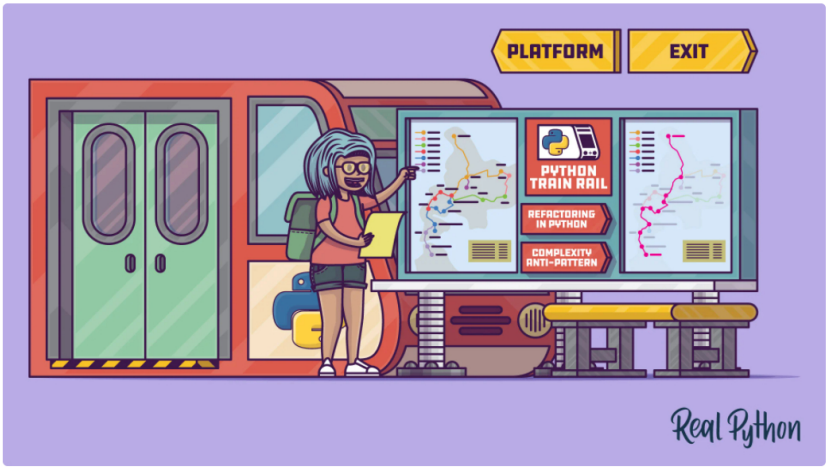


Refactoring Python Applications for Simplicity

by Anthony Shaw Mar 04, 2019 22 Comments best-practices intermediate

Table of Contents

Do you want simpler Python code? You always start a project with the best intentions, a clean codebase, and a nice structure. But over time, there are changes to your apps, and things can get a little messy.

If you can write and maintain clean, simple Python code, then it'll save you lots of time in the long term. You can spend less time testing, finding bugs, and making changes when your code is well laid out and simple to follow.

In this tutorial you'll learn:

- How to measure the complexity of Python code and your applications
- How to change your code without breaking it
- What the common issues in Python code that cause extra complexity are and how you

Table of Contents

- Code Complexity in Python
- Refactoring in Python
- Complexity Anti-Patterns
- Conclusion

can fix them

Throughout this tutorial, I'm going to use the theme of subterranean railway networks to explain complexity because navigating a subway system in a large city can be complicated! Some are well designed, and others seem overly complex.

Code Complexity in Python

The complexity of an application and its codebase is relative to the task it's performing. If you're writing code for NASA's jet propulsion laboratory (literally rocket science), then it's going to be complicated.

The question isn't so much, "Is my code complicated?" as, "Is my code more complicated than it needs to be?"

The Tokyo railway network is one of the most extensive and complicated in the world. This is partly because Tokyo is a metropolis of over 30 million people, but it's also because there are 3 networks overlapping each other.

The author of this article getting lost on the Tokyo Metro

There are the Toei and Tokyo Metro rapid-transport networks as well as the Japan Rail East trains going through Central Tokyo. To even the most experienced traveler, navigating central Tokyo can be mind-bogglingly complicated.

Here is a map of the Tokyo railway network to give you some perspective:

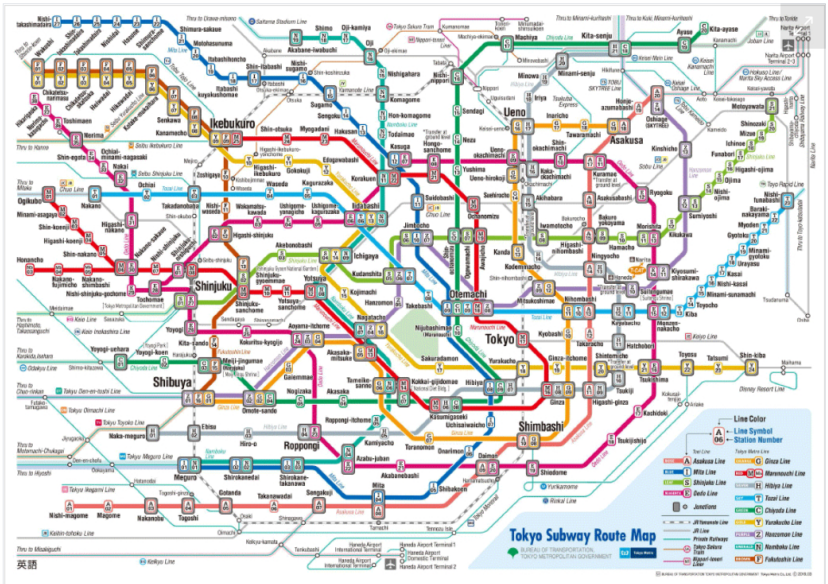
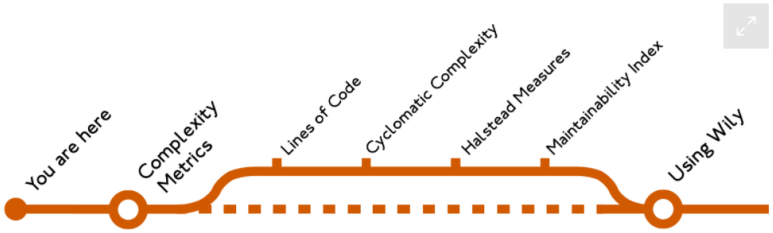


Image: Tokyo Metro Co.

If your code is starting to look a bit like this map, then this is the tutorial for you.

First, we'll go through 4 metrics of complexity that can give you a scale to measure your relative progress in the mission to make your code simpler:



After you've explored the metrics, you'll learn about a tool called `wily` to automate calculating those metrics.

Metrics for Measuring Complexity

Much time and research have been put into analyzing the complexity of computer software. Overly complex and unmaintainable applications can have a very real cost.

The complexity of software correlates to the quality. Code that is easy to read and understand is more likely to be updated by developers in the future.

Here are some metrics for programming languages. They apply to many languages, not just Python.

Lines of Code

LOC, or Lines of Code, is the crudest measure of complexity. It is debatable whether there is any direct correlation between the lines of code and the complexity of an application, but the indirect correlation is clear. After all, a program with 5 lines is likely simpler than one with 5 million.

When looking at Python metrics, we try to ignore blank lines and lines containing comments.

Lines of code can be calculated using the `wc` command on Linux and Mac OS, where `file.py` is the name of the file you want to measure:

Shell

```
$ wc -l file.py
```

If you want to add the combined lines in a folder by recursively searching for all `.py` files, you can combine `wc` with the `find` command:

Shell

```
$ find . -name \*.py | xargs wc -l
```

For Windows, PowerShell offers a word count command in `Measure-Object` and a recursive file search in `Get-ChildItem`:

Windows PowerShell Console

```
PS C:\> Get-ChildItem -Path *.py -Recurse | Measure-Object -Line
```

In the response, you will see the total number of lines.

Why are lines of code used to quantify the amount of code in your application? The

assumption is that a line of code roughly equates to a statement. Lines is a better measure than characters, which would include whitespace.

In Python, we are encouraged to put a single statement on each line. This example is 9 lines of code:

Python

```python
x = 5
value = input("Enter a number: ")
y = int(value)
if x < y:
    print(f"{x} is less than {y}")
elif x == y:
    print(f"{x} is equal to {y}")
else:
    print(f"{x} is more than {y}")
```

If you used only lines of code as your measure of complexity, it could encourage the wrong behaviors.

Python code should be easy to read and understand. Taking that last example, you could reduce the number of lines of code to 3:

Python

```python
x = 5; y = int(input("Enter a number:"))
equality = "is equal to" if x == y else "is less than" if x < y else "is more thar
print(f"{x} {equality} {y}")
```

But the result is hard to read, and PEP 8 has guidelines around maximum line length and line breaking. You can check out How to Write Beautiful Python Code With PEP 8 for more on PEP 8.

This code block uses 2 Python language features to make the code shorter:

- **Compound statements:** using ;
- **Chained conditional or ternary statements:** `name = value if condition else value if condition2 else value2`

We have reduced the number of lines of code but violated one of the fundamental laws of Python:

> "Readability counts"
>
> — Tim Peters, Zen of Python

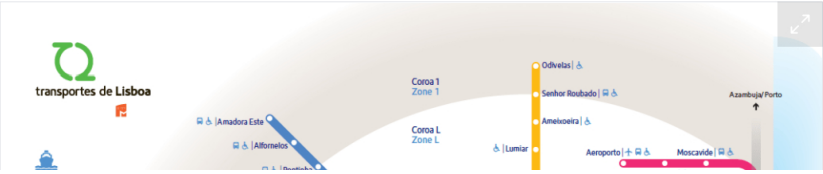
This shortened code is potentially harder to maintain because code maintainers are humans, and this short code is harder to read. We will explore some more advanced and useful metrics for complexity.

Cyclomatic Complexity

Cyclomatic complexity is the measure of how many independent code paths there are through your application. A path is a sequence of statements that the interpreter can follow to get to the end of the application.

One way to think of cyclomatic complexity and code paths is imagine your code is like a railway network.

For a journey, you may need to change trains to reach your destination. The Lisbon Metropolitan railway system in Portugal is simple and easy to navigate. The cyclomatic complexity for any trip is equal to the number of lines you need to travel on:



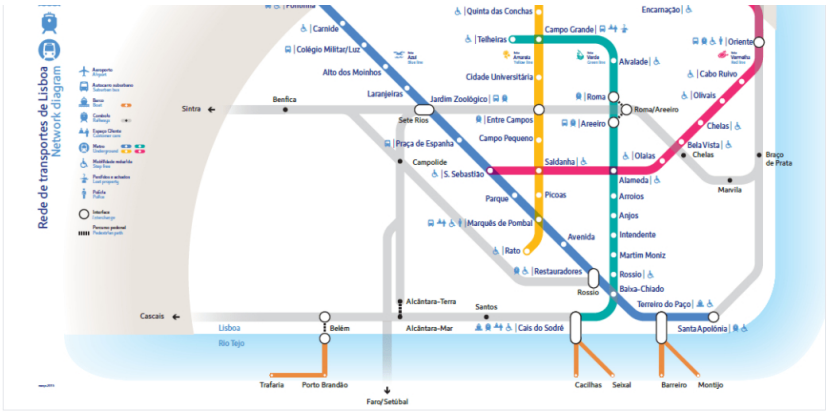


Image: Metro Lisboa

If you needed to get from *Alvalade* to *Anjos*, then you would travel 5 stops on the *linha verde* (green line):

Image: Metro Lisboa

This trip has a cyclomatic complexity of 1 because you only take 1 train. It's an easy trip. That train is equivalent in this analogy to a code branch.

If you needed to travel from the *Aeroporto* (airport) to sample the food in the district of *Belém*, then it's a more complicated journey. You would have to change trains at *Alameda* and *Cais do Sodré*:

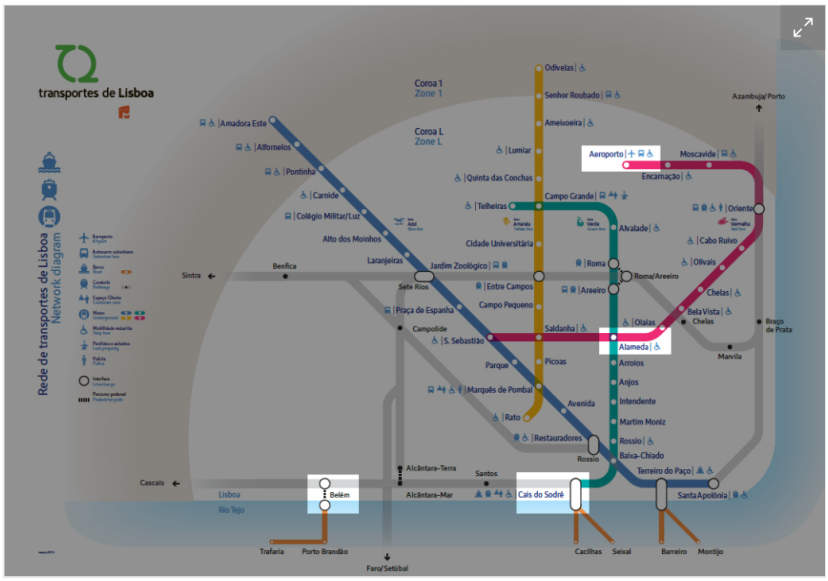


Image: Metro Lisboa

This trip has a cyclomatic complexity of 3, because you take 3 trains. You might be better off taking a taxi!

Seeing as how you're not navigating Lisbon, but rather writing code, the changes of train line become a branch in execution, like an `if` statement.

Let's explore this example:

Python

```
x = 1
```

There is only 1 way this code can be executed, so it has a cyclomatic complexity of 1.

If we add a decision, or branch to the code as an `if` statement, it increases the complexity:

Python

```
x = 1
if x < 2:
    x += 1
```

Even though there is only 1 way this code can be executed, as `x` is a constant, this has a cyclomatic complexity of 2. All of the cyclomatic complexity analyzers will treat an `if` statement as a branch.

This is also an example of overly complex code. The `if` statement is useless as `x` has a fixed value. You could simply refactor this example to the following:

Python

```
x = 2
```

That was a toy example, so let's explore something a little more real.

`main()` has a cyclomatic complexity of 5. I'll comment each branch in the code so you can see where they are:

Python

```
# cyclomatic_example.py
import sys

def main():
    if len(sys.argv) > 1:  # 1
        filepath = sys.argv[1]
    else:
        print("Provide a file path")
        exit(1)
    if filepath:  # 2
        with open(filepath) as fp:  # 3
            for line in fp.readlines():  # 4
                if line != "\n":  # 5
                    print(line, end="")

if __name__ == "__main__":  # Ignored.
    main()
```

There are certainly ways that code can be refactored into a far simpler alternative. We'll get to that later.

> **Note:** The Cyclomatic Complexity measure was developed by Thomas J. McCabe, Sr in 1976. You may see it referred to as the **McCabe metric** or **McCabe number**.

In the following examples, we will use the `radon` library from PyPI to calculate metrics. You can install it now:

Shell

```
$ pip install radon
```

To calculate cyclomatic complexity using `radon`, you can save the example into a file called `cyclomatic_example.py` and use `radon` from the command line.

The `radon` command takes 2 main arguments:

1. The type of analysis (`cc` for cyclomatic complexity)
2. A path to the file or folder to analyze

Execute the `radon` command with the `cc` analysis against the `cyclomatic_example.py` file. Adding `-s` will give the cyclomatic complexity in the output:

Shell

```
$ radon cc cyclomatic_example.py -s
cyclomatic_example.py
    F 4:0 main - B (6)
```

The output is a little cryptic. Here is what each part means:

- `F` means function, `M` means method, and `C` means class.
- `main` is the name of the function.
- `4` is the line the function starts on.
- `B` is the rating from A to F. A is the best grade, meaning the least complexity.
- The number in parentheses, `6`, is the cyclomatic complexity of the code.

Halstead Metrics

The Halstead complexity metrics relate to the size of a program's codebase. They were developed by Maurice H. Halstead in 1977. There are 4 measures in the Halstead equations:

- **Operands** are values and names of variables.
- **Operators** are all of the built-in keywords, like `if`, `else`, `for` or `while`.
- **Length (N)** is the number of operators plus the number of operands in your program.
- **Vocabulary (h)** is the number of *unique* operators plus the number of *unique* operands in your a program.

There are then 3 additional metrics with those measures:

- **Volume (V)** represents a product of the **length** and the **vocabulary**.
- **Difficulty (D)** represents a product of half the unique operands and the reuse of operands.
- **Effort (E)** is the overall metric that is a product of **volume** and **difficulty**.

All of this is very abstract, so let's put it in relative terms:

- The effort of your application is highest if you use a lot of operators and unique operands.
- The effort of your application is lower if you use a few operators and fewer variables.

For the `cyclomatic_complexity.py` example, operators and operands both occur on the first line:

Python

```
import sys  # import (operator), sys (operand)
```

`import` is an operator, and `sys` is the name of the module, so it's an operand.

In a slightly more complex example, there are a number of operators and operands:

Python

```
if len(sys.argv) > 1:
    ...
```

There are 5 operators in this example:

1. `if`
2. `(`
3. `)`
4. `>`
5. `:`

Furthermore, there are 2 operands:

1. `sys.argv`
2. `1`

Be aware that `radon` only counts a subset of operators. For example, parentheses are excluded in any calculations.

To calculate the Halstead measures in `radon`, you can run the following command:

Shell

```
$ radon hal cyclomatic_example.py
cyclomatic_example.py:
    h1: 3
    h2: 6
    N1: 3
    N2: 6
    vocabulary: 9
    length: 9
    calculated_length: 20.264662506490406
    volume: 28.529325012980813
    difficulty: 1.5
    effort: 42.793987519471216
    time: 2.377443751081734
    bugs: 0.009509775004326938
```

Why does `radon` give a metric for time and bugs?

Halstead theorized that you could estimate the time taken in seconds to code by dividing the effort (`E`) by 18.

Halstead also stated that the expected number of bugs could be estimated dividing the volume (`V`) by 3000. Keep in mind this was written in 1977, before Python was even invented! So don't panic and start looking for bugs just yet.

Maintainability Index

The maintainability index brings the McCabe Cyclomatic Complexity and the Halstead Volume measures in a scale roughly between zero and one-hundred.

If you're interested, the original equation is as follows:

$$MI = 171 - 5.2ln(V) - 0.23(C) - 16.2ln(L)$$

In the equation, `V` is the Halstead volume metric, `C` is the cyclomatic complexity, and `L` is the number of lines of code.

If you're as baffled as I was when I first saw this equation, here's it means: it calculates a scale that includes the number of variables, operations, decision paths, and lines of code.

It is used across many tools and languages, so it's one of the more standard metrics. However, there are numerous revisions of the equation, so the exact number shouldn't be taken as fact. `radon`, `wily`, and Visual Studio cap the number between 0 and 100.

On the maintainability index scale, all you need to be paying attention to is when your code is getting significantly lower (toward 0). The scale considers anything lower than 25 as **hard to maintain**, and anything over 75 as **easy to maintain**. The Maintainability Index is also referred to as **MI**.

The maintainability index can be used as a measure to get the current maintainability of your application and see if you're making progress as you refactor it.

To calculate the maintainability index from `radon`, run the following command:

Shell

```
$ radon mi cyclomatic_example.py -s
cyclomatic_example.py - A (87.42)
```

In this result, `A` is the grade that `radon` has applied to the number `87.42` on a scale. On this scale, `A` is most maintainable and `F` the least.

Using `wily` to Capture and Track Your Projects' Complexity

`wily` is an open-source software project for collecting code-complexity metrics, including the ones we've covered so far like Halstead, Cyclomatic, and LOC. `wily` integrates with Git and can automate the collection of metrics across Git branches and revisions.

The purpose of `wily` is to give you the ability to see trends and changes in the complexity of your code over time. If you were trying to fine-tune a car or improve your fitness, you'd start off with measuring a baseline and tracking improvements over time.

Installing `wily`

`wily` is available on PyPI and can be installed using pip:

Shell

```
$ pip install wily
```

Once `wily` is installed, you have some commands available in your command-line:

- **`wily build`:** iterate through the Git history and analyze the metrics for each file
- **`wily report`:** see the historical trend in metrics for a given file or folder
- **`wily graph`:** graph a set of metrics in an HTML file

Building a Cache

Before you can use `wily`, you need to analyze your project. This is done using the `wily build` command.

For this section of the tutorial, we will analyze the very popular `requests` package, used for talking to HTTP APIs. Because this project is open-source and available on GitHub, we can easily access and download a copy of the source code:

Shell

```
$ git clone https://github.com/requests/requests
$ cd requests
$ ls
AUTHORS.rst         CONTRIBUTING.md     LICENSE             Makefile
Pipfile.lock        _appveyor           docs                pytest.ini
setup.cfg           tests               CODE_OF_CONDUCT.md  HISTORY.md
MANIFEST.in         Pipfile             README.md           appveyor.yml
ext                 requests            setup.py            tox.ini
```

Note: Windows users should use the PowerShell command prompt for the following examples instead of traditional MS-DOS Command-Line. To start the PowerShell CLI press `⊞ Win` + `R` and type `powershell` then `Enter ↩`.

You will see a number of folders here, for tests, documentation, and configuration. We're only interested in the source code for the `requests` Python package, which is in a folder called `requests`.

Call the `wily build` command from the cloned source code and provide the name of the

source code folder as the first argument:

Shell

```
$ wily build requests
```

This will take a few minutes to analyze, depending on how much CPU power your computer has:

Collecting Data on Your Project

Once you have analyzed the `requests` source code, you can query any file or folder to see key metrics. Earlier in the tutorial, we discussed the following:

- Lines of Code
- Maintainability Index
- Cyclomatic Complexity

Those are the 3 default metrics in `wily`. To see those metrics for a specific file (such as `requests/api.py`), run the following command:

Shell

```
$ wily report requests/api.py
```

`wily` will print a tabular report on the default metrics for each Git commit in reverse date order. You will see the most recent commit at the top and the oldest at the bottom:

Revision	Author	Date	MI	Lines of Code	Cyclomatic Complexity
f37daf2	Nate Prewitt	2019-01-13	100 (0.0)	158 (0)	9 (0)
6dd410f	Ofek Lev	2019-01-13	100 (0.0)	158 (0)	9 (0)
5c1f72e	Nate Prewitt	2018-12-14	100 (0.0)	158 (0)	9 (0)
c4d7680	Matthieu Moy	2018-12-14	100 (0.0)	158 (0)	9 (0)
c452e3b	Nate Prewitt	2018-12-11	100 (0.0)	158 (0)	9 (0)
5a1e738	Nate Prewitt	2018-12-10	100 (0.0)	158 (0)	9 (0)

This tells us that the `requests/api.py` file has:

- 158 lines of code
- A perfect maintainability index of 100
- A cyclomatic complexity of 9

To see other metrics, you first need to know the names of them. You can see this by running the following command:

Shell

```
$ wily list-metrics
```

You will see a list of operators, modules that analyze the code, and the metrics they provide.

To query alternative metrics on the report command, add their names after the filename. You can add as many metrics as you wish. Here's an example with the Maintainability Rank and the Source Lines of Code:

Shell

```
$ wily report requests/api.py maintainability.rank raw.sloc
```

You will see the table now has 2 different columns with the alternative metrics.

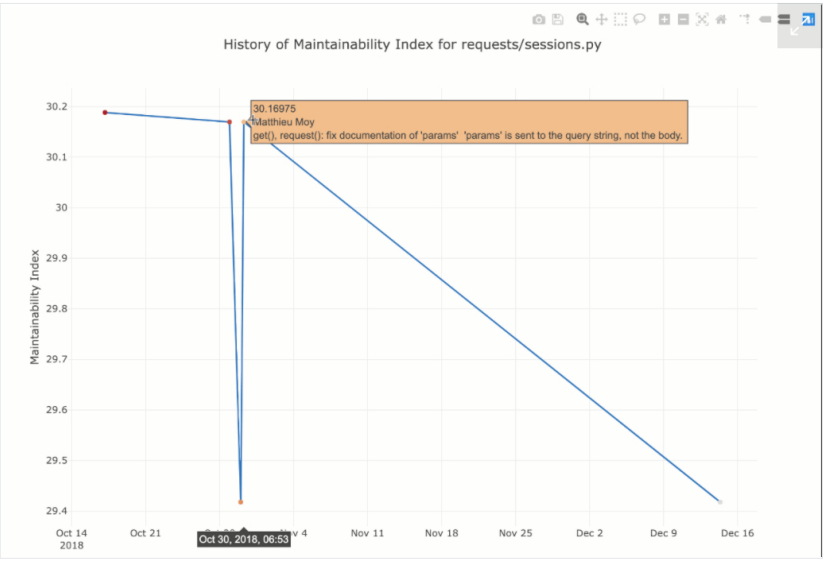
Graphing Metrics

Now that you know the names of the metrics and how to query them on the command line, you can also visualize them in graphs. `wily` supports HTML and interactive charts with a similar interface as the report command:

Shell

```
$ wily graph requests/sessions.py maintainability.mi
```

Your default browser will open with an interactive chart like this:



You can hover over specific data points, and it will show the Git commit message as well as the data.

If you want to save the HTML file in a folder or repository, you can add the `-o` flag with the path to a file:

Shell

```
$ wily graph requests/sessions.py maintainability.mi -o my_report.html
```

There will now be a file called `my_report.html` that you can share with others. This command is ideal for team dashboards.

`wily` as a `pre-commit` Hook

`wily` can be configured so that before you commit changes to your project, it can alert you to improvements or degradations in complexity.

`wily` has a `wily diff` command, that compares the last indexed data with the current working copy of a file.

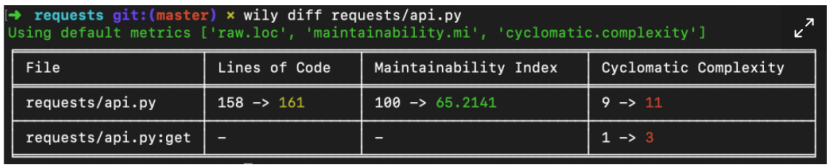
To run a `wily diff` command, provide the names of the files you have changed. For example, if I made some changes to `requests/api.py` you will see the impact on the metrics by running `wily diff` with the file path:

By running `wily diff` with the file path:

Shell

```
$ wily diff requests/api.py
```

In the response, you will see all of the changed metrics, as well as the functions or classes that have changed for cyclomatic complexity:



The `diff` command can be paired with a tool called `pre-commit`. `pre-commit` inserts a hook into your Git configuration that calls a script every time you run the `git commit` command.

To install `pre-commit`, you can install from PyPI:

Shell

```
$ pip install pre-commit
```

Add the following to a `.pre-commit-config.yaml` in your projects root directory:

YAML

```yaml
repos:
-   repo: local
    hooks:
    -   id: wily
        name: wily
        entry: wily diff
        verbose: true
        language: python
        additional_dependencies: [wily]
```

Once setting this, you run the `pre-commit install` command to finalize things:

Shell

```
$ pre-commit install
```

Whenever you run the `git commit` command, it will call `wily diff` along with the list of files you've added to your staged changes.

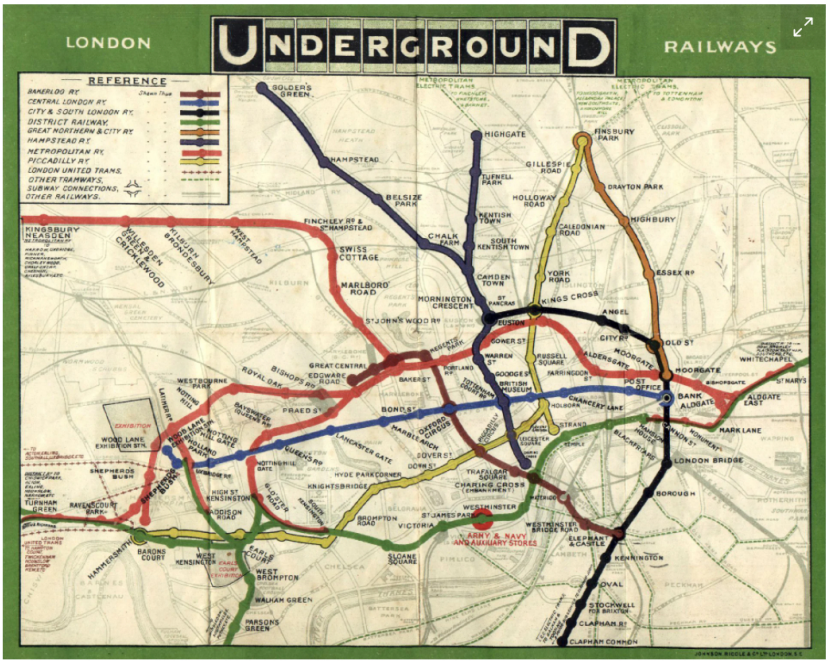
`wily` is a useful utility to baseline the complexity of your code and measure the improvements you make when you start to refactor.

Refactoring in Python

Refactoring is the technique of changing an application (either the code or the architecture) so that it behaves the same way on the outside, but internally has improved. These improvements can be stability, performance, or reduction in complexity.

One of the world's oldest underground railways, the London Underground, started in 1863 with the opening of the Metropolitan line. It had gas-lit wooden carriages hauled by steam locomotives. On the opening of the railway, it was fit for purpose. 1900 brought the invention of the electric railways.

By 1908, the London Underground had expanded to 8 railways. During the Second World War, the London Underground stations were closed to trains and used as air-raid shelters. The modern London Underground carries millions of passengers a day with over 270 stations:

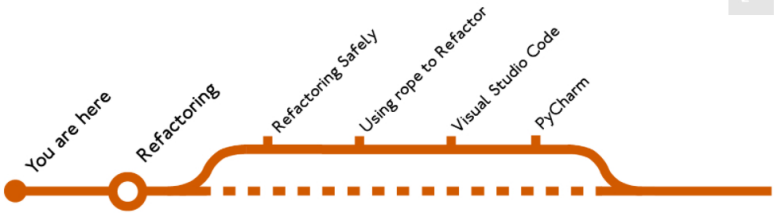


Joint London Underground Railways Map, c. 1908 (Image: Wikipedia)

It's almost impossible to write perfect code the first time, and requirements change frequently. If you would have asked the original designers of the railway to design a network fit for 10 million passengers a day in 2020, they would not design the network that exists today.

Instead, the railway has undergone a series of continuous changes to optimize its operation, design, and layout to match the changes in the city. It has been refactored.

In this section, you'll explore how to safely refactor by leveraging tests and tools. You'll also see how to use the refactoring functionality in Visual Studio Code and PyCharm:



Avoiding Risks With Refactoring: Leveraging Tools and Having Tests

If the point of refactoring is to improve the internals of an application without impacting the externals, how do you ensure the externals haven't changed?

Before you charge into a major refactoring project, you need to make sure you have a solid test suite for your application. Ideally, that test suite should be mostly automated, so that as you make changes, you see the impact on the user and address it quickly.

If you want to learn more about testing in Python, Getting Started With Testing in Python is a great place to start.

There is no perfect number of tests to have on your application. But, the more robust and thorough the test suite, the more aggressively you can refactor your code.

The two most common tasks you will perform when doing refactoring are:

- Renaming modules, functions, classes, and methods
- Finding usages of functions, classes, and methods to see where they are called

You can simply do this by hand using **search and replace**, but it is both time consuming and

risky. Instead, there are some great tools to perform these tasks.

Using `rope` for Refactoring

`rope` is a free Python utility for refactoring Python code. It comes with an extensive set of APIs for refactoring and renaming components in your Python codebase.

`rope` can be used in two ways:

1. By using an editor plugin, for Visual Studio Code, Emacs, or Vim
2. Directly by writing scripts to refactor your application

To use rope as a library, first install `rope` by executing `pip`:

Shell

```
$ pip install rope
```

It is useful to work with `rope` on the REPL so that you can explore the project and see changes in real time. To start, import the `Project` type and instantiate it with the path to the project:

Python >>>

```
>>> from rope.base.project import Project

>>> proj = Project('requests')
```

The `proj` variable can now perform a series of commands, like `get_files` and `get_file`, to get a specific file. Get the file `api.py` and assign it to a variable called `api`:

Python >>>

```
>>> [f.name for f in proj.get_files()]
['structures.py', 'status_codes.py', ...,'api.py', 'cookies.py']

>>> api = proj.get_file('api.py')
```

If you wanted to rename this file, you could simply rename it on the filesystem. However, any other Python files in your project that imported the old name would now be broken. Let's rename the `api.py` to `new_api.py`:

Python >>>

```
>>> from rope.refactor.rename import Rename

>>> change = Rename(proj, api).get_changes('new_api')

>>> proj.do(change)
```

Running `git status`, you will see that `rope` made some changes to the repository:

Shell

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   requests/__init__.py
    deleted:    requests/api.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)

   requests/.ropeproject/
   requests/new_api.py

no changes added to commit (use "git add" and/or "git commit -a")
```

The three changes made by `rope` are the following:

1. Deleted `requests/api.py` and created `requests/new_api.py`
2. Modified `requests/__init__.py` to import from `new_api` instead of `api`
3. Created a project folder named `.ropeproject`

To reset the change, run `git reset`.

There are hundreds of other refactorings that can be done with `rope`.

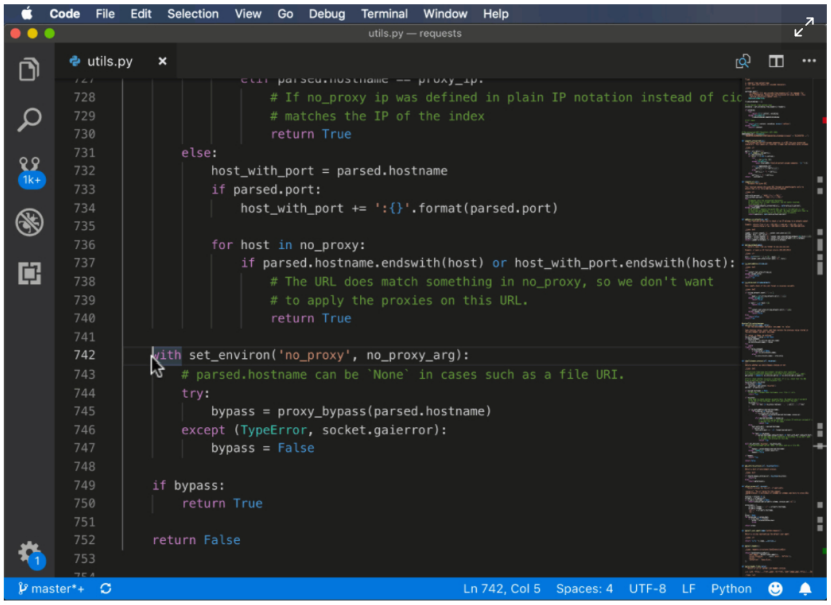
Using Visual Studio Code for Refactoring

Visual Studio Code opens up a small subset of the refactoring commands available in `rope` through its own UI.

You can:

1. Extract variables from a statement
2. Extract methods from a block of code
3. Sort imports into a logical order

Here is an example of using the *Extract methods* command from the command palette:



Using PyCharm for Refactoring

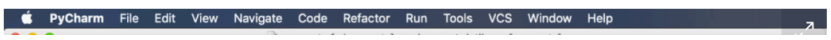
If you use or are considering using PyCharm as a Python editor, it’s worth taking note of the powerful refactoring capabilities it has.

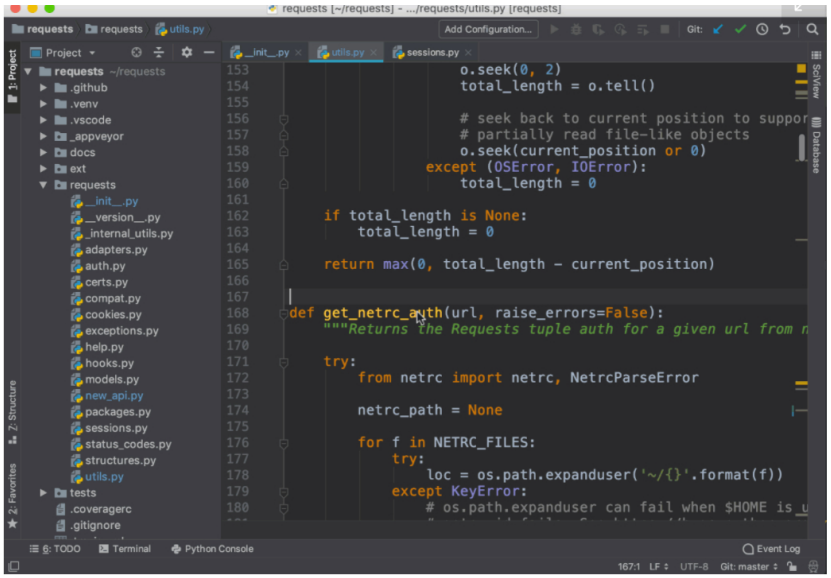
You can access all the refactoring shortcuts with the `^ Ctrl` + `T` command on Windows and macOS. The shortcut to access refactoring in Linux is `^ Ctrl` + `⇧ Shift` + `Alt` + `T`.

Finding Callers and Usages of Functions and Classes

Before you remove a method or class or change the way it behaves, you’ll need to know what code depends on it. PyCharm can search for all usages of a method, function, or class within your project.

To access this feature, select a method, class, or variable by right-clicking and select *Find Usages*:





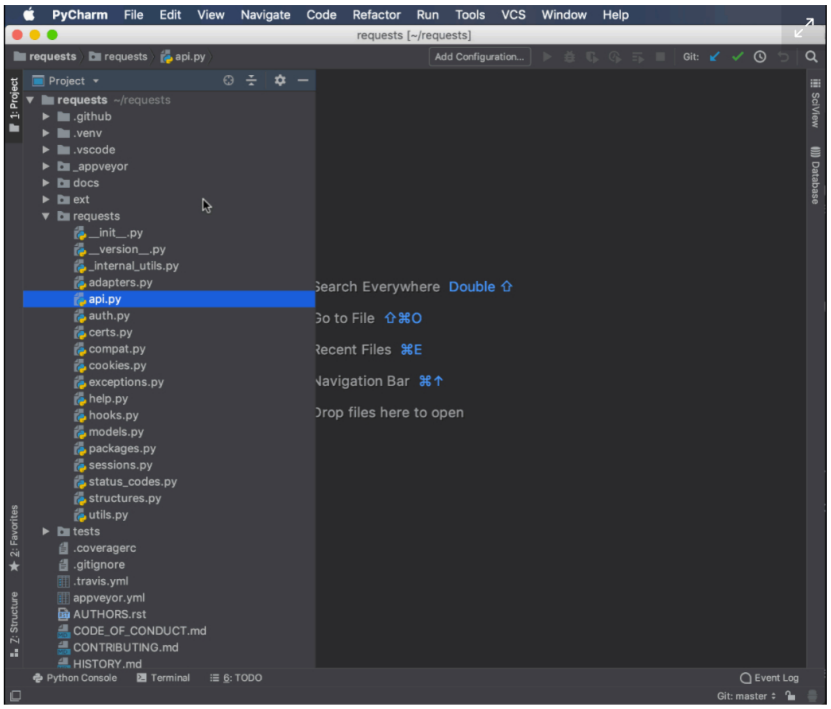
All of the code that uses your search criteria is shown in a panel at the bottom. You can double-click on any item to navigate directly to the line in question.

Using the PyCharm Refactoring Tools

Some of the other refactoring commands include the ability to:

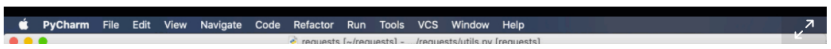
- Extract methods, variables, and constants from existing code
- Extract abstract classes from existing class signatures, including the ability to specify abstract methods
- Rename practically anything, from a variable to a method, file, class, or module

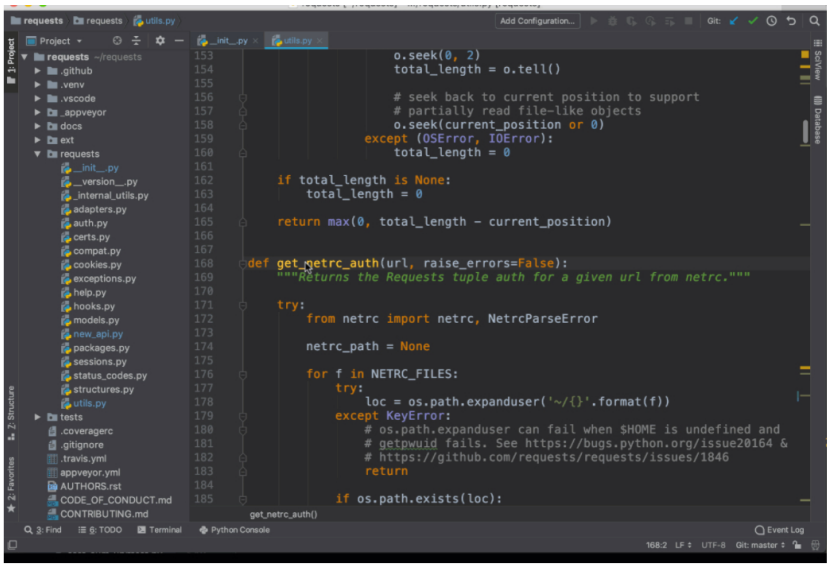
Here is an example of renaming the same `api.py` module you renamed earlier using the `rope` module to `new_api.py`:



The rename command is contextualized to the UI, which makes refactoring quick and simple. It has updated the imports automatically in `__init__.py` with the new module name.

Another useful refactor is the *Change Signature* command. This can be used to add, remove, or rename arguments to a function or method. It will search for usages and update them for you:





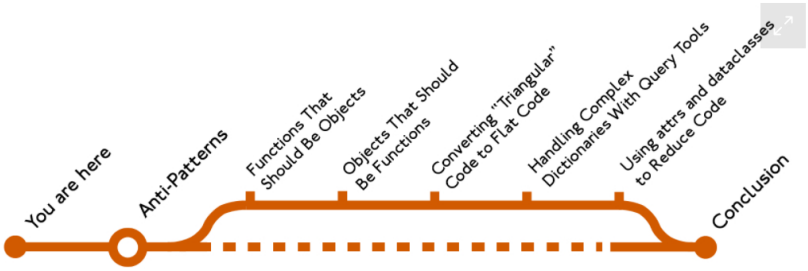
You can set default values and also decide how the refactoring should handle the new arguments.

Summary

Refactoring is an important skill for any developer. As you've learned in this chapter, you aren't alone. The tools and IDEs already come with powerful refactoring features to be able to make changes quickly.

Complexity Anti-Patterns

Now that you know how complexity can be measured, how to measure it, and how to refactor your code, it's time to learn 5 common anti-patterns that make code more complex than it need be:



If you can master these patterns and know how to refactor them, you'll soon be on track (pun intended) to a more maintainable Python application.

1. Functions That Should Be Objects

Python supports procedural programming using functions and also inheritable classes. Both are very powerful and should be applied to different problems.

Take this example of a module for working with images. The logic in the functions has been removed for brevity:

Python

```
# imagelib.py

def load_image(path):
    with open(path, "rb") as file:
```

```
    with open(path, 'rb') as file:
        fb = file.load()
    image = img_lib.parse(fb)
    return image

def crop_image(image, width, height):
    ...
    return image

def get_image_thumbnail(image, resolution=100):
    ...
    return image
```

There are a few issues with this design:

1. It's not clear if `crop_image()` and `get_image_thumbnail()` modify the original `image` variable or create new images. If you wanted to load an image then create both a cropped and thumbnail image, would you have to copy the instance first? You could read the source code in the functions, but you can't rely on every developer doing this.

2. You have to pass the image variable as an argument in every call to the image functions.

This is how the calling code might look:

Python

```
from imagelib import load_image, crop_image, get_image_thumbnail

image = load_image('~/face.jpg')
image = crop_image(image, 400, 500)
thumb = get_image_thumbnail(image)
```

Here are some symptoms of code using functions that could be refactored into classes:

- Similar arguments across functions
- Higher number of Halstead `h2` **unique operands**
- Mix of mutable and immutable functions
- Functions spread across multiple Python files

Here is a refactored version of those 3 functions, where the following happens:

- `.__init__()` replaces `load_image()`.
- `crop()` becomes a class method.
- `get_image_thumbnail()` becomes a property.

The thumbnail resolution has become a class property, so it can be changed globally or on that particular instance:

Python

```
# imagelib.py

class Image(object):
    thumbnail_resolution = 100
    def __init__(self, path):
        ...

    def crop(self, width, height):
        ...

    @property
    def thumbnail(self):
        ...
        return thumb
```

If there were many more image-related functions in this code, the refactoring to a class could make a drastic change. The next consideration would be the complexity of the consuming code.

This is how the refactored example would look:

```python
from imagelib import Image

image = Image('~/face.jpg')
image.crop(400, 500)
thumb = image.thumbnail
```

In the resulting code, we have solved the original problems:

- It is clear that `thumbnail` returns a thumbnail since it is a property, and that it doesn't modify the instance.
- The code no longer requires creating new variables for the crop operation.

2. Objects That Should Be Functions

Sometimes, the reverse is true. There is object-oriented code which would be better suited to a simple function or two.

Here are some tell-tale signs of incorrect use of classes:

- Classes with 1 method (other than `.__init__()`)
- Classes that contain only static methods

Take this example of an authentication class:

```python
# authenticate.py

class Authenticator(object):
    def __init__(self, username, password):
        self.username = username
        self.password = password

    def authenticate(self):
        ...
        return result
```

It would make more sense to just have a simple function named `authenticate()` that takes `username` and `password` as arguments:

```python
# authenticate.py

def authenticate(username, password):
    ...
    return result
```

You don't have to sit down and look for classes that match these criteria by hand: `pylint` comes with a rule that classes should have a minimum of 2 public methods. For more on PyLint and other code quality tools, you can check out Python Code Quality and Writing Cleaner Python Code With PyLint.

To install `pylint`, run the following command in your console:

```shell
$ pip install pylint
```

`pylint` takes a number of optional arguments and then the path to one or more files and folders. If you run `pylint` with its default settings, it's going to give a lot of output as `pylint` has a huge number of rules. Instead, you can run specific rules. The `too-few-public-methods` rule id is `R0903`. You can look this up on the documentation website:

Shell

```
$ pylint --disable=all --enable=R0903 requests
************* Module requests.auth
requests/auth.py:72:0: R0903: Too few public methods (1/2) (too-few-public-methods)
requests/auth.py:100:0: R0903: Too few public methods (1/2) (too-few-public-methods)
************* Module requests.models
requests/models.py:60:0: R0903: Too few public methods (1/2) (too-few-public-methods)

-----------------------------------
Your code has been rated at 9.99/10
```

This output tells us that `auth.py` contains 2 classes that have only 1 public method. Those classes are on lines 72 and 100. There is also a class on line 60 of `models.py` with only 1 public method.

3. Converting "Triangular" Code to Flat Code

If you were to zoom out on your source code and tilt your head 90 degrees to the right, does the whitespace look flat like Holland or mountainous like the Himalayas? Mountainous code is a sign that your code contains a lot of nesting.

Here's one of the principles in the **Zen of Python**:

> "Flat is better than nested"
>
> — Tim Peters, Zen of Python

Why would flat code be better than nested code? Because nested code makes it harder to read and understand what is happening. The reader has to understand and memorize the conditions as they go through the branches.

These are the symptoms of highly nested code:

- A high cyclomatic complexity because of the number of code branches
- A low Maintainability Index because of the high cyclomatic complexity relative to the number of lines of code

Take this example that looks at the argument `data` for strings that match the word `error`. It first checks if the `data` argument is a list. Then, it iterates over each and checks if the item is a string. If it is a string and the value is `"error"`, then it returns `True`. Otherwise, it returns `False`:

Python

```python
def contains_errors(data):
    if isinstance(data, list):
        for item in data:
            if isinstance(item, str):
                if item == "error":
                    return True
    return False
```

This function would have a low maintainability index because it is small, but it has a high cyclomatic complexity.

Instead, we can refactor this function by "returning early" to remove a level of nesting and returning `False` if the value of `data` is not list. Then using `.count()` on the list object to count for instances of `"error"`. The return value is then an evaluation that the `.count()` is greater than zero:

Python

```python
def contains_errors(data):
    if not isinstance(data, list):
        return False
    return data.count("error") > 0
```

Another technique for reducing nesting is to leverage list comprehensions. This common pattern of creating a new list, going through each item in a list to see if it matches a criterion,

then adding all matches to the new list:

```python
results = []
for item in iterable:
    if item == match:
        results.append(item)
```

This code can be replaced with a faster and more efficient list comprehension.

Refactor the last example into a list comprehension and an `if` statement:

```python
results = [item for item in iterable if item == match]
```

This new example is smaller, has less complexity, and is more performant.

If your data is not a single dimension list, then you can leverage the itertools package in the standard library, which contains functions for creating iterators from data structures. You can use it for chaining iterables together, mapping structures, cycling or repeating over existing iterables.

Itertools also contains functions for filtering data, like `filterfalse()`. For more on Itertools, check out Itertools in Python 3, By Example.

4. Handling Complex Dictionaries With Query Tools

One of Python's most powerful and widely used core types is the dictionary. It's fast, efficient, scalable, and highly flexible.

If you're new to dictionaries, or think you could leverage them more, you can read Dictionaries in Python for more information.

It does have one major side-effect: when dictionaries are highly nested, the code that queries them becomes nested too.

Take this example piece of data, a sample of the Tokyo Metro lines you saw earlier:

```python
data = {
 "network": {
  "lines": [
    {
     "name.en": "Ginza",
     "name.jp": "銀座線",
     "color": "orange",
     "number": 3,
     "sign": "G"
    },
    {
     "name.en": "Marunouchi",
     "name.jp": "丸ノ内線",
     "color": "red",
     "number": 4,
     "sign": "M"
    }
  ]
 }
}
```

If you wanted to get the line that matched a certain number, this could be achieved in a small function:

```python
```

```
def find_line_by_number(data, number):
    matches = [line for line in data if line['number'] == number]
    if len(matches) > 0:
        return matches[0]
    else:
        raise ValueError(f"Line {number} does not exist.")
```

Even though the function itself is small, calling the function is unnecessarily complicated because the data is so nested:

Python >>>

```
>>> find_line_by_number(data["network"]["lines"], 3)
```

There are third party tools for querying dictionaries in Python. Some of the most popular are JMESPath, glom, asq, and flupy.

JMESPath can help with our train network. JMESPath is a querying language designed for JSON, with a plugin available for Python that works with Python dictionaries. To install JMESPath, do the following:

Shell

```
$ pip install jmespath
```

Then open up a Python REPL to explore the JMESPath API, copying in the `data` dictionary. To get started, import `jmespath` and call `search()` with a query string as the first argument and the data as the second. The query string `"network.lines"` means return `data['network']['lines']`:

Python >>>

```
>>> import jmespath

>>> jmespath.search("network.lines", data)
[{'name.en': 'Ginza', 'name.jp': '銀座線',
  'color': 'orange', 'number': 3, 'sign': 'G'},
 {'name.en': 'Marunouchi', 'name.jp': '丸ノ内線',
  'color': 'red', 'number': 4, 'sign': 'M'}]
```

When working with lists, you can use square brackets and provide a query inside. The "everything" query is simply `*`. You can then add the name of the attribute inside each matching item to return. If you wanted to get the line number for every line, you could do this:

Python >>>

```
>>> jmespath.search("network.lines[*].number", data)
[3, 4]
```

You can provide more complex queries, like a `==` or `<`. The syntax is a little unusual for Python developers, so keep the documentation handy for reference.

If we wanted to find the line with the number `3`, this can be done in a single query:

Python >>>

```
>>> jmespath.search("network.lines[?number==`3`]", data)
[{'name.en': 'Ginza', 'name.jp': '銀座線', 'color': 'orange', 'number': 3, 'sign': 'G'
```

If we wanted to get the color of that line, you could add the attribute in the end of the query:

Python >>>

```
>>> jmespath.search("network.lines[?number==`3`].color", data)
['orange']
```

JMESPath can be used to reduce and simplify code that queries and searches through complex dictionaries.

5. Using `attrs` and `dataclasses` to Reduce Code

Another goal when refactoring is to simply reduce the amount of code in the codebase while achieving the same behaviors. The techniques shown so far can go a long way to refactoring code into smaller and simpler modules.

Some other techniques require a knowledge of the standard library and some third party libraries.

What Is Boilerplate?

Boilerplate code is code that has to be used in many places with little or no alterations.

Taking our train network as an example, if we were to convert that into types using Python classes and Python 3 type hints, it might look something like this:

Python

```
from typing import List

class Line(object):
    def __init__(self, name_en: str, name_jp: str, color: str, number: int, sign: str
        self.name_en = name_en
        self.name_jp = name_jp
        self.color = color
        self.number = number
        self.sign = sign

    def __repr__(self):
        return f"<Line {self.name_en} color='{self.color}' number={self.number} sign=

    def __str__(self):
        return f"The {self.name_en} line"

class Network(object):
    def __init__(self, lines: List[Line]):
        self._lines = lines

    @property
    def lines(self) -> List[Line]:
        return self._lines
```

Now, you might also want to add other magic methods, like `.__eq__()`. This code is boilerplate. There's no business logic or any other functionality here: we're just copying data from one place to another.

A Case for `dataclasses`

Introduced into the standard library in Python 3.7, with a backport package for Python 3.6 on PyPI, the dataclasses module can help remove a lot of boilerplate for these types of classes where you're just storing data.

To convert the `Line` class above to a dataclass, convert all of the fields to class attributes and ensure they have type annotations:

Python

```
from dataclasses import dataclass

@dataclass
class Line(object):
    name_en: str
    name_jp: str
    color: str
    number: int
    sign: str
```

You can then create an instance of the `Line` type with the same arguments as before, with the same fields, and even `.__str__()`, `.__repr__()`, and `.__eq__()` are implemented:

```
Python >>>
>>> line = Line('Marunouchi', "丸ノ内線", "red", 4, "M")

>>> line.color
red

>>> line2 = Line('Marunouchi', "丸ノ内線", "red", 4, "M")

>>> line == line2
True
```

Dataclasses are a great way to reduce code with a single import that's already available in the standard library. For a full walkthrough, you can checkout The Ultimate Guide to Data Classes in Python 3.7.

Some `attrs` Use Cases

`attrs` is a third party package that's been around a lot longer than dataclasses. `attrs` has a lot more functionality, and it's available on Python 2.7 and 3.4+.

If you are using Python 3.5 or below, `attrs` is a great alternative to `dataclasses`. Also, it provides many more features.

The equivalent dataclasses example in `attrs` would look similar. Instead of using type annotations, the class attributes are assigned with a value from `attrib()`. This can take additional arguments, such as default values and callbacks for validating input:

```
Python
from attr import attrs, attrib

@attrs
class Line(object):
    name_en = attrib()
    name_jp = attrib()
    color = attrib()
    number = attrib()
    sign = attrib()
```

`attrs` can be a useful package for removing boilerplate code and input validation on data classes.

Conclusion

Now that you've learned how to identify and tackle complicated code, think back to the steps you can now take to make your application easier to change and manage:

- Start off by creating a baseline of your project using a tool like `wily`.
- Look at some of the metrics and start with the module that has the lowest maintainability index.
- Refactor that module using the safety provided in tests and the knowledge of tools like PyCharm and `rope`.

Once you follow these steps and the best practices in this article, you can do other exciting things to your application, like adding new features and improving performance.

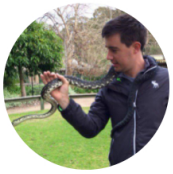
Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Anthony Shaw**



Anthony is an avid Pythonista and writes for Real Python. Anthony is a Fellow of the Python Software Foundation and member of the Open-Source Apache Foundation.

» More about Anthony

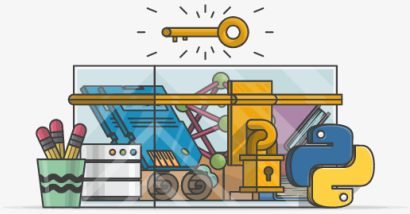
Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

 Aldren

 Geir Arne

 Joanna

Master Real-World Python Skills With Unlimited Access to Real Python



Join us and get access to hundreds of tutorials, hands-on video courses, and a community of expert Pythonistas:

Level Up Your Python Skills »

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally

won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Keep Learning

Related Tutorial Categories: best-practices intermediate