12.9 PROJECTS

Project 12.1 Tracking GPS coordinates

A GPS (short for Global Positioning System) receiver (like those in most mobile phones) is a small computing device that uses signals from four or more GPS satellites to compute its three-dimensional position (latitude, longitude, altitude) on Earth. By recording this position data over time, a GPS device is able to track moving objects. The use of such tracking data is now ubiquitous. When we go jogging, our mobile phones can track our movements to record our route, distance, and speed. Companies and government agencies that maintain vehicle fleets use GPS to track their locations to streamline operations. Biologists attach small GPS devices to animals to track their migration behavior.

In this project, you will write a class that stores a sequence of two-dimensional geographical points (latitude, longitude) with their timestamps. We will call such a sequence a *track*. We will use tracking data that the San Francisco Municipal Transportation Agency (SF MTA) maintains for each of the vehicles (cable cars, streetcars, coaches, buses, and light rail) in its Municipal Railway ("Muni") fleet. For example, the following table shows part of a track for the Powell/Hyde cable car in metropolitan San Francisco.

Time stamp	Longitude	Latitude
2014-12-01 11:03:03	−122.41144	37.79438
2014-12-01 11:04:33	−122.41035	37.79466
2014-12-01 11:06:03	−122.41011	37.7956
2014-12-01 11:07:33	−122.4115	37.79538
2014-12-01 11:09:03	−122.4115	37.79538

Negative longitude values represent longitudes west of the prime meridian and negative latitude values represent latitudes south of the equator. After you implement your track class, write a program that uses it with this data to determine the Muni route that is closest to any particular location in San Francisco.

An abstract data type for a track will need the following pair of attributes.

Instance Variable	Description
points	a list of geographical points with associated times
name	an identifier for the track

In addition, the ADT needs operations that allow us to add new data to the track, draw a picture of the track, and compute various characteristics of the track.

Method	Arguments	Description
append	point, time	add a point and time to the end of the track
length	—	return the number of points on the track
averageSpeed	—	return the average speed over the track
totalDistance	—	return the total distance traversed on the track
diameter	—	return the distance between the two points that are farthest apart on the track
closestDistance	point, error	find the closest distance a point on the track comes to the given point; return this distance and the time(s) when the track comes within error of this distance
draw	conversion function	draw the track, using the given function to convert each geographical point to an equivalent pixel location in the graphics window

Part 1: Write a Time class

Before you implement a class for the `Track` ADT, implement a `Time` class to store the timestamp for each point. On the book website is a skeleton of a `time.py` module that you can use to guide you. The constructor of your `Time` class should accept two strings, one containing the date in `YYYY-MM-DD` format and one containing the time in `HH:MM:SS` format. Store each of these six components in the constructed object. Your class should also include a `duration` method that returns the number of seconds that have elapsed between a `Time` object and another `Time` object that is passed in as a parameter, a `__str__` method that returns the time in `YYYY-MM-DD HH:MM:SS` format, a `date` method that returns a string representing just the date in `YYYY-MM-DD` format, and a `time` method that returns a string representing just the time in `HH:MM:SS` format. Write a short program that thoroughly tests your new class before continuing.

Part 2: Add a timestamp to the `Pair` class

Next, modify the `Pair` class from earlier in the chapter so that it also includes an instance variable that can be assigned a `Time` object representing the timestamp of the point. Also, add a new method named `time` that returns the `Time` object representing the timestamp of the point. Write another short program that thoroughly tests your modified `Pair` class before continuing.

Part 3: Write a Track class

Now implement a `Track` class, following the ADT description above. The points should be stored in a list of `Pair` objects. On the book website is a skeleton of a `track.py` module with some utility methods already written that you should use as a starting point.

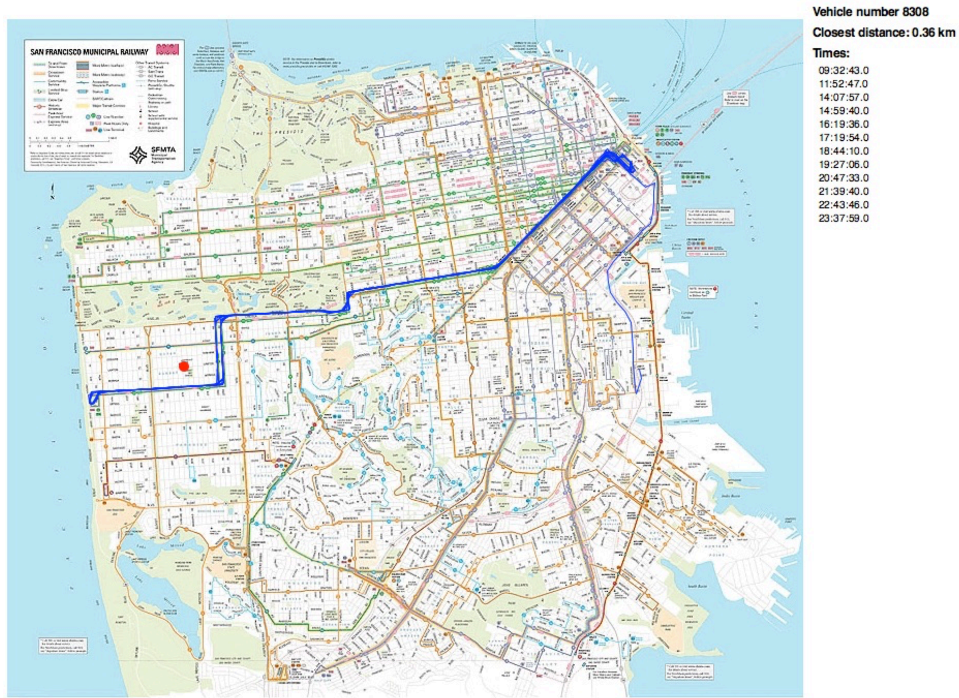
Question 12.1.1 *What does the* `_distance` *method do? Why is its name preceded with an underscore?*

Question 12.1.2 *What is the purpose of the* `degToPix` *function that is passed as a parameter to the* `draw` *method?*

After you write each method, be sure to thoroughly test it before moving on to the next one. For this purpose, design a short track consisting of four to six points and times, and write a program that tests each method on this track.

Part 4: Mysteries on the Muni

You are developing a forensic investigation tool that shows, for any geographical point in metropolitan San Francisco, the closest Muni route and the times at which the vehicle on that route passed by the given point. On the book website is an almost-complete program that implements this tool, named `muni.py`. The program should read one day's worth of tracking data from the San Francisco Municipal Railway into a list or dictionary of `Track` objects, set up a turtle graphics window with a map of the railway system, and then wait for a mouse click on the map. The mouse click triggers a call to a function named `clickMap`. The `clickMap` function draws a red dot where the click occurred, calls the `closestDistance` method of the `Track` class to find the Muni route that is closest to the click, and finally calls the `draw` method to draw the route on the map along with the times that the `Track` passed the click position. The output from the finished tool is visualized on the next page.



http://www.sfmta.com/maps/muni-system-map

In the `main` function of the program, the second-to-last function call

```
screen.onclick(clickMap)
```

registers the function named `clickMap` as the function to be called when a mouse click happens. Then the last function call

```
screen.mainloop()
```

initiates an *event loop* in the background that calls the registered `clickMap` function on each mouse click. The x and y coordinates of the mouse click are passed as parameters into `clickMap` when it is called.

The tracking data is contained in a comma-separated (CSV) file on the book website named `muni_tracking.csv`. The file contains over a million individual data points, tracking the movements of 965 vehicles over a 24-hour period. The only function in `muni.py` that is left to be written is `readTracks`, which should read this data and return a dictionary of `Track` objects, one per vehicle. Each vehicle is labeled in the file with a unique vehicle tag, which you should use for the `Track` objects' names and as the keys in the dictionary of `Track` objects.

Question 12.1.3 *Why is* `tracks` *a global variable? (There had better be a very good reason!)*

Question 12.1.4 *What are the purposes of the seven constant named values in all caps at the top of the program?*

Question 12.1.5 *What do the functions* `degToPix` *and* `pixToDeg` *do?*

Question 12.1.6 *Study the* `clickMap` *function carefully. What does it do?*

Once you have written the `readTracks` function, test the program. The program also uses the `closestDistance` method that you wrote for the `Track` class, so you may have to debug that method to get the program working correctly.

Project 12.2 Economic mobility

In Section 12.5, we designed a `Dictionary` class that assumes that no collisions occur. In this project, you will complete the design of the class so that it properly (and transparently) handles collisions. Then you will use your finished class to write a program that allows one to query data on upward income mobility in the United States.

To deal with collisions, we will use a technique called ***chaining*** in which each slot consists of a *list* of (key, value) pairs instead of a single pair. In this way, we can place as many items in a slot as we need.

Question 12.2.1 *How do the implementations of the* insert*,* delete*, and* lookup *functions need to change to implement chaining?*

Question 12.2.2 *With your answer to the previous question in mind, what is the worst case time complexity of each of these operations if there are n items in the hash table?*

Part 1: Implement chaining

First, modify the `Dictionary` class from Section 12.5 so that the underlying hash table uses chaining. The constructor should initialize the hash table to be a list of empty lists. Each `__getitem__`, `__setitem__`, and `__delitem__` method will need to be modified. Be sure to raise an appropriate exception when warranted. You should notice that these three methods share some common code that you might want to place in a private method that the three methods can call.

In addition, implement the methods named `_printTable`, `__str__`, `__contains__`, `items`, `keys`, and `values` described in Exercises 12.5.1–12.5.4.

Test your implementation by writing a short program that inserts, deletes, and looks up several entries with integer keys. Also test your class with different values of `self._size`.

Part 2: Hash functions

In the next part of the project, you will implement a searchable database of income mobility data for each of 741 commuting zones that cover the United States. A commuting zone is an area in which the residents tend to commute to the same city for work, and is named for the largest city in the zone. This city name will be the key for your database, so you will need a hash function that maps strings to hash table indices. Exercise 12.5.8 suggested one simple way to do this. Do some independent research to discover at least one additional hash function that is effective for general strings. Implement this new hash function.

Question 12.2.3 *According to your research, why is the hash function you discovered better than the one from Exercise 12.5.8?*

Part 3: A searchable database

On the book website is a tab-separated data file named `mobility_by_cz.txt` that contains information about the expected upward income mobility of children in each of the 741 commuting zones. This file is based on data from The Equality of Opportunity Project (`http://www.equality-of-opportunity.org`), based at Harvard and the University of California, Berkeley. The researchers measured potential income mobility in several ways, but the one we will use is the probability that a child raised by parents in the bottom 20% (or "bottom quintile") of income level will rise to the top 20% (or "top quintile") as an adult. This value is contained in the seventh column of the data file (labeled `"P(Child in Q5 | Parent in Q1), 80-85 Cohort"`).

Write a program that reads this data file and returns a `Dictionary` object in which the keys are names of commuting zones and the values are the probabilities described above. Because some of the commuting zone names are identical, you will need to concatenate the commuting zone name and state abbreviation to make unique keys. For example, there are five commuting zones named "Columbus," but your keys should designate Columbus, GA, Columbus, OH, etc. Once the data is in a `Dictionary` object, your program should repeatedly prompt for the name of a commuting zone and print the associated probability. For example, your output might look like this:

```
Enter the name of a commuting zone to find the chance that the
income of a child raised in that commuting zone will rise to
the top quintile if his or her parents are in the bottom quintile.
Commuting zone names have the form "Columbus, OH".

Commuting zone (or q to quit): Columbus, OH
Percentage is 4.9%.
Commuting zone (or q to quit): Columbus
Commuting zone was not found.
Commuting zone (or q to quit): Los Angeles, CA
Percentage is 9.6%.
Commuting zone (or q to quit): q
```

Part 4: State analyses

Finally, enhance your program so that it produces the following output, organized by state. You should create additional `Dictionary` objects to produce these results. (Do not use any built-in Python dictionary objects!)

1. Print a table like the following of all commuting zone data, alphabetized by state then by commuting zone name. (Hints: (a) create another `Dictionary` object as you read the data file; (b) the `sort` method sorts a list of tuples by the first element in the tuple.)

```
AK
  Anchorage: 13.4%
  Barrow: 10.0%
  Bethel: 5.2%
  Dillingham: 11.8%
  Fairbanks: 16.0%
  Juneau: 12.6%
  Ketchikan: 12.0%
  Kodiak: 14.7%
  Kotzebue: 6.5%
  Nome: 4.7%
  Sitka: 7.1%
  Unalaska: 13.0%
  Valdez: 15.4%
AL
  Atmore: 4.8%
  Auburn: 3.5%
  ⋮
```

2. Print a table, like the following, alphabetized by state, of the average probability for each state. (Hint: use another `Dictionary` object.)

```
State  Percent
-----  -------
 AK     11.0%
 AL      5.4%
 AR      7.2%
 ⋮
```

3. Print a table, formatted like that above, of the states with the five lowest and five highest average probabilities. To do this, it may be helpful to know about the following trick with the built-in `sort` method. When the `sort` method sorts a list of tuples or lists, it compares the first elements in the tuples or lists. For example, if `values = [(0, 2), (2, 1), (1, 0)]`, then `values.sort()` will reorder the list to be `[((0, 2), (1, 0), (2, 1)]`. To have the `sort` method use another element as the key on which to sort, you can define a simple function like this:

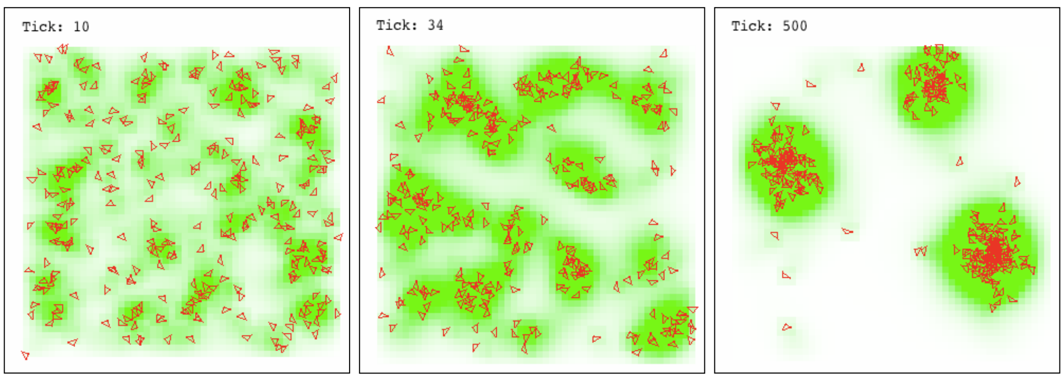
```python
def getSecond(item):
    return item[1]

values.sort(key = getSecond)
```

When the list named `values` is sorted above, the function named `getSecond` is called for each item in the list and the return value is used as the key to use when sorting the item. For example, suppose `values = [(0, 2), (2, 1), (1, 0)]`. Then the keys used to sort the three items will be `2`, `1`, and `0`, respectively, and the final sorted list will be `[(1, 0), (2, 1), (0, 2)]`.

Project 12.3 Slime mold aggregation

In this project, you will write an ***agent-based simulation*** that graphically depicts the emergent "intelligence" of a fascinating organism known as slime mold (*Dictyostelium discoideum*). When food is plentiful, the slime mold exists in a unicellular amoeboid form. But when food becomes scarce, it emits a chemical known as cyclic AMP (or cAMP) that attracts other amoeboids to it. The congregated cells form a ***pseudoplasmodium*** which then scavenges for food as a single multicellular organism. We will investigate how the pseudoplasmodium forms. A movie linked from the book website shows this phenomenon in action.

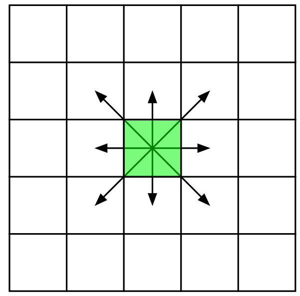
The following sequence of images illustrates what your simulation may look like. The red triangles represent slime mold amoeboids and the varying shades of green represent varying levels of cAMP on the surface. (Darker green represents higher levels.)



Slime world

In our simulation, the slime mold's world will consist of a grid of square patches, each of which contains some non-negative level of the chemical cAMP. The cAMP will be deposited by the slime mold (explained next). In each time step, the chemical in each patch should:

1. Diffuse to the eight neighboring patches. In other words, after the chemical in a patch diffuses, 1/8 of it will be added to the chemical in each neighboring patch. (Note: this needs to be done carefully; the resulting levels should be as if all patches diffused simultaneously.)



2. Partially evaporate. (Reduce the level in each patch to a constant fraction, say 0.9, of its previous level.)

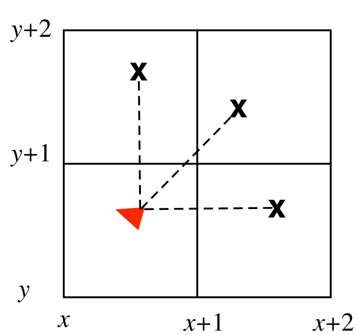
Slime world will be modeled as an instance of a class (that you will create) called `World`. Each patch in slime world will be modeled as an instance of a class called `Patch` (that you will also create). The `World` class should contain a grid of `Patch` objects. You will need to design the variables and methods needed in these new classes.

There is code on the book website to visualize the level of chemical in each patch. Higher levels are represented with darker shades of green on the turtle's canvas. Although it is possible to recolor each patch with a Turtle during each time step, it is far too slow. The supplied code modifies the underlying canvas used in the implementation of the `turtle` module.

Amoeboid behavior

At the outset of the simulation, the world will be populated with some number of slime mold amoeboids at random locations on the grid. At each time step in the simulation, a slime mold amoeboid will:

1. "Sniff" the level of the chemical cAMP at its current position. If that level is above some threshold, it will next sniff for chemical `SNIFF_DISTANCE` units ahead and `SNIFF_DISTANCE` units out at `SNIFF_ANGLE` degrees to the left and right of its current position. `SNIFF_ANGLE` and `SNIFF_DISTANCE` are parameters that can be set in the simulation. In the graphic below, the slime mold is represented by a red triangle pointing at its current heading and `SNIFF_ANGLE` is 45 degrees. The `X`'s represent the positions to sniff.



Notice that neither the current coordinates of the slime mold cell nor the coordinates to sniff may be integers. You will want to write a function that will round coordinates to find the patch in which they reside. Once it ascertains the levels in each of these three patches, it will turn toward the highest level.

2. Randomly wiggle its heading to the left or right a maximum of `WIGGLE_ANGLE` degrees.
3. Move forward one unit on the current heading.
4. Drop `CHEMICAL_ADD` units of cAMP at its current position.

A slime mold amoeboid should, of course, also be modeled as a class. At the very least, the class should contain a `Turtle` object that will graphically represent the cell. Set the speed of the `Turtle` object to 0 and minimize the delay between updates by calling `screen.tracer(200, 0)`. The remaining design of this class is up to you.

The simulation

The main loop of the simulation will involve iterating over some number of time steps. In each time step, every slime mold amoeboid and every patch must execute the steps outlined above.

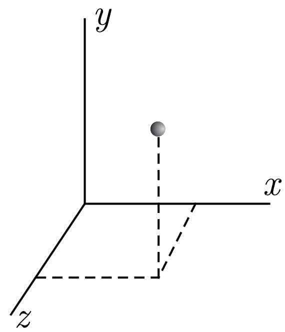
Download a bare-bones skeleton of the classes needed in the project from the book website. These files contain *only the minimum amount of code necessary* to accomplish the drawing of cAMP levels in the background (as discussed earlier).

Before you write any Python code, think carefully about how you want to design your project. Draw pictures and map out what each class should contain. Also map out the main event loop of your simulation. (This will be an additional file.)

Project 12.4 Boids in space

In this project, you will generalize the two-dimensional flocking simulation that we developed in Section 12.3 to three dimensions, and visualize it using three-dimensional graphics software called VPython. For instructions on how to install VPython, visit `http://vpython.org`.

The three-dimensional coordinate system in VPython looks like this, with the positive z-axis coming out of the screen toward you.



The center of the screen is at the origin $(0, 0, 0)$.

Part 1: Generalize the `Vector` *class*

The `Pair` and `Vector` classes that we used to represent positions and velocities, respectively, are limited to representing two-dimensional quantities. For this project, you will generalize the `Vector` class that we introduced in Section 12.3 so that it can store vectors of any length. Then use this new `Vector` class in place of both `Pair` and the old `Vector` class.

The implementation of every method will need to change, except as noted below.

- The constructor should require a list or tuple parameter to initialize the vector. The length of the parameter will dictate the length of the `Vector` object. For example,

  ```
  velocity = Vector((1, 0, 0))
  ```

  will assign a three-dimensional `Vector` object representing the vector $\langle 1, 0, 0\rangle$.
- Add a `__len__` method that returns the length of the `Vector` object.
- The `unit` and `diffAngle` methods can remain unchanged.
- You can delete the `angle` and `turn` methods, as you will no longer need them.
- The dot product of two vectors is the sum of the products of corresponding elements. For example, the dot product of $\langle 1, 2, 3\rangle$ and $\langle 4, 5, 6\rangle$ is $1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 4 + 10 + 18 = 32$. The `dotproduct` method needs to be generalized to compute this quantity for vectors of any length.

Part 2: Make the `World` three-dimensional

Because we were careful in Section 12.3 to make the `World` class very general, there is little to be done to extend it to three dimensions. The main difference will be that instead of enforcing boundaries on the space, you will incorporate an object like a light to which the boids are attracted. Therefore, the swarming behavior will be more similar to moths around a light than migrating birds.

- Generalize the constructor to accept a `depth` in addition to `width` and `height`. These attributes will only be used to size the VPython display and choose initial positions for the boids.
- Once you have `vpython` installed, you can import it with

  ```
  import vpython as vp
  ```

  To create a new window, which is an object belonging to the class `canvas`, do the following:

  ```
  self._scene = vp.canvas(title='Boids', width = 800, height = 800,
                          range = width,
                          background = vp.vector(0.41, 0.46, 0.91))
  ```

  In the `canvas` constructor, the `width` and `height` give the dimensions of the window and the `range` argument gives the visible range of points from the origin. The `background` color is a deep blue; feel free to change it. The `vpython` objects use their own `vp.vector` class to define positions and colors. So you will need to convert between your `Vector` class and `vp.vector` when using `vpython` objects. The following utility functions will do this.

  ```
  def vector2VP(vector):
      a, b, c = vector.get()
      return vp.vector(a, b, c)

  def VP2vector(vpvector):
      return Vector((vpvector.x, vpvector.y, vpvector.z))
  ```

  To place a yellow sphere in the center to represent the light, create a new `sphere` object like this:

  ```
  self._light = vp.sphere(scene = self._scene, color = vp.color.yellow)
  ```

- In the `stepAll` method, make the light follow the position of the mouse with

  ```
  self._light.pos = self._scene.mouse.pos
  ```

  You can change the position of any `vpython` object by changing the value of its `pos` instance variable. In the display object (`self._scene`), `mouse` refers to the mouse pointer within the VPython window.
- Finally, generalize the `_distance` function, and remove all code from the class that limits the position of an agent.

Part 3: Make a `Boid` *three-dimensional*

- In the constructor, initialize the position and velocity to be three-dimensional `Vector` objects. You can represent each boid with a cone, pointing in the direction of the current velocity with:

```
self._turtle = vp.cone(pos = vp.vector(x, y, z),
                       axis = vector2VP(self._velocity * 3),
                       color = vp.color.white,
                       scene = self._world._scene)
```

- The `move` method will need to be generalized to three dimensions, but you can remove all of the code that keeps the boids within a boundary. Once the new position and velocity have been computed, you can move the boid with

```
self._turtle.pos = vp.vector(newX, newY, newZ)
self._turtle.axis = vector2VP(self._velocity * 3)
```

- The `_avoid`, `_center`, and `_match` methods can remain mostly the same, except that you will need to replace instances of `Pair` with `Vector`. Also, have the `_avoid` method avoid the light in addition to avoiding other boids. When getting the position of the light, you will need the `VP2vector` function above.
- Write a new method named `_light` that returns a unit vector pointing toward the current position of the light. Incorporate this vector into your `step` method with another weight

```
LIGHT_WEIGHT = 0.3
```

The main simulation

Once you have completed the steps above, you can remove the turtle graphics setup, and simplify the main program to the following.

```
from world import *
from boid import *
import vpython as vp

WIDTH = 60
HEIGHT = 60
DEPTH = 60
NUM_MOTHS = 20

def main():
    sky = World(WIDTH, HEIGHT, DEPTH)
    for index in range(NUM_MOTHS):
        moth = Boid(sky)

    while True:
        vp.rate(25)    # 1 / 25 sec elapse between computations
        sky.stepAll()

main()
```