Chapter 1. Introducing Asyncio

My story is a lot like yours, only more interesting 'cause it involves robots.

Bender, Futurama episode "30% Iron Chef"

The most common questions I receive about Asyncio in Python 3 are these: "What is it, and what do I do with it?" The following story provides a backdrop for answering these questions. The central focus of Asyncio is about how to best perform multiple concurrent tasks at the same time. And not just any sort of tasks, but specifically tasks that involve waiting periods. The key insight required with this style of programming is that while you wait for *this* task to complete, work on *other* tasks can be performed.

The Restaurant of ThreadBots

The year is 2051, and you find yourself in the restaurant business. Automation, largely by robot workers, powers most of the economy, but it turns out that humans still enjoy going out to eat once in a while. In your restaurant, all the employees are robots—humanoid, of course, but unmistakably robots. The most successful manufacturer of robots is of course Threading Inc., and robot workers from this company have come to be called "ThreadBots."

Except for this small robotic detail, your restaurant looks and operates like one of those old-time restaurants from, say, 2018. Your guests will be looking for that vintage experience. They want fresh food prepared from scratch. They want to sit at tables. They want to wait for their meals—but only a little. They want to pay at the end, and they sometimes even want to leave a tip, for old-time's sake, of course.

Naturally, being new to the robotic restaurant business, you do what every other restaurateur does, and you hire a small fleet of robots: one to greet new diners at the front desk (GreetBot), one to wait tables and take orders (WaitBot), one to do the cooking (ChefBot), and one to manage the bar (WineBot).

Hungry diners will arrive at the front desk and be welcomed by GreetBot, your front-of-house ThreadBot. They are then directed to a table, and once they are seated, WaitBot will take their order. Then, WaitBot will take that order to the kitchen on a slip of paper (because you want to preserve that old-time experience, remember?). The ChefBot will look up the order on the slip and begin preparing the food. The WaitBot will periodically check whether the food is ready, and when it is, will immediately take the dish to the customer's table. When the guests are ready to leave, they return to GreetBot who calculates the bill, and graciously wishes them a pleasant evening.

You open your restaurant, and exactly as you had anticipated, your menu is a hit and you soon grow a large customer base. Your robot employees do exactly what they're told, and they are perfectly good at the tasks you assign them. Everything is going really well, and you really couldn't be happier.

Over time, however, you do begin to notice some problems. Oh, it's nothing truly serious. Just a few things that seem to go wrong. Every other robotic restaurant owner seems to have similar niggling glitches. It is a little worrying that these problems seem to get worse the more successful you become.

Though rare, there are the occasional collisions that are very unsettling: sometimes, when a plate of food is ready in the kitchen, the WaitBot will grab it *before* the ChefBot has even let go of the plate. This usually shatters the plate and leaves a big mess. ChefBot cleans it up of course, but still, you'd think that these top-notch robots would know how to be a bit more synchronized with each other. This happens at the bar too: sometimes WineBot will place a new drink order on the bar and WaitBot will grab it before WineBot has let go, resulting in broken glass and spilled Nederburg Cabernet Sauvignon.

Sometimes GreetBot will seat new diners at exactly the same moment that WaitBot has decided to clean what it thought was an empty table. It's pretty awkward for the diners. You've tried adding delay logic to the WaitBot's cleaning function, or delays to the GreetBot's seating function, but these don't really help, because the collisions still occur. But at least these events are rare.

Well, these used to be rare events. Your restaurant got so popular that you've had to hire a few more ThreadBots. For very busy Friday and Saturday evenings, you've had to add a second GreetBot and two extra WaitBots. Unfortunately the hiring contracts for ThreadBots mean that you have to hire for a whole week, so this effectively means that for most of the quiet part of the week, you're carrying three extra ThreadBots that you don't really need.

The other resource problem, in addition to the extra cost, of course, is that it's more work for you to deal with these extra ThreadBots. It was fine to keep tabs on just four bots, but now you're up to seven. Keeping track of seven ThreadBots is a lot more work, and because your restaurant keeps getting more and more famous, you become worried about taking on even more ThreadBots. It's going to become a full-time job just to keep track of what each ThreadBot is doing. Another thing: these extra ThreadBots are using up a lot more space inside your restaurant. It's becoming a tight squeeze for your customers, what with all these robots zipping around. You're worried that if you need to add even more bots, this space problem is going to get even worse. You want to use the space in your restaurant for customers, not ThreadBots.

The collisions have also become worse since you added more ThreadBots. Now, sometimes two WaitBots take the exact same order from the same table at the same time. It's as if they both noticed that the table was ready to order and moved in to take it, without noticing that the other WaitBot was doing the exact same thing. As you can imagine, this results in duplicated food orders, which causes extra load on the kitchen and increases the chance of collisions when picking up the ready plates. You're worried that if you added more WaitBots, this problem might get worse.

Time passes.

Then, during one very, very busy Friday night service, you have a singular moment of clarity: time slows, lucidity overwhelms you, and you see a snapshot of your restaurant frozen in time. *My ThreadBots are doing nothing!* Not really nothing, to be fair, but they're just…waiting.

Each of your three WaitBots at different tables is waiting for one of the diners at their table to give their order. The WineBot already prepared

17 drinks, which are now waiting to be collected (it took only a few seconds), and is now waiting for a new drink order. One of the GreetBots has greeted a new party of guests, told them they need to wait a minute to be seated, and is waiting for the guest to respond. The other hostbot, now processing a credit card payment for another guest that is leaving, is waiting for confirmation on the payment gateway device. Even the ChefBot, who is currently cooking 35 meals, is not actually doing anything at this moment, but is simply waiting for one of the meals to complete cooking so that it can be plated up and handed over to a WaitBot.

You realize that even though your restaurant is now full of ThreadBots, and you're even considering getting more (with all the problems that entails), the ones that you currently have are not even being fully utilized.

The moment passes, but not the realization. On Sunday, you add a data collection module to your ThreadBots. For each threadbot, you're measuring how much time is spent waiting and how much is spent actively doing work. Over the course of the following week, the data is collected and then on Sunday evening you analyze the results. It turns out that even when your restaurant is at full capacity, the most hardworking ThreadBot is idle about 98% of the time. The ThreadBots are so enormously efficient that they can perform any task in fractions of a second.

As an entrepreneur, this inefficiency really bugs you. You know that every other robotic restaurant owner is running their business the same as you, with many of the same problems. But, you think, slamming your fist on your desk, "There must be a better way."

So the very next day, which is a quiet Monday, you try something very bold: you program a single ThreadBot to do all the tasks. Every time it begins to wait, even for a second, instead of waiting, the ThreadBot switches to the next task, whatever it may be in the entire restaurant. It sounds incredible at face value, only one ThreadBot doing the work of all the others, but you're confident that your calculations are correct. And besides, Monday is a very quiet day; even if something goes wrong, the impact will be small. For this new project, you call the bot "loopbot" because it will loop over all the jobs in the restaurant.

The programming was more difficult than usual. It isn't just that you had to program one ThreadBot with all the different tasks; you also had to program some of the logic of when to switch between tasks. But by this stage, you've had a lot of experience with programming these ThreadBots so you manage to get it done.

You watch your loopbot like a hawk. It moves between stations in fractions of a second, checking whether there is work to be done. Not long after opening, the first guest arrives at the front desk. The loopbot shows up almost immediately, and asks whether the guest would like a table near the window or near the bar. But then, as the loopbot begins to wait, its programming tells it to switch to the next task, and it whizzes off. This seems like a dreadful error, but then you see that as the guest begins to say "window please," the loopbot is back. It receives the answer and directs the guest to table 42. And off it goes again, checking for drinks orders, food orders, table cleanup, and arriving guests, over and over again.

Late Monday evening, you congratulate yourself on a remarkable success. You check the data collection module on the loopbot, and it confirms that even with a single ThreadBot doing the work of seven, the idle time was still around 97%. This result gives you the confidence to continue the experiment all through the rest of the week.

As the busy Friday service approaches, you reflect on the great success of your experiment. For service during a normal working week, you can easily manage the workload with a single loopbot. And there is another thing you've noticed: you don't see any more collisions. It makes sense: since there is only one loopbot, it cannot get confused with itself. No more duplicate orders going to the kitchen, and no more confusion about when to grab a plate or drink.

Friday evening service begins, and as you had hoped, the single ThreadBot keeps up with all the customers and tasks, and service is proceeding even better than before. You imagine that you can take on even more customers now, and you don't have to worry about having to bring on more ThreadBots. You think of all the money you're going to save.

Unfortunately, something goes wrong: one of the meals, an intricate souffle, has flopped. This has never happened before in your restaurant. You begin to study the loopbot more closely. It turns out that at one of your tables, there is a very chatty guest. This guest has come to your restaurant alone, and keeps trying to make conversation with your loopbot, even sometimes holding your loopbot by the hand. When this happens, your loopbot is unable to dash off and attend to the ever-growing list of tasks elsewhere in your restaurant. This is why the kitchen produced its first flopped souffle. Your loopbot was unable to make it back to the kitchen to remove a souffle from the oven, because it was held up by a guest.

Friday service finishes, and you head home to reflect on what you have learned. It's true that the loopbot could still do all the work that was required on a busy Friday service; but on the other hand, your kitchen produced its very first spoiled meal, something that had never happened before. Chatty guests used to keep WaitBots busy all the time, but that never affected the kitchen service at all.

All things considered, you decide, it is still better to continue using a single loopbot. Those worrying collisions no longer occur, and there is much more space in your restaurant, space that you can use for more customers. But you realize something profound about the loopbot: it can only be effective if every task is short; or at least can be performed in a very short period of time. If any activity keeps the loopbot busy for too long, other tasks will begin to suffer neglect.

It is difficult to know in advance which tasks may take too much time. What if a guest orders a cocktail that requires very intricate preparation, much more than usual? What if a guest wants to complain about a meal at the front desk, refuses to pay, and grabs the loopbot by the arm, preventing it from task-switching? You decide that instead of figuring out all of these issues up front, it is better to continue with the loopbot, record as much information as possible, and deal with any problems later as they arise.

More time passes.

Gradually, other restaurant owners notice your operation, and eventually they figure out that they too can get by, and even thrive, with only a

single ThreadBot. Word spreads. Soon every single restaurant operates in this way, and it becomes difficult to remember that robotic restaurants ever operated with multiple ThreadBots at all.

Epilogue

In our story, each of the robot workers in the restaurant is a single thread. The key observation in the story is that the nature of the work in our restaurant involves a great deal of waiting, just as `requests.get()` is waiting for a response from a server.

In a restaurant, the worker time spent waiting isn't huge when slow humans are doing manual work, but when super-efficient and quick robots are doing the work, then nearly all their time is spent waiting. With computer programming, the same is true when network programming is involved. CPUs do "work" and "wait" on network I/O. CPUs in modern computers are extremely fast—hundreds of thousands of times faster than network traffic. Thus, CPUs running networking programs spend a great deal of time waiting.

The insight in the story is that programs can be written to explicitly direct the CPU to move between work tasks as necessary. Although there is an improvement in economy (using fewer CPUs for the same work), the real advantage, compared to a threading (multi-CPU) approach, is the elimination of race conditions.

It's not all roses, however: as we found in the story, there are benefits and drawbacks to most technology solutions. The introduction of the loopbot solved a certain class of problems, but also introduced new problems—not least of which is that the restaurant owner had to learn a slightly different way of programming.