3 Useful Python Code Patterns

Justus Perlwitz

2018-04-06

Contents

Over the years I have found out how important it is to constantly challenge yourself. Coding is craftsmanship. A good craftsman will always try to improve the toolbox they use every day. Code patterns belong in any good toolbox. When used wisely, they allow anyone to clearly communicate their program's intent to both people and machines. Code is not just something that machines execute — it's also something that people read, understand, and maintain.

Some of the programming patterns out there help make code better. "Better" in this case means that the program is:

- easy to understand,
- easy to extend, and
- easy to fix.

At the same time, "better" code — from a machine perspective — is also:

- quick to run,
- reliably when it comes to handling errors and exceptions, and
- efficient in using your computer's available resources.

I'd like to share with you three powerful programming patterns in Python (and similar dynamic object-oriented programming languages) that can make a big difference when it comes to improving your code. Documents like the Zen of Python or the Style Guide for Python Code give lots of useful hints on improving your code's ability to communicate intent. The following code patterns are my personal favorites and can be found (one way or another) in many of the coding guidelines out there.

1 1. Flat, Not Nested

The *Zen of Python* tells you that flat is better than nested.

Flat cose is better than nested code both in terms of how deeply woven your program's function call graph is and from from a practical point of view, how far indented your code ends up being. Emphasizing a sparse use of *nestedness* will always lead to better code.

Flat code is also because layering too many paths of execution in your functions can make them hard to understand. Many paths obscure the *main* path a program will take 99 % of the time and can distract by displaying the minutiae of error handling or data processing with the same priority given to the 2 or 3 lines of code that do most of the productive heavy lifting.

Imagine a web backend that updates user account data after performing a series of validations and input data verifications. In our imaginary example, the last step — updating user account data — can only happen if all the previous steps execute correctly.

```
def update_user(user, new_data, auth):
    if request.has_perm(auth, 'update_user'):
        if new_data.validate():
            # This is the last step
            # despite appearing almost
            # at the top of the function
            if user.data.update(new_data):
                return "Success"
            else:
                return "Could not update"
        else:
            return "Could not validate"
    else:
        return "Invalid permissions"
```

Assuming that `return "Success"` is going to be the most-used exit in this function, we can immediately see that it's not very easy to read. First of all, if the last step is updating the account, then seeing it right in the middle of the function can be confusing. The level of indentation decreases readability as well — scanning left to right with your eyes in order to continue downwards breaks the natural flow of reading code.

An alternative approach would be to return early if any of the assertions fall through. Check out the snippet below. Here we check each individual precondition and return early if it's not satisfied.

```
def update_user(user, new_data, auth):
    if not request.has_perm(auth, 'update_user'):
        return "Invalid permissions"

    if not new_data.validate():
        return "Could not validate"

    if not user.data.update(new_data):
        return "Could not update"

    return "Success"
```

It's tremendously useful to express complicated sets of preconditions in a s flat a way as possible. This allows the reader to quickly scan up and down and see where the actual processing of information happens.

In our `update_user()` function, the actual step lies in `if not user.data.update(new_data)` and you can find it without having to scan left and right.

2 2. Precise Exceptions

Another thing that took me a while to realize is the importance of being specific in error handling. For example, if we take code that reads and processes sensor data, we can imagine a function like the following:

```
def process():
    result = receive_sensor_data()
    frobnicated = frobnicate(result)
    return renice(result)
```

We use `process()` a few times and discover at some point that `receive_sensor_data()` could fail with a `SensorError`. For convenience, the whole block of code is then lifted into a `try...except...` block. Usually, when time is running out, finding exactly where and how failures in an API call could happen becomes a low-priority task. And finding the right exception type is often difficult, since you might want to be able to handle several exception types correctly. It would typically look like this:

```
def process():
    try:
```

```
        # Only this can throw SensorError
        result = receive_sensor_data()
        frobnicated = frobnicate(result)
        renice(result)
    except SensorError:
        handle_failure()
```

Good code is all about readability and maintainability. The snippet above shows that it's too easy to forget 6 months later which of the 3 lines after `try:` could actually raise SensorError. Since we know that only `receive_sensor_data()` can raise this exception, we readjust our code. Take a look and see what has improved.

```
def process():
    try:
        result = receive_sensor_data()
    except SensorError:
        handle_failure()
    else:
        frobnicated = frobnicate(result)
        renice(result)
```

We push the other two lines of code that can't raise a `SensorError` into an `else:` block. The `else:` block is executed when the `try:` block runs through its code without any errors. This makes it very clear that only `receive_sensor_data()` can ever raise this exception, since it's now the only line of code in the `try:` block.

That way, there can never be any confusion — even when the code hasn't been touched in a while — about which line might raise the actual exception.

Another common problem with taking shortcuts when writing error handling code is that the code doesn't catch errors precisely enough. This often happens when the specified exception class is too high up in the inheritance hierarchy of Python exceptions. A useful primer on the hierarchy of built-in errors in Python can be found here.

The useful thing about error handling code is that it not only handles errors, but also serves as documentation for later developers who want to understand the failure modes of a piece of code. This matters if the same failing function is used in another block of code somewhere else. A pattern we often see is the following:

```
def process():
    try:
        something_complex()
    # Underspecified exception
    # How can the try: block fail?
    except Exception:
        handle_error()
```

Correctly implementing error handling is not an easy task. It requires going lots of debugging sessions and studying the documentation of all the different libraries you're using. It might take you 2 weeks to find out that `receive_sensor_data()` will raise the exception `SensorError` and nothing else. You can share this knowledge immediately by precisely specifying this exception class in the `except:` block.

```
def process():
    try:
        something_complex()
    # Now it becomes clear how something_complex
    # can fail
    except SomethingComplexException:
        handle_error()
```

As a bonus, well-specified error handling can prevent bugs and catch other issues when it comes to the functions your code is calling. Maybe you catch `Exception`, and for some reason the code you're using doesn't convert integers properly and a `ValueError` is raised. By specifying that the exception handling should only take place for `SomethingComplexException`, you won't accidentally bury this issue and render your program state invalid.

3 3. Use Generators

Generators in Python are incredibly useful. A generator is a piece of suspendible code that executes until a `yield` statement is encountered. The execution can be continued by calling `next()` on it in Python. We typically use a generator like this:

```
def generator():
    for i in range(10):
        yield i * 2

g = generator()
```

The generator will run through its code until it hits the first `yield` statement. As a matter of fact, generators are s imply Python functions that have at least one `yield` statement. This immediately turns them into a generator. More specifically, this means a generator can't return values directly to its caller. If we look at the contents of `g`, we will see what `generator()` returns.

```
g
```

Output:

```
<generator object generator at 0x10574ed00>
```

We can continue the execution of the generator and retrieve the first value it `yield`s by using `next()` on it:

```
next(g)
```

Output:

```
0
```

As we can see, this is nothing more than `i * 2` for `i = 0`. We can call it one more time and retrieve the next value:

```
next(g)
```

Output:

```
2
```

We can also take a shortcut here and easily retrieve all the items at once:

```
l = list(generator())
l
```

Output:

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

In this case, once it receives a generator as an argument, `list()` will simply run the generator by repeatedly calling `next()`, until it hits a full stop. After that, all the individual items that were retrieved are put into a list.

This can also be done with `tuple()`. In that case, the resulting object will be a `tuple`, not a `list`:

```
t = tuple(generator())
t
```

Output:

```
(0, 2, 4, 6, 8, 10, 12, 14, 16, 18)
```

So we immediately see that generators offer a lot of flexibility. As a more complicated example, let's consider a `get_plots()` function that goes through a list of items, then generates and returns a list of plots to the caller:

```
def get_plots(items):
    result = []
    for item in items:
        process_foobar(item)
        reintensify_verticals(item)
        plot = confusion_matrix(item)
        result.append(plot)
```

In order to use `get_plots()`, you'd call it as follows:

```
>>> plots = get_plots(items)
>>> plots
[<plot_1>, <plot_2>]
>>> draw_plots(plots)
...
```

If we think back to the previous generator example, we can take a productive shortcut when defining `get_plots`:

```
def get_plots(items):
    for item in items:
        process_foobar(item)
        reintensify_verticals(item)
        yield confusion_matrix(item)
```

Instead of generating a list and returning it, we leave it to the caller to process the individual plot items `get_plots()` returns. This allows us to choose how we want the items to be stored in memory:

```
>>> plots = tuple(get_plots(items))
>>> plots = list(get_plots_items)
>>> cache.store_list(plots)
>>> # Etc.
```

We can also skip storing the results and process them further instead. For example, if we want to draw the plots, we can imagine creating a `draw_plots()` function, like so:

```
def draw_plots(items):
    for plot in items:
        window.display(plot)
```

The `for` loop easily operates on generators, and the plots can be displayed like so:

```
>>> plots = get_plots(items)
>>> plots
<generator object get_plots at 0x...>
>>> draw_plots(plots)
...
```