Module 8 Homework - Global Temperatures

Motivation

Extreme weather events are expected to increase in frequency and intensity over the next century. Mapping *individual* events like hurricanes and heat waves to a single root cause is nearly impossible due to the chaotic nature of earth's weather, but we can find surprisingly consistent results when we observe *trends* across the globe.

To assist with research in this area, the World Meteorological Organization (WMO) has hired you to build a research database. Their main concern is that accessing and updating information be rapid - we'll aim for O(1). However, they also have limited hard drive space, so we need to keep memory constrained to O(n).

We'll solve this by using a hash-table mapping. Some important aspects of our solution:

- Our keys will be 2-tuples representing (latitude, longitude). Python supports hashing for tuples, and we should use this to hash our keys (e.g. `hash(position)` instead of `hash(latitude) + hash(longitude)`).
- Each key will map to two values - the maximum and minimum temperature reported for that position (`localrecord.max` and `localrecord.min`)
- Use test-driven development. Write a unittest, implement functionality until it passes, and repeat.

  The examples in this pdf are helpful for understanding what you should do, but you should not use them as your tests - come up with your own numbers, and make sure to test that everything works with a large number of randomly generated inputs. **You should not use the built-in set or dictionary classes when implementing functionality**, but feel free to use them in `TestRecordsMap.py` to help with testing.

Weather Report Locations

Incoming temperature reports from weather satelites have precise geo-location data (latitude and longitude). However, if two points are fairly close to each other, it doesn't really make sense to store separate records for them. We want to instead group nearby points together.

We'll treat any points that round to the same latitude and longitude as the same. In practice, this means we're breaking the planet into roughly 70-mile squares (lattitude and longitude lines are ~70 miles apart).

```
>>> p1 = (41.8067, -72.2522) # 4 decimal point precision (lat, long).
>>> p2 = (41.8097, -72.1473) # close enough to p1
>>> p3 = (41.3005, -72.4533) # latitude is a bit too far south
>>> round(p1[0], 0) == round(p2[0], 0) #  42 == 42
True
>>> round(p1[1], 0) == round(p2[1], 0) # -72 == -72
True
>>> round(p1[0], 0) == round(p3[0], 0) #  42 == 41
False
```

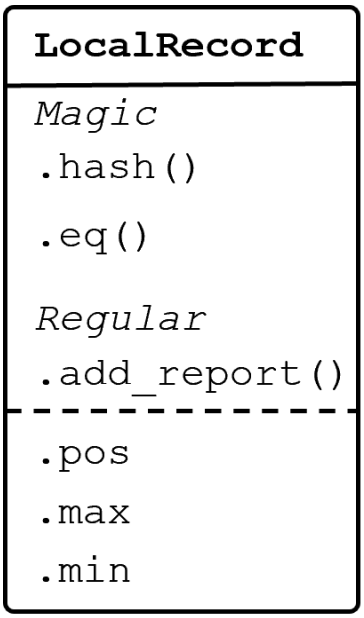
```
>>> round(p1[1], 0) == round(p3[1], 0) # -72 == -72
True
```

In the example above, `p1` and `p2` to corresond to *the same* position, while `p3` corresponds to a *different* position. `p3` is on roughly the same longitude, but a bit too far south to be considered the same position.

Use python's `round()` method for rounding in this assignment.

Part 1 - `class LocalRecord`

Write a data structure `LocalRecord` for storing record temperatures at single grid point. A record needs to support at minimum the methods and variables shown below:

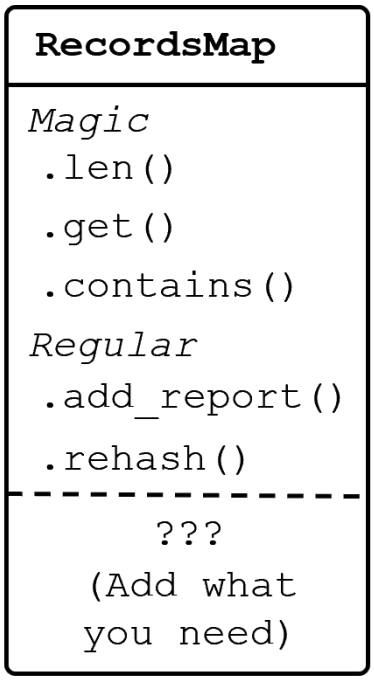


- `hash()` - returns a hash for this object based on its position
- `eq()` - returns `True` iff two records are for the same position
- `add_report(temp)` - Updates `max` and `min` if appropriate
- `pos` - a tuple representing the latitude and longitude of this record. Incoming weather reports will have arbitrary precision, but any that round to the same (lat, long) should be equal, so you'll need to round before storing.
- `max` - maximum temperature reported for this position
- `min` - minimum temperature reported for this position

Go ahead and develop this class using TDD - skeleton code is provided in `TestRecordsMap.py`.

Part 2 - `class RecordsMap`

Write a data structure `RecordsMap` that allows `O(1)` updating of a collection of records whenever a new report comes in. Keys should be a `(lat, long)` tuple and values should be a `(min, max)` tuple.



- `len` - the number of key:value pairs stored
- `get`
    - returns a tuple of `(min, max)` temperatures for a given position.
    - Raises `KeyError` if a point corresponding to the specified tuple is not in the mapping
- `contains(pos)` - returns `True` (`False`) if a given position is (is not) in this `RecordsMap`. It's fine to pass arbitrary precision (lat, long) positions here - LocalRecord takes care of the rounding.
- `add_report(pos, temp)` - updates max and min temperature for the given position as appropriate. There are 2 inputs here - a 2-tuple for position and a float for `temp`.
- `_rehash()` - periodically rehash as number of entries increases. Note that this is a private method - you do not need to write a unittest for it. However, you should write a test to make sure you're O(1) for add_report/contains/get, to make sure this method is getting called correctly.

Your class should support `O(1)` running times and `O(n)` memory allocation for the above methods on average, regardless of how many points are added (with the exception of `_rehash`, which should take `O(n)`).

Do not use the built-in set or dictionary types in this assignment except for testing - you must implement the functionality yourself.

Examples

These examples are intended to be illustrative, not exhaustive. Your code may have bugs even if it behaves as below. Write your own tests and think carefully about edge cases.

```
>>> from RecordsMap import *
>>> rm = RecordsMap()
>>> p1 = (41.8067, -72.2522)
>>> p2 = (41.8097, -72.1473) # rounds to the same as p1
>>> p3 = (41.3005, -72.4533) # rounds to different than p1
```

```
>>> # Examples: `in` and `add_report`
>>> p1 in rm
False
>>> rm.add_report(p1, 25)
>>> p1 in rm
True
>>> p2 in rm
True
>>> p3 in rm
False
>>>
>>> # Examples: `get` and add_report
>>> rm[p1]
(25, 25)
>>> rm.add_report(p1, 25.7) # new high
>>> rm[p1]
(25, 25.7)
>>> rm.add_report(p1, 20)   # new low
>>> rm[p1]
(20, 25.7)
>>> rm.add_report(p1, 24)   # no update
>>> rm[p1]
(20, 25.7)
>>> rm[p3]                  # no entry
Traceback (most recent call last):
...
KeyError: 'No records for pos (41.0, -72.0).'
```

Submission

At a minimum, submit the following file with the classes noted:

- `ReordsMap.py`
    - `class LocalRecord`
    - `class RecordsMap`
- `TestRecordsMap.py`
    - Unittests for:
        - `class LocalRecord`
        - `class RecordsMap`

You should include tests for the entire public interface.

Grading

This assignment is 100% manually graded.

- 30 - `LocalRecord` (tests and functionality)
- 70 - `RecordsMap` (tests and functionality)

Our manual grading will consider the following:

- Did you structure your code according to best practices in object-oriented programming?
- Did you thoroughly test your code using the unittest module?
- Did you choose and correctly implement the best data structures and algorithms?
- Is the code well-organized and documented? (use docstrings, comments, whitespace, and reasonable naming conventions)

Students must submit **individually** by the due date (typically Tuesday at 11:59 pm EST) to receive credit.

Some Pedantry

Feel free to ignore this section - it's just notes that might be of interest if you're curious about GIS, hashing, or general python-weirdness regarading `round()`.

- We are using negative numbers for southern latitudes and western longitudes, so the coordinates `p1` and `p2` in the first example correspond to (41.81° N, 72.25° W) - this is a location you are probably familiar with.
- Tuples can be geometrically close while hashing to very different values. Without rounding, `p1` and `p2` hash as below on my local computer:

  ```
  >>> hash(p1)
  -5786800418736233684
  >>> hash(p2)
  6156801258027466950
  ```

  Alternatively, two points can have very *similar hashes* while corresponding to very *different locations*. This has benefits for reducing collisions when keys are similar, but it makes it tough to find records for nearby points. Such behavior is obviously helpful when mapping local trends. Hashing coordinates such that similar hashes correspond to similar points is beyond the scope of this course, but you can read about one common technique here.

- Python's rounding can be a bit strange. It will round `5`s to the nearest even number instead of always rounding up.

  ```
  >>> round(3.5, 0)
  4.0
  >>> round(4.5, 0)
  4.0
  ```

  This helps minimize rounding bias. There are also occasions where rounding appears incorrect due to the fact that we typically view numbers in decimal notation, but computers handle arithmetic in binary numbers, which cannot always perfectly represent decimal numbers. See here for more information.