
So You Think You Can Program An Elevator

Many of us ride elevators every day. We feel like we understand how they work, how they decide where to go. If you were asked to put it into words, you might say that an elevator goes wherever it's told, and in doing so goes as far in one direction as it can before turning around. Sounds simple, right? Can you put it into code?

In this challenge, you are asked to implement the business logic for a simplified elevator model in Python. We'll ignore a lot of what goes into a real world elevator, like physics, maintenance overrides, and optimizations for traffic patterns. All you are asked to do is to decide whether the elevator should go up, go down, or stop.

How does the challenge work? The simulator and test harness are laid out in this document, followed by several examples. All of this can be run in an actual Python interpreter using Python's built-in `doctest` functionality, which extracts the code in this document and runs it.

A naive implementation of the business logic is provided in the `elevator.py` file in this project. If you run `doctest` using the provided implementation, several examples fail to produce the expected output. Your challenge is to fix that implementation until all of the examples pass.

Open a pull request with your solution. Good luck! Have fun!

Test Harness

Like all elevators, ours can go up and down. We define constants for these. The elevator also happens to be in a building with six floors.

```
1 >>> UP = 1
2 >>> DOWN = 2
3 >>> FLOOR_COUNT = 6
```

We will make an `Elevator` class that simulates an elevator. It will delegate to another class which contains the elevator business logic, i.e. deciding what the elevator should do. Your challenge is to implement this business logic class.

User actions

A user can interact with the elevator in two ways. She can call the elevator by pressing the up or down button on any floor, and she can select a destination floor by pressing the button for that floor on the panel in the elevator. Both of these actions are passed straight through to the logic delegate.

```
1 >>> class Elevator(object):
2 ...     def call(self, floor, direction):
3 ...         self._logic_delegate.on_called(floor, direction)
4 ...
5 ...     def select_floor(self, floor):
6 ...         self._logic_delegate.on_floor_selected(floor)
```

Elevator actions

The logic delegate can respond by setting the elevator to move up, move down, or stop. It can also read the current floor and movement direction of the elevator. These actions are accessed through `Callbacks`, a mediator provided by the `Elevator` class to the logic delegate.

```
1 >>> class Elevator(Elevator):
2 ...     def __init__(self, logic_delegate, starting_floor=1):
3 ...         self._current_floor = starting_floor
4 ...         print "%s..." % starting_floor,
5 ...         self._motor_direction = None
6 ...         self._logic_delegate = logic_delegate
7 ...         self._logic_delegate.callbacks = self.Callbacks(self)
8 ...
9 ...     class Callbacks(object):
10 ...         def __init__(self, outer):
11 ...             self._outer = outer
12 ...
13 ...         @property
14 ...         def current_floor(self):
15 ...             return self._outer._current_floor
16 ...
17 ...         @property
18 ...         def motor_direction(self):
19 ...             return self._outer._motor_direction
20 ...
21 ...         @motor_direction.setter
22 ...         def motor_direction(self, direction):
23 ...             self._outer._motor_direction = direction
```

Simulation

The simulation runs in steps. Each time step consists of the elevator moving a single floor, or pausing at a floor. Either way, the business logic delegate gets notified. Along the way, we print out the movements of the elevator so that we can keep track of it. We also define a few helper methods that advance the simulation to points of interest, for ease of testing.

```
1 >>> class Elevator(Elevator):
```

```
2 ...     def step(self):
3 ...         delta = 0
4 ...         if self._motor_direction == UP: delta = 1
5 ...         elif self._motor_direction == DOWN: delta = -1
6 ...
7 ...         if delta:
8 ...             self._current_floor = self._current_floor + delta
9 ...             print "%s..." % self._current_floor,
10 ...             self._logic_delegate.on_floor_changed()
11 ...         else:
12 ...             self._logic_delegate.on_ready()
13 ...
14 ...         assert self._current_floor >= 1
15 ...         assert self._current_floor <= FLOOR_COUNT
16 ...
17 ...     def run_until_stopped(self):
18 ...         self.step()
19 ...         while self._motor_direction is not None: self.step()
20 ...
21 ...     def run_until_floor(self, floor):
22 ...         for i in range(100):
23 ...             self.step()
24 ...             if self._current_floor == floor: break
25 ...         else: assert False
```

That's it for the framework.

Business Logic

As for the business logic, an example implementation is provided in the `elevator.py` file in this project.

```
1 >>> from elevator import ElevatorLogic
```

As provided, it doesn't pass the tests in this document. Your challenge is to fix it so that it does. To run the tests, run this in your shell:

```
1 python -m doctest -v README.md
```

With the correct business logic, here's how the elevator should behave:

Basic usage

Make an elevator. It starts at the first floor.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
```

Somebody on the fifth floor wants to go down.

```
1 >>> e.call(5, DOWN)
```

Keep in mind that the simulation won't actually advance until we call `step` or one of the `run_until_*` methods.

```
1 >>> e.run_until_stopped()
2 2... 3... 4... 5...
```

The elevator went up to the fifth floor. A passenger boards and wants to go to the first floor.

```
1 >>> e.select_floor(1)
```

Also, somebody on the third floor wants to go down.

```
1 >>> e.call(3, DOWN)
```

Even though the first floor was selected first, the elevator services the call at the third floor...

```
1 >>> e.run_until_stopped()
2 4... 3...
```

...before going to the first floor.

```
1 >>> e.run_until_stopped()
2 2... 1...
```

Directionality

Elevators want to keep going in the same direction. An elevator will serve as many requests in one direction as it can before going the other way. For example, if an elevator is going up, it won't stop to pick up passengers who want to go down until it's done with everything that requires it to go up.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(2, DOWN)
4 >>> e.select_floor(5)
```

Even though the elevator was called at the second floor first, it will service the fifth floor...

```
1 >>> e.run_until_stopped()
2 2... 3... 4... 5...
```

...before coming back down for the second floor.

```
1 >>> e.run_until_stopped()
```

```
2 4... 3... 2...
```

In fact, if a passenger tries to select a floor that contradicts the current direction of the elevator, that selection is ignored entirely. You've probably seen this before. You call the elevator to go down. The elevator shows up, and you board, not realizing that it's still going up. You select a lower floor. The elevator ignores you.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.select_floor(3)
4 >>> e.select_floor(5)
5 >>> e.run_until_stopped()
6 2... 3...
7 >>> e.select_floor(2)
```

At this point the elevator is at the third floor. It's not finished going up because it's wanted at the fifth floor. Therefore, selecting the second floor goes against the current direction, so that request is ignored.

```
1 >>> e.run_until_stopped()
2 4... 5...
3 >>> e.run_until_stopped() # nothing happens, because e.select_floor(2)
                             was ignored
```

Now it's done going up, so you can select the second floor.

```
1 >>> e.select_floor(2)
2 >>> e.run_until_stopped()
3 4... 3... 2...
```

Changing direction

The process of switching directions is a bit tricky. Normally, if an elevator going up stops at a floor and there are no more requests at higher floors, the elevator is free to switch directions right away. However, if the elevator was called to that floor by a user indicating that she wants to go up, the elevator is bound to consider itself going up.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(2, DOWN)
4 >>> e.call(4, UP)
5 >>> e.run_until_stopped()
6 2... 3... 4...
7 >>> e.select_floor(5)
8 >>> e.run_until_stopped()
9 5...
```

```
10 >>> e.run_until_stopped()
11 4... 3... 2...
```

If nobody wants to go further up though, the elevator can turn around.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(2, DOWN)
4 >>> e.call(4, UP)
5 >>> e.run_until_stopped()
6 2... 3... 4...
7 >>> e.run_until_stopped()
8 3... 2...
```

If the elevator is called in both directions at that floor, it must wait once for each direction. You may have seen this too. Some elevators will close their doors and reopen them to indicate that they have changed direction.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.select_floor(5)
4 >>> e.call(5, UP)
5 >>> e.call(5, DOWN)
6 >>> e.run_until_stopped()
7 2... 3... 4... 5...
```

Here, the elevator considers itself to be going up, as it favors continuing in the direction it came from.

```
1 >>> e.select_floor(4) # ignored
2 >>> e.run_until_stopped()
```

Since nothing caused the elevator to move further up, it now waits for requests that cause it to move down.

```
1 >>> e.select_floor(6) # ignored
2 >>> e.run_until_stopped()
```

Since nothing caused the elevator to move down, the elevator now considers itself idle. It can move in either direction.

```
1 >>> e.select_floor(6)
2 >>> e.run_until_stopped()
3 6...
```

En passant

Keep in mind that a user could call the elevator or select a floor at any time. The elevator need not be stopped. If the elevator is called or a floor is selected before it has reached the floor in question, then the request should be serviced.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.select_floor(6)
4 >>> e.run_until_floor(2) # elevator is not stopped
5 2...
6 >>> e.select_floor(3)
7 >>> e.run_until_stopped() # stops for above
8 3...
9 >>> e.run_until_floor(4)
10 4...
11 >>> e.call(5, UP)
12 >>> e.run_until_stopped() # stops for above
13 5...
```

On the other hand, if the elevator is already at, or has passed the floor in question, then the request should be treated like a request in the wrong direction. That is to say, a call is serviced later, and a floor selection is ignored.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.select_floor(5)
4 >>> e.run_until_floor(2)
5 2...
6 >>> e.call(2, UP) # missed the boat, come back later
7 >>> e.step() # doesn't stop
8 3...
9 >>> e.select_floor(3) # missed the boat, ignored
10 >>> e.step() # doesn't stop
11 4...
12 >>> e.run_until_stopped() # service e.select_floor(5)
13 5...
14 >>> e.run_until_stopped() # service e.call(2, UP)
15 4... 3... 2...
```

Fuzz testing

No amount of legal moves should compel the elevator to enter an illegal state. Here, we run a bunch of random requests against the simulator to make sure that no asserts are triggered.

```
1 >>> import random
2 >>> e = Elevator(ElevatorLogic())
```

```
3 1...
4 >>> try: print '-', # doctest:+ELLIPSIS
5 ... finally:
6 ...     for i in range(1000000):
7 ...         r = random.randrange(6)
8 ...         if r == 0: e.call(
9 ...             random.randrange(FLOOR_COUNT) + 1,
10 ...             random.choice((UP, DOWN)))
11 ...         elif r == 1: e.select_floor(random.randrange(FLOOR_COUNT) +
12 ...             1)
13 ...         else: e.step()
13 - ...
```

More Examples

The rest of these examples may be useful for catching bugs. They are meant to be run via doctest, so they may not be very interesting to read through.

An elevator is called but nobody boards. It goes idle.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(5, UP)
4 >>> e.run_until_stopped()
5 2... 3... 4... 5...
6 >>> e.run_until_stopped()
7 >>> e.run_until_stopped()
```

The elevator is called at two different floors.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(3, UP)
4 >>> e.call(5, UP)
5 >>> e.run_until_stopped()
6 2... 3...
7 >>> e.run_until_stopped()
8 4... 5...
```

Like above, but called in reverse order.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(5, UP)
4 >>> e.call(3, UP)
5 >>> e.run_until_stopped()
6 2... 3...
7 >>> e.run_until_stopped()
8 4... 5...
```

The elevator is called at two different floors, but going the other direction.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(3, DOWN)
4 >>> e.call(5, DOWN)
5 >>> e.run_until_stopped()
6 2... 3... 4... 5...
7 >>> e.run_until_stopped()
8 4... 3...
```

The elevator is called at two different floors, going in opposite directions.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(3, UP)
4 >>> e.call(5, DOWN)
5 >>> e.run_until_stopped()
6 2... 3...
7 >>> e.run_until_stopped()
8 4... 5...
```

Like above, but with directions reversed.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(3, DOWN)
4 >>> e.call(5, UP)
5 >>> e.run_until_stopped()
6 2... 3... 4... 5...
7 >>> e.run_until_stopped()
8 4... 3...
```

The elevator is called at two different floors, one above the current floor and one below. It first goes to the floor where it was called first.

```
1 >>> e = Elevator(ElevatorLogic(), 3)
2 3...
3 >>> e.call(2, UP)
4 >>> e.call(4, UP)
5 >>> e.run_until_stopped()
6 2...
7 >>> e.run_until_stopped()
8 3... 4...
```

Like above, but called in reverse order.

```
1 >>> e = Elevator(ElevatorLogic(), 3)
2 3...
3 >>> e.call(4, UP)
4 >>> e.call(2, UP)
```

```
5 >>> e.run_until_stopped()
6 4...
7 >>> e.run_until_stopped()
8 3... 2...
```

The elevator is called while it's already moving.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(5, UP)
4 >>> e.run_until_floor(2)
5 2...
6 >>> e.call(3, UP)
7 >>> e.run_until_stopped()
8 3...
9 >>> e.run_until_stopped()
10 4... 5...
```

If the elevator is already at, or has passed the floor where it was called, it comes back later.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(5, UP)
4 >>> e.run_until_floor(3)
5 2... 3...
6 >>> e.call(3, UP)
7 >>> e.run_until_stopped()
8 4... 5...
9 >>> e.run_until_stopped()
10 4... 3...
```

Two floors are selected.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.select_floor(3)
4 >>> e.select_floor(5)
5 >>> e.run_until_stopped()
6 2... 3...
7 >>> e.run_until_stopped()
8 4... 5...
```

Like above, but selected in reverse order.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.select_floor(5)
4 >>> e.select_floor(3)
5 >>> e.run_until_stopped()
6 2... 3...
7 >>> e.run_until_stopped()
```

```
8 4... 5...
```

Two floors are selected, one above the current floor and one below. The first selection sets the direction, so the second one is completely ignored.

```
1 >>> e = Elevator(ElevatorLogic(), 3)
2 3...
3 >>> e.select_floor(2)
4 >>> e.select_floor(4)
5 >>> e.run_until_stopped()
6 2...
7 >>> e.run_until_stopped()
```

Like above, but selected in reverse order.

```
1 >>> e = Elevator(ElevatorLogic(), 3)
2 3...
3 >>> e.select_floor(4)
4 >>> e.select_floor(2)
5 >>> e.run_until_stopped()
6 4...
7 >>> e.run_until_stopped()
```

If the elevator is called to a floor going up, it should ignore a request to go down.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(5, UP)
4 >>> e.run_until_stopped()
5 2... 3... 4... 5...
6 >>> e.select_floor(6)
7 >>> e.select_floor(4)
8 >>> e.run_until_stopped()
9 6...
10 >>> e.run_until_stopped()
```

Like above, but going in other direction.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(5, DOWN)
4 >>> e.run_until_stopped()
5 2... 3... 4... 5...
6 >>> e.select_floor(6)
7 >>> e.select_floor(4)
8 >>> e.run_until_stopped()
9 4...
10 >>> e.run_until_stopped()
```

Elevator is called to a floor and a passenger also selects the same floor. The elevator should not go

back to that floor twice.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(5, DOWN)
4 >>> e.select_floor(5)
5 >>> e.run_until_stopped()
6 2... 3... 4... 5...
7 >>> e.select_floor(4)
8 >>> e.run_until_stopped()
9 4...
10 >>> e.run_until_stopped()
```

Similarly, if the elevator is called at a floor where it is stopped, it should not go back later.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(3, UP)
4 >>> e.run_until_stopped()
5 2... 3...
6 >>> e.call(3, UP)
7 >>> e.call(5, DOWN)
8 >>> e.run_until_stopped()
9 4... 5...
10 >>> e.run_until_stopped()
```

Elevator is ready to change direction, new call causes it to keep going in same direction.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.call(2, DOWN)
4 >>> e.call(4, UP)
5 >>> e.run_until_stopped()
6 2... 3... 4...
7 >>> e.call(5, DOWN) # It's not too late.
8 >>> e.run_until_stopped()
9 5...
10 >>> e.run_until_stopped()
11 4... 3... 2...
```

When changing directions, wait one step to clear current direction.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.select_floor(5)
4 >>> e.call(5, UP)
5 >>> e.call(5, DOWN)
6 >>> e.run_until_stopped()
7 2... 3... 4... 5...
8 >>> e.select_floor(4) # ignored
9 >>> e.run_until_stopped()
```

```
10 >>> e.select_floor(6) # ignored
11 >>> e.select_floor(4)
12 >>> e.run_until_stopped()
13 4...
14 >>> e.run_until_stopped()
```

Like above, but going in other direction.

```
1 >>> e = Elevator(ElevatorLogic(), 6)
2 6...
3 >>> e.select_floor(2)
4 >>> e.call(2, UP)
5 >>> e.call(2, DOWN)
6 >>> e.run_until_stopped()
7 5... 4... 3... 2...
8 >>> e.select_floor(3) # ignored
9 >>> e.run_until_stopped()
10 >>> e.select_floor(1) # ignored
11 >>> e.select_floor(3)
12 >>> e.run_until_stopped()
13 3...
14 >>> e.run_until_stopped()
```

If other direction is not cleared, come back.

```
1 >>> e = Elevator(ElevatorLogic())
2 1...
3 >>> e.select_floor(5)
4 >>> e.call(5, UP)
5 >>> e.call(5, DOWN)
6 >>> e.run_until_stopped()
7 2... 3... 4... 5...
8 >>> e.select_floor(6)
9 >>> e.run_until_stopped()
10 6...
11 >>> e.run_until_stopped()
12 5...
13 >>> e.select_floor(6) # ignored
14 >>> e.select_floor(4)
15 >>> e.run_until_stopped()
16 4...
17 >>> e.run_until_stopped()
```