Keeping Time with Python

How to Train Your Robot

Chapter 2

Brandon Rohrer

How to Train Your Robot

Chapter 1:
Can't Artificial Intelligence
Already Do That?

Chapter 2:
Keeping Time with Python

About This Project

How to Train Your Robot is a long term side project. I've been working on it for 20 years, and I don't know if I'll ever finish it. But I find it deeply satisfying to share progress as I go, and who knows, maybe someone will find it useful. This is already the second chapter. I'm pretty pleased with that, because it means that the project has a quantifiable momentum.

Onward we go.

Brandon
Boston, USA
October 18, 2022

Keeping Time with Python

Chapter 2

In which we synchronize computers
with the rest of the world.

In chapter 1 we stood on top of a high hill, pointed at the horizon, and declared a destination: sample-efficient human directed reinforcement learning. Now we start down the path, putting one foot in front of the other. Before, we were surveying vast landscapes and distant mountain ranges. Now we're going to have to pay attention to twists in the trail and pebbles in our shoes. But it's all part of the same journey.

The Wall Clock

As long as we stay inside the computer, time is not too terribly difficult to manage. Everything happens in lockstep, synchronized to the nanosecond by a master clock. There are some subtleties. Because the processor has to juggle so many different activities with so many different priorities, sometimes programs get neglected and even freeze entirely, but thankfully this is the exception rather than the rule.

This bubble of control is still in place when working with simulated robots. In a simulation, all the pushing and pulling between particles and linkages that lead to forces and accelerations are represented by a lot of math, expressed in computer code. But however complex the calculations needed to advance the simulation physics from one time step to the next, however tedious the computation and planning of the controller, each step can be made to wait patiently for the other. As far as the processes are concerned, time is elastic. It can be stretched or compressed to fit the calculations.

One of the things that makes robotics hard is that it takes place outside of the computer's realm, out in the rest of the world. Sometimes this is called the "real world", but that implies that the transistors and electrons inside the CPU are imaginary. It’s also called the "physical world", but that label doesn’t sit quite right for the same reason. Probably the most descriptive term I’ve heard for this is "meatspace", suggesting that it is a realm where interaction with bulk biological tissue becomes possible. But instead of these, we’ll go with a semantic wrecking ball and just use the term "world" to describe anything that happens outside of the signals governed by the CPU clock.

In the world wild things can happen. Batteries drain. Lightning strikes. Earthquakes shake. Monkey wrenches get dropped into gears. A robust approach to robot control will be able to fail gracefully when any of these happen. But one thing that does not ever change, even on the weirdest day, is the progression of time. When working with robots, world time is absolute.

When we have a computer controller driving a robot in the world, we no longer have the luxury of ignoring the wall clock. If code in a critical control loop takes too long to run, it may not send instructions on time. The robot's performance may degrade or it might fail spectacularly, damaging itself or someone else. If sensor information isn’t read in on a regular cadence, the robot may wrongly estimate how fast things are changing and lose the ability to respond appropriately. It might even miss important events entirely. When a computer is taking in inputs, and more importantly, when it is sending out commands, it needs to be able to keep time with the world, or at the very least be aware of when it’s not keeping up.

The critical role of time and timing makes it a cornerstone of everything that comes after. It’s important enough to get right that we're going to dedicate a chapter to it. And it’s foundational enough they were going to tackle it before we build anything else.

Measuring the Passage of Time

It’s worthwhile to take a minute and get very clear about how we are going to measure time. A tried and true method is to find something that happens on a cycle and count it. The classic example of this is the sun passing overhead. This physical phenomenon happened consistently enough that it made a reliable unit of measure. When subdivided into hours, minutes, and seconds and agglomerated into months, years, and centuries, it provided a collection of measurement scales suitable for almost every purpose.

Using the principle of counting cycles, enterprising inventors found they could also use pendulums to keep time for clocks and metronomes. Rather than a 24-hour cycle, pendulums had cycles measured in seconds. A weight attached to a stick will swing back and forth with a highly repeatable number of swings per minute. Surprisingly, this frequency is the same, whether the swings are tiny or wide. If you stop to think about it, it's not at all obvious that this should be the case. Shouldn't a pendulum that is barely moving oscillate faster than one that is really pumping? Or slower? Physics says no (at least until the amplitude gets large enough that the linear approximation $\sin(x)\cong x$ breaks down). Fortunately for time-keepers, this insensitivity of oscillation frequency to amplitude makes pendulums a fantastic tool for time keeping. Curiously, this phenomenon is also exploited by parents pushing two year olds on swings. A child will need pushing just as often, no matter how high they're going. This predictability lets the parent push with one hand while doomscrolling with the other.

Another great feature of pendulums is that their frequency is adjustable. Moving the weight closer to the pivot makes the pendulum swing faster, and sliding it further down the stick will make it swing more slowly. Adding a small gravity-fed or spring-driven kick at the end of each swing will keep that pendulum going at a precise rate for as long as someone is willing to keep the spring wound or the weights re-hung. Carefully adjusting the frequency of oscillation enables mechanical clocks to keep pretty good time and help pianists to play Mozart's sonatas at the tempo he intended.

Tail wags were briefly considered as a way to track time, but were abandoned due to being distractingly adorable and notoriously unreliable.

Then came computers. At their heart is an oscillator, a small crystal that wiggles ever so slightly millions of times per second. It is similar to a tiny pendulum, but instead of swinging back-and-forth, it rings like a champagne flute struck with a cake fork. And instead of getting nudged by a spring driven gear, it gets gently kicked at the extreme of each vibration with a small voltage pulse. This hypercaffeinated tick-tock is also extremely regular, thanks to the laws of physics. It gives us another way to represent time. We can turn on the computer and start counting the number of oscillations in the clock crystal. If a program needs to communicate to another program a time at which something happened, or should happen, the number of oscillator cycles is a precise unit of measure that means the same thing to both of them.

This system works great until the computer needs to do something according to a wall clock. If I have an alarm clock app, I want to be able to set a wake up for 6:15 AM, rather than figure out how many billions of oscillator cycles have elapsed between the time I turned the computer on and when I want the alarm to sound. This problem is solved by having the computer check the Internet when it's powered up to find out what time all the humans think it is. Then, knowing how many oscillator cycles it goes through in a second, it can inch its own internal copy of the wall clock forward. This arrangement essentially lets the computer track the passage of the sun overhead with an immense amount of precision. When I set an alarm for 6:15 AM, I know it’s going to go off right at 6:15. If my computer tells me an email arrived at 2:12 PM and 17 seconds, I can have a high degree of confidence that is exactly what my wall clock said when the email hit my mail server.

This approach works great for computers that are close to each other. It allows a computer to report the wall clock time for any event, past or planned, with millisecond accuracy. Nearby computers can compare notes and can agree whether one thing happened before another.

However another piece of complexity now steps forward. It was always there, but ironically, computers' ability to reach around the world in less than a second brought it to the fore. There are lots of wall clocks in the world. If my wall clock says 10 AM, that doesn’t tell me what your wall clock says. If your computer tells me an event happened at 5:55 PM, I could not tell you whether that happened before or after an event that my computer reported at 6 PM. Wall clocks can only be compared if you know where they are.

The nuances of dealing with time zones are famously tangled and difficult to get correct. They are a running joke among software developers. Every time you think you have a good set of rules for handling time zones, one more exception pops up: irregular time zone boundaries, daylight savings time, the International Date Line, non-contiguous time zones, time zones offset by thirty minutes, or fifteen, time zones tied to shifting laws and political boundaries. Accurately comparing two wall clocks is an exercise in logic, history, politics, and patience. It requires knowing where both of the clocks are, sometimes with a high level of accuracy. It is not to be undertaken lightly.

A commonly accepted way to get around this mess is to pick one time zone for all computer programs everywhere in the world and only convert to the local time zone when necessary. This trick isn't without its hiccups, but it works surprisingly well. Time in computer programs is most often represented in UTC (a modified acronym for Coordinated Universal Time) which matches the local time in London during winter, when daylight savings time is not in force. With this common representation, computers regain their ability to compare events no matter where they're located in the world. It short circuits the chaos introduced by our patchwork of time zones. It works well, because computers could not care less whether it's sunny outside at 12 PM or dark at 2 AM. They experience no cognitive dissonance from disconnecting time of day to position of the sun.

There's one other innovation that makes working with time in code much easier: representing it as a floating point number. If you try to calculate what day of the month it will be 600,000 hours from now, you quickly discover that dates and times are an arithmetic nightmare. The hack to simplify this is the UNIX Epoch. It is a collective delusion that time began in 1970.

January 1, 1970 at 12:00:00 AM UTC to be exact. With that datum in place, it becomes possible to define **UNIX time**[1] as the number of seconds since the UNIX Epoch. Any date, any time can be represented as a simple floating point number. (As long as it wasn't during the 60s. Or before that.) This universal, arithmetic-friendly representation of time is the standard for software developers worldwide. It's considered best practice to keep all of your time representations in UNIX time, converting to a local human readable date and time only when necessary for the benefit of human users.

There is yet one more layer of the onion to peel back, one step deeper into the rabbit hole we need to descend, before we complete our side trip into time measurement. Coordinating CPU clocks with Internet wall clocks works pretty well, but still allows some amount of error. It takes time for distant computers to talk to each other, which introduces some delay in synchronizing them, but a system called the network time protocol (NTP) does a robust job of accounting for that. It's usually off by no more than a few milliseconds, and only very occasionally by much more than that.

More disruptive still are leap seconds. It turns out that the rotation of the earth and the passage of the sun through the sky is not quite so regular as we originally thought. The oceans sloshing about as they are tugged on by the moon dissipate a non-negligible amount of the earth's rotational momentum. The world turns a little slower every day. The net effect is that approximately once every 800 days an extra second needs to be added to a day in order to keep the passage of the sun and our wall clocks in precise synchronization. UNIX time makes a concession to the solar cycle here, and ignores leap seconds entirely. This decision gives Unix time the convenient property that 12 AM UTC on any given day will be evenly divisible by 86,400 seconds. It artificially enforces a fixed number of seconds in a day. This also means once every 2 1/2 years there is a second just before midnight and another just after midnight that are referred to by the same UNIX times. For a UNIX time during that interval, it's impossible to determine which of the two seconds it is referring to. If an event happens during that two second period, Unix time can't ensure that it will be correctly interpreted. This ambiguity is likely to offend anyone with a sense of rigor, and it shows that we haven’t entirely escaped the primacy of the natural world in governing our measurement of time.

Between leap seconds and wall clock updates from NTP, our computers' internal wall clocks (called **system clocks**) will fluctuate somewhat, probably on the order of milliseconds, but almost never more than a second. For time stamping emails and setting alarms, this is perfectly fine. However, for some applications this is unacceptable. For example, in robotics it is common practice to sense position frequently, say several hundred times per second, and use it to estimate velocity. To do this well, we have to know to a high degree of accuracy how much time has elapsed between each measurement. Our jumpy system clock will not give us this.

For sensitive time operations like these, where precise time intervals matter much more than synchronizing with distant systems, we can fall back to counting cycles of our CPU clock. This is called a monotonic clock, because the time stamp of any given event is always guaranteed to be greater than those of all of the events that came previously. This guarantee doesn’t exist in a world where leap seconds and clock adjustments can happen. Monotonic clocks have no reference, that is, we don't know exactly what time it was when they started counting. They are not useful for telling us what’s going on anywhere outside of our CPU or giving us any notion of what our wall clock says. But they are great at strictly ordering and measuring internal events.

That catches us up with everything we need to know about time to build kick-ass robots. If you are sad that this rabbit hole has come to an end, never fear. I invite you to explore three other tunnels: 1) propagation delay in integrated circuits, 2) relativistic effects of moving GPS satellites, and 3) strontium clocks. At the sub nanosecond level, you have to start worrying about how long it takes an electron to travel a few inches down a wire and also about how the notion of time is fundamentally tied to position and starts to distort when you are very far away or moving very fast. And if you want to measure things happening that quickly, you're going to need an extremely accurate clock. But we have what we need for our robotics journey, so we'll pack up and move on.

Let’s write some code!

We’re going to be working exclusively in Python. A lot of robotics work is done in C++, because it gives you a lot of low level control, and it’s screaming fast. Coding in C++ is a little bit like wielding a lightsaber. In the right hands it can be quite effective, but when you’re just starting off you’re more likely to cut off your own foot.

Likewise, a lot of scientific computation libraries are implemented in C and C++ because of their raw speed. Working in these has a steeper learning curve, and using them would make this project less accessible to beginners. And to be frank, my own C coding experience is now a distant memory.

Luckily, we don't take too big of a hit working in Python. For robotics, even the native C++ code is often deployed with a Python wrapper. This leaves all of the grittyness wrapped up tightly in a box and gives us some nice Python handles to grab it by. Similarly, scientific computation in Python is greatly accelerated thanks to libraries like **numpy**, which has highly optimized C code under the hood for common numerical operations, and **numba**, which will compile your Python code down to C and make it run nearly as fast as raw C code.

It has been said that Python is the second best language for everything, and we are going to test this assertion. A far reaching project like this one will eventually touch many different computing applications: robotics and hardware integration, scientific computation and linear algebra, visualization and animation, sensing and communication, process monitoring and coordination. In addition to all these, I have the goal to make this work as broadly understandable and understood as possible. Python was designed for readability. It doesn't always succeed, but compared to the alternatives it does an exceptional job.

I’ve set this up so that if you choose to, you can easily fetch and run this code yourself. There’s no better way to get a feel for how things work than to play with them. Run it, change it, run it again. Break it and fix it. Put your own spin on it. All of the scripts we walk through here you can find on GitHub. The code is available at brandonrohrer.com/httyr2files.

My coding explanations assume that you have a little bit of Python experience. That said, if you are new to Python and want to learn, don’t be frightened. Although teaching Python is beyond the scope of this book, I put together a list of great resources for it here (e2eml.school/python_resources.html). I also created a course for brand new Python coders here (e2eml.school/201) that covers a lot of the same ground, working through tricks you can do with time, including some simple games. I also try to keep my code as learner friendly as possible. I aim for readability, descriptive variable names, one operation per line, and using the simplest approach I can get away with.

It runs on my machine

I’m going to apologize in advance; I have not taken steps to guarantee that my code will run for you. Writing code that will run on any computer is a gargantuan task. It becomes especially thorny when you start interacting with the world via pixels, microphones, keyboards, networks–really any inputs and outputs. Even when we stay within the confines of the computer, different versions of Python, different versions of each package, different operating systems, different versions of the same operating system, and different processor chipsets can all introduce their own subtle quirks into how code is interpreted and what will run.

That said, I've tried to stack the deck in our favor by not getting too fancy. I try not to do anything that requires the latest version of Python or operating system-specific behaviors, at least when I can avoid it. I try not to require the latest, bleeding edge package versions or recently released features. In fact, if there's anything we can code up ourselves, I'll avoid relying on third-party packages altogether. It's possible that some of this code won't run on your computer, but I'll do what I can to avoid that.

For reference, all these examples were written and run on Python 3.8.10, under Ubuntu 20.04.5 LTS. Full specifications of my machine are in an endnote[2] for reference if you want to compare your setup.

If you're not already set up to run Python, I wrote some bare-bones instructions for first time Python users under Windows (e2eml.school/112) and under MacOS (e2eml.school/111). If you’re a Linux user, then I’ll leave you to your own devices.

I have a strong opinion on development environments: The best one is the one you’re most comfortable with. Please don’t let anyone waste your time by telling you there’s one right way to do it or that you’re doing it wrong. Any combination of tools that lets you write code, run it, and find the bugs in it is plenty for our purposes. Test drive as many environments as you like until you feel at home. It just so happens that my comfort zone is writing code in the vim text editor and executing at the command line. But the fact that my favorite flavor of ice cream is burnt caramel grape nut should have no bearing on yours. Find your flavor and grab a spoon. If you really have no idea what to try first, a lot of people rave about Visual Studio Code. It’s as good a place as any to start.

Checking the Time

Every operating system has its own way to check the time. Python hides all of this behind a single package, appropriately named `time`.[3] No matter what operating system you're using, you can check the system clock with the `time.time()` function.

```
import time

current_unix_time = time.time()
print(current_unix_time)
```

00_unix_time.py

When run, the result is the current Unix time, a little over 1.66 billion seconds.

```
$ python3 00_unix_time.py

1662637918.4849603
```

I could call a friend in Mumbai and if we both run this code as I say "one, two, three, go!", we'll get answers that differ only by the 150 ms it takes for my voice signal to be digitized and hop there via satellite.

In its native form, Unix time is tough to interpret. If we needed to pull out time of day to set a microwave clock, we could do that with a little arithmetic.

```
import time

# UTC offset for my local time
# (US Eastern Daylight Time)
utc_offset_hours = -4
seconds_per_hour = 3600
seconds_per_day = seconds_per_hour * 24  # 86,400

unix_time = time.time()
utc_time_of_day_seconds = unix_time % seconds_per_day
utc_time_of_day_hours = (
    utc_time_of_day_seconds / seconds_per_hour)
local_time_of_day_hours = (
    utc_time_of_day_hours + utc_offset_hours)
local_hour = int(local_time_of_day_hours)

print("local hour:", local_hour)
```

01_time_hour.py

This code does two non-obvious things. The first is to divide the Unix time by the number of seconds in a day (86,000) and just keep the remainder. Shorthand for this remainder-keeping operation is "modulo" and Python uses the `%` operator for it. Thanks to the fact that Unix time ignores leap seconds, Unix time modulo 86,400 will always give seconds since midnight according to UTC.

Converting seconds to hours is straightforward, but getting from UTC to my local timezone is a manual lookup exercise[4]. I have to know that for my particular location in Boston, USA the UTC offset is -5 hours, that is, that the local time here lags UTC by an even five hours. I also have to know that there is an additional +1 correction since we are currently on daylight savings time. This is hard-coded into the variable `utc_offset_hours` as -4.

With these two pieces in place, when I run this code I can see that it is currently the 10am hour here.

```
$ python3 01_time_hour.py

local hour: 10
```

My friend in Mumbai would get a different result of course. The UTC offset there is +5.5 and they don't observe daylight savings. Luckily for us, we don't plan to go any further into the mess that is resolving local times and their differences across the globe. I'll save that for braver people than me. We're going to explore in the opposite direction–small events that happen very nearby and very fast.

But is it dinner time?

Timing your code

Modern processors compute at such blinding speed that it often seems instantaneous, but that speed can evaporate quickly when you start playing with algorithms and fat data streams. When you bump up against the limits of what your system can do, it suddenly becomes very important to understand where that time is going. Is it spent reading a file? Multiplying arrays? Waiting for another process to return a result? The only way to know for sure is to time your code.

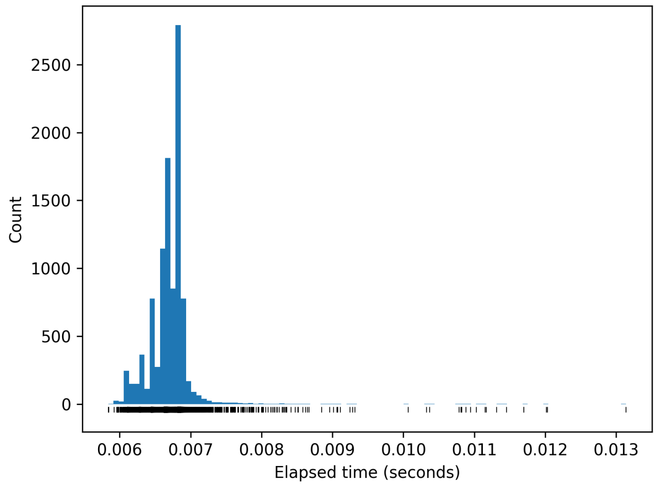
We can repurpose our `time.time()` function for this. By calling it immediately before and immediately after some snippet of interest, we can take the difference of the two to find the elapsed time.

```
import time
import numpy as np

n_iterations = 10000
total_execution_time = 0

for _ in range(n_iterations):
    start_time = time.time()

    # Do some time consuming busywork computation
    size_3D = (100, 100, 100)
    random_array = np.random.sample(size=size_3D)
    total = np.sum(random_array)

    end_time = time.time()
    elapsed_time = end_time - start_time
    total_execution_time += elapsed_time

average_time = (
    total_execution_time / n_iterations)
print(
    "Average execution time:",
    f"{average_time:.09} seconds")
```

02_code_timing.py

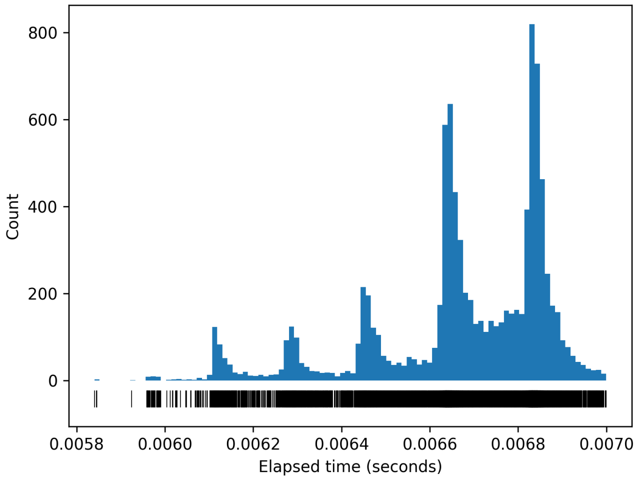
There's some variation from run to run, so to get a robust estimate we can take a lot of measurements and calculate the mean. For this particular code running on my machine, the average execution time is about 6.55 milliseconds.

```
$ python3 02_code_timing.py

Average execution time: 0.00654980178 seconds
```



03_code_timing_distribution.py

Average execution time is a useful number, but it hides a whole mess of complexity. If we run this again and plot a histogram of all the 10,000 execution times, we can see that most of them fall pretty close to the peak. But some of them are much higher, almost double. The ticks along the bottom show each individual measurement. we can see that the top 10 fall above 10 ms. That’s one in 1000 measurements coming in at more than 150% of the mean. That may not sound like a lot, but it illustrates an underlying principle: every so often code execution will take much longer than you expect. And I collected this data on a lean system, operating way beneath its limits without a lot of competing processes. It’s not uncommon for these hiccups to reach several seconds on some systems. The important lesson to take away from this is that code execution times can have a very long tail. However long you think they might take, they will occasionally take even longer than that. Any code we write involving time and timing will need to be able to handle these long tail events.



03_code_timing_distribution.py

There's more we can learn from this plot. Zooming in on the main cluster, some fascinating structure emerges. There isn't just a single peak in the distribution. Instead, there are five of them. They are narrow and distinct. They are equally spaced. The gaps between them are approximately 0.2 milliseconds wide. And they are arrayed in ascending order of height. A simple model of random noise would result in one smooth peak, a gentle hill that rises and falls with an apex not too far from the mean. The intricate structure of this distribution shows that the variation is anything but random. Multiple, evenly spaced peaks betray some mechanism at work behind the curtain. Fully explaining each of these wiggles is beyond the scope of this chapter (that’s author-speak for "I have no idea what’s going on here"), and we don’t need to be able to track the motion of every cog to describe the operation of the clockwork mechanism as a whole. But if you find this type of investigation interesting, I recommend the book Street Coder[5] by Sedat Kapanoğlu. The author will introduce you to the gears that mesh to create patterns like these.

You may be thinking "Wait! Checking the time is also a process. How much time does that take?" I’m so glad you asked.

We can time the `time()` function by taking our previous code and stripping out the busywork computation. What we’re left with is how much time passes between subsequent calls to `time()`.

```
import time
import numpy as np

n_iterations = 10000

elapsed_times = np.zeros(n_iterations)
for i_iteration in range(n_iterations):
        start_time = time.time()
        end_time = time.time()
        elapsed_time = end_time - start_time
        elapsed_times[i_iteration] = elapsed_time

average_time_us = np.mean(elapsed_times) * 1e6
print(f"Average time: {average_time_us:.06} us")
for time in elapsed_times[:12]:
        print(time * 1e6)
```

04_time_timing.py

Taking the mean of 100,000 instances, we find that the average execution time for `time.time()` is 77 nanoseconds. 77 *billionths* of a second. This is very small compared to the 6.55 ms we measured for our busywork code, so it serves a nice check that we were safe to ignore the cost of checking the time.

Inspecting the first dozen reported times reveals something interesting. Most of them are zeros. And those that aren't zero are exactly 0.2384185791015625 microseconds (μs).

```
$ python3 04_time_timing.py

Average time: 0.07689 us
0.2384185791015625
0.0
0.0
0.2384185791015625
0.2384185791015625
0.0
0.0
0.0
0.0
0.0
0.0
0.2384185791015625
```

There is obviously something weird going on here. To understand what, imagine a digital clock in the dashboard of a car and a five-year-old in the back seat eager to get home before their favorite show starts. She asks what time it is every 15 seconds. Your sequence of answers might be [3:47, 3:47, 3:47, 3:48, 3:48, 3:48, 3:48, 3:49, 3:49]. Your child, mentally calculating the elapsed time between these answers, would come up with [0, 0, 1, 0, 0, 0, 1, 0], a pattern very similar to what we are seeing. This is induced by a **discretization** of time. For the child in the car, time is discretized into one minute intervals. In our case, it is 0.238 μs intervals. We don’t know why this is the case, but knowing that it's happening is valuable.

The implication of working with discretized time is that we can’t hope to measure anything with an accuracy higher than half the discretization interval. If you tell the child in the back seat that it's 3:49, she doesn't know whether it's 3:49:00 or 3:49:59 or somewhere in between. If she guesses 3:49:30, she'll be off by at most 30 seconds, 15 seconds on average.

For time differences, this error can be doubled. A time difference is the result of two separate measurements, and their errors compound. If I tell you I started my workout at 11:13 and ended it at 11:23, that might mean I started at 11:13:00 and ended at 11:23:59 (almost 11 minutes of cardio!), or at the other extreme it might mean it was barely over 9 minutes. If you guess my workout was 10 minutes long, your maximum possible error is 60 seconds.

Similarly, for time discretized at 0.238 μs, we can expect time difference errors as as large as 0.238 μs. In measuring `time.time()` we cheated and measured the same function 10,000 times. We were able to average the coarsely discretized measurements to get an estimate that is much finer than a single measurement would allow. It is the same effect as timing a single code block that consists of 10,000 calls to `time()`. The total 770 ms this would take is far larger than the 0.238 μs discretization period, hiding its discretization error.

Even though we can use the trick of repetitive measurement to get an accurate timing estimate, Python doesn't want us to have to work that hard. For timing code there is a better tool than `time.time()`, and that is `time.monotonic()`. It references the monotonic clock we described earlier. It doesn't give a fig about the wall clock or Unix time or NTP. It starts sometime (usually when you power up your computer) and starts counting steadily. When we substitute a `monotonic()` call into our previous code, this is what we get.

```
$ python3 05_monotonic_timing.py

Average time: 0.0732865 us
0.2150190994143486
0.12799864634871483
0.08198549039661884
0.08899951353669167
0.0709842424839735
0.07101334631443024
0.07200287654995918
0.07101334631443024
0.07200287654995918
0.06999471224844456
```

The `monotonic()` function itself runs just a bit faster than `time()`, 73 ns compared to 77 ns. But what is really interesting here is that the zeros are gone. We are nowhere near the discretization limit. Each measurement appears to be fairly accurate on its own.

This lets us see cool patterns that weren't visible before. For instance, notice how the first few measurements are longer than the average. The very first one takes about three times longer than the average. We wouldn't have been able to pull this out from `time()` measurements, when all we had was the mean. If we were so inclined we could also look at how these individual execution times are distributed and search for patterns and dig into the stories behind them. With additional temporal resolution comes great power. (Responsibility is optional.)

There is actually an underlying discretization still,[6] but while working with robots and their sensors in the world, it will be so small that we won't have to consider it. For our purposes, we have found a perfect timer.

Pausing your code

Sometimes you just need your code to sit still and do nothing for a while. This brings us to what is possibly the most boring function of all time: `sleep()`. It does exactly what it sounds like. It puts your code down for a little nap.

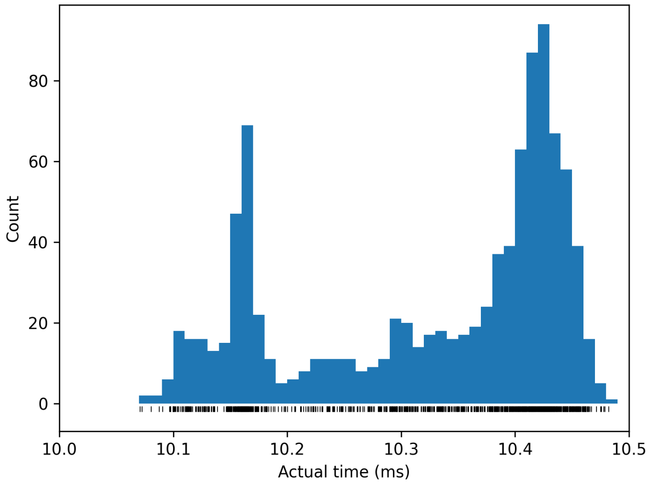
By now we know not to expect that anything time-related is going to go exactly according to plan. `sleep()` is no different. We can check how accurate it is by telling it to sleep for 10 ms and then measuring how long it actually sleeps.

```
import time
import numpy as np

n_iterations = 1000
sleep_duration = .01

sleep_times = np.zeros(n_iterations)
for i_iteration in range(n_iterations):
        start_time = time.monotonic()
        time.sleep(sleep_duration)
        end_time = time.monotonic()
        sleep_time = end_time - start_time
        sleep_times[i_iteration] = sleep_time

average_time = np.mean(sleep_times)
print(
        "Average sleep time:",
        f"{average_time:.09} seconds")
```

06_sleep_timing.py

```
$ python3 06_sleep_timing.py

Average sleep time: 0.0102642315 seconds
```

As expected, the average sleep time is not precisely 10 ms. It's about 2.5% higher than what we asked for. The picture gets even blurrier when we look at the distribution of 10,000 different `sleep(.01)` requests.

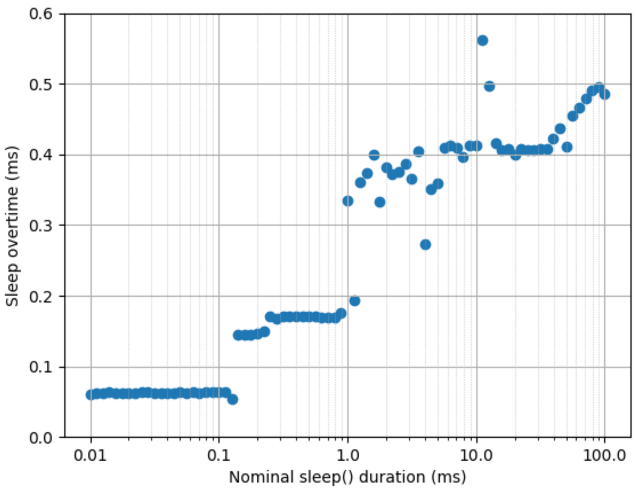


07_sleep_timing_distribution.py

The first thing that jumps out is that all of the measured sleep times are well above the requested 10 ms. This is a feature of how `sleep()` is implemented in Python. It's guaranteed to sleep *at least* the time specified. But as we can see here, it can go over by quite a bit. The median of this distribution is pretty close to 10.4 ms, a full 4% higher than the requested time.

Another thing we can see is that there's not just a single peak. There is a cluster of measurements down between 10.1 and 10.2, then another larger cluster up above 10.4. If the overtime were consistent, we could correct for it and increase the accuracy. But as it is, on any given call to sleep for 10 ms, we don't know if it's going to sleep for 10.1 or 10.4. It's nondeterministic with a significant range.

When we dig a little more and see how this varies for different durations of sleep, even more weirdness emerges. This plot shows the median sleep time (not the average) across many sleep durations.



08_sleep_overhead.py

The first thing that jumps out is that there is a lot of structure here. If the overtime were due to some random process, we might expect it to be either constant across sleep durations or proportional to sleep duration. Either way it would be a straight line. This shows neither of those. To a first approximation it is stepwise constant with jumps at around the .1 ms, .15 ms, and 1 ms durations. The size of the jumps don't follow any obvious pattern. There's also a steady climb that begins at about 40 ms and seems to peak just below 100 ms. And there seems to be quite a bit of variability in the 1 to 10 ms range.

At this point, we have two options for dealing with Python's odd sleep behavior. The first is to create an ever more intricate model of overtime, and explicitly compensate for it. The other is to make sure that whatever code we write doesn't depend on `sleep()` being highly accurate.

The first of these two options has a magnetic appeal. Data such as this, with its intricate structure, is practically screaming for an explanation and mathematical representation. If you can resist the temptation to craft a story explaining what's going on here, you are a stronger person than I.

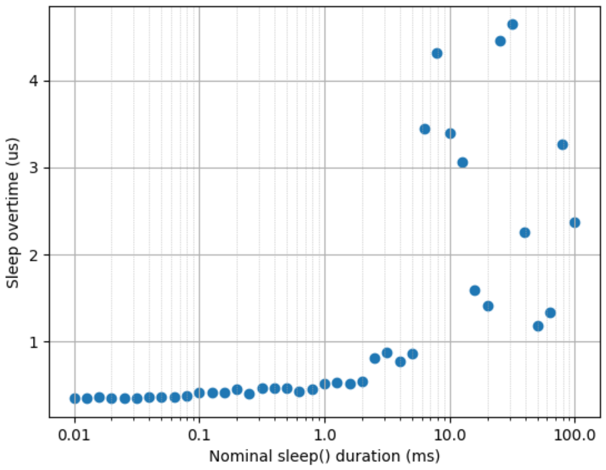
However, we are going to exercise a great deal of discipline and take the second of these options. We'll invest in code that is immune to `sleep()`'s oddities. This will prove to be a wise decision. It turns out that the situation is even more complicated than what we've shown. I've observed on my machine that sleep overtime also depends on what else is running at the moment. And the subtle differences I've observed on my machine will be swamped by differences across different hardware platforms and operating systems. Trying to model away sleep overtime is a losing game. We won't even try to play.

Before moving on, there is a secret third thing: we could write our own hyper-accurate sleep function. Rather than calling `time.sleep()`, we can write a custom sleep function that brute force checks the clock over and over again until the specified duration has passed.

```
def sleep(duration):
    start = time.monotonic()
    end = start + duration
    while time.monotonic() < end:
        pass
```

09_sleep_precise.py

When testing across the same range of durations, the biggest thing that jumps out is that the y-axis is no longer measured in milliseconds, but microseconds. For the most part overtime is about 1000 times less with our brute force method.



09_sleep_precise.py

There's also some interesting behavior where overtime increases slightly up to and just past sleeps of 1 ms, then increases dramatically at around the 10 ms point. I have no idea why this might be. I suspect it's some type of automatic throttling that goes on when the CPU clock gets too many requests from the same process, but that is only speculation. The best part is that the overtime is still so small that we can ignore it. If we needed a sleep function that is accurate to within about a microsecond, we could have one. All it requires is for us to be willing to spam our CPU's monotonic clock. This is a trick that we'll squirrel away, just in case we need it in the future or want to make conversation at parties.[7]

[insert photo of Reign sleeping]

```
time.sleep = True
```

Pacing your code

`sleep()` on its own is only marginally useful for us when working with robots. But we are going to apply it to create a rhythmic timekeeper, a pacemaker for robot code that needs to be executed on a fixed cadence and behave on a predictable timeline.

The first step toward building this is making a metronome.

```
import time

clock_freq_Hz = 2
clock_period = 1 / float(clock_freq_Hz)

test_duration = 10  # seconds
n_iterations = int(clock_freq_Hz * test_duration)

t0 = time.monotonic()
last_completed = t0

for i_iter in range(n_iterations):
    end = t0 + (i_iter + 1) * clock_period
    wait = end - time.monotonic()
    if wait > 0:
        time.sleep(wait)

    completed = time.monotonic()
    duration = completed - last_completed
    print(duration * 1000)
    last_completed = completed
```

10_metronome.py

To kick things off, the metronome needs a pace to set, `clock_freq_Hz`. In this code, we use 2 Hz and calculate a `clock_period` to be 0.5 seconds (500ms). Then, the code starts counting metronome ticks with `i_iter`.

For each tick, the code calculates

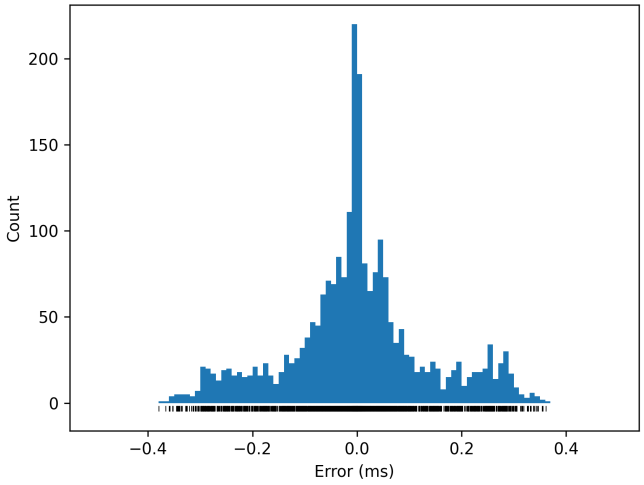
1. when the tick should end
2. what the current time is
3. the difference between the two

Then it waits as long as it needs to via a call to `sleep()`.

The measured tick durations match our target of 500 ms very nicely.

```
$ python3 10_metronome.py

500.85251999553293
499.75492199882865
500.24256703909487
499.7710359748453
499.94934001006186
500.05020102253184
500.1911209546961
499.7375410166569
500.29632402583957
499.922044982668
499.76122297812253
500.0822790316306
500.2248259843327
```

The deviations are all less than half a millisecond, within a tenth of a percent of the nominal tick length. We can definitely work with that. Letting the metronome run longer and looking at the distribution of errors (how far off the actual tick duration was from the ideal), we see a gratifying peak at zero. The shoulders of the peak fall mostly within plus-or-minus a tenth of a millisecond and the maximum and minimum values are both off by less than half a millisecond. This is in line with the behavior of `sleep()` we explored a few pages ago. This relatively small amount of jitter will be negligible for our purposes: coordinating code that controls a robot fumbling around in the world.



11_metronome_errors.py

The central tendency of the tick durations is very close to zero. The median is within a couple of microseconds and the mean is a tenth of that.

```
$ python3 11_metronome_errors.py

Average error 0.00019090167673337 58 ms
Median error -0.0018999970052419046 ms
```

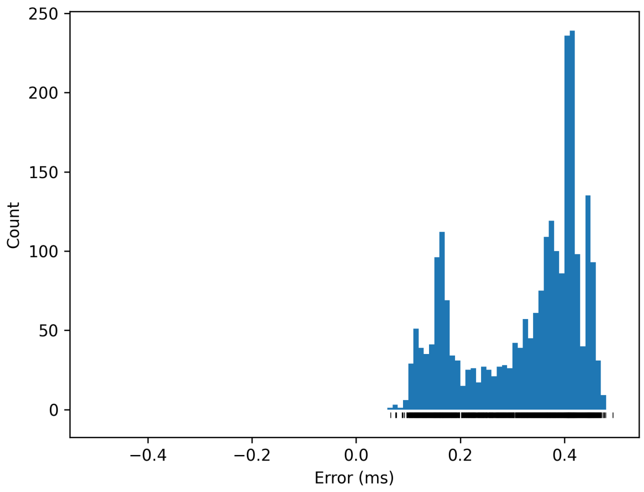
The accuracy of our ticks is not an accident. It comes from how we calculate the end target for each tick. This is the magic line of code:
`end = t0 + (i_iter + 1) * clock_period`

It calculates the end of the current tick by adding up the total time that should have passed since the beginning of the first tick, that is, the number of ticks so far multiplied by the nominal tick duration. That means that even if there is a hiccup and one tick ends up taking a lot longer than expected, the next will automatically compensate for that.

Alternatively, it's also possible to look back just one tick, to check when the previous tick completed and add the nominal tick duration to that to find when the next tick should end.

```
end = last_completed + clock_period
```

While this is not wrong, it's much less robust. If we happen to have an abnormally long tick, the next will do nothing to compensate for it. It will just pick up where the last one left off. Even if the ticks are all consistent, we're stuck with the fact that `sleep()` tends to run long. This version of the metronome creates tick durations that all fall above zero.



11_metronome_errors.py

Their average and median are both measured in hundreds of microseconds, rather than a singular or fraction of a microsecond.

```
$ python3 11_metronome_errors.py

Average error 0.3257934350161398 ms
Median error 0.36869199830107113 ms
```

Even more problematic, these errors add up over time. Given enough ticks of this metronome, even a small average error will add up and cause drift away from the ideal tick sequence. Positive errors result in a shift of the metronome frequency. Two such metronomes, set at the same frequency but running on separate computers, will drift apart and end up counting different numbers of ticks in a day.

Our approach of tying the end time of a tick back to the beginning of tick zero avoids all of this. In fact, average error will always approach zero if you let your metronome run long enough, even with the noisiest and most error prone `sleep()`. Two metronomes set to the same frequency will always report the same number of ticks in a day. Calculating tick end time in this way is a small thing, but it introduces a robustness to the performance of our code that will save us some horrendous debugging experiences down the line.

A Pacemaker

At last, we get to put the pieces together to make a rhythmic timekeeper for our code, a pacemaker that helps it run not too fast and not too slow. When we have multiple processes running in parallel, each overseeing a different part of the robot's activities, it will be quite useful to know that each is running according to its own beat, and will continue to do so consistently until they finish or fail.

To turn our metronome into a pacemaker, all we need to do is to add a workload. Here, we run a stopwatch. The code checks how much time has elapsed since the beginning of tick zero and reports the seconds and milliseconds.

```python
import time

clock_freq_Hz = 4
clock_period = 1 / float(clock_freq_Hz)

test_duration = 10  # seconds
n_iterations = int(clock_freq_Hz * test_duration)

t0 = time.monotonic()
last_completed = t0

for i_iter in range(n_iterations):

    elapsed = time.monotonic() - t0
    seconds = int(elapsed)
    milliseconds = int(elapsed * 1000) % 1000
    print(f"   {seconds}:{milliseconds:03}")

    end = t0 + (i_iter + 1) * clock_period
    wait = end - time.monotonic()
    if wait > 0:
        time.sleep(wait)
    else:
        print("We're running behind!")

    completed = time.monotonic()
    duration = completed - last_completed
    last_completed = completed
```

12_pacemaker.py

Thanks to our carefully considered calculation of tick duration, the result is rock solid and doesn't differ from the ideal enough to register at all at the millisecond level.

```
$ python3 12_pacemaker.py

      0:000
       0:250
       0:500
       0:750
       1:000
       1:250
       1:500
       1:750
       2:000
       2:250
       2:500
       2:750
       3:000
```

If we run this by calculating tick end times from the end of the previous tick we see more than 20 ms of deviation over the course of a ten second run. Not very satisfying! And possibly enough of a quirk to violate the assumptions of code it might interact with in the future.

```
$ python3 12_pacemaker.py


        ...
        8:267
        8:518
        8:768
        9:019
        9:269
        9:519
        9:770
```

The final piece we added was a check in each tick that it finished its work in time. If the workload is too big, it will take longer than what is allotted for the tick. In our pacemaker nothing terrible happens–it just immediately advances to the next tick and will fast forward through as many ticks as necessary to get back on track. But if it's important that our code runs within the tick duration, or more concerning, if our workload is longer than the tick duration every time and the code just gets later and later, then this needs to be reported somewhere. The print statement we added is a placeholder. In later chapters we'll see how to log these too-long ticks and raise large or repeated violations as exceptions to let our process know something is really wrong.

What's next?

Time and pacemakers give us a solid base to start working with more than one process. Multi-process development is an exercise in choreography, and we now have the synchronization tools we need to keep our dancers from kicking each other in the head.

The next step is to practice creating these processes and get them talking to each other. Inter-process message passing is famously hard to get right and even harder to debug. We're going to apply the same philosophy we used to great effect here: Use the simplest tools we can in the most robust way we can find. We want to give our bugs minimal cover to hide.

Recap

You can count the cycles of any regular phenomenon to keep time. Physics is very helpful here.

Unix time is a useful way to represent time in code. It is (roughly) the number of seconds since the beginning of 1970.

Python's `time.time()` checks the Unix time. Beware time zones and leap seconds.

`time.monotonic()` taps into an accurate monotonic clock that works beautifully for timing code.

`time.sleep()` is conservative tool for pausing your code. It never undersleeps, but always oversleeps just a bit.

These tools can be used to construct a pacemaker for keeping repetitive code executing on a strict cadence.

```
time.sleep = True
```

Resources

1. There's an online Unix time converter that I refer to often for quick conversions to and from.
https://www.epochconverter.com/

2. My computer's specifications
Memory: 15.4 GiB
Processor: Intel® Core™ i7-8650U CPU @ 1.90GHz × 8
Graphics: Mesa Intel® UHD Graphics 620 (KBL GT2)
Disk Capacity: 512.1 GB
OS: Ubuntu 20.04.5 LTS, 64 bit
Lenovo Thinkpad T480 that I call Loki
Purchased September 2018

3. The documentation for Python's time module is excellent. I referred to it numerous times while writing this.
https://docs.python.org/3/library/time.html

4. If you are doing the time zone lookup yourself, there's a slick Wikipedia reference for that.
https://en.wikipedia.org/wiki/List_of_UTC_offsets

5. Street Coder by Sedat Kapanoğlu.
streetcoder.org

6. An engaging exploration of the granularity of reported time.
https://shipilev.net/blog/2014/nanotrusting-nanotime/

7. As I was writing this, the ever helpful Twitter Python community informed me that new in Python 3.11 `sleep()` is about to become much more accurate.
https://docs.python.org/3.11/whatsnew/3.11.html#time

About the Author

Robots made their way into Brandon's imagination while he watched the theatrical release The Empire Strikes Back as a child, and they never left. 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 Shih Tzu.

To see more of his work, visit brandonrohrer.com