Polling places

Due: Friday, November 13, 3pm CST

You may work alone or in a pair on this assignment.

The goal of this assignment is to give you practice designing and implementing classes and methods, and working with class instances.

Getting started

Before you start working on the assignment's tasks, please take a moment to follow the steps described in Coursework Basics (https://classes.cs.uchicago.edu/archive/2020/fall/12100-1/resources/coursework-basics.html#coursework-basics/) page to get the files for this assignment (these steps will be the same as the ones you followed to get the files for previous assignments).

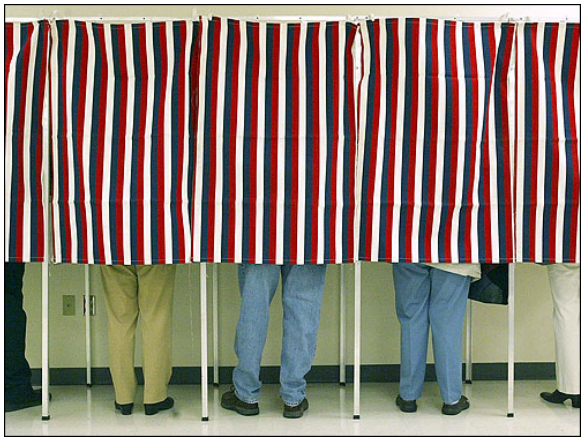
If you are going to work in a pair, please follow the "Working in a Pair" instructions in Coursework Basics (https://classes.cs.uchicago.edu/archive/2020/fall/12100-1/resources/coursework-basics.html#working-in-a-pair) to get started. Students working in pairs should **not** use their individual repositories. Please not that you will not be able to start working on the assignment until you request and setup your pair repository.

Your `pa4` directory will include:

- a file named `simulate.py`, which you will modify,
- a file named `util.py` that contains a few useful functions,
- a directory named `data`, which contains precinct configuration files (described below),
- and a series of test files.

The `README` file contains information on the contents of your `pa4` directory. Please put all of your code for this assignment in `simulate.py`. Do not edit other files or add extra files for the required classes.

Introduction



(../../_images/voting_booths.png)
Every two years, many people around the country wait in long lines to vote. In some states, voters have the option to vote a "straight-ticket" where they select candidates for every office from one political party with a single mark on their ballot. If a voter does not want to vote a straight-ticket, they can vote a "split-ticket" by filling out the entire ballot, selecting candidates for each office one-by-one.

Straight-ticket voting is a popular voting method where it is available. In Texas, two-thirds of voters voted a straight-ticket in the 2018 midterm election (https://www.texasmonthly.com/politics/straight-ticket-voting-says-farewell-texas-big-way/). However, this was not an option for Texas voters in November 2020. Removing the option to straight-ticket vote may have lead to longer waiting times since it can take more time to fill out a lengthy ballot than voting with a single mark.

In this assignment, you will write code to simulate the flow of voters through precincts and analyze the effect of straight-ticket voting on waiting time.

We will view this problem as an instance of the more general problem of simulating an $M/M/N$ queue, where:

- the first M indicates that arrivals (voters arriving at the polls) obey a Markov process;
- the second M indicates that the service times obey a Markov process (the amount of time a voter takes to complete a ballot); and finally
- the N indicates that there are N servers (the number of voting booths).

We will tell you how to structure your implementation at a high level, but you will be responsible for the design details.

Precinct Configuration

The configuration for a precinct is specified using a dictionary containing this information:

- The name of the precinct (`name`)
- Number of hours the polls are open (`hours_open`)

- Number of voters assigned to the precinct (`num_voters`)
- The number of voting booths in the precinct (`num_booths`)
- The rate at which voters arrive (`arrival_rate`)
- The percentage of straight-ticket voters (`percent_straight_ticket`) and the voting time for straight-ticket voters (`straight_ticket_duration`)
- The rate at which split-ticket voters complete their voting (`voting_duration_rate`). The significance of these rates is explained later.

Note that `percent_straight_ticket` is, strictly speaking, the probability that a voter is a straight-ticket voter written as a number between 0 and 1. That is, the expected percentage of straight-ticket voters is `percent_straight_ticket` .

For example:

```
{
 "name": "Downtown",
 "hours_open": 1,
 "num_voters": 5,
 "num_booths": 1,
 "voting_duration_rate": 0.1,
 "arrival_rate": 0.11,
 "percent_straight_ticket": 0.5,
 "straight_ticket_duration": 2
}
```

Information about precincts is stored in a JSON file that contains a dictionary with two items: the key `"precincts"` maps to a list of precinct configurations as specified above, and the key `"seed"` maps to a random seed. Similar to PA1, the random seed will be used to ensure that a simulation involving randomness produces the same result every time it runs (which will allow us to test whether your code produces the expected results). We recommend reviewing PA1 (and how it used random seeds) if you're unsure of how random seeds are used in the context of running a simulation.

We have provided a function, `load_precincts` , in `util.py` that takes the name of a JSON precincts file and returns two values: a list of precinct configurations (as specified above) and the random seed.

This function is already called from the code we provide for you, so you should not make any calls to `load_precincts` in the code you'll add to `simulate.py` . However, it can be a useful function when testing your code. In particular, you should take a moment to familiarize yourself with the data stored in these JSON files, and how to access each value once you've loaded the file with `load_precincts` . We suggest you start by looking at `data/config-single-precinct-0.json` , and loading that file from IPython:

```
In [1]: import util

In [2]: precincts, seed = util.load_precincts("data/config-single-precinct-0.json")

In [3]: precincts
Out[3]:
[{'name': 'Downtown',
  'hours_open': 1,
  'num_voters': 5,
  'num_booths': 1,
  'voting_duration_rate': 0.1,
  'arrival_rate': 0.11,
  'percent_straight_ticket': 0.5,
  'straight_ticket_duration': 2}]

In [4]: seed
Out[4]: 1468604453

In [5]: len(precincts)
Out[5]: 1

In [6]: p = precincts[0]

In [7]: p["num_voters"]
Out[7]: 5

In [8]: p["arrival_rate"]
Out[8]: 0.11

In [9]: p["percent_straight_ticket"]
Out[9]: 0.5
```

Representing Voters

In this assignment you will write a `Voter` class to model voters. This class must include the time the voter arrives at the polls, voting duration, that is, the amount of time the voter takes to vote, and the time the voter is assigned to a voting booth (aka, the start time). For simplicity, we will model time as a floating point number, representing the number of minutes since the polls opened. So, if we say a voter arrived at the polls at time 60.5, that means they arrived an hour and thirty seconds after the polls opened.

While not strictly necessary, we found it convenient for debugging purposes to store the time the voter leaves the voting booth (i.e., the departure time) as well.

The start time for a voter, which is not available at construction time, should be initialized to `None` when the voter object is constructed. This value should be reset to the correct value when the voter is assigned to a voting booth. Similarly, voters' departure times cannot be determined until we know their start times.

Your implementation **must** include at a minimum: a constructor and public attributes named `arrival_time`, `voting_duration`, and `start_time`. Our utility function for printing voters, described below, relies on the existence of these attributes.

This class is straightforward. It merely serves as a way to encapsulate information about a voter. Our implementation is less than 20 lines.

Please note, the `Precinct` class, described next, is the only class that is allowed to call the `Voter` constructor.

Precincts

Cities are broken up into voting precincts. You will complete the definition of a `Precinct` class that represents a single precinct, and which will be responsible for simulating the behaviour of voters in that precinct.

A precinct is open for some number of hours, has some maximum number of voters who can vote in the precinct (you can think of this as the number of voters who are registered to vote in that precinct), has some percentage of voters who vote straight-ticket, and has some number of voting booths. We will assume that all registered voters will go to the polls on election day, but not all of them may get to vote, because some could arrive after the polls close.

Notice how having a `Precinct` class allows us to have multiple `Precinct` objects in our program, each with its own set of parameters. This will make it easier for us to simulate the behavior of multiple precincts, as well as compare the effects of those parameters. For example, we could create two `Precinct` objects with the exact same parameters, except that 20% of voters vote a straight-ticket in one precinct, and 70% in the other. Simulating both can provide insights into how the percentage of straight-ticket voters can affect waiting times.

We have provided a skeleton for the `Precinct` class that includes the constructor and a `simulate` method to simulate a single voting day for that precinct. While you are allowed to add additional methods to this class, you *cannot* modify the parameter list of the constructor or the `simulate` method.

Please note that we will consider each `Precinct` as completely independent from each other: voters cannot go from one precinct to another, and there is **no shared information between the precincts**.

Generating Voters

One of the main responsibilities of the `Precinct` class is to generate the voters, so let's take a closer look at how those voters will be generated.

The arrival time of our voters will follow a Poisson process. From Wikipedia: *a Poisson process, named after the French mathematician Siméon-Denis Poisson (1781-1840), is a stochastic process in which the time to the next event is independent of past events.* We can use the following fact to generate a voter sample: if the number of arrivals in a given time interval $[0, t]$ follows the Poisson distribution with mean $t \cdot \lambda$, then the gap between arrivals times follows the exponential distribution with rate parameter λ.

This means that a `Precinct` object must keep track of time. We will assume that we keep track of time in minutes starting at time zero. To generate a voter, you will need to generate an arrival time and a voting duration. To generate the arrival time of the next voter, you will generate an inter-arrival time (gap) from the exponential distribution using the specified arrival rate as λ and add it to the current time (t). You'll then move time forward by updating the time from t to $t + gap$. Notice how gap represents the time that elapses between the arrival of the previous voter and the arrival of the next voter.

For example, let's say we want to generate a voter in a precinct, and the random number generator returns `5.7` for that voter's gap. Since this is the first voter we generate, their arrival time will be `5.7`, and we advance the precinct's current time from `0` to `5.7`. We then generate another voter, and the random number generator re-

turns `2.4` for the voter's gap. This means that this second voter arrives `2.4` minutes *after* the previous voter, so the second voter's arrival time will be `8.1` (`5.7 + 2.4`) and we advance the precinct's current time to `8.1`.

When we generate a voter, we also need to assign a voting duration to that voter. The voting duration of a voter depends on whether the voter is a straight-ticket voter or not. If a voter is a straight-ticket voter, the voting duration is a fixed number of minutes. Otherwise, the voting duration follows a Poisson process so you will draw the voter's voting duration from the exponential distribution using the specified voting duration rate as λ.

For testing purposes, it is important that you draw the gap, decide whether a voter is a straight-ticket voter, and decide the voting duration in the same order as our code. To reduce the likelihood that the testing process fails simply because the values were drawn in the wrong order, we have provided a function, `gen_voter_parameters`, in `util.py` that takes the arrival rate, the voting duration rate, percentage of straight-ticket voters, and the straight-ticket voting duration as arguments and returns a pair of floats to use for the gap and voting duration (in that order). We recommend familiarizing yourself with this function to understand how arrival gaps, straight-ticket voters, and voting duration times are chosen.

We also recommend that you include a `next_voter` method in your `Precinct` class. This method would return the next voter in that precinct, and can be useful both for testing and for keeping your simulation code clean.

Task 0: Implement voter generation

You will start by implementing the `Voter` class, as well as writing only the voter generation logic in the `simulate` method in `Precinct`. In other words, instead of carrying out a simulation, you will just generate all the `Voter` objects for the voters who will be able to vote in the precinct, and will return a list of these objects.

When doing so, you must generate voters until you reach one of the simulation's *stopping conditions*. In this case, we need to stop the simulation once there are no more voters who can vote. This can happen in two situations: (1) you have generated the maximum number of the voters specified for the precinct, or (2) you generate a voter who arrives after closing time. In the second case, the late voter should be discarded and no further voters should be generated. We will refer to voters who arrive before the polls close as *valid*.

Doing this partial implementation of the `simulate` method will ensure that you've correctly implemented the voting generation logic we described earlier. The code for generating the voters in your `Precinct` should be fairly simple (our implementation is only 15 lines long, not including comments).

Testing

To manually test your implementation, we have provided two sets of voter generation parameters. The first set (see `data/config-single-precinct-0.json`) includes the following:

- number of voters in the precinct: 5
- number of booths in the precinct: 1
- hours open: 1
- arrival rate: 0.11 (corresponds to mean time between voters of approximately 9 minutes)
- voting duration rate: 0.1 (corresponds to a mean voting duration of 10 minutes for split-ticket voters)
- percentage of straight-ticket voters: 0.5 (50% of voters)
- voting duration of straight-ticket voters: 2 (all straight-ticket voters take 2 minutes to vote)
- seed for random number generator: 1468604453

and corresponds to a voter population in which all the voters arrive before the polls close.

The random seed

Remember that the precincts file includes a seed value (returned by `load_precincts`). This seed must be passed to `simulate` (using the `seed` parameter) and you must use the `random.seed(seed)` function to set the seed *inside* `simulate`. The seed only needs to be set once per simulation.

Remember that PA1 includes a longer discussion of random seeds, and their significance in testing simulations.

We have also provided a function named `print_voters` in `util.py` that takes a list of `Voter` instances and an optional file name. This function will either print a representation of the voters to screen or to a file, depending on whether a filename is specified (you likely won't have to write to a file yourselves, but it is something that we've found useful on occasion).

You can use this function to print the return value of the `simulate` method, and check whether the returned values look correct:

```
In [1]: from simulate import Precinct

In [2]: import util

In [3]: precincts, seed = util.load_precincts("data/config-single-precinct-0.json")

In [4]: p = precincts[0]

In [5]: precinct = Precinct(p["name"], p["hours_open"], p["num_voters"],
   ...:                     p["num_booths"], p["arrival_rate"], p["voting_duration_rate"])

In [6]: voters = precinct.simulate(percent_straight_ticket=0.5,
   ...:                            straight_ticket_duration=2,
   ...:                            seed=seed)

In [7]: util.print_voters(voters)
Arrival Time   Voting Duration   Start Time    Departure Time
      0.23         2.11             None             None

     18.49         2.00             None             None

     20.11         2.00             None             None

     26.70         2.00             None             None

     29.50         2.54             None             None
```

Notice how the `Start Time` and the `Departure Time` will be initially `None` because you are not yet calculating the time when the voters start voting (they may need to wait in line once they arrive at the precinct, and we describe this in more detail in the next task).

The second set, `data/config-single-precinct-1.json`, is the same except the arrival rate is set to 0.04, which corresponds to a mean gap of 25 minutes between arrivals:

```
In [8]: precincts, seed = util.load_precincts("data/config-single-precinct-1.json")

In [9]: p = precincts[0]

In [10]: precinct = Precinct(p["name"], p["hours_open"], p["num_voters"],
    ...:                     p["num_booths"], p["arrival_rate"], p["voting_duration_rate"]

In [11]: voters = precinct.simulate(percent_straight_ticket=0.5,
    ...:                            straight_ticket_duration=2,
    ...:                            seed=seed)

In [12]: util.print_voters(voters)
Arrival Time   Voting Duration   Start Time    Departure Time
      0.64         2.11             None              None

     50.85         2.00             None              None

     55.30         2.00             None              None
```

Notice how, in the above set, only three voters get to vote. If you actually generated the fourth voter, it would have an arrival time of `73.44`, which means they would arrive after the polls closed since this precinct is only open for one hour.

Tip: Instead of having to type in all the above code every time, you may want to write a short `load_and_print` function to help you.

Once you've done some manual testing, you can run additional automated tests with py.test:

```
$ py.test -xv -k voter
```

(Recall that we use `$` to indicate the Linux command-line prompt. You should not include it when you run this command.)

As usual, remember that you can find more detailed instructions on how manual and automated testing work in the Testing Your Code (../../resources/testing.html#testing-your-code) page.

Task 1: Simulate an election day

In this task, you will be simulating an election for a single precinct. You will do this by completing the implementation of the `simulate` method in `Precinct` and by implementing a `VotingBooths` class that models the voting booths in that precinct.

Simulating a Precinct

The simulation itself will be similar to the M/D/1 queue example from class. In the M/D/1 example, we have a single server, such as a toll booth, that serves a queue of clients, such as cars, that arrive with a specified distribution and have a fixed service time. In this assignment, your simulation will need to simulate multiple servers (voting booths) and variable service times (voting durations), but the structure of your simulation will be similar.

The biggest difference is that you will be implementing a `VotingBooths` class which models the "multiple servers" in the queue. This class will be responsible for determining whether a voting booth is available and, if not, when a booth will become available. This will be necessary to determine when a given voter will finish voting (since it depends on when they can access a booth), and to ensure that the number of booths in use to be no more than the number of booths assigned to the precinct. We discuss the `VotingBooths` class in more detail below.

Example

Let's walk through an example. The file `data/config-single-precinct-2.json` contains the following precinct:

```
{
 "name": "Little Rodentia",
 "hours_open": 1,
 "num_voters": 10,
 "num_booths": 2,
 "voting_duration_rate": 0.1,
 "arrival_rate": 0.16666666666666666,
 "percent_straight_ticket": 0.5,
 "straight_ticket_duration": 2
}
```

And the seed `1438018945`.

The precinct has 10 voters who arrive at the precinct once every 6 minutes on average. Straight-ticket voters take exactly 2 minutes to vote and split-ticket voters take 10 minutes on average to vote.

As you work on your implementation, it can be useful to print out the voters returned by your `simulate` method, even if you have not yet filled in the start and departure times of the voters:

```
In [3]: precincts, seed = util.load_precincts("data/config-single-precinct-2.json")

In [4]: p = precincts[0]

In [5]: precinct = Precinct(p["name"], p["hours_open"], p["num_voters"],
   ...:                     p["num_booths"], p["arrival_rate"], p["voting_duration_rate"])

In [6]: voters = precinct.simulate(percent_straight_ticket=0.5,
   ...:                            straight_ticket_duration=2,
   ...:                            seed=seed)

In [7]: util.print_voters(voters)
Arrival Time   Voting Duration   Start Time    Departure Time
     13.38         2.00             None              None

     15.58         0.12             None              None

     17.92         4.15             None              None

     18.98        11.31             None              None

     22.29         2.00             None              None

     22.71        22.74             None              None

     35.05         2.00             None              None

     48.89         2.00             None              None

     51.34         2.00             None              None

     51.48         2.00             None              None
```

Let's consider what happens when we simulate this precinct which has two voting booths. The first call to `util.gen_voter_parameters` generates `(13.38, 2.00)` as the gap and voting duration. This means that our first voter arrives at 13.38, enters a voting booth immediately at time 13.38, and finishes voting at 15.38 (13.38 + 2.00). (All times in this example have been rounded to two digits for clarity; your code should not round times.)

The second call to `util.gen_voter_parameters` yields `(2.20, 0.12)`, which means that the second voter arrives at time 15.58 (13.38 + 2.20). This second voter starts voting immediately, because a booth is open, and finishes voting at time 15.70 (15.58 + 0.12).

The third voter arrives at time 17.92 (gap: 2.33) and has a voting duration of 4.15 minutes. Since the first voter departed at 15.38 and the second voter departed at 15.70, this voter can start voting immediately, and finishes voting at 22.07.

The fourth voter arrives at 18.98 (gap: 3.10) and has a voting duration of 11.31 minutes. Only one booth is occupied when the fourth voter arrives, so this voter can start voting immediately, and finishes voting at 30.29.

The fifth voter arrives at time 22.29 (gap: 3.31), after the third voter has departed, and starts voting immediately. This voter departs at 24.29. Notice that this voter finishes *before* the fourth voter.

The sixth voter arrives at 22.71 (gap: 0.43) and has a voting duration of 22.74 minutes. Both booths are occupied when the sixth voter arrives, so this voter must wait until a booth becomes free, which happens when the fifth voter finishes at time 24.29. Thus, unlike the previous voters, the sixth voter's start time (24.29) will be different from their arrival time (22.71) and their finish time will be 47.03.

And so on.

All ten voters arrive before the polls close and so all ten get to vote. Remember, if a voter arrives before the polls close, they still get to vote even if they will not start voting until after the polls have closed. In this example, the polls close at time 60 (the precinct dictionary specifies the number of *hours* the polls are open, while all other times are measured in *minutes*).

Here's the result of our simulation:

```
In [3]: precincts, seed = util.load_precincts("data/config-single-precinct-2.json")

In [4]: p = precincts[0]

In [5]: precinct = Precinct(p["name"], p["hours_open"], p["num_voters"],
   ...:                     p["num_booths"], p["arrival_rate"], p["voting_duration_rate"])

In [6]: voters = precinct.simulate(percent_straight_ticket=0.5,
   ...:                            straight_ticket_duration=2,
   ...:                            seed=seed)

In [7]: util.print_voters(voters)
Arrival Time   Voting Duration   Start Time    Departure Time
     13.38         2.00            13.38             15.38

     15.58         0.12            15.58             15.70

     17.92         4.15            17.92             22.07

     18.98        11.31            18.98             30.29

     22.29         2.00            22.29             24.29

     22.71        22.74            24.29             47.03

     35.05         2.00            35.05             37.05

     48.89         2.00            48.89             50.89

     51.34         2.00            51.34             53.34

     51.48         2.00            51.48             53.48
```

Modelling voting booths within a precinct

We will encapsulate the behavior of the voting booths using a `VotingBooths` class. In turn, this class will need a data structure to keep track of the voters who are currently occupying voting booths and to determine which voter will depart next.

Your implementation should use the `PriorityQueue` class the from the Python queue (https://docs.python.org/3/library/queue.html) library for this purpose. A *minimum priority queue* is a data structure that includes operations for (1) adding items along with a *priority* to the queue (where the priority is just a number), (2) removing the item with the *lowest* priority value from the queue, and (3) determining whether the queue is empty. Removing the elements with the lowest priority value may seem counter-intuitive but, in Computer Science, a lower value usually denotes a higher priority (think of it as a ranking: if you're #1, your ranked above someone who is ranked #2).

The `PriorityQueue` class also allows you to specify an upper bound on the number of items allowed to be in the queue at the same time and provides a method for determining whether the queue is full. Please look through the queue (https://docs.python.org/3/library/queue.html) library for details on how to use the this class.

For this assignment, the priorities will be the voters' departure times.

Blocking on `get` and `put`

You should be careful with one specific aspect of priority queues: the `get` method is a *blocking* method, meaning that if you call `get` (to extract a value from the priority queue) and the queue has no values, the operation will *block* (or hang) until a value is available.

In this assignment, you should never call `get` on an empty queue so, if your code seems to mysteriously hang, the first thing you should check is whether you're calling `get` on an empty queue somewhere in your code. Alternatively, you can call `get` like this:

```
my_queue.get(block=False)
```

This way, `get` will raise an `Empty` exception if you try to extract a value from an empty queue (immediately alerting you to the fact that there may be an error somewhere in your code, since you should never be calling `get` on an empty queue in this assignment)

Similarly, if you call `put` on a queue that is full, your code will also block. However, if you call `put` like this:

```
my_queue.put(value, block=False)
```

Then putting a value into a full queue will raise a `Full` exception.

Note: the `PriorityQueue` class does not have a "peek" function (to look at the element at the front of the queue without removing it), but you should be able to implement your `VotingBooths` class without relying on a "peek" function.

Take into account that your implementation should not need a separate instance of `VotingBooths` for each individual voting booth. There could certainly be models that require modelling each voting booth as its own object but, in the model we are assuming, it is enough to have a `VotingBooths` class that uses just one priority queue to model the state of all the voting booths.

Furthermore, your `Precinct` code should not need to access the priority queue directly. Instead, you should implement methods in the `VotingBooths` class that encapsulate operations like adding a voter to a booth, or removing the next voter who will finish voting. Your `Precinct` code will then call those. An easy way to make sure that your `Precinct` doesn't directly access the priority queue is by making sure that the priority queue inside `VotingBooths` is a *private* attribute (by adding two underscores at the start of the attribute's name).

Don't queue voters!

You may be tempted to create a queue of `Voter` objects. Don't! This won't be necessary, since we're already using a priority queue to model the voting booths. This allows us to efficiently answer the question "When will a voting booth become available in this precinct?". If we can easily answer that question, then we will know the voting start time of any new `Voter`.

Returning the voters

Your `simulate` function must return a list of `Voter` objects with the `start_time` attribute filled in. Furthermore, for a given precinct, the list of voters should be sorted by the voters' arrival time. Since your `Precinct` should generate voters in the order in which they arrive, producing a list of voters sorted by arrival time should not require any additional steps. If you do find that you need to sort the list explicitly, it is likely that your solution is headed in the wrong direction; utilize office hours or ask on Piazza if that is the case.

Testing

We have provided several automated tests that will test your implementation of the `simulate` method. The first tests you should run are the ones that simulate a single precinct (and, thus, create only a single `Precinct` object). You can run these tests like this:

```
$ py.test -xv -k single test_simulate_election_day.py
```

However, remember that the first approach to testing and debugging your code should not rely exclusively on the tests. If you run the tests, and it is not immediately clear what the issue is, you should manually run your simulation code either from IPython (in the way we've been doing so far) or by using a command we describe below. These two approaches will generally produce error messages that may be easier to interpret and debug.

If you run a simulation from IPython, and would like to manually verify whether the values are correct, each configuration file has a corresponding CSV file with the expected values (take into account that these are the files the automated tests use; if you find that your implementation seems to produce correct values, you may want to switch to running the automated tests).

Once the single precinct tests are passing, you can then try the tests that simulate multiple precincts. Take into account that the precinct configuration files we have used up to this point contain only a single precinct. However, the file format allows for multiple precincts to be specified in the same file. For example, the `config-multiple-precincts-1.json` file specifies two identical precincts, except for the number of booths

```
{
 "name": "Little Rodentia (1 booth)",
 "hours_open": 1,
 "num_voters": 10,
 "num_booths": 1,
 "voting_duration_rate": 0.1,
 "arrival_rate": 0.16666666666666666,
 "percent_straight_ticket": 0.5,
 "straight_ticket_duration": 2
}
```

```
{
 "name": "Little Rodentia (2 booths)",
 "hours_open": 1,
 "num_voters": 10,
 "num_booths": 2,
 "voting_duration_rate": 0.1,
 "arrival_rate": 0.16666666666666666,
 "percent_straight_ticket": 0.5,
 "straight_ticket_duration": 2
}
```

In most cases, if you are passing the single precinct tests, you should also pass the multiple precinct tests effortlessly. However, there are a few bugs that will only be caught if we create multiple `Precinct` objects.

You can run the automated tests that use multiple precincts like this:

```
$ py.test -xv -k multiple test_simulate_election_day.py
```

If you want to run all the simulation tests (with both single and multiple precincts), you can just run this:

```
$ py.test -xv -k day
```

We also provide a main function in `simulate.py` that allows you to run `simulate.py` from the command-line. If you run `simulate.py` with just the name of a configuration file, it will print a summary of the simulation results. For example:

```
$ python3 simulate.py data/config-single-precinct-0.json

PRECINCT 'Downtown'
- 5 voters voted.
- Polls closed at 60.0 and last voter departed at 32.04.
- Avg wait time: 0.08
```

You can use the `--print-voters` options to, instead, print all the voters:

```
$ python3 simulate.py data/config-single-precinct-0.json --print-voters

PRECINCT 'Downtown'
Arrival Time   Voting Duration   Start Time    Departure Time
      0.23         2.11             0.23             2.34

     18.49         2.00            18.49            20.49

     20.11         2.00            20.49            22.49

     26.70         2.00            26.70            28.70

     29.50         2.54            29.50            32.04
```

And here is an example with multiple precincts:

```
$ python3 simulate.py data/config-multiple-precincts-1.json

PRECINCT 'Little Rodentia (1 booth)'
- 10 voters voted.
- Polls closed at 60.0 and last voter departed at 66.12.
- Avg wait time: 8.45

PRECINCT 'Little Rodentia (2 booths)'
- 10 voters voted.
- Polls closed at 60.0 and last voter departed at 53.48.
- Avg wait time: 0.16
```

Notice how the results make sense intuitively: with more booths, it is possible to accommodate new voters sooner, which means waiting times will go down.

Task 2: Finding the average waiting time of a precinct

At this point, we have the ability to simulate precincts under a variety of parameters. One interesting parameter to tweak is the percentage of straight-ticket voters since it has a direct effect on voting durations, and thus waiting times.

In this task, you will implement a function that, given a single precinct and a percentage of straight-ticket voters, computes the average waiting time when simulating that precinct with that percentage of straight-ticket voters:

```
def find_avg_wait_time(precinct, percent_straight_ticket, ntrials, initial_seed=0):
```

Where:

- `precinct` is a precinct dictionary. You will use this to construct a `Precinct` object.
- `percent_straight_ticket` is the percentage of straight ticket voters (written as a number between 0 and 1, inclusive) to use in the simulation. Notice that this will override whatever value is specified in the `precinct` dictionary. However **do not** modify the `percent_straight_ticket` field of the `precinct` dictionary.
- `ntrials` is a number of trials (this is explained below).
- `initial_seed` is an initial seed (also explained below).

Given a single simulation of a precinct, computing the average waiting time of the voters is not difficult. In fact, if you look at the code we provide (see the `cmd` function in `simulate.py`), you'll find some code within it to do just that! However, as with any simulation, it is unreasonable to accept the result of a single simulation (or "trial"). So, instead you will simulate the precinct `ntrials` times, compute the mean wait time for each trial, sort the resulting mean wait times, and return the median value (which, for simplicity, we will define simply as element `ntrials//2` in the sorted list of average wait times).

In this task, you will need to pass a different random seed to `simulate` in each trial. You will do so by passing `initial_seed` to the first trial, and then, before each subsequent trial, incrementing the seed by one.

To test your implementation of the function, you can call the function from IPython. For example:

```
In [1]: %load_ext autoreload

In [2]: %autoreload 2

In [3]: import util

In [4]: import simulate

In [5]: precincts, seed = util.load_precincts("data/config-single-precinct-3.json")

In [6]: p = precincts[0]

In [7]: simulate.find_avg_wait_time(p, percent_straight_ticket=0.1, ntrials=20, initial_se
Out[7]: 97.2304785523948

In [8]: simulate.find_avg_wait_time(p, percent_straight_ticket=0.5, ntrials=20, initial_se
Out[8]: 5.236052395682352
```

You can also run the automated tests for this function like this:

```
$ py.test -xv -k avg
```

Task 3: Bounding the average waiting time

If split-ticket voters take longer on average to vote than straight-ticket voters, we can expect waiting times to increase with the percentage of split-ticket voters. If enough voters vote a split-ticket, there could be a point at which the average waiting time at a precinct is so large that it deters voters from voting. Your goal in this task compute that point.

More precisely, in this task you will write code to answer the following question: "Given a target waiting time W, what is the smallest percentage of split-ticket voters that will result in an average waiting time greater than W?" You can think of W as the longest acceptable waiting time.

You will write the following function to answer this question:

```
def find_percent_split_ticket(precinct, target_wait_time, ntrials, seed=0):
```

Where:

- `precinct` is a precinct dictionary. You will use this to construct a `Precinct` object.
- `target_wait_time` is W as defined above, the longest acceptable waiting time.
- `ntrials` is the number of trials to run when finding the average waiting time of a precinct.
- `seed` is a random seed.

Your function will do a simple linear search: you will first simulate the precinct with 0% split-ticket voters and check whether the average waiting time of the voters (as produced by `find_avg_wait_time` from Task 2) is strictly above `target_wait_time`. If it isn't, you will simulate the precinct with 10% split-ticket voters, and check the average waiting time, and so on in 10% increments until you find the first percentage that achieves an average waiting time larger than `target_wait_time`. This is the first percentage that achieves an unacceptably long waiting time. The 10% increment is a fixed amount, not a parameter to this function.

Careful! In this function, you want to find the percentage of *split-ticket* voters as described above. However, `find_avg_wait_time` takes the percentage of *straight-ticket* voters as a parameter. You will need to perform a transformation from percent split-ticket to percent straight-ticket in order to utilize `find_avg_wait_time`.

Take into account that it could also be the case that the average waiting time is less than the provided `target_wait_time` even if all voters are split-ticket voters. This happens when we reach 100% split-ticket voters and the average wait time is still acceptable. In this case, we say that `target_wait_time` is always feasible.

So, the result of this search will be a tuple with two values: the percentage of split-ticket voters (as a number between 0 and 1, inclusive), and the average waiting time with that percentage. If the provided target waiting time is always feasible, return `(1, None)`.

To test your implementation of the function, you can call the function from IPython. For example:

```
In [1]: %load_ext autoreload

In [2]: %autoreload 2

In [3]: import util

In [4]: import simulate

In [5]: precincts, seed = util.load_precincts("data/config-single-precinct-3.json")

In [6]: p = precincts[0]

In [7]: simulate.find_percent_split_ticket(p, ntrials=20, target_wait_time=10, seed=seed)
Out[7]: (0.6, 11.665725140016315)

In [8]: simulate.find_percent_split_ticket(p, ntrials=20, target_wait_time=160, seed=seed)
Out[8]: (1, None)
```

You can also run the function from the command-line by using the `--target-wait-time` option:

```
$ python3 simulate.py data/config-single-precinct-3.json --target-wait-time 10
Precinct 'Sahara Square' exceeds average waiting time of 11.67 with 60.0 percent split-tic

$ python3 simulate.py data/config-single-precinct-3.json --target-wait-time 160.0
Waiting times are always below 160.00 in precinct 'Sahara Square'
```

Note: The command-line tool sets the value of `ntrials` to `20`.

And, finally, you can run the automated tests for this function like this:

```
$ py.test -xv -k find
```

Floating point errors

Throughout this assignment, we've been using decimal values to represent percentages (e.g., 0.3 for 30%). These non-integer values are called *floating point numbers* in computer science (float in Python). Since computers store numbers in binary, floats cannot always be represented exactly on a computer. Furthermore, per-

forming arithmetic with floats can incur small errors that you might see while you're working on your code. For example, 3*0.1 should evaluate exactly to 0.3, but look at what happens in IPython:

```
In [1]: 3*0.1
Out[1]: 0.30000000000000004
```

If you see strange floating point numbers while you work through this task, inexact floating-point arithmetic is likely the culprit.

Unexpected results

As you play around with the data, you might find some unexpected results. It is possible, for example, for the average waiting time to *decrease* with more split-ticket voters in some simulations. While the average waiting time should increase on average, it's easy to find outliers, especially if we simulate a precinct with a small number of voters or if we use few trials.

Grading

Programming assignments will be graded according to a general rubric. Specifically, we will assign points for completeness, correctness, design, and style. (For more details on the categories, see our PA Rubric page (../rubric.html).)

The exact weights for each category will vary from one assignment to another. For this assignment, the weights will be:

- **Completeness:** 50%
- **Correctness:** 10%
- **Design:** 30%
- **Style:** 10%

The completeness part of your score will be determined using automated tests. To get your score for the automated tests, simply run the grader script, as described in our Testing Your Code (../../resources/testing.html#testing-your-code) page.

Design is an important part of this assignment, because you have to design the `Voter` and `VotingBooths` classes. There are a few things we will be looking out for in your design:

- `Precinct` is the only class that is allowed to construct `Voter` objects. Creating `Voter` objects anywhere else typically denotes a bad design.
- This assignment does not require implementing a class to model an *individual* voting booth. If you do, you are likely overthinking how to implement a precinct.
- Your `find_avg_wait_time` function (Task 2), and `find_percent_split_ticket` function (Task 3) should be fairly short and straightforward, with absolutely no simulation logic in them (which should, instead, be in your `Precinct` class). If these functions become long and convoluted, it may be because your classes are not yet well designed.

You must include header comments in all the methods you implement. You do not need to include a docstring for a class (but it certainly doesn't hurt to do so).

Cleaning up

Before you submit your final solution, you should, remove

- any `print` statements that you added for debugging purposes and
- all in-line comments of the form: "YOUR CODE HERE" and "REPLACE …"

Also, check your code against the style guide. Did you use good variable names? Do you have any lines that are too long, etc.

Make sure you have included header comments, that is, the triple-quote strings that describe the purpose, inputs, and return values of each function, for every function you have written.

As you clean up, you should periodically save your file and run your code through the tests to make sure that you have not broken it in the process.

Submission

You must submit your work through Gradescope (linked from our Canvas site). In the "Programming Assignment #4" assignment, simply upload file `simulate.py` (do not upload any other file!). Please note:

- You are allowed to make as many submissions as you want before the deadline.
- For students working in a pair, one student should upload the pair's solution and use GradeScope's mechanism for adding group members (https://help.gradescope.com/article/m5qz2xsnjy-student-add-group-members) to add the second person in the pair.
- Please make sure you have read and understood our Late Submission Policy (../../syllabus.html#late-submissions)
- Your completeness score is determined solely based on the automated tests, but we may adjust your score if you attempt to pass tests by rote (e.g., by writing code that hard-codes the expected output for each possible test input).
- Gradescope will report the test score it obtains when running your code. If there is a discrepancy between the score you get when running our grader script, and the score reported by Gradescope, please let us know so we can take a look at it.

Back to top