Python For Large Applications

Aaron Maxwell

aaron@powerfulpython.com

Contents

Chapter 1

Exceptions and Errors

Errors happen. That's why every practical programming language provides a rich framework for dealing with them.

Python's error model is based on *exceptions*. Some of you reading this are familiar with exceptions, and some are not; some of you have used exceptions in other languages, and not yet with Python. This chapter is for all of you.

If you are familiar with how exceptions work in Java, C++ or C#, you'll find Python uses similar concepts, even if the syntax is completely different. And beyond those similarities lie uniquely Pythonic patterns.

We'll start with the basics some of you know. Even if you've used Python exceptions before, I recommend reading all of this chapter. Odds are you will learn useful things, even in sections which appear to discuss what you've seen before.

1.1 The Basic Idea

An *exception* is a way to interrupt the normal flow of code. When an exception occurs, the block of Python code will stop executing - literally in the middle of the line - and immediately jump to *another* block of code, designed to handle the situation.

Often an exception means an error of some sort, but it doesn't have to be. It can be used to signal anticipated events, which are best handled in an interrupt-driven way. Let's illustrate the common, simple cases first, before exploring more sophisticated patterns.

You've already encountered exceptions, even if you didn't realize it. Here's a little program using a `dict`:

```
# favdessert.py
def describe_favorite(category):
    "Describe my favorite food in a category."
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    return "My favorite {} is {}.".format(
        category, favorites[category])

message = describe_favorite("dessert")
print(message)
```

When run, this program exits with an error:

```
Traceback (most recent call last):
  File "favdessert.py", line 12, in <module>
    message = describe_favorite("dessert")
  File "favdessert.py", line 10, in describe_favorite
    category, favorites[category])
KeyError: 'dessert'
```

When you look up a missing dictionary key like this, we say Python *raises* a *KeyError*. (In other languages, the terminology is "throw an exception". Same idea; Python uses the word "raise" instead of "throw".) That `KeyError` is an *exception*. In fact, most errors you see in Python are exceptions. This includes `IndexError` for lists, `TypeError` for incompatible types, `ValueError` for bad values, and so on. When an error occurs, Python responds by raising an exception.

An exception needs to be handled. If not, your program will crash. You handle it with *try-except* blocks. They look like this:

```
# Replace the last few lines with the following:
try:
    message = describe_favorite("dessert")
    print(message)
except KeyError:
    print("I have no favorite dessert. I love them all!")
```

Notice the structure. You have the keyword `try`, followed by an indented block of code, immediately followed by `except KeyError`, which has its own block of code. We say the `except` block *catches* the `KeyError` exception.

Run the program with these new lines, and you get the following output:

```
I have no favorite dessert. I love them all!
```

Importantly, the new program exits successfully; its exit code to the operating system indicates "success" rather than "failure".

Here's how `try` and `except` work:

- Python starts executing lines of code in the `try:` block.
- If Python gets to the end of the `try` block and no exceptions are raised, Python skips over the `except` block completely. None of its lines are executed, and Python proceeds to the next line after (if there is one).
- If an exception is raised anywhere in the `try` block, the program immediately stops - literally in the middle of the line; no further lines in the `try` block will be executed. Python then checks whether the exception type (`KeyError`, in this case) matches an `except` clause. If so, it jumps to the matching block's first line.
- If the exception does *not* match the `except` block, the exception ignores it, acting like the block isn't even there. If no higher-level code has an `except` block to catch it, the program will crash.

Let's wrap these lines of code in a function:

```
def print_description(category):
    try:
        message = describe_favorite(category)
        print(message)
    except KeyError:
        print("I have no favorite {}. I love them all!".format(
            category))
```

Notice how `print_description` behaves differently, depending on what you feed it:

```
>>> print_description("dessert")
I have no favorite dessert. I love them all!
>>> print_description("appetizer")
My favorite appetizer is calamari.
>>> print_description("beverage")
My favorite beverage is coffee.
>>> print_description("soup")
I have no favorite soup. I love them all!
```

Exceptions aren't just for damage control. You will sometimes use them as a flow-control tool, to deal with ordinary variations you know can occur at runtime. Suppose, for example, your program loads data from a file, in JSON format. You import the `json.load` function in your code:

```
from json import load
```

`json` is part of Python's standard library, so it's always available. Now, imagine there's an open-source library called `speedyjson`,[1] with a `load` function just like what's in the standard library - except twice as fast. And your program works with BIG json files, so you want to preferentially use the `speedyjson` version when available. In Python, importing something that doesn't exist raises an `ImportError`:

```
# If speedyjson isn't installed...
>>> from speedyjson import load
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ImportError: No module named 'speedyjson'
```

[1] Not a real library, so far as I know. But after this book is published, I'm sure one of you will make a library with that name, just to mess with me.

How can you use `speedyjson` if it's there, yet gracefully fall back on `json` when it's not? Use a try-except block:

```
try:
    from speedyjson import load
except ImportError:
    from json import load
```

If `speedyjson` is installed and importable, `load` will refer to its version of the function in your code. Otherwise you get `json.load`.

A single `try` can have multiple `except` blocks. For example, `int()` will raise a `TypeError` if passed a nonsensical type; it raises `ValueError` if the type is acceptable, but its value can't be converted to an integer.

```
try:
    value = int(user_input)
except ValueError:
    print("Bad value from user")
except TypeError:
    print("Invalid type (probably a bug)")
```

More realistically, you might log different error events[2] with different levels of severity:

```
try:
    value = int(user_input)
except ValueError:
    logging.error("Bad value from user: %r", user_input)
except TypeError:
    logging.critical(
        "Invalid type (probably a bug): %r", user_input)
```

If an exception is raised, Python will check whether its type matches the first `except` block. If not, it checks the next. The first matching `except` block is executed, and all others are skipped over entirely - so you will never have more than one of the `except` blocks executed for a given `try`. Of course, if none of them match, the exception continues rising until something catches it. (Or the process dies.)

[2] Especially in larger applications, exception handling often integrates with logging. See the logging chapter for details.

There's a good rule of thumb, which I suggest you start building as a habit now: *put as little code as possible in the* `try` *block*. You do this so your `except` block(s) will not catch or mask errors they should not.

Sometimes you will want to have clean-up code that runs *no matter what*, even if an exception is raised. You can do this by adding a `finally` block:

```
try:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

The code in the `finally` block is *always* executed. If an exception is raised in the `try` block, Python will immediately jump to the `finally` block, run its lines, then raise the exception. If an exception is not raised, Python will run all the lines in the `try` block, then run the lines in the `finally` block. It's a way to say, "run these lines no matter what".

You can also have one (or more) except clauses:

```
try:
    line1
    line2
    # etc.
except FirstException:
    line1
    line2
    # etc.
except SecondException:
    line1
    line2
    # etc.
finally:
    line1
    line2
    # etc.
```

What's executed and when depends on whether an exception is raised. If not, the lines in the `try` block run, followed by the lines in the `finally` block; none of the `except` blocks run. If

an exception *is* raised, and it matches one of the `except` blocks, then the `finally` block runs *last*. The order is: the `try` block (up until the exception is raised), then the matching `except` block, and then the finally block.

What if an exception is raised, but there is no matching except block? The `except` blocks are ignored, because none of them match. The lines of code in `try` are executed, up until the exception is raised. Python immediately jumps to the `finally` block; when its lines finish, only then is the exception raised.

It's important to understand this ordering. When you include a `finally` block, and an exception is raised, the code in the `finally` block interjects itself between the code that could run in the `try` block, and the raising of the exception. A `finally` block is like insurance, for code which *must* run, no matter what.

Here's a good example. Imagine writing control code that does batch calculations on a fleet of cloud virtual machines. You issue an API call to rent them, and pay by the hour until you release them. Your code might look something like:

```
# fleet_config is an object with the details of what
# virtual machines to start, and how to connect them.
fleet = CloudVMFleet(fleet_config)
# job_config details what kind of batch calculation to run.
job = BatchJob(job_config)
# .start() makes the API calls to rent the instances,
# blocking until they are ready to accept jobs.
fleet.start()
# Now submit the job. It returns a RunningJob handle.
running_job = fleet.submit_job(job)
# Wait for it to finish.
running_job.wait()
# And now release the fleet of VM instances, so we
# don't have to keep paying for them.
fleet.terminate()
```

Now imagine `running_job.wait()` raises a `socket.timeout` exception (which means the network connection has timed out). This causes a stack trace, and the program crashes, or maybe some higher-level code actually catches the exception.

Regardless, now `fleet.terminate()` is never called. Whoops. That could be *really* expensive.

To save your bank balance (or keep your job), rewrite the code using a `finally` block:

```
fleet = CloudVMFleet(fleet_config)
job = BatchJob(job_config)
try:
    fleet.start()
    running_job = fleet.submit_job(job)
    running_job.wait()
finally:
    fleet.terminate()
```

This code expresses the idea: "no matter what, terminate the fleet of rented virtual machines." Even if an error in `fleet.submit_job(job)` or `running_job.wait()` makes the program crash, it calls `fleet.terminate()` with its dying breath.

Let's look at dictionaries again. When working directly with a dictionary, you can use the "if key in dictionary" pattern to avoid a `KeyError`, instead of try/except blocks:

```
# Another approach we could have taken with favdessert.py
def describe_favorite_or_default(category):
    'Describe my favorite food in a category.'
    favorites = {
        "appetizer": "calamari",
        "vegetable": "broccoli",
        "beverage": "coffee",
    }
    if category in favorites:
        message = "My favorite {} is {}.".format(
        category, favorites[category])
    else:
        message = "I have no favorite {}. I love them all!".format(
           category)
    return message

message = describe_favorite_or_default("dessert")
print(message)
```

The general pattern is:

```
# Using "if key in dictionary" idiom.
if key in mydict:
    value = mydict[key]
else:
    value = default_value

# Contrast with "try/except KeyError".
try:
    value = mydict[key]
except KeyError:
    value = default_value
```

Many developers prefer using the "if key in dictionary" idiom, or using `dict.get()`. But these aren't always the best choice. They are only options if your code has direct access to the dictionary, for one thing. Maybe `describe_favorite()` is part of a library, and you can't change it. Even if it's open-source, you have better things to do than fork a library every time a function interface isn't convenient. Or maybe `describe_favorite()` is code you control, but you just don't *want* to change it, for any number of good reasons. A try-except block catching `KeyError` solves all these problems, because it lets you handle the situation without modifying any code inside `describe_favorite()` itself.

1.2 Exceptions Are Objects

An exception is an object: an instance of an exception class. `KeyError`, `IndexError`, `TypeError` and `ValueError` are all built-in classes, which inherit from a base class called `Exception`. Writing code like `except KeyError:` means "if the exception just raised is of type `KeyError`, run this block of code."

So far, we haven't dealt with those exception objects directly. And often, you don't need to. But sometimes you want more information about what happened, and capturing the exception object can help. Here's the structure:

```
try:
    do_something()
except ExceptionClass as exception_object:
    handle_exception(exception_object)
```

where *ExceptionClass* is some exception class, like `KeyError`, etc. In the except block,

exception_object will be an instance of that class. You can choose any name for that variable; no one actually calls it exception_object, preferring shorter names like ex, exc, or err. The methods and contents of that object will depend on the kind of exception, but almost all will have an attribute called args. That will be a tuple of what was passed to the exception's constructor. The args of a KeyError, for example, will have one element - the missing key:

```
# Atomic numbers of noble gasses.
nobles = {'He': 2, 'Ne': 10,
  'Ar': 18, 'Kr': 36, 'Xe': 54}
def show_element_info(elements):
   for element in elements:
       print('Atomic number of {} is {}'.format(
             element, nobles[element]))
try:
    show_element_info(['Ne', 'Ar', 'Br'])
except KeyError as err:
    missing_element = err.args[0]
    print('Missing data for element: ' + missing_element)
```

Running this code gives you the following output:

```
Atomic number of Ne is 10
Atomic number of Ar is 18
Missing data for element: Br
```

The interesting bit is in the except block. Writing except KeyError as err stores the exception object in the err variable. That lets us look up the offending key, by peeking in err.args. Notice we could not get the offending key any other way, unless we want to modify show_element_info (which we may not want to do, or perhaps *can't* do, as described before).

Let's walk through a more sophisticated example. In the os module, the makedirs function will create a directory:

```
# Creates the directory "riddles", relative
# to the current directory.
import os
os.makedirs("riddles")
```

By default, if the directory already exists, makedirs will raise FileExistsError:[3] Imagine you are writing a web application, and need to create an upload directory for each new user.

[3] Python 2 does something different, and more complex. We'll cover that in detail later. For now, keep reading.

That directory should not exist; if it does, that's an error and needs to be logged. Our upload-directory-creating function might look like this:

```
# First version....
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError:
        logging.error(
            "Upload dir for new user already exists")
```

It's great we are detecting and logging the error, but the error message isn't informative enough to be helpful. We at least need to know the offending username, but it's even better to know the directory's full path (so you don't have to dig in the code to remind yourself what `UPLOAD_ROOT` was set to).

Fortunately, `FileExistsError` objects have an attribute called `filename`. This is a string, and the path to the already-existing directory. We can use that to improve the log message:

```
# Better version!
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

Only the `except` block is different. That `filename` attribute is perfect for a useful log message.

1.3 Raising Exceptions

`ValueError` is a built-in exception that signals some data is of the correct type, but its format isn't valid. It shows up everywhere:

```
>>> int("not a number")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'not a number'
```

Your own code can raise exceptions, just like `int()` does. You should, in fact, so you have better error messages. (And sometimes for other reasons - more on that later.) You do so with the *raise* statement. The most common form is this:

```
raise ExceptionClass(arguments)
```

For `ValueError` specifically, it might look like:

```
def positive_int(value):
    number = int(value)
    if number <= 0:
        raise ValueError("Bad value: " + str(value))
    return number
```

Focus on the `raise` line in `positive_int`. You simply create an instance of `ValueError`, and pass it directly to `raise`. Really, the syntax is `raise exception_object` - though usually you just create the object inline. `ValueError`'s constructor takes one argument, a descriptive string. This shows up in stack traces and log messages, so be sure to make it informative and useful:

```
>>> positive_int("-3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in positive_int
ValueError: Bad value: -3
>>> positive_int(-7.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in positive_int
ValueError: Bad value: -7.0
```

Let's show a more complex example. Imagine you have a Money class:

```
class Money:
    def __init__(self, dollars, cents):
        self.dollars = dollars
        self.cents = cents
    def __repr__(self):
        'Renders the object nicely on the prompt.'
        return "Money({},{})".format(
            self.dollars, self.cents)
    # Plus other methods, which aren't important to us now.
```

Your code needs to create Money objects from string values, like "$140.75". The constructor takes dollars and cents, so you create a function to parse that string and instantiate Money for you:

```
import re
def money_from_string(amount):
    # amount is a string like "$140.75"
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

This function[4] works like this:

```
>>> money_from_string("$140.75")
Money(140,75)
>>> money_from_string("$12.30")
Money(12,30)
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in money_from_string
AttributeError: 'NoneType' object has no attribute 'group'
```

This error isn't clear; you must read the source and think about it to understand what went wrong. We have better things to do than decrypt stack traces. You can improve this function's

[4] It's better to make this a class method of Money, rather than a separate function. That is a separate topic, though; see @classmethod in the object-oriented patterns chapter for details.

usability by having it raise a `ValueError`.

```
import re
def money_from_string(amount):
    match = re.search(
        r'^\$(?P<dollars>\d+)\.(?P<cents>\d\d)$', amount)
    # Adding the next two lines here
    if match is None:
        raise ValueError("Invalid amount: " + repr(amount))
    dollars = int(match.group('dollars'))
    cents = int(match.group('cents'))
    return Money(dollars, cents)
```

The error message is now much more informative:

```
>>> money_from_string("Big money")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in money_from_string
ValueError: Invalid amount: 'Big money'
```

1.4 Catching And Re-raising

In an `except` block, you can re-raise the current exception. It's very simple; just write `raise` by itself, with no arguments:

```
try:
    do_something()
except ExceptionClass:
    handle_exception()
    raise
```

Notice you don't need to store the exception object in a variable. It's a shorthand, exactly equivalent to this:

```
try:
    do_something()
except ExceptionClass as err:
    handle_exception()
    raise err
```

This "catch and release" only works in an `except` block. It requires that some higher-level code will catch the exception and deal with it. Yet it enables several useful code patterns. One is when you want to delegate handling the exception to higher-level code, but also want to inject some extra behavior closer to the exception source. For example:

```
try:
    process_user_input(value)
except ValueError:
    logging.info("Invalid user input: %s", value)
    raise
```

If `process_user_input` raises a `ValueError`, the `except` block will execute the logging line. Other than that, the exception propagates as normal.

It's also useful when you need to execute code before deciding whether to re-raise the exception at all. Earlier, we used a try/except block pair to create an upload directory, logging an error if it already exists:

```
# Remember this? Python 3 code, from earlier.
import os
import logging
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except FileExistsError as err:
        logging.error("Upload dir already exists: %s",
            err.filename)
```

This relies on `FileExistsError`, which was introduced in Python 3. How could you do this in Python 2? Even if you no longer write code in Python 2, it's worth studying the different approach required, as it demonstrates a widely useful exception-handling pattern. Let's take a look.

`FileExistsError` subclasses the more general `OSError`. This exception type has been around since the early days of Python, and in Python 2, `makedirs` simply raises `OSError`. But `OSError` can indicate many problems other than the directory already existing: a lack of filesystem permissions, a system call getting interrupted, even a timeout over a network-mounted filesystem. We need a way to discriminate between these possibilities.

`OSError` objects have an `errno` attribute, indicating the precise error. These correspond to the variable `errno` in a C program, with different integer values meaning different error conditions. Most higher-level languages - including Python - reuse the constant names defined in the C API; in particular, the standard constant for "file already exists" is `EEXIST` (which happens to be set to the number 17 in most implementations). These constants are defined in the `errno` module in Python, so we just type `from errno import EEXIST` in our program.

In versions of Python with `FileExistsError`, the general pattern is:

- Optimistically create the directory, and
- if `FileExistsError` is raised, catch it and log the event.

In Python 2, we must do this instead:

- Optimistically create the directory.
- if `OSError` is raised, catch it.
- Inspect the exception's `errno` attribute. If it's equal to `EEXIST`, this means the directory already existed; log that event.
- If `errno` is something else, it means we don't want to catch this exception here; re-raise the error.

The code:

```
# How to accomplish the same in Python 2.
import os
import logging
from errno import EEXIST
UPLOAD_ROOT = "/var/www/uploads/"
def create_upload_dir(username):
    userdir = os.path.join(UPLOAD_ROOT, username)
    try:
        os.makedirs(userdir)
    except OSError as err:
        if err.errno != EEXIST:
            raise
        logging.error("Upload dir already exists: %s",
            err.filename)
```

The only difference between the Python 2 and 3 versions is the "except" clause. But there's a lot going on there. First, we're catching `OSError` rather than `FileExistsError`. But we may or may not re-raise the exception, depending on the value of its `errno` attribute. Basically, a value of `EEXIST` means the directory already exists. So we log it and move on. Any other value indicates an error we aren't prepared to handle right here, so re-raise in order to pass it to higher level code.

1.5 The Most Diabolical Python Anti-Pattern

You know about "design patterns": time-tested solutions to common code problems. And you've probably heard of "anti-patterns": solutions people often choose to a code problem, because it *seems* to be a good approach, but actually turn out to be harmful.

In Python, one antipattern is most harmful of all.

I wish I could not even tell you about it. If you don't know it exists, you can't use it in your code. Unfortunately, you might stumble on it somewhere and adopt it, not realizing the danger. So, it's my duty to warn you.

Here's the punchline. The following is the most self-destructive code a Python developer can write:

```
try:
    do_something()
except:
    pass
```

Python lets you completely omit the argument to `except`. If you do that, it will catch *every exception*. That's pretty harmful right there; remember, the more pin-pointed your `except` clauses are, the more precise your error handling can be, without sweeping unrelated errors under the rug. And typing `except:` will sweep *every* unrelated error under the rug.

But it's much worse than that, because of the `pass` in the `except` clause. What `except: pass` does is silently and invisibly hide error conditions that you'd otherwise quickly detect and fix.

(Instead of `"except:"`, you'll sometimes see variants like `"except Exception:"` or `"except Exception as ex:"`. They amount to the same thing.)

This creates the **worst kind of bug**. Have you ever been troubleshooting a bug, and just couldn't figure out where in the code base it came from, even after hours of searching, getting

more and more frustrated as the minutes and hours roll by? *This is how you create that in Python.*

I first understood this anti-pattern after joining an engineering team, in an explosively-growing Silicon Valley start-up. The company's product was a web service, which needed to be up 24/7. So engineers took turns being "on call" in case of a critical issue. An obscure Unicode bug somehow kept triggering, waking up an engineer - in the middle of the night! - several times a week. But no one could figure out how to reproduce the bug, or even track down exactly how it was happening in the large code base.

After a few months of this nonsense, some of the senior engineers got fed up and devoted themselves to rooting out the problem. One senior engineer did nothing for *three full days* except investigate it, ignoring other responsibilities as they piled up. He made some progress, and took useful notes on what he found, but in the end did not figure it out. He ran out of time, and had to give up.

Then, a second senior engineer took over. Using those notes as a starting point, he also dug into it, ignoring emails and other commitments for *another three full days* to track down the problem. And he failed. He make progress, adding usefully to the notes. But in the end, he had to give up too, when other responsibilities could no longer be ignored.

Finally, after these six days, they passed the torch to me - the new engineer on the team. I wasn't too familiar with the code base, but their notes gave me a lot to go on. So I dove in on Day 7, and completely ignored everything else for six hours straight.

And finally, late in the day, I was able to isolate the problem to a single block of code:

```
try:
    extract_address(location_data)
except:
    pass
```

That was it. The data in `location_data` was corrupted, causing the `extract_address` call to raise a `UnicodeError`. Which the program then *completely silenced*. Not even producing a stack trace; simply moving on, as if nothing had happened.

After nearly *seven full days* of engineer effort, we pinpointed the error to this one block of code. I un-suppressed the exception, and was almost immediately able to reproduce the bug - with a full and very informative stack trace.

Once I did that, can you guess how long it took us to fix the bug?

TEN MINUTES.

That's right. A full WEEK of engineer time was wasted, all because this anti-pattern somehow snuck into our code base. Had it not, then the first time it woke up an engineer, it would have been obvious what the problem was, and how to fix it. The code would have been patched by the end of the day, and we would all have moved on to bigger and better things.

The cruelty of this anti-pattern comes from how it completely hides *all* helpful information. Normally, when a bug causes a problem in your code, you can inspect the stack trace; identify what lines of code are involved; and start solving it. With The Most Diabolical Python Antipattern (TMDPA), none of that information is available. What line of code did the error come from? Which *file* in your Python application, for that matter? In fact, what was the exception type? Was it a `KeyError`? A `UnicodeError`? Or even a `NameError`, coming from a mis-typed variable name? Was it `OSError`, and if so, what was its `errno`? You don't know. You *can't* know.

In fact, TMDPA often **hides the fact that an error even occurs**. This is one of the ways bugs hide from you during development, then sneak into production, where they're free to cause real damage.

We never did figure out why the original developer wrote `except: pass` to begin with. I think that at the time, `location_data` may have sometimes been empty, causing `extract_address` to innocuously raise a `ValueError`. In other words, if `ValueError` was raised, it was appropriate to ignore that and move on. By the time the other two engineers and I were involved, the code base had changed so that was no longer how things worked. But the broad `except` block remained, like a land mine lurking in a lush field.

So why do people do this? Well, no one *wants* to wreak such havoc in their Python code, of course. People do this because they expect errors to occur in the normal course of operation, in some specific way. They are simply catching too broadly, without realizing the full implications.

So what do you do instead? There are two basic choices. In most cases, it's best to modify the `except` clause to catch a more specific exception. For the situation above, this would have been a much better choice:

```
try:
    extract_address(location_data)
except ValueError:
    pass
```

Here, `ValueError` is caught and appropriately ignored. If `UnicodeError` raises, it propagates and (if not caught) the program crashes. That would have been *great* in our situation. The error

log would have a full stack trace clearly telling us what happened, and we'd be able to fix it in ten minutes.

As a variation, you may want to insert some logging:

```
try:
    extract_address(location_data)
except ValueError:
    logging.info(
        "Invalid location for user %s", username)
```

The other reason people write `except: pass` is a bit more valid. Sometimes, a code path simply must broadly catch all exceptions, and continue running regardless. This is common in the top-level loop for a long-running, persistent process. The problem is that `except: pass` hides all information about the problem, including that the problem even exists.

Fortunately, Python provides an easy way to capture that error event, and all the information you need to fix it. The `logging` module has a function called `exception`, which will log your message *along with the full stack trace of the current exception*. So you can write code like this:

```
import logging
def get_number():
    return int('foo')
try:
    x = get_number()
except:
    logging.exception('Caught an error')
```

The log will contain the error message, followed by a formatted stack trace spread across several lines:

```
ERROR:root:Caught an error
Traceback (most recent call last):
  File "example-logging-exception.py", line 5, in <module>
    x = get_number()
  File "example-logging-exception.py", line 3, in get_number
    return int('foo')
ValueError: invalid literal for int() with base 10: 'foo'
```

This stack trace is *priceless*. Especially in more complex applications, it's often not enough to know the file and line number where an error occurs. It's at least as important to know *how* that

function or method was called... what path of executed code led to it being invoked. Otherwise you can never determine what conditions lead to that function or method being called in the first place. The stack trace, in contrast, gives you everything you need to know.

I wish `"except: pass"` was not valid Python syntax. I think much grief would be spared if it was. But it's not my call, and changing it now is probably not practical. Your only defense is to be vigilant. That includes educating your fellow developers. Does your team hold regular engineering meetings? Ask for five minutes at the next one to explain this antipattern, the cost it has to everyone's productivity, and the simple solutions.

Even better: if there are local Python or technical meetups in your area, volunteer to give a short talk - five to fifteen minutes. These meetups almost always need speakers, and you will be helping so many of your fellow developers in the audience.

There is a longer article explaining this situation at https://powerfulpython.com/blog/the-most-diabolical-python-antipattern/ . Simply sharing the URL will educate people too. And feel free to write your own blog post, with your own explanation of the situation, and how to fix it. Serve your fellow engineers by evangelizing this important knowledge.

Chapter 2

Logging in Python

Logging is critical in many kinds of software. For long-running software systems, it enables continuous telemetry and reporting. And for *all* software, it can provide priceless information for troubleshooting and post-mortems. The bigger the application, the more important logging becomes. But even small scripts can benefit.

Python provides logging through the `logging` module. In my opinion, this module is one of the more technically impressive parts of Python's standard library. It's well-designed, flexible, thread-safe, and richly powerful. It's also complex, with many moving parts, making it hard to learn well. This chapter gets you over most of that learning curve, so you can fully benefit from what `logging` has to offer. The payoff is well worth it, and will serve you for years.

Broadly, there are two ways to use `logging`. One, which I'm calling the *basic interface*, is appropriate for scripts - meaning, Python programs that are small enough to fit in a single file. For more substantial applications, it's typically better to use *logger objects*, which give more flexible, centralized control, and access to logging hierarchies. We'll start with the former, to introduce key ideas.

2.1 The Basic Interface

Here's the easiest way to use Python's logging module:

```
import logging
logging.warning('Look out!')
```

Save this in a script and run it, and you'll see this printed to standard output:

```
WARNING:root:Look out!
```

You can do useful logging right away, by calling functions in the `logging` module itself. Notice you invoke `logging.warning()`, and the output line starts with `WARNING`. You can also call `logging.error()`, which gives a different prefix:

```
ERROR:root:Look out!
```

We say that `warning` and `error` are at different *message log levels*. You have a spectrum of log levels to choose from, in order of increasing severity:[1]

debug Detailed information, typically of interest only when diagnosing problems.

info Confirmation that things are working as expected.

warning An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.

error Due to a more serious problem, the software has not been able to perform some function.

critical A serious error, indicating that the program itself may be unable to continue running.

You use them all just like `logging.warning()` and `logging.error()`:

```
logging.debug("Small detail. Useful for troubleshooting.")
logging.info("This is informative.")
logging.warning("This is a warning message.")
logging.error("Uh oh. Something went wrong.")
logging.critical("We have a big problem!")
```

Each has a corresponding uppercased constant in the library (e.g., `logging.WARNING` for `logging.warning()`). You use these when defining the *log level threshold*. Run the above, and here is the output:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

[1] These beautifully crisp descriptions, which I cannot improve upon, are taken from https://docs.python.org/-3/howto/logging.html .

Where did the debug and info messages go? As it turns out, the default logging threshold is `logging.WARNING`, which means only messages of that severity or greater are actually generated; the others are ignored completely. The order matters in the list above; debug is considered strictly less severe than `info`, and so on. Change the log level threshold using the `basicConfig` function:

```
logging.basicConfig(level=logging.INFO)
logging.info("This is informative.")
logging.error("Uh oh. Something went wrong.")
```

Run this new program, and the `INFO` message gets printed:

```
INFO:root:This is informative.
ERROR:root:Uh oh. Something went wrong.
```

Again, the order is `debug()`, `info()`, `warning()`, `error()` and `critical()`, from lowest to highest severity. When we set the log level threshold, we declare that we only want to see messages of that level or higher. Messages of a lower level are not printed. When you set `level` to `logging.DEBUG`, you see everything; set it to `logging.CRITICAL`, and you only see critical messages, and so on.

The phrase "log level" means two different things, depending on context. It can mean the severity of a message, which you set by choosing which of the functions to use - `logging.warning()`, etc. Or it can mean the threshold for ignoring messages, which is signaled by the constants: `logging.WARNING`, etc.

You can also use the constants in the more general `logging.log` function - for example, a debug message:

```
logging.log(logging.DEBUG,
    "Small detail. Useful for troubleshooting.")
logging.log(logging.INFO, "This is informative.")
logging.log(logging.WARNING, "This is a warning message.")
logging.log(logging.ERROR, "Uh oh. Something went wrong.")
logging.log(logging.CRITICAL, "We have a big problem!")
```

This lets you modify the log level dynamically, at runtime:

```
def log_results(message, level=logging.INFO):
    logging.log(level, "Results: " + message)
```

2.1.1 Why do we have log levels?

If you haven't worked with similar logging systems before, you may wonder why we have different log levels, and why you'd want to control the filtering threshold. It's easiest to see this if you've written Python scripts that include a number of `print()` statements - including some useful for diagnosis when something goes wrong, but a distraction when everything is working fine.

The fact is, some of those `print()` statements are more important than others. Some indicate mission-critical problems you always want to know about - possibly to the point of waking up an engineer, so they can deploy a fix immediately. Some are important, but can wait until the next work day - and you definitely do NOT want to wake anyone up for that. Some are details which may have been important in the past, and might be in the future, so you don't want to remove them; in the meantime, they are just line noise.

Having log levels solves all these problems. As you develop and evolve your code over time, you continually add new logging statements of the appropriate severity. You now even have the freedom to be proactive. With "logging" via `print()`, each log statement has a cost - certainly in signal-to-noise ratio, and also potentially in performance. So you might debate whether to include that print statement at all. But with `logging`, you can insert `info` messages, for example, to log certain events occurring as they should. In development, those INFO messages can be very useful to verify certain things are happening, so you can modify the log level to produce them. On production, you may not want to have them cluttering up the logs, so you just set the threshold higher. Or if you are doing some kind of monitoring on production, and temporarily need that information, you can adjust the log level threshold to output those messages; when you are finished, you can adjust it back to exclude them again.

When troubleshooting, you can liberally introduce `debug`-level statements to provide extra detailed statements. When done, you can just adjust the log level to turn them off. You can leave them in the code without cost, eliminating any risk of introducing more bugs when you go through and remove them. This also leaves them available if they are needed in the future.

The log level symbols are actually set to integers. You can theoretically use these numbers instead, or even define your own log levels that are (for example) a third of the way between `WARNING` and `ERROR`. In normal practice, it's best to use the predefined logging levels. Doing otherwise makes your code harder to read and maintain, and isn't worthwhile unless you have a compelling reason.

For reference, the numbers are 50 for `CRITICAL`, 40 for `ERROR`, 30 for `WARNING`, 20 for `INFO`,

and 10 for DEBUG. So when you set the log level threshold, it's actually setting a number. The only log messages emitted are those with a level greater than or equal to that number.

2.2 Configuring The Basic Interface

You saw above you can change the loglevel threshold by calling a function called basicConfig:

```
logging.basicConfig(level=logging.INFO)
logging.debug("You won't see this message!")
logging.error("But you will see this one.")
```

If you use it at all, basicConfig must be called exactly once, and it must happen before the first logging event. (Meaning, before the first call to debug(), or warning(), etc.) Additionally, if your program has several threads, it must be called from the main thread - and *only* the main thread. [2]

You already met one of the configuration options, level. This is set to the log level threshold, and is one of DEBUG, INFO, WARNING, ERROR, or CRITICAL. Some of the other options include:

filename Write log messages to the given file, rather than stderr.

filemode Set to "a" to append to the log file (the default), or "w" to overwrite.

format The format of log records.

level The log level threshold, described above.

By default, log messages are written to standard error. You can also write them to a file, one per line, to easily read later. Do this by setting filename to the log file path. By default it appends log messages, meaning that it will only add to the end of the file if it isn't empty. If you'd rather the file be emptied before the first log message, set filemode to "w". Be careful about doing that, of course, because you can easily lose old log messages if the application restarts:

```
# Wipes out previous log entries when program restarts
logging.basicConfig(filename="log.txt", filemode="w")
logging.error("oops")
```

[2] These restrictions aren't in place for logger objects, described later.

The other valid value is "a", for append - that's the default, and probably will serve you better in production. "w" can be useful during development, though.

`format` defines what chunks of information the final log record will include, and how they are laid out. These chunks are called "attributes" in the `logging` module docs. One of these attributes is the actual log message - the string you pass when you call `logging.warning()`, and so on. Often you will want to include other attributes as well. Consider the kind of log record we saw above:

```
WARNING:root:Collision imminent
```

This record has three attributes, separated by colons. First is the log level name; last is the actual string message you pass when you call `logging.warning()`. (In the middle is the name of the underlying logger object. `basicConfig` uses a logger called "root"; we'll learn more about that later.)

You specify the layout you want by setting `format` to a string that defines certain named fields, according to percent-style formatting. Three of them are `levelname`, the log level; `message`, the message string passed to the logging function; and `name`, the name of the underlying logger. Here's an example:

```
logging.basicConfig(
    format="Log level: %(levelname)s, msg: %(message)s")
logging.warning("Collision imminent")
```

If you run this as a program, you get the following output:

```
Log level: WARNING, msg: Collision imminent
```

It turns out the default formatting string is

```
%(levelname)s:%(name)s:%(message)s
```

You indicate named fields in percent-formatting by `%(FIELDNAME)X`, where "X" is a type code: `s` for string, `d` for integer (decimal), and `f` for floating-point.

Many other attributes are provided, if you want to include them. Here's a select few from the full list:[3]

[3] https://docs.python.org/3/library/logging.html#logrecord-attributes

Attribute	Format	Description
asctime	`%(asctime)s`	Human-readable date/time
funcName	`%(funcName)s`	Name of function containing the logging call
lineno	`%(lineno)d`	The line number of the logging call
message	`%(message)s`	The log message
pathname	`%(pathname)s`	Full pathname of the source file of the logging call
levelname	`%(levelname)s`	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
name	`%(name)s`	The logger's name

You might be wondering why log record format strings use Python 2's percent-formatting style, when everything else in Python 3 uses the newer, brace-style string formatting. As it turns out, the conversion was attempted, but backwards-compatibility reasons made percent-formatting the only practical choice for the logging module, even after the Python 3 reboot.

If you want to use the newer string formatting badly enough, there are things you can do - there's even a standard recipe.[4] But doing so is complicated enough that it may not be worth the effort, and it won't help with legacy code. I recommend you simply cooperate with the situation, and use percent formatting with your Python logging.

2.3 Passing Arguments

You often want to include some kind of runtime data in the logged message. When you construct the message to log, specify the final log message like this:

```
num_fruits = 14
fruit_name = "oranges"
logging.info(
    "We ate %d of your %s. Thanks!",
    num_fruits, fruit_name)
```

The output:

[4] https://docs.python.org/3/howto/logging-cookbook.html#use-of-alternative-formatting-styles

```
INFO:root:We ate 14 of your oranges. Thanks!
```

We call `info` with three parameters. First is the format string; the second and third are arguments. The general form is

```
logging.info(format, *args)
```

You can pass zero or more arguments, so long as each has a field in the format string:

```
logging.info("%s, %s, %s, %s, %s, %s and %s",
    "Doc", "Happy", "Sneezy", "Bashful",
    "Dopey", "Sleepy", "Grumpy")
```

You *must* resist the obvious temptation to format the string fully, and pass that to the logging function:

```
num_fruits = 14
fruit_name = "oranges"
logging.warning(
    "Don't do this: %d %s" % (num_fruits, fruit_name))
logging.warning(
    "Or even this: {:d} {:s}".format(
    num_fruits, fruit_name))
```

This works, of course, in the sense that you will get correct log messages. However, it's wasteful, and surrenders important benefits `logging` normally provides. Remember: when the line of code with the log message is executed, it may not actually trigger a log event. If the log level threshold is higher than the message itself, the line does nothing. In that case, there is no reason to format the string.

In the first form, the string is formatted if and only if a log event actually happens, so that's fine. But if you format the string yourself, it's *always* formatted. That takes up system memory and CPU cycles even if no logging takes place. If the code path with the logging call is only executed occasionally, that's not a big deal. But it impairs the program when a debug message is logged in the middle of a tight loop. When you originally code the line, you never really know where it might migrate in the future, or who will call your function in ways you never imagined.

So just use the supported form, where the first argument is the format string, and subsequent arguments are the parameters for it. You can also use named fields, by passing a dictionary as the second argument:

```
fruit_info = {"count": 14, "name": "oranges"}
logging.info(
    "We ate %(count)d of your %(name)s. Thanks!",
    fruit_info)
```

2.4 Beyond Basic: Loggers

The basic interface is simple and easy to set up. It works well in single-file scripts. Larger Python applications tend to have different logging needs, however. `logging` meets these needs through a richer interface, called *logger objects* - or simply, *loggers*.

Actually, you have been using a logger object all along: when you call `logging.warning()` (or the other log functions), they convey messages through what is called the *root logger* - the primary, default logger object. This is why the word "root" shows in some example output.

`logger.basicConfig` operates on this root logger. You can fetch the actual root logger object by calling `logging.getLogger`:

```
>>> logger = logging.getLogger()
>>> logger.name
'root'
```

As you can see, it knows its name is "root". Logger objects have all the same functions (methods, actually) the `logging` module itself has:

```
import logging
logger = logging.getLogger()
logger.debug("Small detail. Useful for troubleshooting.")
logger.info("This is informative.")
logger.warning("This is a warning message.")
logger.error("Uh oh. Something went wrong.")
logger.critical("We have a big problem!")
```

Save this in a file and run it, and you'll see the following output:

```
This is a warning message.
Uh oh. Something went wrong.
We have a big problem!
```

This is different from what we saw with `basicConfig`, which printed out this instead:

```
WARNING:root:This is a warning message.
ERROR:root:Uh oh. Something went wrong.
CRITICAL:root:We have a big problem!
```

At this point, we've taken steps backward compared to `basicConfig`. Not only is the log message unadorned by the log level, or anything else useful. The log level threshold is hard-coded to `logging.WARNING`, with no way to change it. The logging output will be written to standard error, and no where else, regardless of where you actually need it to go.

Let's take inventory of what we want to control here. We want to choose our log record format. And further, we want to be able to control the log level threshold, and write messages to different streams and destinations. You do this with a tool called *handlers*.

2.5 Log Destinations: Handlers and Streams

By default, loggers write to standard error. You can select a different destination - or even *several* destinations - for each log record:

- You can write log records to a file. Very common.
- You can, while writing records to that file, *also* parrot it to stderr.
- Or to stdout. Or both.
- You can simultaneously log messages to two different files.
- In fact, you can log (say) `INFO` and higher messages to one file, and `ERROR` and higher to another.
- You can write log records to a remote log server, accessed via a REST HTTP API.
- Mix and match all the above, and more.
- And you can set a different, custom log format for each destination.

This is all managed through what are called *handlers*. In Python logging, a handler's job is to take a log record, and make sure it gets recorded in the appropriate destination. That destination can be a file; a stream like stderr or stdout; or something more abstract, like inserting into a queue, or transmitting via an RPC or HTTP call.

By default, logger objects don't have any handlers. You can verify this using the `hasHandlers` method:

```
>>> logger = logging.getLogger()
>>> logger.hasHandlers()
False
```

With no handler, a logger has the following behavior:

- Messages are written to stderr.
- Only the message is written, nothing else. There's no way to add fields or otherwise modify it.
- The log level threshold is `logging.WARNING`. There is no way to change that.

To change this, your first step is to create a handler. Nearly all logger objects you ever use will have custom handlers. Let's see how to create a simple handler that writes messages to a file, called "log.txt".

```
import logging
logger = logging.getLogger()
log_file_handler = logging.FileHandler("log.txt")
logger.addHandler(log_file_handler)
logger.debug("A little detail")
logger.warning("Boo!")
```

The `logging` module provides a class called `FileHandler`. It takes a file path argument, and will write log records into that file, one per line. When you run this code, `log.txt` will be created (if it doesn't already exist), and will contain the string "Boo!" followed by a newline. (If `log.txt` did exist already, the logged message would be *appended* to the end of the file.)

But "A little detail" is not written, because it's below the default logger threshold of `WARNING`. We change that by calling a method named `setLevel` on the logger object:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
log_file_handler = logging.FileHandler("log.txt")
logger.addHandler(log_file_handler)
logger.debug("A little detail")
logger.warning("Boo!")
```

This writes the following in "log.txt":

```
A little detail
Boo!
```

Confusingly, you can call `setLevel` on a logger with no handlers, *but it has no effect*:

```
# Doing it wrong:
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG) # No effect.
logger.debug("This won't work :(")
```

To change the threshold from the default of `logging.WARNING`, you must both add a handler, *and* change the logger's level.

What if you want to log to stdout? Do that with a `StreamHandler`:

```
import logging
import sys
logger = logging.getLogger()
out_handler = logging.StreamHandler(sys.stdout)
logger.addHandler(out_handler)
logger.warning("Boo!")
```

If you save this in a file and run it, you'll get "Boo!" on standard output. Notice that `logging.StreamHandler` takes `sys.stdout` as its argument. You can create a `StreamHandler` without an argument too, in which case it will write its records to standard error:

```
import logging
logger = logging.getLogger()
# Same as StreamHandler(sys.stderr)
stderr_handler = logging.StreamHandler()
logger.addHandler(stderr_handler)
logger.warning("This goes to standard error")
```

In fact, you can pass any file-like object; The object just needs to define compatible `write` and `flush` methods. Theoretically, you could even log to a file by creating a handler like `StreamHandler(open("log.txt", "a"))` - but in that case, it's better to use a `FileHandler`, so it can manage opening and closing the file.

When creating a handler, your needs are nearly always met by either `StreamHandler` or `FileHandler`. There are other predefined handlers, too, useful when logging to certain specialized destinations:

- `WatchedFileHandler` and `RotatingFileHandler`, for logging to rotated log files
- `SocketHandler` and `DatagramHandler` for logging over network sockets
- `HTTPHandler` for logging over an HTTP REST interface
- `QueueHandler` and `QueueListener` for queuing log records across thread and process boundaries

See the official docs[5] for more details.

2.6 Logging to Multiple Destinations

Suppose you want your long-running application to log all messages to a file, including debug-level records. At the same time, you want warnings, errors, and criticals logged to the console. How do you do this?

We've given you part of the answer already. A single logger object can have multiple handlers: all you have to do is call `addHandler` multiple times, passing a different handler object for each. For example, here is how you parrot all log messages to the console (via standard error) and also to a file:

```
import logging
logger = logging.getLogger()
# Remember, StreamHandler defaults to using sys.stderr
console_handler = logging.StreamHandler()
logger.addHandler(console_handler)
# Now the file handler:
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)
logger.warning(
    "This goes to both the console, AND log.txt.")
```

[5] https://docs.python.org/3/library/logging.handlers.html

This is combining what we learned above. We create two handlers - a `StreamHandler` named `console_handler`, and a `FileHandler` named `logfile_handler` - and add both to the same logger (via `addHandler`). That's all you need to log to multiple destinations in parallel. Sure enough, if you save the above in a script and run it, you'll find the messages are both written into "log.txt", as well as printed on the console (through standard error).

We aren't done, though. How do we make it so every record is written in the log file, but only those of `logging.WARNING` or higher get sent to the console screen? Do this by setting log level thresholds for both the logger object and the individual handlers. Both logger objects and handlers have a method called `setLevel`, taking a log level threshold as an argument:

```
my_logger.setLevel(logging.DEBUG)
my_handler.setLevel(logging.INFO)
```

If you set the level for a logger, but not its handlers, the handlers inherit from the logger:

```
my_logger.setLevel(logging.ERROR)
my_logger.addHandler(my_handler)
my_logger.error("This message is emitted by my_handler.")
my_logger.debug("But this message will not.")
```

And you can override that at the handler level. Here, I create two handlers. One handler inherits its threshold from the logger, while the other does its own thing:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)

verbose_handler = logging.FileHandler("verbose.txt")
logger.addHandler(verbose_handler)

terse_handler = logging.FileHandler("terse.txt")
terse_handler.setLevel(logging.WARNING)
logger.addHandler(terse_handler)

logger.debug("This message appears in verbose.txt ONLY.")
logger.warning("And this message appears in both files.")
```

There's a caveat, though: a handler can only make itself *more* selective than its logger, not less. If the logger chooses a threshold of `logger.DEBUG`, its handler can choose a threshold of `logger.INFO`, or `logger.ERROR`, and so on. But if the logger defines a strict threshold -

say, `logger.INFO` - an individual handler cannot choose a lower one, like `logger.DEBUG`. So something like this won't work:

```
# This doesn't quite work...
import logging
my_logger = logging.getLogger()
my_logger.setLevel(logging.INFO)
my_handler = logging.StreamHandler()
my_handler.setLevel(logging.DEBUG) # FAIL!
my_logger.addHandler(my_handler)
my_logger.debug("No one will ever see this message :(")
```

There's a subtle corollary of this. By default, a logger object's threshold is set to `logger.WARNING`. So if you don't set the logger object's log level at all, it implicitly censors all handlers:

```
import logging
my_logger = logging.getLogger()
my_handler = logging.StreamHandler()
my_handler.setLevel(logging.DEBUG) # FAIL!
my_logger.addHandler(my_handler)
# No one will see this message either.
my_logger.debug(
    "Because anything under WARNING gets filtered.")
```

The logger object's default log level is not always permissive enough for all handlers you might want to define. So you will generally want to start by setting the logger object to the lowest threshold needed by any log-record destination, and tighten that threshold for each handler as needed.

Bringing this all together, we can now accomplish what we originally wanted - to verbosely log everything into a log file, while duplicating only the more interesting messages onto the console:

```
import logging
logger = logging.getLogger()
logger.setLevel(logging.DEBUG)
# Warnings and higher only on the console.
console_handler = logging.StreamHandler()
console_handler.setLevel(logging.WARNING)
logger.addHandler(console_handler)
# But allow everything to into the log file.
logfile_handler = logging.FileHandler("log.txt")
logger.addHandler(logfile_handler)

logger.warning(
    "This goes to both the console, AND into log.txt.")
logger.debug("While this only goes to the file.")
```

Add as many handlers as you want. Each can have different log levels. You can log to many different destinations, using the different built-in handler types mentioned above. If those don't do what you need, implement your own subclass of `logging.Handler` and use that.

2.7 Record Layout with Formatters

We haven't covered one important detail. So far, we've only shown you how to create logger objects that will write just the log message and nothing else. At the very least, you probably want to annotate that with the log level. You may also want to insert the time, or some other information. How do you do that?

The answer is to use a *formatter*. A formatter converts a log record into something that is recorded in the handler's destination. That's an abstract way of saying it; more simply, a typical formatter just converts the record into a usefully-formatted string. That string contains the actual log message, as well as the other fields you care about.

The procedure is to create a `Formatter` object, then associate with a handler (using the latter's `setHandler` method). Creating a formatter is easy - it normally takes just one argument, the format string:

```
import logging
my_handler = logging.StreamHandler()
fmt = logging.Formatter("My message is: %(message)s")
my_handler.setFormatter(fmt)
my_logger = logging.getLogger()
my_logger.addHandler(my_handler)
my_logger.warning("WAKE UP!!")
```

If you run this in a script, the output will be:

```
My message is this: WAKE UP!!
```

Notice the attribute for the message, `%(message)s`, included in the string. This is just a normal formatting string, in the older, percent-formatting style. It's exactly equivalent to using the `format` argument when you call `basicConfig`. For this reason, you can use the same attributes, arranged however you like - here's the attribute table again, distilled from the full official list:[6]

Attribute	Format	Description
asctime	`%(asctime)s`	Human-readable date/time
funcName	`%(funcName)s`	Name of function containing the logging call
lineno	`%(lineno)d`	The line number of the logging call
message	`%(message)s`	The log message
pathname	`%(pathname)s`	Full pathname of the source file of the logging call
levelname	`%(levelname)s`	Text logging level for the message ('DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL')
name	`%(name)s`	The logger's name

[6] https://docs.python.org/3/library/logging.html#logrecord-attributes

Chapter 3

Automated Testing and TDD

Writing automated tests is one of those things that separates average developers from the best in the world. Master this skill, and you will be able to write *far* more complex and powerful software than you ever could before. It's a superpower, and changes the arc of your career.

There are roughly two kinds of readers for this chapter. Some of you have, so far, little or no experience writing automated tests, in any language. This chapter is primarily written for you. It introduces many fundamental ideas of test automation, explains the problems it is supposed to solve, and teaches how to apply Python's tools for doing so.

Or you might be someone with extensive experience using standard test frameworks in other languages: JUnit in Java, PHPUnit in PHP, and so on. Generally speaking, if you have mastered an xUnit framework in another language, and are fluent in Python, you may be able to start skimming the Python's `unittest` module docs[1] and be productive in minutes. Python's test library, `unittest`, was originally based on JUnit 3, and maps very closely to how most xUnit libraries work.[2]

If you are more experienced, I believe it's worth your time to at least skim the chapter, and perhaps study it thoroughly. In writing, I invested a great deal of effort weaving in useful, real-world wisdom - both for software testing in general, and for Python specifically. This includes topics like how to organize Python test code; writing test code which is maintainable; useful, Python-only features like subtests; and even cognitive aspects of programming... getting into an enjoyable, highly productive "flow" state via test-driven development.

[1] https://docs.python.org/3/library/unittest.html

[2] You may be in a third category, having a lot of experience with a non-xUnit testing framework. If so, you should probably pretend you're in the first group. You'll be able to move quickly.

With that in mind, let's start with the core ideas for writing automated tests. We'll then focus on writing them for Python programs.

3.1 What is Test-Driven Development?

An *automated test* is a program that tests another program. Generally, it tests a specific portion of that program: a function, a method, a class, or some group of these things. We call that portion the "system under test". If the system under test is working correctly, the test passes; if not, our test catches that error, and immediately tells us what is wrong. Real applications accumulate many of these tests as development proceeds.

People have different names for different kinds of automated tests: unit tests, integration tests, end-to-end tests, etc. These distinctions can be useful, but we won't need to worry about them right now. They all share the same foundation.

In this chapter, we do *test-driven development*, or TDD. Test-driven development means you start working on each new feature or bugfix by writing the automated test for it **first**. You run that test, verify it fails, and only then do you write the actual code for the feature. You know you are done when the test passes.

This is a different process from implementing the feature first, **then** writing a test for it after. Writing the test first forces you to think through the interfaces of your code, answering the question "how will I know my code is working?" That immediate benefit is useful, but not the whole story.

The greatest mid-term benefits are mostly cognitive. As you become competent and comfortable with test-driven development, you learn to easily get into a state of flow - where you find yourself repeatedly implementing feature after feature, keeping your focus with ease for long periods of time. You can honestly surprise and delight yourself with how much you've accomplished in a few hours of coding.

But the greatest benefits emerge over time. We've all done substantial refactorings of a large code base, changing fundamental aspects of its architecture.[3] Such refactorings - which threaten to break the application in confusing, hidden ways - become straightforward and safe using TDD. You take the existing body of tests, updating where needed and introducing new tests as appropriate. Then all you have to do is make them pass. It may still be a ton of work.

[3] If you haven't done one of these yet, you will.

But you can be fairly confident in the correctness and robustness of the result, instead of hoping and praying.

Among developers who know how to write tests, some love to do test-driven development in their day to day work. Some like to do it part of the time; some hate it, and do it rarely, or never. However, the absolute best way to quickly master unit testing is to strictly do test-driven development for a while. So I'll teach you how to do that. You don't have to do it forever if you don't want to.

Python's standard library ships with two modules for creating unit tests: `doctest` and `unittest`. Most engineering teams prefer `unittest`, as it is more full-featured than `doctest`. This isn't just a convenience. There is a real ceiling of complexity that `doctest` can handle, and real applications will quickly bump up against that limit. With `unittest`, the sky is more or less the limit.

In addition - as noted above - `unittest` maps almost exactly to the xUnit libraries used in many other languages. If you are already familiar with Python, and have used JUnit, PHPUnit, or any other xUnit library in any language, you will feel right at home with unittest. That said, `unittest` has some unique tools and idioms - partly because of differences in the Python language, and partly from unique extensions and improvements. We will learn the best of what `unittest` has to offer as we go along.

3.2 Unit Tests And Simple Assertions

Imagine a class representing an angle:

```
>>> small_angle = Angle(60)
>>> small_angle.degrees
60
>>> small_angle.is_acute()
True
>>> big_angle = Angle(320)
>>> big_angle.is_acute()
False
>>> funny_angle = Angle(1081)
>>> funny_angle.degrees
1
>>> total_angle = small_angle.add(big_angle)
>>> total_angle.degrees
20
```

As you can see, `Angle` keeps track of the angle size, wrapping around so it's in a range of 0 up to 360 degrees. There is also an `is_acute` method, to tell you if its size is under 90 degrees, and an `add` method for arithmetic.[4]

Suppose this `Angle` class is defined in a file named `angle.py`. Here's how we create a simple test for it - in a separate file, named `test_angle.py`:

[4]The object-oriented chapter talks about "magic methods" like `__add__`, which provide a more natural syntax for math-like operations on custom types. This chapter just uses regular methods, in case you haven't read that chapter yet.

```
import unittest
from angle import Angle

class TestAngle(unittest.TestCase):
    def test_degrees(self):
        small_angle = Angle(60)
        self.assertEqual(60, small_angle.degrees)
        self.assertTrue(small_angle.is_acute())
        big_angle = Angle(320)
        self.assertFalse(big_angle.is_acute())
        funny_angle = Angle(1081)
        self.assertEqual(1, funny_angle.degrees)

    def test_arithmetic(self):
        small_angle = Angle(60)
        big_angle = Angle(320)
        total_angle = small_angle.add(big_angle)
        self.assertEqual(20, total_angle.degrees,
                         'Adding angles with wrap-around')
```

As you look over this code, notice a few things:

- There's a class called `TestAngle`. You just define it, not create any instance of it. This subclasses `TestCase`.
- You define two methods, `test_degrees` and `test_arithmetic`.
- Both `test_degrees` and `test_arithmetic` have assertions, using some methods of `TestCase`: `assertEqual`, `assertTrue`, and `assertFalse`.
- The last assertion includes a custom message, as its third argument.

To see how this works, let's define a stub for the `Angle` class in `angles.py`:

```
# angle.py, version 1
class Angle:
    def __init__(self, degrees):
        self.degrees = 0
    def is_acute(self):
        return False
    def add(self, other_angle):
        return Angle(0)
```

This `Angle` class defines all the attributes and methods it is expected to have, but otherwise can't do anything useful. We need a stub like this to verify the test can run correctly, and alert us to the fact that the code isn't working yet.

The `unittest` module is not just used to define tests, but also to run them. You do so on the command line like this:

```
python3 -m unittest test_angles.py
```

When you run the test,[5] and you'll see the following output:

[5] Python 2 requires you to drop the test file's `.py` extension - in other words, passing the test module name. So you invoke it like `python2.7 -m unittest test_angles`. Python 3 lets you do either; we'll always use the test filename in this chapter, but you can use whichever you prefer.

```
$ python3 -m unittest test_angle.py
FF
======================================================
FAIL: test_arithmetic (test_angle.TestAngle)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_angle.py", line 18, in test_arithmetic
    self.assertEqual(20, total_angle.degrees, 'Adding angles with
       wrap-around')
AssertionError: 20 != 0 : Adding angles with wrap-around

======================================================
FAIL: test_degrees (test_angle.TestAngle)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_angle.py", line 7, in test_degrees
    self.assertEqual(60, small_angle.degrees)
AssertionError: 60 != 0

------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=2)
```

Notice:

- Both test methods are shown. They both have a failed assertion highlighted.
- `test_degrees` makes several assertions, but only the first one has been run - once it fails, the others are not executed.
- For each failing assertion, you are given the line number; the expected and actual values; and its test method.
- The custom message in `test_arithmetic` shows up in the output.

This demonstrates one useful way to organize your test code. In a single test module (`test_angle.py`), you define one or more subclasses of `unittest.TestCase`. Here, I just define `TestAngle`, containing tests for the `Angle` class. Within this, I create several test methods, for testing different aspects of the class. And in each of these test methods, I can have as many assertions as makes sense.

Some of the naming conventions matter. It's traditional to start a test class name with the string `Test`, but that is not required; `unittest` will find all subclasses of `TestCase` automatically. But every method must start with the string "test". If it starts with anything else (even "Test"!), `unittest` will not run its assertions.

Running the test and watching it fail is an important first step. It verifies that the test does, in fact, actually test your code. As you write more and more tests, you'll occasionally create the test; run it, expecting it to fail; and find it unexpectedly passes. That's a bug in your test code! Fortunately you ran the test first, so you caught it right away.

In the test code, we defined `test_degrees` before `test_arithmetic`, but they were actually run in the opposite order. It's important to craft your test methods to be self-contained, and not depend on one being run before the other; the order of execution is essentially random.[6]

At this point, we have a correctly failing test. If I'm using version control and working in a branch, this is a good commit point - check in the test code, because it specifies the correct behavior (even if it's presently failing). The next step is to actually make that test pass. Here's one way to do it:

```
# angle.py, version 2
class Angle:
    def __init__(self, degrees):
        self.degrees = degrees % 360
    def is_acute(self):
        return self.degrees < 90
    def add(self, other_angle):
        return Angle(self.degrees + other_angle.degrees)
```

Now when I run my test again, the test passes:

```
python3 -m unittest test_angle1.py
..
----------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

This becomes your second commit in version control.

[6] If you find yourself wanting to run tests in a certain order, this might be better handled with `setUp` and `tearDown`, explained in the next section.

`assertEqual`, `assertTrue` and `assertFalse` will be the most common assertion methods you'll use, along with `assertNotEqual` (which does the opposite of `assertEqual`). Many others are provided, such as `assertIs`, `assertIsNone`, `assertIn`, and `assertIsInstance` - along with "not" variants (e.g. `assertIsNot`). Each takes a optional final message-string argument, like "Adding angles with wrap-around" in `test_arithmetic` above. If the test fails, this is printed in the output, which can give very helpful advice to whomever is troubleshooting a broken test.[7]

If you try checking that two dictionaries are equal, and they are not, the output is tailored to the data type: highlighting which key is missing, or which value is incorrect, for example. This also happens with lists, tuples, and sets, making troubleshooting much easier. What's actually happening is that `unittest` provides certain type-specialized assertions, like `assertDictEqual`, `assertListEqual`, and more. You almost never need to invoke them directly: if you invoke `assertEqual` with two dictionaries, it automatically dispatches to `assertDictEqual`, and similar for the other types. So you get this usefully detailed error reporting for free.

Notice the `assertEqual` lines take two arguments, and I always wrote the expected, correct value first:

```
small_angle = Angle(60)
self.assertEqual(60, small_angle.degrees)
```

It does not matter whether the expected value is first, or second. But it's smart to pick an order and stick with it - at least throughout a single codebase, and maybe for all code you write. Sticking with a consistent order greatly improves the readability of your test output, because you never have to decipher which is which. Believe me, this will save you a lot of time in the long run. If you're on a team, negotiate with them to agree on a consistent order.

3.3 Fixtures And Common Test Setup

As an application grows and you write more tests, you will find yourself writing groups of test methods that start or end with the same lines of code. This repeated code - which does some kind of pretest set-up, and/or after-test cleanup - can be consolidated in the special methods `setUp` and `tearDown`. When defined in your `TestCase` subclass, `setUp` is executed just before each test method starts; `tearDown` is run just after. This is repeated for every single test method.

[7] Which could be you, months or years down the road. Be considerate of your future self!

Here's a realistic example of when you might use it. Imagine working on a tool that saves its state between runs in a special file, in JSON format. We'll call this the "state file". On start, it reads the state from the file; on exit, it rewrites it, if there are any changes. A stub of this class might look like

```
# statefile.py
class State:
    def __init__(self, state_file_path):
        # Load the stored state data, and save
        # it in self.data.
        self.data = { }
    def close(self):
        # Handle any changes on application exit.
```

In fleshing out this stub, we want our tests to verify the following:

- If I add a new key-value pair to the state, it is recorded correctly in the state file.
- If I alter the value of an existing key, that updated value is written to the state file.
- If the state is not changed, the state file's content stays the same.

For each test, we want the state file to be in a known starting state. Afterwards, we want to remove that file, so our tests don't leave garbage files on the filesystem. Here's how the `setUp` and `tearDown` methods accomplish this:

```
import os
import unittest
import shutil
import tempfile
from statefile import State

INITIAL_STATE = '{"foo": 42, "bar": 17}'

class TestState(unittest.TestCase):
    def setUp(self):
        self.testdir = tempfile.mkdtemp()
        self.state_file_path = os.path.join(
            self.testdir, 'statefile.json')
        with open(self.state_file_path, 'w') as outfile:
            outfile.write(INITIAL_STATE)
        self.state = State(self.state_file_path)

    def tearDown(self):
        shutil.rmtree(self.testdir)

    def test_change_value(self):
        self.state.data["foo"] = 21
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertEqual(21,
            reloaded_statefile.data["foo"])

    def test_remove_value(self):
        del self.state.data["bar"]
        self.state.close()
        reloaded_statefile = State(self.state_file_path)
        self.assertNotIn("bar", reloaded_statefile.data)

    def test_no_change(self):
        self.state.close()
        with open(self.state_file_path) as handle:
            checked_content = handle.read()
        self.assertEqual(checked_content, INITIAL_STATE)
```

In `setUp`, we create a fresh temporary directory, and write the contents of `INITIAL_DATA` inside. Since we know each test will be working with a `State` object based on that initial data, we go ahead and create that object, and save it in `self.state`. Each test can then work with that object, confident it is in the same consistent starting state, regardless of what any other test method does. In effect, `setUp` creates a private sandbox for each test method.

The tests in `TestState` would all work reliably with just `setUp`. But we also want to clean up the temporary files we created; otherwise, they will accumulate over time with repeated test runs. The `tearDown` method will run after each `test_*` method completes, even if some of its assertions fail. This ensures the temp files and directories are all removed completely.

The generic term for this kind of pre-test preparation is called a *test fixture*. A test fixture is whatever needs to be done before a test can properly run. In this case, we set up the text fixture by creating the state file, and the `State` object. A text fixture can be a mock database; a set of files in a known state; some kind of network connection; or even starting a server process. You can do all these with `setUp`.

`tearDown` is for shutting down and cleaning up the text fixture: deleting files, stopping the server process, etc. For some kinds of tests, a tear-down might not be at all optional. If `setUp` starts some kind of server process, for example, and `tearDown` fails to terminate it, then `setUp` may not be able to run for the next test.

The camel-casing matters: people sometimes misspell them as `setup` or `teardown`, then wonder why they don't seem to be invoked. Also, any uncaught exception in either `setUp` or `tearDown` will cause `unittest` to mark the test method as failing (which means it will clearly show up in the test output), then immediately skip to the next test. For errors in `setUp`, this means none of that test's assertions will run (though it's still marked as failing). For `tearDown`, the test is marked as failing, even if all the individual assertions passed.

3.4 Asserting Exceptions

Sometimes your code is supposed to raise an exception, under certain exceptional conditions. If that condition occurs, and your code does *not* raise the correct exception, that's a bug. How do you write test code for this situation?

You can verify that behavior with a special method of `TestCase`, called `assertRaises`. It's used in a `with` statement in your test; the block under the `with` statement is asserted to raise the exception. For example, suppose you are writing a library that translates Roman numerals into integers. You might define a function called `roman2int`:

```
>>> roman2int("XVI")
16
>>> roman2int("II")
2
```

In thinking about the best way to design this function, you decide that passing nonsensical input to `roman2int` should raise a `ValueError`. Here's how you write a test to assert that behavior:

```
import unittest
from roman import roman2int

class TestRoman(unittest.TestCase):
    def test_roman2int_error(self):
        with self.assertRaises(ValueError):
            roman2int("This is not a valid roman numeral.")
```

If you run this test, and `roman2int` does NOT raise the error, this is the result:

```
$ python3 -m unittest test_roman2int.py
F
======================================================
FAIL: test_roman2int_error (test_roman2int.TestRoman)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_roman2int.py", line 7, in test_roman2int_error
    roman2int("This is not a valid roman numeral.")
AssertionError: ValueError not raised

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

When you fix the bug, and `roman2int` raises `ValueError` like it should, the test passes.

3.5 Using Subtests

Python 3 has a new feature called subtests, allowing you to conveniently iterate through a collection of test inputs. Imagine a function called `numwords`, which counts the number of unique words in a string (ignoring punctuation, spelling and spaces):

```
>>> numwords("Good, good morning. Beautiful morning!")
3
```

Suppose you want to test how `numwords` handles excess whitespace. You can easily imagine a dozen different reasonable inputs that will result in the same return value, and want to verify it can handle them all. You might create something like this:

```
class TestWords(unittest.TestCase):
    def test_whitespace(self):
        self.assertEqual(2, numwords("foo bar"))
        self.assertEqual(2, numwords("    foo bar"))
        self.assertEqual(2, numwords("foo\tbar"))
        self.assertEqual(2, numwords("foo   bar"))
        self.assertEqual(2, numwords("foo bar    \t   \t"))
        # And so on, and so on...
```

Seems a bit repetitive, doesn't it? The only thing varying is the argument to `numwords`. We might benefit from using a for loop:

```
    def test_whitespace_forloop(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            self.assertEqual(2, numwords(text))
```

At first glance, this is certainly more maintainable. If we add new variants, it's just another line in the `texts` list. And if I rename `numwords`, I only need to change it in one place in the test.

However, using a for loop like this creates more problems than it solves. Suppose you run this test, and get the following failure:

```
$ python3 -m unittest test_words_forloop.py
F
======================================================
FAIL: test_whitespace_forloop (test_words_forloop.TestWords)
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_forloop.py", line 17, in
     test_whitespace_forloop
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=1)
```

Look closely, and you'll realize that `numwords` returned 3 when it was supposed to return 2. Pop quiz: out of all the inputs in the list, which caused the bad return value?

The way we've written the test, there is no way to know. All you can infer is that at least one of the test inputs produced an incorrect value. You don't know which one. That's the first problem. The second is that the test stops when the first failure happens. If several test inputs are causing errors, it would be helpful to know that right away. (Of course, the original version has this shortcoming too.) Knowing all the failing inputs, and the incorrect results they create, would be *very* helpful for quickly understanding what's going on.

Python 3.4 introduced a new feature, called *subtests*, that gives you the best of all worlds. Our for-loop solution is actually quite close. All we have to do is add one line - can you spot it below?

```
    def test_whitespace_subtest(self):
        texts = [
            "foo bar",
            "    foo bar",
            "foo\tbar",
            "foo   bar",
            "foo bar    \t   \t",
            ]
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

Just inside the for loop, we write `with self.subTest(text=text)`. This creates a context in which assertions can be made, and even fail. Regardless of whether they pass or not, the test continues with the next iteration of the for loop. At the end, *all* failures are collected and reported in the test result output, like this:

```
$ python3 -m unittest test_words_subtest.py

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='
   foo\tbar')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in
     test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 3

======================================================
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (text='
   foo bar    \t   \t')
------------------------------------------------------
Traceback (most recent call last):
  File "/src/test_words_subtest.py", line 16, in
     test_whitespace_subtest
    self.assertEqual(2, numwords(text))
AssertionError: 2 != 4

------------------------------------------------------
Ran 1 test in 0.000s

FAILED (failures=2)
```

Behold the opulence of information in this output:

- Each individual failing input has its own detailed summary.
- We are told what the full value of `text` was.
- We are told what the actual returned value was, clearly compared to the expected value.
- No values are skipped. We can be confident that these two are the *only* failures.

This is MUCH better. The two offending inputs are `"foo\tbar"` and `"foo bar \t \t"`. These are the only values containing tab characters, so you can quickly realize the nature of the bug: tab characters are being counted as separate words.

Let's look at the three key lines of code again:

```
        for text in texts:
            with self.subTest(text=text):
                self.assertEqual(2, numwords(text))
```

The key-value arguments to `self.subTest` are used in reporting the output. They can be anything that helps you understand exactly what is wrong when a test fails. Often you will want to pass everything that varies from the test cases; here, that's only the string passed to `numwords`.

Be clear that in these three lines, the symbol `text` is used for two different things. The `text` variable in the for loop is the same variable that is passed to `numwords` on the last line. In the call to `subTest`, the left-hand side of `text=text` is actually a parameter that is used in the reporting output if the test fails. For example, suppose we wrote it as `input_text` instead:

```
        for text in texts:
            with self.subTest(input_text=text):
                self.assertEqual(2, numwords(text))
```

Then the failure output might look like:

```
FAIL: test_whitespace_subtest (test_words_subtest.TestWords) (
   input_text='foo\tbar')
```

In other words, the l-value `text` in the `assertEqual` line has nothing to do with the argument to `subTest`. Just remember that the arguments to `subTest` are only used in the error output when something goes wrong, and are otherwise ignored completely.

3.6 Final Thoughts

Let's recap the big ideas. Test-driven development means we create the test first, and whatever stubs we need to make the test run. We then run it, and watch it fail. **This is an important step.** You must run the test and see it fail.

This is important for two reasons. You don't really know if the test is correct until you verify that it **can** fail. As you write automated tests more and more over time, you will probably be surprised at how often you write a test, confident in its correctness, only to discover it passes when it should fail. As far as I can tell, every good, experienced software engineer I know still

commonly experiences this... even after doing TDD for many years! This is why we build the habit of always verifying the test fails first.

The second reason is more subtle. As you gain experience with TDD and become comfortable with it, you will find the cycle of writing tests and making them pass lets you get into a state of flow. This means you are enjoyably productive and focused, in a way that is easy to maintain over time.

Is it important that you strictly follow test-driven development? People have different opinions on this, some of them *very* strong. Personally, I went through a period of almost a year where I followed TDD to the letter, very strictly. As a result, I got *very* good at writing tests, and writing high-quality tests very quickly.

Now that I've developed that level of skill, I prefer instead to follow the 80-20 rule, and sometimes the 70-30 or even 50-50 rule. I have noticed that TDD is most powerful when I have great clarity on the software's design, architecture and APIs; it helps me get into an cognitive state that seems accelerated, so that I can more easily maintain my mental focus, and produce quality code faster.

But I find it very hard to write good tests when I don't yet have that clarity... when I am still thinking through how I will structure the program and organize the code. In fact, I find TDD slows me down in that phase, as any test I write will probably have to be completely rewritten several times, if not deleted, before things stabilize. In these situations, I prefer to get a first version of the code working through manual testing, then write the tests afterwards.

To close with the obvious: Experiment to find what approach works best for you, and not just follow what someone else writes that you "should" do. I encourage you to try TDD for a period of time, because of what it will teach you. But be flexible, and at some point step back and evaluate how you want to integrate it into your daily routine.

Chapter 4

String Formatting

The situation with string formatting is complicated.

Once upon a time, Python introduced *percent formatting*. It uses `"%"` as a binary operator to render strings:

```
>>> drink = "coffee"
>>> price = 2.5
>>> message = "This %s costs $%.2f." % (drink, price)
>>> print(message)
This coffee costs $2.50.
```

Later in Python 2's history, a different style was introduced, called simply *string formatting* (yes, that's the official name). Its very different syntax makes any Python string a potential template, inserting values through the `str.format()` method.

```
>>> template = "This {} costs ${:.2f}."
>>> print(template.format(drink, price))
This coffee costs $2.50.
```

Python 3.6 introduces a third option, called *f-strings*. This lets you write literal strings, prefixed with an "f" character, interpolating values from the immediate context:

```
>>> message = f"This {drink} costs ${price:.02f}."
>>> print(message)
This coffee costs $2.50.
```

So… which do you use? Here's my guidance in a nutshell:

- Go ahead and master `str.format()` now. Everything you learn transfers entirely to f-strings, and you'll sometimes want to use `str.format()` even in cutting-edge versions of Python.
- Prefer f-strings when working in a codebase that supports it - meaning, all developers and end-users of the code base are certain to have Python 3.6 or later.
- Until then, prefer `str.format()`.
- Exception: for the `logging` module, use percent-formatting, even if you're otherwise using f-strings.
- Aside from `logging`, don't use percent-formatting unless legacy reasons force you to.

"Which should I use?" is a separate question from "which should a Python book use for its code examples?" As you've noticed, I am using `str.format()` throughout this book. That's because all modern Python versions support it, so I know everyone reading this book can use it.

Someday, when Python versions before 3.6 are a distant memory, there will be no reason not to use f-strings. But when that happens, `str.format()` will still be important. There are string formatting situations where f-strings are awkward at best, and `str.format()` is well suited. In the meantime, there is a lot of Python code out there using `str.format()`, which you'll need to be able to read and understand. Hence, I'm choosing to focus on `str.format()` in this chapter. Conveniently, this also teaches you much about f-strings; they are more similar than different, because the formatting codes are nearly identical. `str.format()` is also the only practical choice for most people reading this, and will be for years still.

You might wonder if the old percent-formatting has any place in modern Python. In fact, it does, due to the `logging` module. As you'll read in its chapter, this important module is built on percent-formatting in a deep way. It's possible to use `str.format()` in new logging code, but requires special steps; and legacy logging code cannot be safely converted in an automated way. I recommend you just cooperate with the situation, and use percent-formatting for your log messages.

For those interested, this chapter ends with sections briefly explaining f-strings and percent-formatting. For now, we'll focus on `str.format()`. While reading, I highly recommend you have a Python interpreter prompt open, typing in the examples as you go along. The goal is to make its expressive power automatic and easy for you to use, so that it's *mentally* available to you... giving you the easy ability to use it in the future, without digging into the reference docs. Most people never master it to this threshold, effectively denying them most of the benefits of this rich tool. You won't have that problem.

4.1 Replacing Fields

`str.format()` lets you start simple, leveraging more complex extensions as needed. You start by creating a format string. This is just a regular string, and acts as a kind of template. It contains, among other text, one or more *replacement fields*. These are simply pairs of opening and closing curly braces: `"Good {}, my friend"`. You then invoke the `format` method on that string, passing in one argument for each replacement field:

```
>>> "Good {}, my friend".format("morning")
'Good morning, my friend'
>>> "Good {}, my friend".format("afternoon")
'Good afternoon, my friend'
>>> offer = "Give me {} dollars and I'll give you a {}."
>>> offer.format(2, "cheeseburger")
"Give me 2 dollars and I'll give you a cheeseburger."
>>> offer.format(7, "nice shoulder rub")
"Give me 7 dollars and I'll give you a nice shoulder rub."
```

(If possible, type these examples in an interpreter as you go along, so you learn them deeply.) `.format()` is a method returning a new string; the format string itself is not modified.

Notice how fields line up by position, and no type information is needed. The integer 2 and the string "cheeseburger" are both inserted without complaint. We'll see how to specify more precise types for the fields later.

Within the curly braces of the replacement field, you can specify numbers starting at 0. These reference the positions of the arguments passed to `format`, and allow you to repeat fields:

```
>>> "{0} is {1}; {1}, {0}".format("truth", "beauty")
'truth is beauty; beauty, truth'
```

You can also reference fields by a name, and pass the fields as key-value pairs to `format`:

```
>>> "Good {when}, {user}!".format(
        when="morning", user="John")
'Good morning, John!'
```

The arguments to `format()` don't actually have to be strings: they can be objects or lists. Reference within them as you normally would, within the curly braces:

```
>>> class Point:
...     def __init__(self, x, y):
...         self.x = x
...         self.y = y
...
>>> point = Point(3, 7)
>>> 'The coordinates are {point.x}, {point.y}'.format(
        point=point)
'The coordinates are 3, 7'
>>> 'The coordinates are {0.x}, {0.y}'.format(point)
'The coordinates are 3, 7'
```

Notice the difference in how you use the point with a named field (the first format call), versus a numbered field (the second). Here's how it looks with a list:

```
>>> params = ["morning", "user"]
>>> "Good {params[0]}, {params[1]}!".format(params=params)
'Good morning, user!'
>>> "Good {0[0]}, {0[1]}!".format(params)
'Good morning, user!'
```

You can do the same thing with dictionaries too, though there is one subtle quirk - can you spot it?

```
>>> params = {"when": "morning", "user":"John"}
>>> "Good {0[when]}, {0[user]}!".format(params)
'Good morning, John!'
```

See it? The key `when` in `{0[when]}` does not have quotation marks around it! In fact, if you do put them in, you get an error. This quirk was intentionally put in, to make it easier to reference keys within a string that is bounded by quotes already.

4.2 Number Formats (and "Format Specs")

Now that you know how to substitute values into different replacement fields (i.e., pairs of curly braces), you may next want to format a field as a number. Do this by inserting a colon between the curly braces, followed by one or more descriptive characters. For example, use ":d" to format as an integer, and ":f" to format as a floating-point number. Here's how it works:

```
>>> 'The magic number is {:d}'.format(42)
'The magic number is 42'
>>> 'The magic number is actually {:f}'.format(42)
'The magic number is actually 42.000000'
>>> 'But this will cause an error: {:d}'.format("foo")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Unknown format code 'd' for object of type 'str'
```

The number 42 is rendered as either `42` or `42.000000`, depending on whether the replacement field is `{:d}` or `{:f}`. And when we try to stuff something that isn't a number in `{:d}`, it triggers a fatal error.

You can combine this with field numbering and naming. Just put that label before the colon:

```
>>> "The time is {hour:d} o' clock.".format(hour=11)
"The time is 11 o' clock."
>>> "The answer is {0:f}, not {1:f}. But you can round it to {0:d}."
    .format(12, 14)
'The answer is 12.000000, not 14.000000. But you can round it to 12.
    '
```

The rule is easy: if you label a field - whether it's a string name, or a number - always put that label first within the curly brackets.

The part after after the colon is called a *format spec*. There's actually many options you can stuff in there, but let's focus on those related to formatting numbers first. As you saw, the code for an integer is `d`, converting the argument to an integer, if possible. If not, it throws a `ValueError`. (By the way, "d" stands for "decimal number", as in a base-ten number.)

Then there's floating point numbers. With the `f` code, we get six decimal places of precision by default - which are filled with zeros if necessary:

```
>>> from math import pi
>>> 'The ratio is about {:f}'.format(pi)
'The ratio is about 3.141593'
>>> '{:f} is NOT a good approximation.'.format(3)
'3.000000 is NOT a good approximation.'
```

"f" actually stands for "fixed-point number", not "floating point" - we'll see some variations later. We can change the number of fixed points: writing a period followed by a number means to use that many decimal places. We put it between the colon and the letter "f", like so:

```
>>> 'The ratio is about {:.3f}'.format(pi)
'The ratio is about 3.142'
>>> '{:.1f} is NOT a good approximation.'.format(3)
'3.0 is NOT a good approximation.'
```

It's easier for humans to read numbers with many digits if they have commas. You can tell the formatter to do this by putting a comma after the colon:

```
>>> "Billions and {:,d}'s".format(10**9)
"Billions and 1,000,000,000's"
>>> "It works with floating point too: {:,f}".format(10**9)
'It works with floating point too: 1,000,000,000.000000'
```

For large numbers, sometimes we want scientific notation, also called exponent notation. We can use the code "E" for that instead:

```
>>> "Billions and {:E}'s".format(10**9)
"Billions and 1.000000E+09's"
>>> "Precision works the same: {:.2E}".format(10**9)
'Precision works the same: 1.00E+09'
```

So far, we have seen codes for three presentation types: d (decimal) for integers; and f (fixed-point) and E (exponential) for floating-point numbers. We actually have many other choices for both: read the format-specification mini-language section[1] in the Python docs.

4.3 Width, Alignment, and Fill

In the examples above, the substituted values will take only as much space as they need, but no more. So `"a{}b".format(n)` will render as `a7b` if n is 7 - but not `a07b`, for example. But if n is 77, it will expand to take up four characters instead of three: `a77b`.

We can change this default behavior, placing the value in a field of a certain number of characters. If it's small enough to fit in there (i.e. not too many digits or chars), then it will be right-justified. We specify the width by putting the number of characters between the colon and the type code:

```
>>> "foo{:7d}bar".format(753)
'foo    753bar'
```

[1] https://docs.python.org/3/library/string.html#format-specification-mini-language

Let's count the character columns here:

```
foo    753bar
0123456789012
```

Positions 3 through 9 are taken up by the replacement field value. That value only has three characters (753), so the others are *filled* by the space character. We say that the space is the *fill* character here.

By default, the value is right-justified in the field for numbers. But for strings, it's left justified:

```
>>> "foo{:7s}bar".format("blah")
'fooblah   bar'
```

Generally speaking, it will default to right-justifying for any kind of number, and left-justify for everything else. We can override the default, or even just be explicit about what we want, by inserting an *alignment* right before the field width. For right-justifying, this is the greater-than sign:

```
>>> "foo{:>7d}bar".format(753)
'foo    753bar'
>>> "foo{:>7s}bar".format("blah")
'foo   blahbar'
```

To left-justify, use a less-than sign:

```
>>> "foo{:<7d}bar".format(753)
'foo753    bar'
>>> "foo{:<7s}bar".format("blah")
'fooblah   bar'
```

Or we can center it, with a caret:

```
>>> "foo{:^7d}bar".format(753)
'foo  753  bar'
>>> "foo{:^7s}bar".format("blah")
'foo blah  bar'
```

So far, the extra characters in the field have been spaces. That extra character is called the *fill character*, or the *fill*. We can specify a different fill character by placing it just before the alignment character (<, > or ^):

```
>>> "foo{:_>7d}bar".format(753)
'foo____753bar'
>>> "foo{:+<7d}bar".format(753)
'foo753++++bar'
>>> "foo{:X^7d}bar".format(753)
'fooXX753XXbar'
```

This is a good time to remember you can combine field names or indices with these format annotations - just put the name or index to the left side of the colon:

```
>>> "alpha{x:_<6d}beta{y:+^7d}gamma".format(x=42, y=17)
'alpha42____beta++17+++gamma'
>>> "{0:_>6d}{1:-^7d}{0:_<6d}".format(11, 333)
'____11--333--11____'
```

Historically, over the long and ongoing lifetime of `printf`, it's been common to use zero as a fill character for right-justified integer fields. So if the number 42 is put in a five-character-wide field, it shows up at "00042". Since people often want to do this, a shorthand evolved. You can omit the alignment character if the fill is "0" (zero), and the type is decimal:

```
>>> "foo{:0>7d}bar".format(753)
'foo0000753bar'
>>> "foo{:07d}bar".format(753)
'foo0000753bar'
```

That doesn't generally work for other fill values, though:

```
>>> "foo{:_7d}bar".format(753)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Invalid format specifier
```

What happens if the value is too big to fit in a tiny field? The width is actually a *minimum* width. So it will expand as needed to fill it out:

```
>>> "red{:4d}green".format(123456789)
'red123456789green'
```

There isn't any way to specify a maximum width. If you need that, you can convert it to a string, implement your own trimming logic, then inject that trimmed string:

```
>>> value = 123456789 # Or some other number.
>>> trimmed_value = str(value)[:4] # Or last 4, etc.
>>> "red{:>4s}green".format(trimmed_value)
'red1234green'
```

4.4 F-Strings

Python 3.6 introduced an alternative to `str.format()`, called **f-strings**. The formal name is "formatted string literal". Instead of a `.format()` method, you prefix the string with the letters "f" or "F", putting variable names directly inside the replacement field:

```
>>> time_of_day = "afternoon"
>>> f"Good {time_of_day}, my friend"
'Good afternoon, my friend'
>>> F"Good {time_of_day}, my friend"
'Good afternoon, my friend'
```

It's exactly equivalent to this:

```
>>> # This...
... "Good {time_of_day}, my friend".format(
...     time_of_day=time_of_day)
'Good afternoon, my friend'
>>> # Or this:
... "Good {}, my friend".format(time_of_day)
'Good afternoon, my friend'
```

If your values are already stored in locally-readable variables, using f-strings is more succinct. But you can do more than that. In fact, the replacement field (i.e. the curly braces) can contain not only a variable name, but a full Python expression!

```
>>> f"Good {time_of_day.upper()}, my friend"
'Good AFTERNOON, my friend'

>>> def reverse(string):
...     return string[::-1]
>>> f"Good {reverse(time_of_day)}, my friend"
'Good noonretfa, my friend'

>>> groceries = ["milk", "bread", "broccoli"]
>>> f"I need to get some {groceries[2]}."
'I need to get some broccoli.'
```

You can do this with `str.format()`, too, but f-strings express it a bit more naturally. Aside from that, you can use the normal number formatting codes. After the expression name, simply write a colon, and the same code you would use for `str.format()`:

```
>>> from math import pi
>>> f"The ratio is about {pi:f}"
'The ratio is about 3.141593'
>>> f"Which is roughly {pi:0.2f}"
'Which is roughly 3.14'
>>> f"{pi:.0f} is NOT a good approximation."
'3 is NOT a good approximation.'

>>> number = 10**9
>>> f"Billions and {number:,d}'s"
"Billions and 1,000,000,000's"
>>> f"Billions and {number:E}'s"
"Billions and 1.000000E+09's"
```

As well as the width, alignment, and fill:

```
>>> num = 753
>>> word = "WOW"
>>> f"foo{num:7d}bar"
'foo    753bar'
>>> f"foo{word:7s}bar"
'fooWOW    bar'
>>> f"foo{word:>7s}bar"
'foo    WOWbar'
>>> f"foo{num:<7d}bar"
'foo753    bar'
>>> f"foo{num:^7d}bar"
'foo  753  bar'
>>> f"foo{num:X^7d}bar"
'fooXX753XXbar'
```

Now it's clear why I emphasize `str.format()` in this chapter, and this book. Practically speaking, all Python programmers need to know `str.format()` anyway; and once you've learned it, you are fluent with f-strings almost immediately.

The main downside to f-strings will become less important over time. It only works with Python 3.6 and later. That means in order to use f-strings in your code, you must be working on a codebase which will only ever be executed on those versions. This applies not only to your fellow developers, but - more problematically - all end users. If a customer has installed Python 3.5 on a server, and your program uses f-strings, they won't be able to run it unless you can convince them to upgrade. Which, unfortunately, probably isn't as high a priority for them as it is for you.

As I write this, f-strings have been out a short while. But many Python developers seem to have already fallen in love with them. The next edition of this book may emphasize them more, depending how quickly the Python community moves to versions of Python which support f-strings, and how popular they become. Fortunately, it's quite easy to switch back and forth between `str.format()` and f-strings; there seems to be very little mental energy needed to switch. So if you want to use f-strings, you can do so when it's practical, and then easily switch to `str.format()` when needed.

4.5 Percent Formatting

Modern Python still needs percent formatting in a few places, mainly when you work with the `logging` module. Thankfully, you don't need to know all its details. Learning just a few parts of percent formatting will cover 95% of what you're likely to need. I'll focus on that high-impact portion here; for more detail, consult the official Python reference.[2]

Percent formatting is officially called "printf-style string formatting", and - like the string formatting in *many* languages - has its roots in C's `printf()`. So if you have experience with that, you'll skim through quickly - though there *are* a few differences.

It uses the percent character in two ways. Here's a simple example:

```
>>> "Hello, %s, today is %s." % ("Aaron", "Tuesday")
'Hello, Aaron, today is Tuesday.'
```

Notice percent is used as a binary operator. On its left is a string; on its right, a tuple of two strings. That string on the left is called the *format*; you can see it has two percent characters inside. Specifically, it has the sequence `%s` twice. These `%s` sequences are called *conversion specifiers*. There's two of them, and two values in the tuple on the right; they map to each other. Each value is substituted for its corresponding `%s`.

The "s" means that the inserted value is converted to a string; these are already strings, though, so they are just placed in. You can also use %d for an integer:

```
>>> "Here's %d dollars and %d cents." % (14, 25)
"Here's 14 dollars and 25 cents."
```

Sometimes it doesn't matter much which specifier you use. If that last format string was `"Here's %s dollars and %s cents."`, it would render the same. But I recommend you choose the strictest type that will work for your data; if you expect the value to be an integer, use %d, so that if it's *not* an integer, you'll get a clear stack trace telling you what's wrong:

```
>>> "Here's %d dollars and %d cents." % (14, "a quarter")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: %d format: a number is required, not str
```

(I'm assuming you'd rather discover a lurking bug now, during development, instead of through an angry customer's bug report later.)

[2] https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting

Sometimes you'll have just one value to interject. A tuple of one value must be written with an extra comma, like `(foo,)` - because `(foo)` becomes simply `foo` in the normal meaning of parentheses for grouping. So to format a string of one variable, you can type this:

```
>>> "High %d!" % (5,)
'High 5!'
```

Early in Python's history, people decided typing those extra characters was a bit annoying, especially given how common it is to format a string with a single value. So as a special case, when you have a single specifier, you can pass in a single value instead of a tuple:

```
>>> "High %d!" % 5
'High 5!'
```

In addition to `%d` and `%s`, you can use `%f` for a floating-point number:

```
>>> "I owe you $%f." % 7.05
'I owe you $7.050000.'
```

You'll often want to specify its precision - the number of digits to the right of the decimal. Do this by inserting a *precision code* between `%` and `f` - which will be a `.` (dot) followed by an integer:

```
>>> "I owe you $%.2f." % 5.05
'I owe you $5.05.'
```

You can also use `%r` (which formats the `repr()` of the object). It's especially useful for logging and troubleshooting:

```
>>> class Money:
...     def __init__(self, dollars, cents):
...         self.dollars = dollars
...         self.cents = cents
...     def __repr__(self):
...         return 'Money({},{})'.format(
...             self.dollars, self.cents)
>>> cash = Money(127, 82)
>>> "Cash on hand: %r" % cash
'Cash on hand: Money(127,82)'
```

There is much more to percent formatting than this, but what we've covered lets you read and write most of what you need.

Now, as mentioned, in modern Python you'll mainly need to use percent formatting with the `logging` module. However, in that case, you use it a bit differently. As described in its chapter, the `logging` module includes functions for log events at different levels of urgency - whether that's an error, a warning, or even a non-urgent informational message:

```
logging.info("So far, so good!")
```

You will *very* often want to inject run-time values into the message. For example, if a customer spends a certain amount of money:

```
logging.info("User %s spent $%0.2f", username, amount)
```

Notice there's no binary percent operator! That's deliberate. The logging functions are designed to take a format string as the first argument, and the values to substitute as subsequent arguments. That's because not every log message needs to be executed. You can - and often will - configured your logger to filter out all those boring `info` messages, for example, or omit the overly detailed debug messages. In other words, don't do this:

```
# NO! Bad code!
logging.info("User %s spent $%0.2f" % (username, amount))
```

Suppose you are filtering out `info` messages right now, so the `info` message doesn't need to actually log. In the first, recommended form, that `logging.info` line in your code is cheap; it's essentially treated by Python as a no-op. In the second form, it will still be translated as a no-op, *but only after that string is rendered.* So you unnecessarily incur the cost of rendering the string, just to throw it away. This is all explained in more detail in the logging chapter; for now, just be aware of the idea.

Index