Due Saturday April 30th, by 11:59:00pm

NOTE: no late submissions allowed on this project! Make the deadline, plan ahead.
If you have any tokens left, they will each count as 0.25pts extra towards semester total.

- The purpose of this assignment is to explore creating classes, making objects, and generally using the abstraction that comes with the OO approach. We will also create our own exception type and raise it at appropriate places.
- You will turn in a single python file following our naming convention: `gmason76_2B5_P6.py`
- Include the following in comments at the top of your file: Name, G#, lecture and lab sections, and any references or comments we ought to know. You don't have to use the template from previous projects.
- Use the Piazza discussion forums (and professor/TA office hours) to obtain assistance.
- Remember, do not publicly post code for assignments on the forum! Use private posts as needed (or when you're not sure). Have a specific question ready, and both show what you're thinking and show what you've tried independently before you got stuck. We will prod for more details if needed.

Background:

Classes allow us to create our own datatypes with their own rich structure and functionality. If the built-in types such as lists, tuples, dictionaries, sets, and so on don't seem to capture the exact structure you want, or if it leaves things perilously unnamed, then class definitions are the way to go. We get to define what attributes (interior values) each object must have, as well as the functionality that must be available over these objects. Classes let us model the problem as we see it, rather than having to just do the bookkeeping with built-in types and keep all the abstractions buried in our minds.

Exceptions allow us to represent unusable states of a program, decide where to deal with the problem, and how to recover from them. Anything from files not being present to inconsistent states of data (e.g., birthday=="1/1/2001" and age=30) can be represented as an exception, raised, and then we can force the program to acknowledge the situation and provide a solution, or perish. As extra credit for this project, we will create our own exception type and raise it.

Grab http://cs.gmu.edu/~yzhong/112/p6/tester6p.py for testing.

What's allowed?

Here is the exhaustive list of things you can use on the project.

- all basic expressions/operators, indexing/slicing
- all basic statements: assignment, selection, and loop statements, break/continue, return.
- dictionaries, sets, tuples, and their methods. I don't expect these will be useful; just make sure you still obey the types listed for parameters, attributes, and return values.
- functions: len(), range(), abs()
- lists: `insert(), remove(), append(), extend(), pop()`
- string methods including formatting
- classes, exceptions and their methods.

This means that…

- you can't call anything not listed above. Focus on applying these functions to solve the task.
- you can't import any modules for this project

Task

Our goal is to create the infrastructure for a Global Positioning System (GPS) application that can store a map as a list of points of interest, search the map, and set a route. In order to do this, we will create classes that represent a GPS location, a GPS POI (point of interest), and a GPS. We won't actually be creating the full application, as that would be too large a project – but you could imagine storing the map data to a text file, creating a GUI to show it to the user and accept commands to search and route in the map.

You will complete the definitions as described below; you can test your code with testing file:

- Get the testing file: http://cs.gmu.edu/~yzhong/112/p6/tester6p.py

- Invoke it similar to prior assignments: `python3 testerfile.py yourcode.py just these things`
    - note: you can only give specific `things` as the targeted tests, generally either a class name or a method name, but with certain restrictions:
        - class names: `GPS_Location`, `GPS_POI`, `GPS` (only checks a few general things in `GPS` class)
        - methods for `GPS`: `add_dest, relocate, display_map, search_name_kind, search_within_dist, closest_kind` (only works after you have a working `GPS constructor`)
        - more methods for `GPS`: `display_route, arrive_first, drop_dest, dist_to_travel` (only works after you have a working `add_dest`)
        - `extra_credit` (only checks the extra credit tests involving `GPS_Error`)

Classes

Note:

- Non-constructor methods should return `None` unless specified otherwise.
- We use notation `name::type` to describe variables and parameters. For example, `x::int` specifies that we need a variable/parameter named `x` and the expected type of `x` is `int` (integer).
- The class definitions are built upon each other. Make sure you define them in the order as given below.

`class GPS_Location:` This class represents one location in a 2D space and deals with its representation.

- **Instance variables**:
    - `x::int`. One non-negative integer representing horizontal coordinate.
    - `y::int`. A second non-negative integer representing vertical coordinate.
- **Methods**:
    - `__init__(self,x,y)`: constructor; must store the `x` parameter into the `x` attribute and `y` parameter into the `y` attribute. You can assume the parameters are always non-negative integers unless you are attempting extra credit. **Extra credit:** if `x` or `y` is negative, raise a `GPS_Error` with the message `"GPS_Location error: coordinate cannot be negative"`. See "**Extra credit**" for the details of `GPS_Error` class.
    - `__str__(self)`: returns a string representation of the coordinates. For example, if `x==3`, `y==4`, then the returned string must be `"(3,4)"` (note: no space in the string).
    - `__repr__(self)`: returns a computer-centric view of a `GPS_Location`. Specifically, it's a string whose contents looks like a constructor call that would recreate the object. Returns a string such as `"GPS_Location(3,4)"` (note: no space in the string).
    - `__eq__(self,other)`: compares to see whether two locations (`self` and another `GPS_Location` object named as `other`) are identical. We should compare the two pairs of coordinate values, return `True` if both match and return `False` otherwise.

- `dist(self,other)`: calculates and returns Manhattan distance between two locations (`self` and another `GPS_Location` object named as `other`). You can find the definition of Manhattan distance here: http://xlinux.nist.gov/dads//HTML/manhattanDistance.html. Example: Manhattan distance between `(1,2)` and `(3,4)` is `(3-1)+(4-2) = 4`.

`class GPS_POI`: This class represents a "point of interest" in a map.

- **Instance variables**:
  - `location :: GPS_Location`. A `GPS_Location` object representing the coordinates of the POI.
  - `name :: str`. A string name of the point of interest, for example, `"Panera Bread"`.
  - `kind :: str`. A string name as the classification of the POI, for example, `"food"` or `"gas"`.
- **Methods**:
  - `__init__(self,location,name,kind)`: constructor; must accept arguments for the location, name, and kind. They must be stored to instance variables named `location`, `name` and `kind`.
  - `__str__(self)`: returns a human-centric string representation with the following format: the string must start with the `str()` representation of `location` followed by a colon, then `name` and `kind`, separated by a comma. Example: `"(3,4): Panera Bread, food"`(note: two single spaces after colon and comma in the string).
  - `__repr__(self)`: returns a computer-centric string looking like the constructor call which could recreate the object. See the "**Examples**" at the last page and test cases for an exact representation. Be sure to use GPI_Location's repr() definition!

`class GPS`: This class represents a GPS system, including a current location, a map as a list of POIs, and a route as a list of destination locations. We can display it, search the map based on different criterions, and set and update the route.

- **Instance variables**:
  - `current :: GPS_Location`. A `GPS_Location` object representing our current coordinates.
  - `map :: [GPS_POI]`. A list of `GPS_POI` objects.
  - `route :: [GPS_Location]`. A list of `GPS_Location` objects representing our destinations. Note that we assume the starting point of our route is always `current` although it is not included in the list.
- **Methods**:
  - `__init__(self, current, map=None)`: constructor. Must set the instance variables as:
    - `self.current`: set as `current` parameter
    - `self.map`: set as `map` parameter, or (when it is the **None** value) set it to be an empty list.
    - `self.route`: always set it to be an empty list (thus no need for a parameter). This also implies that we must start with an empty `route` and add destinations to it later.
    - **parameters**:
      - `current :: GPS_Location`
      - `map ::[GPS_POI]` (or, `map ::None`, optional).
  - `relocate(self,location)`: accepts a `GPS_Location` object `location` and change `current` attribute to be the this value.
  - `add_dest(self,location)`: accepts a `GPS_Location` object `location` and appends this value to the end of `route`.
  - `drop_dest(self,location)`: accepts a `GPS_Location` object `location` and remove the first occurrence of this value from `route`. You can assume there is at least one occurrence of `location` included in `route` unless you are attempting extra credit. **Extra credit:** if the specified `location` is not included in `route`, raise a `GPS_Error` with the message `'GPS drop_dest error: not in route'`.

- `arrive_first(self)` : simulates the operations to take when we arrive at the first destination in `route`: updates `current` to be the first destination location in `route`, and removes that value from `route`. You can assume `route` has at least one destination unless you are attempting extra credit. **Extra credit:** if `route` is currently empty, raise a `GPS_Error` with the message `'GPS arrive error: empty route'`.
- `display_map(self)`: returns a string representation of `map`: it should be a multi-line representation, each includes the `str()` representation of one POI value in `map` and ends with a newline. Return an empty string if `map` has no POIs. *See the examples and test cases for an exact representation.* Be sure to use GPS_POI's str() definition!
- `display_route(self)`: returns a string representation of `route`: it should start with the `str()` representation of `current`, followed the `str()` representation of all locations in `route`, each preceded by a dash(`'-'`). For example, '`(2,3)-(4,4)-(5,5)`'. Return an empty string if `route` has no locations. *See the examples and test cases for an exact representation.* Be sure to use GPS_Location's str() definition!
- `dist_to_travel(self)` : calculates and returns the total Manhattan distance to travel according to `current` and locations in `route`. Be sure to use GPS_Location's dist() definition! Return zero if `route` is empty.
  - **return value**: `int`
- `search_name_kind(self,name,kind)` : finds all point of interest values in `map` that match the supplied `name` and `kind` arguments, and returns them as a list.
  - **parameters**:
    - `name :: str`
    - `kind :: str`
  - **return value**: `[GPS_POI]` (a list of `GPS_POI` objects, could be empty).
- `search_within_dist(self,dist,kind=None)` : finds all point of interest values in `map` that match `kind` and are located within(<=) `dist` of `current` location, and returns them as a list. If `kind` is not specified, return point of interest values of any kind within the indicated distance. Be sure to use GPS_Location's dist() definition!
  - **parameters**:
    - `dist :: int`
    - `kind :: str`, optional
  - **return value**: `[GPS_POI]` (a list of `GPS_POI` objects, could be empty).
- `closest_kind(self,kind)` : finds all point of interest values in `map` that match `kind` and are closest to `current` location, and returns them as a list. The list might have one or more `GPI_POI` objects depending on whether there is a tie in the distance between the pois and `current`.
  - **parameters**:
    - `kind :: str`
  - **return value**: `[GPS_POI]` (a list of `GPS_POI` objects, could be empty).

Extra Credit (+5)

Create an Exception class to represent various abnormal situations in our GPS implementation. First, define our own Exception class:

`class GPS_Error(Exception)`: A subclass of `Exception` that represents any issue with our GPS.

- **Instance variable**:
  - `msg :: str`. The message describing what sort of GPS-related problem we've got.
- **Methods**:
  - `__init__(self,msg)`: constructor; must store the `msg` parameter into the `msg` attribute.

- `__str__(self)`: returns the `msg` attribute.
- `__repr__(self)`: returns the string looking like the constructor call that could recreate the object, such as `"GPS_Error('GPS_Location error: coordinate cannot be negative')"`. You need to use single-quotes inside the string to match the test cases.

Then augment your implementation of `GPS_Location` and `GPS` classes to raise `GPS_Error` exceptions for special cases. The required ones are described above, marked by **Extra credit:**. Keep the extra credit in mind when implementing the regular version, but it should be attainable to implement the extra credit portion entirely after the rest of the project.

Grading Rubric

```
Code passes shared tests:     90 (zero points for hard-coding)
Well-documented/submitted:     5
No extra globals used:         5
-------------------------------
TOTAL:                       100  +5 extra credit for Exception
```

Note

- You should always test your code not only with the provided testing script, but also by directly calling your functions. ***Just be sure to remove global variables and debug printings before turning in your work – globals are not allowed in your final submission.*** Consider the file below on the left, named `shouter.py`, which you can run as shown below on the right using interactive mode (-i).

```
def shout(msg):
     print(msg.upper())

mystring1 = "hello"
mystring2 = "another one"
```

```
demo$ python3 -i shouter.py
>>> shout("i wrote this")
'I WROTE THIS'
>>> shout(mystring1)
'HELLO'
>>> shout(mystring2)
'ANOTHER ONE'
```

Reminders on Turning In Projects

For this last project, no late work is accepted! We do drop the lowest project grade, so if you miss the deadline it doesn't have to be a semester-ruining event, but please note the difference on this last project.

You can turn in your code as many times as you want; we will only grade the last submission that is on time. If you are getting perilously close to the deadline, it may be worth it to turn in an "almost-done" version about 30 minutes before the clock strikes midnight just to be sure you get something turned in.

You can (and should) check your submitted files. If you re-visit BlackBoard and navigate to your submission, you can double-check that you actually submitted a file (it's possible to skip that crucial step and turn in a no-files submission!), you can re-download that file, and then you can re-test that file to make sure you turned in the version you intended to turn in. It is your responsibility to turn in the correct file, on time, to the correct assignment.

Use a backup service. Do future you an enormous favor, and just keep all of your code in some automatically synced location, such as a Dropbox or Google Drive folder. Every semester someone's computer is lost/drowned/dead, or their USB drive is lost/destroyed, or their hard drive busts. Don't give these situations the chance to doom your project work!

Examples

```
>>> l1 = GPS_Location(1,2)
>>> str(l1)
'(1,2)'
>>> repr(l1)
'GPS_Location(1,2)'
>>> l2 = GPS_Location(1,2)
>>> l3 = GPS_Location(3,1)
>>> l1==l2
True
>>> l1==l3
False
>>> l1.dist(l3)
3
>>>
>>> p1 = GPS_POI(l3,"panera bread","food")
>>> str(p1)
'(3,1): panera bread, food'
>>> repr(p1)
"GPS_POI(GPS_Location(3,1),'panera
bread','food')"
>>>
>>> g = GPS(l1)
>>> g.map
[]
>>> g.display_map()
''
>>> g.route
[]
>>> g.display_route()
''
>>> g.current
GPS_Location(1,2)
>>> g.relocate(l3)
>>> g.current
GPS_Location(3,1)
>>> p2 = GPS_POI(GPS_Location(4,5),"red hot
blue","food")
>>> p3 =
GPS_POI(GPS_Location(3,3),"ups","service")
>>> g = GPS(l1,[p1,p2,p3])
>>> g.display_map()
'(3,1): panera bread, food\n(4,5): red hot
blue, food\n(3,3): ups, service\n '
>>> print(g.display_map())
(3,1): panera bread, food
(4,5): red hot blue, food
(3,3): ups, service

>>> g.search_name_kind("panera bread","food")
[GPS_POI(GPS_Location(3,1),'panera
bread','food')]
>>> g.search_within_dist(4,"food")
[GPS_POI(GPS_Location(3,1),'panera
bread','food')]
>>> g.search_within_dist(8,"food")
[GPS_POI(GPS_Location(3,1),'panera
bread','food'),
GPS_POI(GPS_Location(4,5),'red hot
blue','food')]
>>> g.search_within_dist(5)
[GPS_POI(GPS_Location(3,1),'panera
bread','food'),
GPS_POI(GPS_Location(3,3),'ups','service')]
>>>
>>> g.closest_kind("food")
[GPS_POI(GPS_Location(3,1),'panera
bread','food')]
>>> g.closest_kind("gas")
[]
>>>
>>> g.add_dest(p1.location)
>>> g.add_dest(GPS_Location(4,5))
>>> g.display_route()
'(1,2)-(3,1)-(4,5)'
>>> g.dist_to_travel()
8
>>> g.arrive_first()
>>> g.display_route()
'(3,1)-(4,5)'
>>> g.current
GPS_Location(3,1)
>>> g.drop_dest(GPS_Location(4,5))
>>> g.display_route()
''
>>>
>>> e = GPS_Error("something wrong")
>>> str(e)
'something wrong'
>>> repr(e)
"GPS_Error('something wrong')"
>>> raise e
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.GPS_Error: something wrong
>>>
>>> l = GPS_Location(-1,0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ... # some lines of msg omitted
__main__.GPS_Error: GPS_Location error:
coordinate cannot be negative
>>> g = GPS(l1)
>>> g.drop_dest(l1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
__main__.GPS_Error: GPS drop_dest error: not
in route
>>> g.arrive_first()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  ...
__main__.GPS_Error: GPS arrive error: empty
route
>>>
```

GPS_Location

GPS_POI

GPS

GPS with a map

GPS map searching

GPS route manipulation

Extra credit