
print dash_date.weekday() # 2
```

Keep in mind that this method **is only available for datetime objects**, but you can convert `datetime` objects to `date` objects with your own custom function. Can you figure out how?