Project 6

Background

The acronym API is short for "Application Programming Interface." It is usually a collection of functions, methods or classes that supports the interaction of an application program (the program you write as a developer) with other programs that run on a remote server. Google has several API's available from its iGoogle (http://www.google.com/ig) website for personal queries, including weather, movies, news, etc. It doesn't advertise how you can use them from other applications, but they are there for anyone to use. To interact with iGoogle, a fairly simple web API has been developed. Our goal is to create a user class that, when instantiated, has local values to be used in a personal query. The instance will query the iGoogle API's for user-specified information as needed, and report back some useful information.

The API

We will be using the map API for purposes of finding distances between cities in the U.S. As with all iGoogle API's, it is based on creating a query using the http protocol of the web. Essentially, you send a web address (with the query embedded in the web address) and a web page is returned with the requested information, albeit in a rather odd form. Thus, you have to create a particular web query (a particular kind of web page request) to get the information back. The nice part is that you can do the queries on the web (to see the results) and can then do it in Python (discussed below).

The general form of a distance query is:

`http://maps.googleapis.com/maps/api/distancematrix/`*`json`*`?`*`parameters`*

where parameters denotes a series of parameter. The parameters we will use are:

- origins: the origin city and state[1]
- destinations: the destination city and state
- sensor: false, which tells the server that your application does not use a GPS locator
- mode: one of 'driving', 'bicycling', or 'walking'

There are no spaces in the query. Parameter assignments are separated using the ampersand (&) symbol and words in the city are separated using the plus (+) sign.

Some examples are listed next. Copy and paste each one into a browser to see what is returned. Then create your own examples to see how changing parameter values changes the response you get back. What happens, for instance, if you make up some city names and/or state abbreviations? Or if the cities are not connected by roadways?

http://maps.googleapis.com/maps/api/distancematrix/json?origins=New+York+NY&destinations=Lansing+MI&mode=driving&sensor=false

[1] The parameter names are plural because you can request a distance matrix, indicating distances between a set of potential origins and a set of potential destinations. However, we will just use a single city for each.

http://maps.googleapis.com/maps/api/distancematrix/json?origins=Lansing+MI&destinations=Sacramento+CA&mode=bicycling&sensor=false

You can find additional information about the Google map API at https://developers.google.com/maps/documentation/webservices/

Parsing the Result

As distance calculation is part of the map API, and not its own API, the entire result is returned as a single string, which must be parsed. The string returned by the first query above, which finds the driving distance between New York, NY and Lansing, MI, is as follows:

{ "destination_addresses" : ["Lansing, MI, USA"], "origin_addresses" : ["New York, NY, USA"], "rows" : [{ "elements" : [{ "distance" : { "text" : "1,092 km", "value" : 1091822 }, "duration" : { "text" : "10 hours 11 mins", "value" : 36652 }, "status" : "OK" }] }], "status" : "OK" }

You will need to parse this string to extract the information required. In this response, you will need to extract the value 1091822, which is the number of meters from Lansing to New York.

In testing queries, you should have discovered that the server "guesses" a location if it cannot exactly match the values indicated for the cities. Your program does not need to check that the response is for the correct location—just assume that the distance returned in a response is correct. But your program *does* need to handle the situation where the response does not contain a distance. (See below.)

Querying the web from Python

The Python library `urllib.request` has tools to create a connection to a web server and download data from the web server as if it were a file. The function to create a connection is `urlopen`. Like any file, you open it, read the contents, and then close it. Here is an example:

```
import urllib.request
web_obj =
urllib.request.urlopen("http://maps.googleapis.com/maps/api/distancematrix/json?origins=New+York+NY&destinations=Lansing+MI&mode=driving&sensor=false")
results_str = str(web_obj.read())
web_obj.close()
```

It is important to note that the result of `read()` is not a *string*, but an *array of bytes*. It must therefore be converted to a string, or you will not be able to do anything with it. Once it is converted, you will be able to perform normal string methods on the result in order to extract the information desired. You can use this approach to confirm that the response indicates a distance and to extract the value (as an int) associated with the distance.

Project Description / Specification

Create a class called `Tour`. An instance of this class is to be instantiated with an arbitrary number of US cities, and is used to fetch information from the web. Specifically, the class must define:

1. `__init__` The constructor takes zero or more strings as arguments, each giving a city name and state abbreviation, and indicating a destination along a tour of US cities. For example: `Tour("New York, NY", "Lansing, MI", "Los Angeles, CA")` represents a tour that starts in New York city, proceeds to Lansing, and ends in Los Angeles.
2. `distance` This method takes a single (optional) argument indicating a mode – one of the strings `'driving'` (default), `'bicycling'`, or `'walking'`. It returns the total distance (in meters) covered by the tour for the indicated mode. This method is where you will use urllib functions to get data from the web to find the distances between consecutive locations in the tour and calculate the total distance. If a response does not contain a distance value, the method should raise a `ValueError` exception.
3. The following operations must be implemented. All methods except those in (d) should handle operands of type `Tour`. Multiplication is defined for a Tour and a non-negative integer only. Thus, for example, if `tour1` is of type `Tour`, you must handle both `tour1*2` and `3*tour1`.
   a. `__str__` and `__repr__`: Return as a string the cities to be visited by this tour, where the cities are listed in order, separated by a semicolon (`;`), and formatted the same as the arguments provided at construction of this tour.
   b. `__add__` : Concatenate a tour to the end of this tour.
   c. `__mul__` and `__rmul__`: Repeated concatenation of this tour, where the single argument indicates the number of times to cycle through the cities and must therefore be a non-negative integer. Raises a `TypeError` if the argument is not an `int` and a `ValueError` if it is a negative `int`.
   d. `__gt__` : Compare the driving distance of this tour to that of another tour; return `True` if the driving distance for this tour is greater than that for the other, and `False`, otherwise.
   e. `__lt__` : Compare the driving distance of this tour to that of another tour; return `True` if the driving distance for this tour is less than that for the other, and `False`, otherwise.
   f. `__eq__`: Compare this tour to another tour for equality, where the tours are considered equal if they visit precisely the same cities in the same order; for simplicity, consider two cities in different tours the same only if the strings given for construction of the tours are equal.
4. Include a `main` function that tests all (!) of your methods. See the partial `main` function below that you can use as a starting point.

Example main program to demonstrate that your class works (this example is incomplete because it doesn't test all the methods—you need to add tests to make it complete):

```
def main():
  t1 = Tour("New York, NY", "Lansing, MI", "Sacramento, CA")
  t2 = Tour("Oakland, CA")
  t3 = Tour("Sacramento, CA", "Oakland, CA")

  print("t1: {}¥nt2:{}¥nt3:{}".format(t1,t2,t3))
  print("t1 distances: driving-{} km; biking-{} km; walking-{} km".format(
    round(t1.distance()/1000), round(t1.distance('bicycling')/1000),
    round(t1.distance('walking')/1000)))
  print("Using driving distances from here on.")
  t4 = t1 + t2
  print("t4:", t4)
  print("t4 driving distance:", round(t4.distance()/1000),"km")
  print("t4 == t1 + t2:", t4 == t1 + t2)
```

Example output from running the above main program:

```
>>> main()
t1: New York, NY; Lansing, MI; Sacramento, CA
t1 distances: driving-4706 km; biking-5149 km; walking-4608 km
Using driving distances from here on.
t4: New York, NY; Lansing, MI; Sacramento, CA; Oakland, CA
t4 driving distance: 4836 km
t4 == t1 + t2: True
```

Notes and Hints:

1. Python provides the `raise` statement to force an exception to occur. For example:
   ```
   >>> raise NameError('Message describing the reason')
   Traceback (most recent call last):
     File "<stdin>", line 1, in ?
   NameError: Message describing the reason
   ```
   You will need to use this statement in the methods that need to raise exceptions.
2. Python allows you to define a function that takes a variable number of arguments: the final parameter in a function definition, if preceded by a star (*) symbol, indicates that the parameter is assigned a tuple of all remaining arguments. See your textbook for details and examples. You will need to use this in defining the constructor.
3. Enter the example URLs in a browser to see the web pages that result from the queries. Experiment with changing the parameter values. These experiments give you a good idea of what responses you will need to parse.

4. Experiment in the Python shell with the `urllib.request` functions (as shown above) to fetch a URL and see what you get.
5. Experiment with one such fetch and see how to parse the response to extract the required information.
6. Use various string functions to break the response down, bit by bit, until it is in a form you can easily work with.
7. Start writing the easiest methods: `__init__` and `__str__`. Test them before moving on, e.g. test the first 4 lines of the example main program.
8. Next work on the `distance` method using what you learned from steps 1-4. Test it using the first 7 lines from the example main program.
9. Next work on the `__add__` method. Think about this one! Test it using the first 9 lines of the example main program.
10. Next work on the `__eq__` method. You can now test the whole example main program.

Continue implementing and testing the remaining operations. Don't forget to test the exceptions, too.