Getting Processes to Talk to Each Other

How to Train Your Robot

Chapter 3

Brandon Rohrer

How to Train Your Robot

About This Project

How to Train Your Robot is a side project I've been working on for 20 years. It's a consistent source of satisfaction. A big part of the joy is sharing my progress as I go, and who knows? maybe someone will find it useful.

This is already the third chapter. It's also the longest. If you extrapolate, chapter 17 will be 500 pages.

Up, up, and away.

Brandon
Boston, USA
November 1, 2022

Getting Processes to Talk to Each Other

Chapter 3

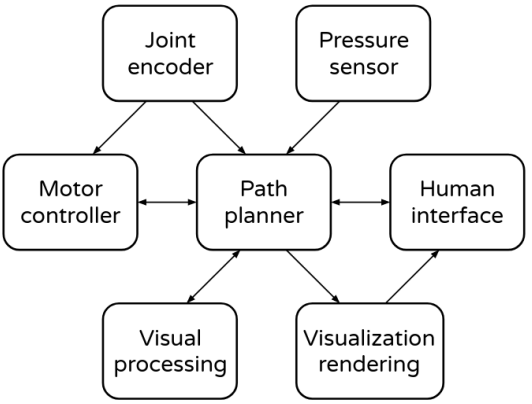
In which we teach programs to share.

We've made some good progress already on our journey toward human directed reinforcement learning. In chapter 2, we laid an important foundation block with timekeeping by building a pacemaker. In this chapter, we're going to lay another cornerstone: inter-process communication. We'll construct a way to get different programs to talk to each other that is as straightforward and as unbreakable as we can make it.

Do we really need more than one process?

It's not immediately obvious why enabling programs to talk to each other is important for doing robotics, but it turns out that it is. The nature of a robot is that there is a lot going on. Motors need controlling. Sensors need reading. Actions need planning. Visualizations need

animating. While it's theoretically possible to do all of this in a single process, breaking these tasks out into separate processes streamlines development and lets the robot harness the computing power of multiple processors.



Hypothetical process diagram for a robot system.

Also, it is often the case that processes are running on different pieces of hardware. Sensor and actuator controllers are often on the robot itself, very close to the action. A process driving the human interaction will likely sit on a laptop in front of the operator. And, depending on the nature of the planning algorithm, it might be running on the operator's laptop, a separate workstation, in the cloud, or perhaps a combination of all three.

These processes typically work at very different time scales. The planner may update several times per second. Actuator control might be adjusted dozens of

times per second. Sensors can report readings hundreds of times per second. And the human interface might be entirely asynchronous, driven by keyboard inputs whenever the human decides to make it happen. The need for working at different timescales also lends itself to having separate processes.

The ability to pass messages between processes is so critical to robots that it is the core functionality of the Robot Operating System (ROS[1]). ROS helped out a lot of robotics researchers by providing a message passing framework. It set up a reliable way for processes to send and receive messages. This has saved many collective years of grad students' time by removing the need for them to hack something together to meet that need.

In this chapter, we won't try to re-create anything of the sophistication or performance levels of ROS, but we're going to build something with a similar function. As we will see, it's not trivial to get this right. But by the time we're done, we'll have something that is good enough for our purposes–simple, serviceable, and straightforward to explain.

Why not threads?

Before we go to all the work to help processes talk to each other, it's worth asking whether there is an easier way. Each process has its own memory and resources. There is no simple way to share variables. Any interaction between processes requires the involvement of the operating system. It's all kind of a pain in the butt.

Threads are a different story. They mostly act like processes. They let you handle separate lines of computational reasoning in parallel. They don't have to operate in lockstep with each other. On top of this, they have a huge advantage: they can share memory. You can create variables that multiple threads can see and modify. It would solve all of our communications problems if we could just use threads.

Unfortunately, threads aren't the tool we're looking for. Even though they seem to run in parallel, they actually just take turns with each other really fast. They are still limited by having to take place on a single processor. This is a problem when you are trying to do a lot of computation. You could quickly run out of CPU cycles. Threads don't let you parallelize across multiple processors.

Threads also make the system fragile. If one thread crashes or freezes, the entire program ceases to function. This isn't consistent with our design goal of having a robot that can keep functioning when an individual piece fails. Threads are good for quick, lightweight tasks. They are not a good fit for the fiercely independent parallel processes we plan to construct in our robot work. The extra robustness we get from processes is definitely worth a little extra labor up front.

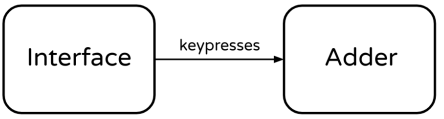
All this thread talk is exhausting. Get to the good part.

Counting concertgoers

It's time to take these abstract ideas and make them real. We're going to build a people counter. Imagine you've been asked to watch the entrance to a Billie Eilish[2] concert and count the number of people going in.

Hand tally knitting row counter.
Storye book, CC BY 3.0, via Wikimedia Commons.

You could use one of those handy little thumb clickers that lets you push once for each person and keeps a tally. But there are so many people going in so quickly that you decide you'd be better off being able to increment by several people at a time. They are flowing by fast, but you can visually pick out groups of three, four, and five, and key them in much faster than you could one at a time. And so you set out to build a better people counter.



Process diagram for a people counter.

There are two programs doing most of the work here. The first one is the keyboard interface that keeps track of each time you hit a digit. The second is an adder that sums the numerical values of all of the keys that you hit, keeps track of the total, and shows it back to you. Of course it would be straightforward to include these in the same program, but this example is just a thin pretext for building out some functionality that we are really going to appreciate later.

Install the getkey package

Even though our people counter process diagram is simple (Two boxes and one arrow! How cute!) it will be helpful to build it up methodically, one piece at a time. We'll start with the key pressing interface.

Surprisingly, there's no built-in Python function for getting individual keypresses. There is an `input()` function that reads typed input from the keyboard, but you have to hit the Enter key each time to get it to read. This requires much more coordination and attention than we want for our application. Imagine that, while counting people, someone misses the Enter key between a 2 and a 3. Then they will accidentally report 23 people, rather than 5. That's a whole mess of mistakes waiting to happen.

Python can read single keypresses, but the details of how it does this vary by operating system. To handle it correctly, Python has to look at what operating system it is using it and then adapt its behavior. It can do all of this just fine, but it's more complicated than a single built-in function.

Thankfully, enterprising developers have run into this problem before and done all of this work for us. The **getkey** package is a piece of open source software that we can download and drop into our program. It's common practice, when one discovers a gap in Python's capabilities, to write a package meeting that need and share it with the community. A popular place to host these packages is PyPI[3], the Python Package Index. It is integrated with Python's built-in package manager, **pip**, to let you download and install new packages from the command line. In my case I would use this command.

```
$ python3 -m pip install getkey

Collecting getkey
  Downloading getkey-0.6.5.tar.gz (13 kB)
  Preparing metadata (setup.py) ... done
Building wheels for collected packages: getkey
  Building wheel for getkey (setup.py) ... done
  Created wheel for getkey:
filename=getkey-0.6.5-py3-none-any.whl size=11439
sha256=135debb14bce179bbe08959e3dde229f66b7575b5b75f1db
6b865f5efada85c8
  Stored in directory:
/tmp/pip-ephem-wheel-cache-hxuahrzl/wheels/46/31/db/887
db397dcf4c0cda0fcca2918c2d2233dd2266bb7efa11dc4
Successfully built getkey
Installing collected packages: getkey
  Attempting uninstall: getkey
        Found existing installation: getkey 0.6.5
        Uninstalling getkey-0.6.5:
        Successfully uninstalled getkey-0.6.5
Successfully installed getkey-0.6.5
```

This tells pip to go search PyPI for a package named **getkey**, download it, and install it. The `python3 -m` at the beginning ensures that the package will be associated with the particular instance of Python that gets called when I type `python3` at the command line. All the output that follows is just my computer's verbose way of telling me that everything went fine.

Avoiding Python environment hell

A common and valid criticism of Python is that it can be hard to keep track of your environment. It's completely legal, and even common, to have several different

versions of Python on your computer at once. As of the day I wrote this, the most recent Python version is 3.10, although 3.11 is getting very close to being released. But there are also a handful of earlier versions all being actively maintained. At any given moment several or all of these might be installed on your computer.

It may be helpful to think of Python versions as being separate programs entirely. It's like having Chrome and Firefox and Safari all installed on your laptop. You can change your settings and add bookmarks in Chrome, but that won't affect your experience in Firefox. There are web pages that work in Firefox that won't work in Safari. They are separate programs, separate executable files in different directories, and even though their behavior is mostly similar, it can be different in important ways.

To make things more complicated, you can have more than one **environment**, that is, more than one unique collection of Python packages and settings, associated with a given Python installation. If you add a new package, it's a different environment, if you change the version of a package, it's a different environment. And very often programs that run in one environment will behave a little bit differently in another, or not run at all.

There are some handy tools for juggling environments, but they come with their own learning curves. We are going to save them for a later chapter. For the meantime, the cheat I recommend is to always tell the computer exactly which Python interpreter I want it to use. When I

type `python3` at the command line, that refers to a particular instance of Python, the same one every time.

If you are using an IDE or a notebook, you may have to do a little extra legwork to make sure you know which version of Python it's using. If you run out of patience with that approach, a convenient end run is to use your IDE as a code editor only, and run all your scripts from the command line.

Anyone with experience can tell you that this is a pretty lame answer. It's like when you go to your doctor and say "Doctor it hurts when I do this!" and she says "Well maybe stop doing that?" Yes, it removes the immediate source of pain, but doesn't do anything to treat underlying symptoms. The only excuse I have for serving such weak tea is that when treating the underlying symptoms requires a bit of surgery, sometimes a quick fix Band-Aid is OK while we get our affairs in order.

Get keypresses

Back to business–now we have the **getkey** package. It's an instant upgrade to our abilities. It's as if we dialed Tank from within the Matrix and asked him to upload a pilot program for a B-212 helicopter. Suddenly we can do things we couldn't do before.

In this case, we can get a keypress. It's not as impressive as flying a helicopter, but it's a lot more helpful for counting Billie Eilish fans. The way we invoke this is by

importing a function called getkey from the package module called getkey. With this trick we can create a sandbox to test the keypress interface.

```
from getkey import getkey

while True:
    key = getkey()
    if key in ["1", "2", "3", "4", "5", "6", "7", "8"]:
        print(f"key pressed: {key}")
```

interface_sandbox.py

The `getkey()` function sits and waits for a key to be pressed. Since we want to count concert attendees, we are only interested in non-zero digits. There is a check whether the key pressed is 1 through 8. If so, it gets acknowledged with a print statement. Our sandbox sets up an infinite loop and will keep doing this until we get bored or its battery dies.

```
$ python3 interface_sandbox.py

key pressed: 8
key pressed: 4
key pressed: 5
key pressed: 7
key pressed: 3
key pressed: 6
key pressed: 4
```

You can also try pressing other keys to verify that they get ignored.

Add numbers together

Now that we can read number keys in, the next step is to build a process that adds them together. This adder will be a little fancier than most. Not only do we want it to collect our keypresses and calculate a total, but we are going to constrain it to work periodically, on the clock, with a heartbeat.

```
from pacemaker import Pacemaker
import numpy as np

clock_freq_hz = 2
pacemaker = Pacemaker(clock_freq_hz)

ppl_count = 0
while True:
    overtime = pacemaker.beat()

    ppl = np.random.randint(9)
    ppl_count += int(ppl)
    print(f"ppl_added: {ppl},  total_ppl: {ppl_count}")
```

adder_sandbox.py

For now, we'll generate some small, random integers to be added.

```
ppl = np.random.randint(9)
```

For the heartbeat, we'll use the pacemaker method we developed in chapter 2. We've wrapped this neatly in its own module so that all we need to do now is to import the Pacemaker class,

```
from pacemaker import Pacemaker
```

initialize it with our update frequency,

```
clock_freq_hz = 2
pacemaker = Pacemaker(clock_freq_hz)
```

and have it keep pace in our main loop.

```
overtime = pacemaker.beat()
```

When we put these pieces together and run it, The adder does exactly what we hope, generating a running sum of these numbers on a regular cadence.

```
$ python3 adder_sandbox.py

ppl_added: 2,  total_ppl: 2
ppl_added: 4,  total_ppl: 6
ppl_added: 7,  total_ppl: 13
ppl_added: 6,  total_ppl: 19
ppl_added: 4,  total_ppl: 23
ppl_added: 2,  total_ppl: 25
ppl_added: 5,  total_ppl: 30
ppl_added: 5,  total_ppl: 35
```

The regular clockwork nature of this code is a little bit forced, but it will set us up well when we go to solve more complicated problems.

Our pacemaker code from chapter 2 we folded into a class.

```
import time

class Pacemaker:
    def __init__(self, clock_freq_Hz):
        self.clock_period = 1 / float(clock_freq_Hz)
        self.last_run_completed = time.monotonic()
        self.start_time = time.monotonic()
        self.i_iter = -1

    def beat(self):
        self.i_iter += 1
        end = (
            self.start_time +
            (self.i_iter + 1) * self.clock_period)
        sleep_time = end - time.monotonic()
        if sleep_time > 0:
            time.sleep(sleep_time)

        this_run_completed = time.monotonic()
        dt = (
            this_run_completed -
            self.last_run_completed)
        overtime = dt - self.clock_period
        self.last_run_completed = this_run_completed
        return overtime
```

pacemaker.py

If this is your first exposure to classes, it's OK to ignore them for now. The important bits to understand are that there is an `__init__()` function that gets called to set the pacemaker up, and then a separate `beat()` function that gets called to advance to the next time step. The advantage of having `Pacemaker` be a class is that it can

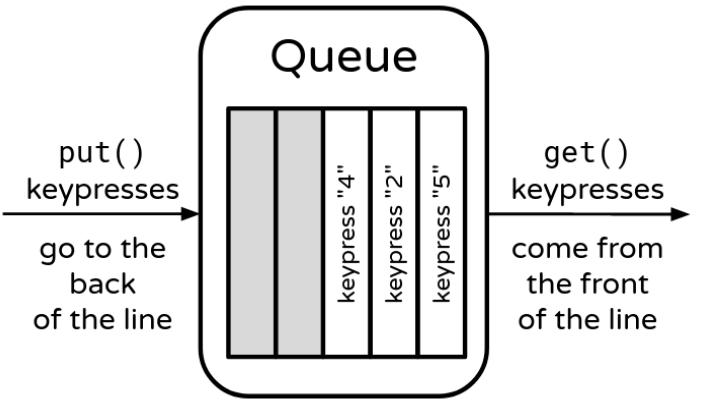
track its internal attributes, like when it started (`self.start_time`) and how many beats it has executed so far (`self.i_iter`). It hides the details so we don't have to keep track of them in the main code. It keeps our adder code cleaner.

Pass keypresses through a Queue

Now we get to the critical nugget of this chapter, connecting these two separate programs, so that they can talk, specifically so that the interface can pass keypresses to the adder.

In our process diagram, this is the arrow that connects the two. In Python, we do this by using a queue (pronounced like the letter "Q", for all us Americans who have never encountered the word at a supermarket).

A queue serves as a separate staging ground, a temporary holding pen for objects we want to pass between processes. We'll be working with a first-in, first-out (FIFO) queue.



There are two things we are going to want to do with our queue, put objects and get objects. The interface will put keypresses into the queue and the adder will get those keypresses after they happen. To see how this is implemented, let's take a look at the interface, including the queue.

```
import time
from getkey import getkey

def run(q):
    last_key = "0"
    while True:
        key = getkey()
        key_ts = time.time()

        if key in [
                "1", "2", "3", "4", "5",
                "6", "7", "8", "9"]:
            q.put((key_ts, key))
            print(
                f"ts: {key_ts}, ppl_reported: {key}")
            last_key = key

        if key == " ":
            remove_ppl = str(-int(last_key))
            q.put((key_ts, remove_ppl))
            print(
                f"ts: {key_ts}, ppl_undo: {last_key}")
            last_key = "0"
```

interface.py

The first difference we see is that we've defined a `run()` function that takes the queue variable (cleverly named `q`) as an input argument. Doing it this way is necessary for reasons we will see in a few pages. For now, we'll skip over it.

The really important bit here is where the pressed key gets put into the queue.

```
q.put((key_ts, key))
```

This function call puts whatever object it receives as an argument into the queue. In this case, we are passing a tuple containing the timestamp when the key was read, and the key that was pressed, `(key_ts, key)`.

In a nod toward the fallibility and comfort of our users, we'll also add the option to undo one keypress. If the user hits the space bar, it will undo their previous keypress by sending the negative of it.

```
if key == " ":
    remove_ppl = str(-int(last_key))
    q.put((key_ts, remove_ppl))
```

For consistency, this has to be a string, so we convert the `last_key` to an integer, take the negative, then convert it back into a string. In both cases, we also print the keypress sent and the timestamp to the console just so that we can monitor what's going on.

The calls to `q.put()` are the base of the arrow connecting the interface to the adder. To build the head of the arrow, we need to modify the adder as well.

```
import time
from pacemaker import Pacemaker

clock_freq_hz = 4
clock_period = 1 / float(clock_freq_hz)
pacemaker = Pacemaker(clock_freq_hz)

def run(q):
    ppl_count = 0
    while True:
        overtime = pacemaker.beat()
        if overtime > clock_period:
            print(
                f"ts: {time.time()}",
                f"overtime: {overtime}")

        while not q.empty():
            timestamp, ppl = q.get()
            print(f"ts: {timestamp}, ppl_added: {ppl}")

            ppl_count += int(ppl)
            print(
                f"ts: {timestamp}",
                f"ppl_count: {ppl_count}")
```

adder.py

Here the modifications are similar. After initialization, the adding code is wrapped in a function called `run()`, which takes a queue as an input argument, and then there is the code that reads from the queue.

When reading a queue we have to be prepared for several outcomes. There might be one thing in the queue, or there might be many, or there might be none. To handle all of these cases, we set up a while loop.

```
while not q.empty():
    timestamp, ppl = q.get()
```

On each iteration it first checks whether the queue is empty. If it's not, it gets an object from the queue–the one that has reached the front of the line, the one that has been in the queue the longest. Because we know we passed a tuple containing the time stamp and the keypress showing the number of people counted, we unpack that tuple from the queue into `timestamp` and `ppl`. For visibility, we also print them in the console. Then our adder does its work and increments our people count accordingly, and we print the total as well.

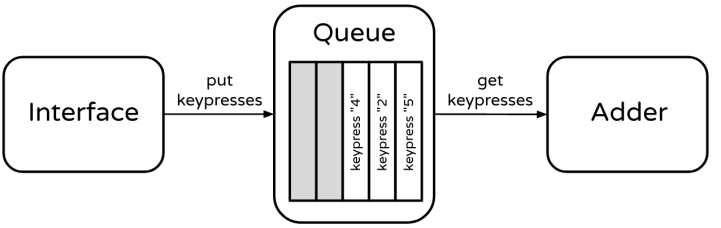
Another change here to make our code more robust is that when we call `pacemaker.beat`, we catch the return value, the amount that the beat has gone overtime.

```
overtime = pacemaker.beat()
if overtime > clock_period:
```

We've added a check to make sure the overtime is not ridiculous, for instance, an entire clock period. If it is, then we print an error notice at the console to let the user know. In a later version, we might want to take this and raise an Exception, or take some other kind of corrective action. But for now, we will be satisfied with giving the user notice.

Tie the parts together

This gets us to the point where both our interface and adder processes have `run()` functions that accept a queue as an argument. The next step is to tie them all together.



To do this we will make use of the **multiprocessing** package in a top level script that kicks off both of our processes and connects them.

```
import multiprocessing as mp
import interface
import adder

instructions = """
  Welcome to the People Counter

  Use keys 1-9 to count people.
  Use space bar to undo your last keypress.
"""
print(instructions)

q = mp.Queue()
p_interface = mp.Process(
    target=interface.run,
    args=(q,))
p_adder = mp.Process(
    target=adder.run,
    args=(q,))

p_interface.start()
p_adder.start()
```

ppl_counter.py

The important parts of this are creating the queue,

```
q = mp.Queue()
```

creating two new separate processes, one from `interface.run` and one from `adder.run`,

```
p_interface = mp.Process(
    target=interface.run,
    args=(q,))
p_adder = mp.Process(
    target=adder.run,
    args=(q,))
```

then kicking off both of the processes by calling their start method.

```
p_interface.start()
p_adder.start()
```

This script builds our process diagram for us. It creates the interface and the adder block, creates the arrow that connects them, connects the arrow to both blocks, then starts the whole thing rolling.

Perhaps the trickiest part of this is the syntax for creating a new process. For both the interface and the adder processes, we instantiate a `Process` object and pass it two arguments.

```
p_interface = mp.Process(
    target=interface.run,
    args=(q,))
```

The first is a named argument called `target`, to which we pass the function that the process will run. In our case, we want one process to have a target of `interface.run` and the other process to have a target of `adder.run`. The second named argument is `args`, a tuple containing all of the arguments that should be passed to the target function. (If this feels confusing, that's because it is.) For our processes, both of our `run` functions look for a single input argument containing the queue that connects them. To indicates a tuple with a single element, we use the construction `(q,)`. It's how we show that both of the run functions should be called with the input argument `q`.

Despite the awkwardness of the syntax, this script isn't doing anything more complicated than creating all the parts in our process diagram and connecting them together. Thanks to the multiprocessing package, this can occur in a readable way that is the same for every operating system. Like the getkey package, multiprocessing hides all of the operating system specific details and takes care of all of the bookkeeping and error catching that needs to happen for this to work smoothly. I don't actually know exactly how it works, and thanks to the multiprocessing package I don't need to.

Unlike the getkey package, multiprocessing has been adopted by Python's core maintainers and is officially supported and gets distributed with every Python download. There's no need to pull it from PyPI. You already have it.

Now that our process diagram is fully defined, and wired together, we can run the whole thing.

```
$ python3 ppl_counter.py

 Welcome to the People Counter

  Use keys 1-9 to count people.
  Use space bar to undo your last keypress.

ts: 1664583456.8063445, ppl_reported: 7
ts: 1664583456.8063445, ppl_added: 7
ts: 1664583456.8063445, ppl_count: 7
ts: 1664583457.415506, ppl_reported: 4
ts: 1664583457.415506, ppl_added: 4
ts: 1664583457.415506, ppl_count: 11
ts: 1664583457.973959, ppl_reported: 8
ts: 1664583457.973959, ppl_added: 8
ts: 1664583457.973959, ppl_count: 19
ts: 1664583458.50361, ppl_reported: 6
ts: 1664583458.50361, ppl_added: 6
ts: 1664583458.50361, ppl_count: 25
```

And behold! We have a people counter that uses two processes at once.

Keeping an eye on everything

We also have a really noisy stream of console output. In its current form, we print a line every time a new key is pressed, every time a new keypress is read, and every time the total is updated. It's confusing to follow. We are in a bit of a bind, though. This level of verbosity is indispensable when we are writing multi-process code like this and need to debug it. And if anything should happen while it's running, say one process freezes or throws an exception, we would need to have some way to detect it. Otherwise we are setting ourselves up for some hair pulling debugging nightmares. You can just imagine how this problem compounds as we scale up to 4 or 8 or 16 separate processes, all connected to each other by several queues passing objects.

Luckily, as is often the case, someone else has already had this problem and has come up with a great solution: logging.

Logging is a lot like printing to the console, except everything gets written to a text file instead. This keeps the console nice and clean and preserves this precious real estate for things that make the user happy. All the historical records and useful information for debugging is squirreled away in the log file.

The other thing this does is allow us to programmatically scan and interpret these logged outputs. Rather than needing to stare at the screen and scroll around to find

events of interest, we can read them in with a script and pull out the parts we care about.

The only thing I don't like about logging is that it makes the code itself slightly harder to read. Logging requires some extra initialization, and the logging statements themselves are longer and not as straightforward as a print statement.

Over/Under-engineering

This is a good example of the over-/under-engineering trade-off. As code gets larger, more complex, used by more people in a wider variety of situations, it can be helpful to introduce tricks and structures to manage it. Like logging, for instance. Knowing when to introduce these measures is an art and a subject of heated debate.

On the one hand, you can wait too long and err on the side of under-engineering. Under-engineering is natural. It's inertia at work. Doing things the way we've always done them is a good way to keep up steady progress. Upgrades and migrations mean that everything has to go on hold for a while. But given enough time and growth, progress will gradually slow and eventually stop altogether. What worked for a thousand-line code base doesn't work for a million. What worked for three developers doesn't work for three hundred. What worked for a million users doesn't work for a billion. A hint that you might be under-engineering is when you're spending most of your time either waiting around or

dealing with problems other than writing, testing, and improving your code.

On the other hand, you can jump the gun and err on the side of over-engineering. There are good reasons to avoid introducing extra tools and structure. They often add a dependency on someone else's code that you can't control and may not fully understand. They can add complexity to your code and bloat your code base. They can make it harder for new developers to understand how your code works, discouraging them from contributing. They can introduce new bugs of an entirely different nature. (These can be devious and pernicious.) Prematurely adding new engineering features can make your system grow so complex that it becomes practically impossible to change. Things you might hear in an overengineering culture are "we're going to need it sooner or later", "I really want to pull in this new framework I read about", "let's just abstract that out", "we should buy the xyz platform to take care of that for us", "we have to prepare to scale by a factor of a thousand".

The under-/over-engineering trade-off is hard because there is no right answer. There are always really good reasons to go either way, depending on what you most care about, and smart well-intentioned people can care about different things. It makes for a lot of fascinating conversations and eye-opening discussions. It can also make for some awkward compromises and frustrating confrontations. But it's never boring.

My own guiding principle here is a rule of three. I wait until the third time that something hurts before I engineer it away. This seems to strike the right balance. If I tried to fix everything the first time it hurt, I would end up investing a lot of time in engineering solutions for problems that I might never see again. But if I wait until the tenth time something hurts, I've probably done a lot of preventable suffering. Once a pain point comes up three times, that indicates it's likely to come up again.

Overengineering isn't always a bad thing

My experience with logging is an example of this. When I first released some open source code a while back, I naïvely made a blanket solicitation for contributions. It was maybe a thousand lines of code in all, and there were several different modules and algorithms involved. I had built and tested it using primitive print debugging.

One contributor made the not-unreasonable addition of logging capability. It brought the code a step closer to being ready for production. It also made the code a little longer and added a dependency on another package, which I had been trying to avoid. Most importantly, it made the code just a little less readable. The workings of the algorithms were obscured a bit by the code required to set up logging and write the log messages. It wasn't wrong, but it addressed a pain point that I hadn't yet experienced, and it introduced some small, but non-negligible cost. I politely rejected the pull request and learned to be more specific about my requests for contribution.

Fast forward to me trying to get multiprocess communication working. There were print lines in multiple processes printing to the same console in rapid succession. Deciphering what wasn't working and why required scrolling through pages of console output, making pen and paper notes, and trying to come up with more creative ways to make informative print lines. After the third time I sat thinking "There has got to be a better way!" I stepped back and took the time to do it a better way. I embraced the small investment of time it took to properly set up logging and the small hit my code took for conciseness and readability, and counted them a bargain for the visibility it gave me into how my code was working and what was going wrong.

The rule of three has more exceptions than not. Sometimes a pain is recurring, but it's just not that big.

For example, I have to go through a bizarre dance of unplugging and replugging HDMI cables to get my Linux laptop to display on an external monitor. It takes an extra five seconds every time I connect. There is almost certainly a way to fix it, but even if it only took me an hour to figure it out, that's still more time than I would spend fiddling with cables over the next few years. It's just not worth it to fix a problem that small.

On the other hand, sometimes problems are so big or so predictable that it is worth it to engineer them out before they even happen. I'm not waiting for my third automobile accident before I start wearing a seatbelt. And I'm not going to lose this chapter three times before I start backing it up.

The biggest mistake I've seen people make in over-/under-engineering decisions is being dogmatic and inflexible. Some red flags: We should do it this way because it's the way we've always done it. Or because that's how they did it at my old company. Or because that's how BigTechCo does it. Or because I read a blog post from that one person that said we should. Or because it's new and shiny.

Once you stop making engineering decisions based on the particulars of your use case, you set yourself up for a lot of waste and pain. The only truly wrong engineering decisions I've seen are the ones that have been made blindly without consideration for the quirks of the situation as it is.

One convenience that logging gives us is the ability to toggle different logging messages on and off. By default, there are five levels, helpfully named DEBUG, INFO, WARNING, ERROR, and CRITICAL in order of increasing severity. When we create a logger, we can specify what level it's going to operate at. For instance, if we specify WARNING, then only log statements at level WARNING and higher (ERROR and CRITICAL) will be logged to the file. It's an easy way to get your code to ignore unnecessary log statements. Then when something goes wrong and you really need to dig deep, you can go back into your code and set the logging level to DEBUG and force it to log everything.

The mechanics of setting up logging aren't pretty, but they are only moderately onerous. To fully take control over what gets written and where, we have to add a few extra lines.

After importing the logging package and a couple of helpful classes, we have to give it a name

```
import logging
from logging import FileHandler, Formatter
logger = logging.getLogger("interface")
```

and give the log file a name

```
log_name = f"{int(time.time())}_interface.log"
```

and specify the string format that we want to write.

```
logger_file_handler.setFormatter(
    Formatter("%(message)s"))
```

The default formatter adds some helpful information, but we are going all in on being control freaks, so here we specify a format that writes exactly the message that we tell it to write and nothing else.

By including `int(time.time())` in the filename, we add the whole number of seconds in the Unix time, ensuring that the log filename will be unique and timestamped (assuming that we wait at least one second before creating a new log file). It's an extra touch of convenience.

We have total freedom over what text we log. Literally any string. It wouldn't be unreasonable to log the same strings we were previously printing to the console. That gets all of the information we need stashed out of the way where we can find it.

Unfortunately, that still leaves us with the problem of potentially having to scroll through a lot of text to find events of interest or patterns. To help out our future selves even more, we're going to use a trick that will let us go back later with another Python script to read the log file and make sense of it. We are going to write each line as a JSON formatted string.

Logging with JSON strings

This might surprise you, but people and computers speak very different languages. This gap gets bridged in different ways. We have accepted ways for people to tell computers what to do, called programming languages. To complement that, it's also helpful to have ways for people to tell computers about data. That's where JSON comes in.

JSON is a way to describe data in text. It is unambiguous; any two people or any two programs can read the same JSON string and agree on what it means. It's awkward for humans to read. If it helps, it's also awkward for computers to read. It's a middle ground, a compromise, a place where we can make a hand-off and be confident that our data won't get mangled during re-interpretation.

To help us with this, Python has a **json** package. It is especially helpful for taking Python dictionaries and converting them to JSON strings with the function `json.dumps()`, meaning "dump string". Not coincidentally, when you create a Python dict and convert it to a JSON string, the JSON string looks a whole lot like the dict creation syntax.

The script

```
import json
log_dict = {"ts": 7777777, "ppl_reported": 7}
print(json.dumps(log_dict))
```

results in the console output

```
{"ts": 7777777, "ppl_reported": 7}
```

In our log files, the difference between our original print logging and JSON-formatted logging is cosmetic. If the console printed line looked like

```
ts: 1664317821.256342, ppl_reported: 9
```

the JSON-formatted version in the log file will look like

```
{"ts": 1664317821.256342, "ppl_reported": "9"}
```

Formatting each line of our log file as its own JSON string gives it a structure and machine readability that we can use to great advantage later.

Once we've created the logger, we can add a line to the log file with one of five functions–`debug()`, `info()`, `warning()`, `error()`, and `critical()`–depending on the severity of the message. If the severity level is included in the logger's current severity range, it gets added to the end of the log file. Here's how it all comes together in the keypress interface.

```
import json
import logging
from logging import FileHandler, Formatter
import time
from getkey import getkey

# valid levels are
# {DEBUG, INFO, WARNING, ERROR, CRITICAL}
logging_level = logging.INFO
log_name = f"{int(time.time())}_interface.log"
logger = logging.getLogger("interface")
logger.setLevel(logging_level)
logger_file_handler = FileHandler(log_name)
logger_file_handler.setLevel(logging_level)
logger_file_handler.setFormatter(
    Formatter("%(message)s"))
logger.addHandler(logger_file_handler)

# continued ...
```

```
def run(q):
    last_key = "0"
    while True:
        key = getkey()
        key_time = time.time()

        if key in [
                "1", "2", "3", "4", "5",
                "6", "7", "8", "9"]:
            q.put((key_time, key))
            log_dict = {
                "ts": key_time,
                "ppl_reported": key}
            logger.info(json.dumps(log_dict))
            last_key = key

        if key == " ":
            remove_ppl = str(-int(last_key))
            q.put((key_time, remove_ppl))
            log_dict = {
                "ts": key_time,
                "ppl_undo": last_key}
            logger.info(json.dumps(log_dict))
            last_key = "0"
```

interface_logging.py

Despite the unsightly 13 lines dedicated to setting up logging at the beginning of the script, the net effect for the user is quite nice. The blast of printed console lines goes away. Birds sing. Squirrels come out of their holes. Peace returns to the meadow.

The adder with logging undergoes a similar transformation.

```
import json
import logging
from logging import FileHandler, Formatter
import time
from pacemaker import Pacemaker

# valid levels are
# {DEBUG, INFO, WARNING, ERROR, CRITICAL}
logging_level = logging.INFO
log_name = f"{int(time.time())}_adder.log"
logger = logging.getLogger("adder")
logger.setLevel(logging_level)
logger_file_handler = FileHandler(log_name)
logger_file_handler.setLevel(logging_level)
logger_file_handler.setFormatter(
    Formatter("%(message)s"))
logger.addHandler(logger_file_handler)

clock_freq_hz = 4
clock_period = 1 / float(clock_freq_hz)
pacemaker = Pacemaker(clock_freq_hz)

# ...
```

```
def run(q):
    ppl_count = 0
    while True:
        overtime = pacemaker.beat()
        if overtime > clock_period:
            log_dict = {
                "ts": time.time(),
                "overtime": overtime}
            logger.error(json.dumps(log_dict))

        while not q.empty():
            timestamp, ppl = q.get()
            log_dict = {
                "ts": timestamp,
                "ppl_added": ppl}
                logger.debug(json.dumps(log_dict))

            ppl_count += int(ppl)
            log_dict = {
                "ts": timestamp,
                "ppl_count": ppl_count}
            logger.info(json.dumps(log_dict))
        print(f"    {ppl_count} people ", end="\r")
```

adder_logging.py

Take note of how we use the different logging levels here. In the interface, we register keypresses at the INFO level, and in the adder we log them at the DEBUG level. We are anticipating that if something is broken and the processes are not communicating, we will need a copy of what one process sent and what the other received so that we can compare them. To do this, we would set the logging level to DEBUG, as we have, and both of these logging statements would get acted on. But if all we

want is a nice, chatty report of everything that happened, we don't need two copies of the keypress. The copy generated by the interface process will suffice. In that case, we would set the logging level INFO. We also include a copy of the updated total each time a new keypress is registered at the INFO level.

One thing here that we would really like to know about is when the pacemaker clock pace is not kept. If a beat goes overtime by more than one clock period, that suggests something is not working as hoped. Because this is a little higher priority, we log it as a WARNING. We don't have anything happening in this code that would warrant an actual error log. Even if the pacemaker detects a few missed beats, it can always recover the next time around. But if anything horrible did happen, we could log it as an ERROR. And if our code was capable of detecting any existential threat to humanity, we could log that as CRITICAL.

With all of our information being cleverly stashed away into log files now, we leave the user without any visual information about what's going on. We can fix that with a single line that prints the current people total.

```
print(f"    {ppl_count} people ", end="\r")
```

I'm a big fan of the carriage return end character here, `end="\r"`. That means that after printing, the cursor will return to the beginning of the line, rather than to a new line. The net effect is that each successive line will print over the previous one. Rather than filling up a screen with print lines, it will just continue to update the one.

With logging in place, we need a new top level people counter script to pull in the new interface and adder scripts.

```
import multiprocessing as mp
import interface_logging as interface
import adder_logging as adder

instructions = """
  Welcome to the People Counter

  Use keys 1-9 to count people.
  Use space bar to undo your last keypress.
"""
print(instructions)

q = mp.Queue()
p_interface = mp.Process(
    target=interface.run, args=(q,))
p_adder = mp.Process(
    target=adder.run, args=(q,))

p_interface.start()
p_adder.start()
```

ppl_counter_logging.py

Now we can run it and check the results. As hoped, we see a single line status at the bottom of the console. It's not strictly necessary, but it's always satisfying to the user to know what's going on, that the program hasn't crashed, and to see their keypress actions have some immediate effect. What's really great about this is how we can focus on the user's experience here. We no longer have to worry about capturing all the information we might need for reporting and debugging. We are taking care of that elsewhere.

The console can now focus on being a user interface. We can strip away anything that might be distracting to someone using the tool in the heat of the moment. The logs are for the developers (us). The console is for the user.

A peek at the logs show that they are doing exactly what we hoped. 1664318883_interface.log shows that it was kicked off at UNIX time 1664318883, Tuesday, September 27, 2022 6:48:03 PM in my time zone. An arbitrary snippet from the middle of the log shows a sequence of keypresses.

```
...
{"ts": 1664318928.6324704, "ppl_reported": "5"}
{"ts": 1664318930.604963, "ppl_reported": "2"}
{"ts": 1664318934.3606267, "ppl_reported": "4"}
{"ts": 1664318936.424587, "ppl_reported": "3"}
{"ts": 1664318942.1853814, "ppl_reported": "7"}
{"ts": 1664318943.0344594, "ppl_reported": "2"}
{"ts": 1664318943.7940567, "ppl_reported": "5"}
{"ts": 1664318945.192151, "ppl_reported": "3"}
{"ts": 1664318945.9686284, "ppl_reported": "2"}
{"ts": 1664318946.5910223, "ppl_reported": "2"}
{"ts": 1664318947.3454957, "ppl_reported": "2"}
...
```

1664318883_interface.log

Because we set the interface logging level at INFO, we can see all of the info logs about the process, and when they occurred. We don't have any other log statements here, so it's relatively uniform, but if we did, they would all be interleaved in the order that they were added by the code.

Taking a look at 1664318883_adder.log, we can also see that it was kicked off at UNIX time 1664318883. This happens to be the same UNIX time that the interface was kicked off at, but it won't necessarily be that way every time. We are creating these log files in two separate processes. This happens very quickly, so they will often fall on the same second, but there's nothing that guarantees that they will.

```
...
{"ts": 1664318993.9616911, "ppl_count": 286}
{"ts": 1664318994.4360094, "ppl_count": 289}
{"ts": 1664318995.121171, "ppl_count": 293}
{"ts": 1664318997.585884, "ppl_count": 295}
{"ts": 1664319000.7577388, "ppl_count": 298}
{"ts": 1664319002.3956523, "ppl_count": 303}
{"ts": 1664319003.0966415, "ppl_count": 298}
{"ts": 1664319003.377368, "ppl_count": 304}
{"ts": 1664319006.4533072, "ppl_count": 311}
{"ts": 1664319007.957886, "ppl_count": 313}
{"ts": 1664319008.7823172, "ppl_count": 316}
{"ts": 1664319009.1996095, "ppl_count": 321}
...
```

1664318883_adder.log

When we open it up, we again see just one type of log message. Because we set the logger level to INFO for the adder, it doesn't show keypresses received from the queue, only the updates to the total. Our pacemaker was very well behaved, so we don't have any overtime warnings to report. It's wonderfully convenient to have all of this information hidden away out of sight in case we should need it.

Catch bugs before they happen

While we have some momentum engineering our pain away, there are two other tools that are worth putting in our box: a code checker called **flake8** and a code formatter called **black**.

flake8 is a linter, a program that goes through our code and looks for mistakes. It can't catch everything, but what it does catch is very helpful. When I am disciplined and run flake8 on my Python code each time I make a change, I save myself a whole lot of time. flake8 catches dumb mistakes that might not crash my code until after it has been running for a minute or an hour. Even better, sometimes flake8 catches mistakes that wouldn't have crashed my code at all. I would have happily gone on thinking everything was fine, but getting wrong answers. flake8 also flags weird things like when I assign a value to a variable but never do anything with it. Sometimes it means I created a nasty bug. Sometimes it just means I got sloppy. Either way, flake8 helps my code be better and saves me time down the line.

You can install flake8 from PyPI with pip

```
$ python3 -m pip install flake8
```

and run it from the command line .

```
$ python3 -m flake8 <filename>
```

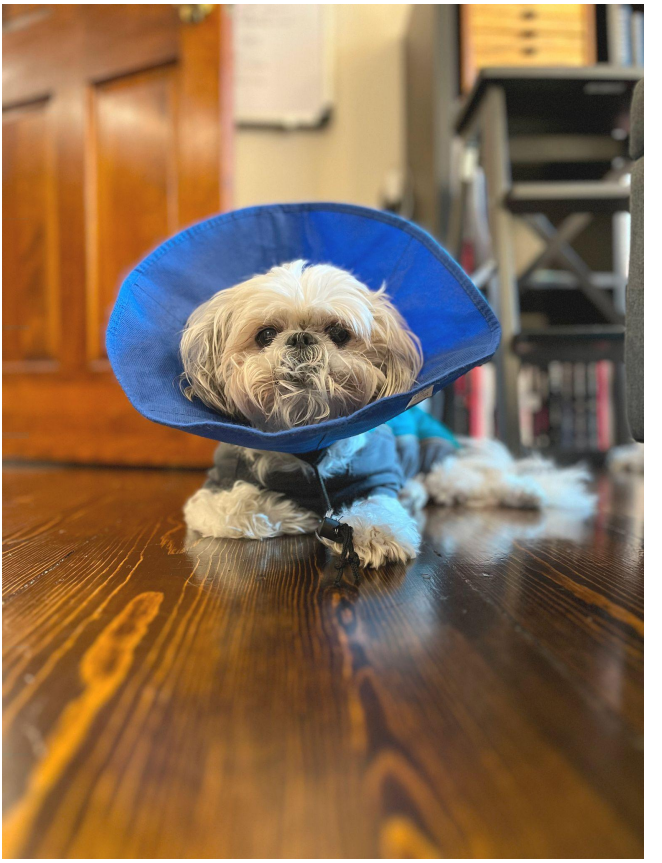
To get a sense of what it can do, we can run it on some code we know to be flawed.

```
infinity = 3/0
 print(
               infinity
```

bad_code.py

```
$ python3 -m flake8 bad_code.py

bad_code.py:2:1: E999 IndentationError: unexpected
indent
```

On flake8's first pass through the code it finds an indentation error on line 2, character 1. That's obviously not the only thing wrong with this code, but if it's broken so badly that flake8 can't make sense of it, flake8 just reports what it has found so far and exits.



Some errors hurt more than others.

After fixing the indentation error, we can give flake8 another chance.

```
infinity = 3/0
print(
                infinity
```

bad_code.py (indentation error fixed)

```
$ python3 -m flake8 bad_code.py

bad_code.py:3:20: E999 SyntaxError: unexpected EOF
while parsing
```

This time flake8 finds the missing parenthesis at the end of line 3, which manifests as "Hey I got to the end of the file (EOF) too soon. Something's messed up!"

```
infinity = 3/0
print(
               infinity)
```

bad_code.py (fixed)

After adding this in, we can run flake8 yet again and, like a grumpy teacher, it expresses its approval by saying nothing at all. Even though this code has passed the flake8 test, it's still not pretty. This is where black comes in.

You can also install black from PyPI with pip

```
$ python3 -m pip install black
```

and run it from the command line

```
$ python3 -m black <filename>
```

When we run black on our bad code, it cheerfully lets us know that it made some formatting changes.

```
$ python3 -m black bad_code.py

reformatted bad_code.py

All done!
1 file reformatted.
```

Taking a quick look at the reformatted code shows a couple of changes that make it easier to read. black added a space to either side of the division operator, and it put the print statement all on one line.

```
infinity = 3 / 0
print(infinity)
```

bad_code.py (reformatted)

The name black is inspired by Henry Ford's assurance: "A customer can have any color they want, as long as it is black".[4] It reflects the opinionated formatting approach that black takes.

There are many ways to format a particular piece of Python code. You can break lines with a backslash or using parentheses or not at all. When breaking lines, you can indent them any amount you like. You can have any number of spaces around operators and any number of blank lines between functions, including none at all. If these sound inconsequential, the heated arguments they spark are anything but. And may heaven help you if you wade into the battle over whether to indent with tabs or spaces.

There is a more-or-less official style guide in the Python Extension Proposal (PEP) 8. PEP-8[5] lays out a lot of suggestions for the best way to do these things, but even it leaves some wiggle room in places, such as whether to break a line before or after a binary operator. black eliminates this. According to black, there is only one correct way to write any piece of Python, and it will enforce that way on your code without asking and without apologizing. On the one hand, you give up some control. On the other hand, you stop having to worry about where to break lines and add spaces. It frees you up to spend less time agonizing over formatting and more time watching unboxing videos.

Now that we've used flake8 and black to get our code in order, we can finally run it.

```
$ python3 bad_code.py

Traceback (most recent call last):
  File "bad_code.py", line 1, in <module>
        infinity = 3 / 0
ZeroDivisionError: division by zero
```

Sadly, there is still an error in it that not even flake8 could catch. Because Python is an interpreted language, rather than a compiled one, the code doesn't actually get thoroughly vetted until you run it. Value errors (like dividing by zero) and type errors (like treating an integer as a string) can't be caught until the interpreter tries to run the code and stumbles. That's the deal with the devil we make when we work with Python. We get flexibility, readability, ease of use. We lose the optimization and error checking that gets done during precompiling. (If this bothers you tremendously, I recommend looking at Rust[6] as an alternative.)

Despite the fact that they don't fix everything, flake8 and black save me enough time that they clear my very high bar for inclusion in my typical workflow. As with everything I suggest, try it out, see if it works for you, and take or leave it as you please. There's no wrong answer here. You're the ultimate judge of what works for you.

As a sidenote, I do run black on all my code, and the scripts in the repository are all black-formatted. However, I've tweaked the code you see in the book

here. black tries to limit lines of code to no more than 79 characters long. At a readable font size, this page doesn't allow for quite that many. I go in and add some extra line breaks by hand to accommodate us, but I try to do it in a black-consistent manner.

Reporting

It's finally time to take advantage of our fancy logging format. We're going to show how this is done by making a report of the people that have been counted so far.

To do this, we'll want to pull the running total of people counted from the adder log and plot each update against the time that it was observed. We can use the standard Python incantation for reading in a text file one line at a time.

```
with open(filename, "rt") as f:
    lines = f.readlines()
```

Then we can iterate over each line, converting it from string to JSON, then converting that to a Python dictionary.

```
    for line in lines:
        row_dict = dict(json.loads(line))
```

And at this point we can run any checks we need to in order to see whether this particular line of the log file is interesting to us. We can check to make sure that it includes a people counted field. If we wanted to, we could also ensure that it fell within a particular time

range. For more complex and larger logs, this filtering step is where we get to implement whatever logic tricks we can conjure up to isolate the information that we care about.

Because we are interested in collecting a history, we will walk through every line in the log file and add each new observation to a list.

```
times.append(row_dict["ts"])
count.append(row_dict["ppl_count"])
```

We didn't have to use JSON formatting in each line of our log file. We could've written anything we wanted in any text format. It might have made the logging code a little simpler. It definitely would have made the logging file more concise and require less memory. But all of that gets paid back triple when we go to read and interpret the logs. There's no need to write regex string parsers. There's no need to reinvent a good way to know whether a number is a float or an integer. There's no need to go in and update the parsing code every time we add a new log message or want to extract a different piece of information from the log. The formality that JSON imposes on us, together with Python's built-in JSON parsing capabilities, save us all of this pain. This isn't the best answer for every logging system, or even most of them, but it sure works well for us here.

Now that we parsed the logs, we have a running list of people totals and a matching list of the UNIX time at which each of those people totals occurred. Data like this

benefits from a visual representation. My own favorite tool for making plots is **Matplotlib**. It gives us complete control over what we plot. It is essentially a general purpose drawing program with a few plotting routines built on the top of it. The cost of having that much power and flexibility is that it has a steeper learning curve than other tools. For me, the trade-off is worth it, but I won't judge you if you decide to go a different way.

The first step in getting our data ready to plot is to convert it from a list to a **Numpy** array. Numpy is a Python scientific computing library that we are going to see a lot of in upcoming chapters. But all we really need to know about it to get started is that Numpy arrays are an excellent way to hold groups of numbers.

These you can also install from PyPI.

```
$ python3 -m pip install numpy
$ python3 -m pip install matplotlib
```

And in our code we can use Numpy to convert our lists to arrays.

```
times = numpy.array(times)
count = numpy.array(count)
```

Of our two arrays, `count` is straightforward. It's the total number of people counted so far. `times` is less easy to interpret. One thing we can do to help with that is to convert it from a UNIX timestamp to the number of seconds since we started counting. To do that, we

subtract the smallest time to be found in the whole bunch from every other UNIX time. What we have left is the elapsed value since the very first measurement was made, in seconds.

```
minutes = (times - np.min(times)) / 60
```

Another small tweak that I make out of personal preference is to convert the time from seconds to minutes. I'm assuming here that people counting might go on for an hour or two. Reasoning in terms of dozens of minutes requires less mental effort than thinking about thousands of seconds. Although the underlying information is the same, this is a small thing we can do to make the analysts' life easier, and, to be honest, "the analyst" is probably us. It's not at all a matter of catering to laziness or coddling the plot reader. It is an acknowledgment that we can only think about so many things at once. Every logic step, every mental transformation, every visual leap we can save the reader frees up their brain to make connections and see patterns that otherwise might remain hidden. We want to exploit any advantage we can give the reader.

As a bonus, if we are making this report for someone else, they will appreciate it. Making insights pop out for a reader makes them feel smarter, and when something we give them makes them feel smarter, that makes us look smarter. Everybody wins.

Now that we have our data pre-formatted, prepared and pre-processed, we can create the plot itself. The code here shows a bare-bones set up for plotting a line and adding some axis labels. It meets our needs just fine.

```
fig = plt.figure()
ax = fig.gca()
ax.plot(minutes, count)
ax.set_xlabel("Counting time (minutes)")
ax.set_ylabel("Total people")
```

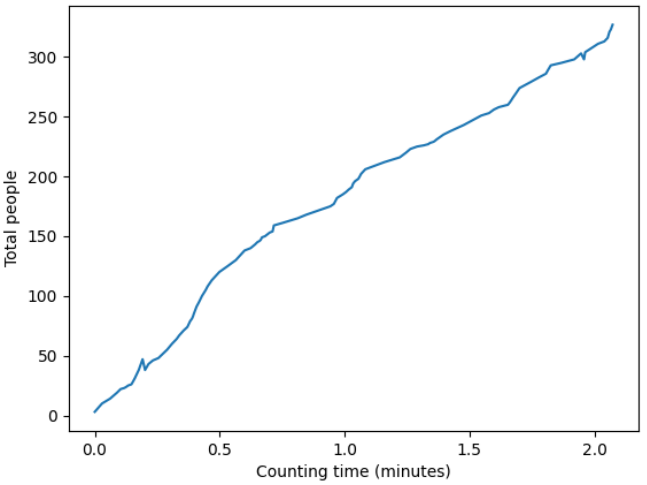
There are 1000 other things we can do with Matplotlib, and will end up doing a lot of them before we're done, but for the meantime, this is all we need for looking at our counting history.

In the completed script I added a few other niceties. It iterates over all the adder logs present and creates a separate plot for each one. It saves the plots with the same UNIX time prefix as the log file. It ignores entries in the log file that have nothing to do with people count. It handles the case where the log file is empty.

```
import json
import os
import numpy as np
import matplotlib.pyplot as plt

filenames = os.listdir()
for filename in filenames:
    if filename.__contains__("adder.log"):
        log_id = filename.split("_")[0]
        times = []
        count = []
        with open(filename, "rt") as f:
            lines = f.readlines()
            for line in lines:
                try:
                    row_dict = dict(json.loads(line))
                    times.append(row_dict["ts"])
                    count.append(row_dict["ppl_count"])
                except Exception:
                    pass
        times = np.array(times)
        count = np.array(count)

        try:
            minutes = (times - np.min(times)) / 60
        except ValueError:
            minutes = 0
            count = 0

        fig = plt.figure()
        ax = fig.gca()
        ax.plot(minutes, count)
        ax.set_xlabel("Counting time (minutes)")
        ax.set_ylabel("Total people")
        plt.savefig(f"{log_id}_report.png")
```

report.py

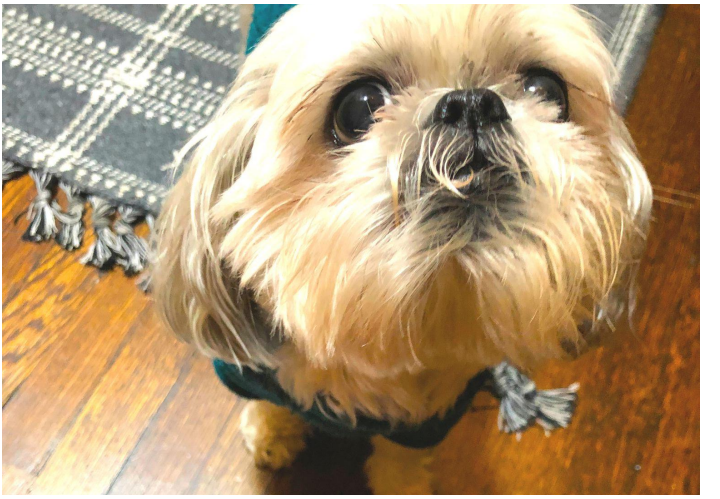
Plotting the data helps us see relationships that we couldn't have easily gotten by browsing the log file.



At the far right hand side, it shows the total number of people counted so far. But it also shows how we got there. Areas of rapid people flow show up as steep sections of the curve. We can even see a couple of places at around 0.2 and 1.9 minutes where a keypress was undone, resulting in a brief drop in the count.

A cursory visual inspection of the plot will also reveal any unusual failures. If the interface had stopped responding for a while and failed to report keypresses, that would show up as a perfectly flat portion of the plot. If somehow a bug caused a keypress of 9 to be counted as 99, that would show up as an unexplained vertical jump in the plot. What we see here is a mostly smooth,

mostly increasing plot of people count, a plausible representation of what happened. A plausible plot isn't a guarantee that nothing bad happened. It's still possible that people were counted incorrectly or a keypress was missed or something was counted twice. But it does give a way to catch bugs that would be difficult to catch any other way, as well as an intuitive sense of everything that happened during this people counting session. It shows that through this particular gate for this particular two minute interval, about 330 people passed through. You can imagine extending this through the duration of the concert and aggregating it across all gates to get the full report of just how many happy fans got to see Billie Eilish that night.



"Please, sir, can I have some more Numpy?"

Shell scripting

Before we finish, there's one more piece of engineering I want to introduce: the shell script. If you are working at the command line, you probably find yourself typing the same things over and over.

```
python3 -m flake8 *.py
python3 -m black *.py
```

And after running the `ppl_counter.py` and `report.py` scripts a few times, you have probably generated a healthy collection of .log files and .png's. Given time, these will seem to replicate and take over your development directory. To handle this, I like to frequently move any .log and .png files to an `archive` directory. This becomes another set of commands that I type frequently.

```
mv *.log archive
mv *.png archive
```

Typing this out is only a minor inconvenience, and they are only a handful of lines, but even so, they add up. For me the bigger factor than the inconvenience is that sometimes I forget to flake8 my scripts. I forget to run black and clean up the formatting. I forget to move my log files and png's to the archive. And so I create a little bit of a mess, and worse, I slow myself down with avoidable bugs.

All of this is to say that even though I remain opposed to clever engineering for its own sake, here the cost is low and the benefits significant enough to be worth it. The solution I'm talking about is a shell script.

First off, an apology to those of you working in Windows. Windows has options for doing things very like this, either in a DOS terminal or in a PowerShell or a Cygwin console. In each case the details are a little different, and I am going to abdicate my teaching responsibilities here to others who are better qualified.

The macOS and Linux environments aren't identical, but they are close enough that this approach will work on both.

A shell script is a whole lot like a Python script, except it is interpreted by the shell, rather than by a Python interpreter. It is a text file and usually ends in .sh, although this is convention only. The shell doesn't care what you name it. This is mostly to help us so that when we see the file in three weeks, we remember what's in it.

In our shell script, we can add everything that we would normally type at the command line, exactly as we would type it.

```
mkdir -p archive
mv *.log archive/
mv *.png archive/

python3 -m black *.py
python3 -m flake8 *.py
```

clean.sh

The one nicety we add is a check whether an archive directory exists, and if it doesn't, we create one: `mkdir -p archive`. And that's it. That's a shell script.

To run it, we type `source` and the name of the shell script at the command line.

```
$ source clean.sh

All done!
10 files left unchanged.
```

The shell interpreter finds the text file containing the shell script and runs the lines, one by one, as if we had typed them.

We go from having to remember several things to remembering one thing, from having to type several lines to having to typing one short line. It's a small win, but since the whole thing takes maybe one minute to set up, it's a small cost. The first time it helps us avoid a 10 minute debugging loop, it will have paid for itself many times over. And I'm not gonna lie, the first time you get

to casually drop in conversation that you use shell scripts in your everyday workflow, it's a bit of a rush.

What's next?

We covered a lot of ground in this chapter. We are now armed with a working knowledge of processes and interprocess communication. That's kind of a big deal. It will be one of our big foundation stones for everything that follows. We also covered call logging, reporting, and a handful of software engineering tricks. This rounds out our ring of skeleton keys for getting into robotics.

In the next chapter, we are going to use everything we've learned about processes, time, and pacemakers to start animating simulated robots. To do this we are going to have to talk about plotting, visualization, and animation. We're just getting to the good part. Stick around.

Recap

multiprocessing is the Python package that helps spawn and connect processes from our code. Create new processes with a call to **Process()**. Use a **Queue** to help processes pass objects back and forth.

The **getkey** package is super helpful for grabbing keypresses.

The **flake8** package is worth its weight in gold for catching syntax errors ahead of time.

The **black** package is also a useful tool. It reformats Python code to a readable and PEP 8-compliant format.

The **logging** package is useful for creating, managing, and writing to log files.

JSON is a helpful text format for communicating data between human and machine. Python's **json** package unlocks a toolset for working with it.

Numpy and **Matplotlib** are invaluable numerical processing and plotting packages. We will become much better acquainted with them in future chapters.

Packages

black	https://pypi.org/project/black/
flake8	https://pypi.org/project/flake8/
getkey	https://pypi.org/project/getkey/
json	https://docs.python.org/3/library/json.html
logging	https://docs.python.org/3/library/logging.html
matplotlib	https://matplotlib.org/
multiprocessing	https://docs.python.org/3/library/multiprocessing.html
numpy	https://numpy.org/
pip	https://pypi.org/project/pip/

Resources

1. The Robot Operating System
https://en.m.wikipedia.org/wiki/Robot_Operating_System

2. Billie Eilish is a popular musician.
https://www.youtube.com/watch?v=DyDfgMOUjCI

3. PyPI, the Python Package Index.
https://pypi.org/

4. It looks like it wasn't actually Ford who first said this. But as (probably someone other than) Mark Twain said, "Never let the truth get in the way of a good story."
https://collectorsautosupply.com/blog/true-or-false-did-ford-really-say-any-color-the-customer-wants-as-long-as-its-black/

5. PEP-8. Python Extension Proposals (https://peps.python.org/) are a catch-all for several diverse types of documents, such as proposed syntax changes, reference materials, philosophical essays, and third party packages. They are mostly numbered in order of creation, so PEP-8 is an early one. Although most of it is in the form of recommendations rather than commandments, it has the weight of consensus and tenure behind it. Unless you have a good reason to do otherwise, you're always safe following PEP-8 style.
https://peps.python.org/pep-0008/

6. Rust is a popular programming language. It emphasizes speed, but with fewer opportunities to shoot yourself in the foot than are offered by C and C++.
https://www.rust-lang.org/

About the Author

Robots made their way into Brandon's imagination as a child while he watched The Empire Strikes Back on the big screen, one buttery hand lying forgotten in a tub of popcorn. He went on to study robots and their ways at MIT and has been puzzling over them ever since. His lifetime goal is to make a robot as smart as his pup. The pup is skeptical.

To see more of his work, visit brandonrohrer.com