Python Package Structure

Dead Simple Python: Project Structure and Imports

#python #beginners #coding

Jason C. McDonald

Jan 15, 2019 *Updated on May 01, 2020* · 13 min read

Dead Simple Python (13 Part Series)

1 Introducing "Dead Simple Python"

2 Dead Simple Python: Virtual Environments and pip

... 11 more parts...

3 Dead Simple Python: Project Structure and Imports

4 Dead Simple Python: Data Typing and Immutability

5 Dead Simple Python: Classes

6 Dead Simple Python: Errors

7 Dead Simple Python: Loops and Iterators

8 Dead Simple Python: Iteration Power Tools

9 Dead Simple Python: List Comprehensions and Generator Expressions

10 Dead Simple Python (An Announcement)

11 Dead Simple Python: Generators and Coroutines

12 Dead Simple Python: Lambdas, Decorators, and Other Magic

13 Dead Simple Python: Working with Files

The worst part of tutorials is always their simplicity, isn't it? Rarely will you find one with more than one file, far more seldom with multiple directories.
I've found that **structuring a Python project** is one of the most often overlooked components of teaching the language. Worse, many developers get it wrong, stumbling through a jumble of common mistakes until they arrive at something that at least *works*.
Here's the good news: you don't have to be one of them!
In this installment of the Dead Simple Python series, we'll be exploring `import` statements, modules, packages, and how to fit everything together without tearing your hair out. We'll even touch on VCS, PEP, and the Zen of Python. Buckle up!

Setting Up The Repository

Before we delve into the actual project structure, let's address how this fits into our Version Control System [VCS]...starting with the fact you *need* a VCS! A few reasons are...

- Tracking every change you make,
- Figuring out exactly when you broke something,
- Being able to see old versions of your code,
- Backing up your code, and
- Collaborating with others.

You've got plenty of options available to you. **Git** is the most obvious, especially if you don't know what else to use. You can host your Git repository for free on GitHub, GitLab, Bitbucket, or Gitote, among others. If you want something other than Git, there's dozens of other options, including Mercurial, Bazaar, Subversion (although if you use that last one, you'll probably be considered something of a dinosaur by your peers.)
I'll be quietly assuming you're using Git for the rest of this guide, as that's what I use exclusively.
Once you've created your repository and cloned a *local copy* to your computer, you can begin setting up your project. At minimum, you'll need to create the following:

- `README.md`: A description of your project and its goals.
- `LICENSE.md`: Your project's license, if it's open source. (See opensource.org for more information about selecting one.)
- `.gitignore`: A special file that tells Git what files and directories to ignore. (If you're using another VCS, this file has a different name. Look it up.)
- A directory with the name of your project.

That's right...**our Python code files actually belong in a separate subdirectory!** This is very important, as our repository's root directory is going to get mighty cluttered with build files, packaging scripts, virtual environments, and all manner of other things that aren't actually part of the source code.
Just for the sake of example, we'll call our fictional project `awesomething`.

PEP 8 and Naming

Python style is governed largely by a set of documents called **Python Enhancement Proposals**, abbreviated **PEP**. Not all PEPs are actually adopted, of course - that's why they're called "Proposals" - but some are. You can browse the master PEP index on the official Python website. This index is formally referred to as PEP 0.
Right now, we're mainly concerned with **PEP 8**, first authored by the Python language creator Guido van Rossum back in 2001. It is the document which officially outlines the coding style all Python developers should generally follow. Keep it under your pillow! Learn it, follow it, encourage others to do the same.
(Side Note: PEP 8 makes the point that there are always exceptions to style rules. It's a *guide*, not a *mandate*.)
Right now, we're chiefly concerned with the section entitled "Package and Module Names"...
Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.
We'll get to what exactly *modules* and *packages* are in a moment, but for now, understand that **modules are named by filenames**, and **packages are named by their directory name**.
In other words, **filenames should be all lowercase, with underscores if that improves readability.** Similarly, **directory names should be all lowercase, without underscores if at all avoidable**. To put that another way...

- Do This: `awesomething/data/load_settings.py`
- NOT This: `awesomething/Data/LoadSettings.py`

I know, I know, long-winded way to make a point, but at least I put a little PEP in your step. (*Hello? Is this thing on?*)

Packages and Modules

This is going to feel anticlimactic, but here are those promised definitions:

Any Python (`.py`) file is a *module*, and a bunch of modules in a directory is a *package*.

Well...almost. There's one other thing you have to do to a directory to make it a package, and that's to stick a file called `__init__.py` into it. You actually don't have to put anything *into* that file. It just has to be there.

There is other cool stuff you can do with `__init__.py`, but it's beyond the scope of this guide, so go read the docs to learn more.

If you *do* forget `__init__.py` in your package, it's going to do something much weirder than just failing, because that makes it an **implicit namespace package**. There's some nifty things you can do with that special type of package, but I'm not going into that here. As usual, you can learn more by reading the documentation: PEP 420: Implicit Namespace Packages.

So, if we look at our project structure, `awesomething` is actually a package, and it can contain other packages. Thus, we might call `awesomething` our *top-level package*, and all the packages underneath its *subpackages*. This is going to be really important once we get to importing stuff.

Let's look at one a snapshot of my real-world projects, `omission`, to get an idea of how we're structuring stuff...

```
omission-git
├── LICENSE.md
├── omission
│   ├── app.py
│   ├── common
│   │   ├── classproperty.py
│   │   ├── constants.py
│   │   ├── game_enums.py
│   │   └── __init__.py
│   ├── data
│   │   ├── data_loader.py
│   │   ├── game_round_settings.py
│   │   ├── __init__.py
│   │   ├── scoreboard.py
│   │   └── settings.py
│   ├── game
│   │   ├── content_loader.py
│   │   ├── game_item.py
│   │   ├── game_round.py
│   │   ├── __init__.py
│   │   └── timer.py
│   ├── __init__.py
│   ├── __main__.py
│   ├── resources
│   └── tests
│       ├── __init__.py
│       ├── test_game_item.py
│       ├── test_game_round_settings.py
│       ├── test_scoreboard.py
│       ├── test_settings.py
│       ├── test_test.py
│       └── test_timer.py
├── pylintrc
├── README.md
└── .gitignore
```

(In case you're wondering, I used the UNIX program `tree` to make that little diagram above.)

You'll see that I have one top-level package called `omission`, with four sub-packages: `common`, `data`, `game`, and `tests`. I also have the directory `resources`, but that only contains game audio, images, etc. (omitted here for brevity). `resources` is NOT a package, as it doesn't contain an `__init__.py`.

I also have another special file in my top-level package: `__main__.py`. This is the file that is run when we execute our top-level package directly via `python -m omission`. We'll talk about what goes in that `__main__.py` in a bit.

How import Works

If you've written any meaningful Python code before, you're almost certainly familiar with the `import` statement. For example...

```
import re
```

It is helpful to know that, when we import a module, we are actually running it. This means that any `import` statements in the module are also being run.

For example, `re.py` has several import statements of its own, which are executed when we say `import re`. That doesn't mean they're available to the file we imported `re` *from*, but it does mean those files have to exist. If (for some unlikely reason) `enum.py` got deleted on your environment, and you ran `import re`, it would fail with an error...

Traceback (most recent call last):
File "weird.py", line 1, in
import re
File "re.py", line 122, in
import enum
ModuleNotFoundError: No module named 'enum'
Naturally, reading that, you might get a bit confused. I've had people ask me why the outer module (in this example, `re`) can't be found. Others have wondered why the inner module (`enum` here) is being imported at all, since they didn't ask for it directly in their code. The answer is simple: we imported `re`, and that imports `enum`.
Of course, the above scenario is fictional: `import enum` and `import re` are never going to fail under normal circumstances, because both modules are part of Python's core library. It's just a silly example. ;)

Import Dos and Don'ts

There are actually a number of ways of importing, but most of them should rarely, if ever be used.
For all of the examples below, we'll imagine that we have a file called `smart_door.py`:

```
# smart_door.py
def close():
    print("Ahhhhhhhhhhhh.")

def open():
    print("Thank you for making a simple door very happy.")
```

Just for example, we will run the rest of the code in this section in the Python interactive shell, from the same directory as `smart_door.py`.
If we want to run the function `open()`, we have to first import the module `smart_door`. The easiest way to do this is...

```
import smart_door
smart_door.open()
smart_door.close()
```

We would actually say that `smart_door` is the **namespace** of `open()` and `close()`. Python developers really like namespaces, because they make it obvious where functions and whatnot are coming from.
(By the way, don't confuse *namespace* with *implicit namespace package*. They're two different things.)
The **Zen of Python**, also known as PEP 20, defines the philosophy behind the Python language. The last line has a statement that addresses this:
Namespaces are one honking great idea -- let's do more of those!
At a certain point, however, namespaces can become a pain, especially with nested packages. `foo.bar.baz.whatever.doThing()` is just ugly. Thankfully, we do have a way around having to use the namespace *every time* we call the function.
If we want to be able to use the `open()` function without constantly having to precede it with its module name, we can do this instead...

```
from smart_door import open
open()
```

Note, however, that neither `close()` nor `smart_door.close()` will not work in that last scenario, because we didn't import the function outright. To use it, we'd have to change the code to this...

```
from smart_door import open, close
open()
close()
```

In that terrible nested-package nightmare earlier, we can now say `from foo.bar.baz.whatever import doThing`, and then just use `doThing()` directly. Alternatively, if we want a LITTLE bit of namespace, we can say `from foo.bar.baz import whatever`, and say `whatever.doThing()`.
The `import` system is deliciously flexible like that.
Before long, though, you'll probably find yourself saying "But I have hundreds of functions in my module, and I want to use them all!" This is the point at which many developers go off the rails, by doing this...

```
from smart_door import *
```

This is very, very bad! Simply put, it imports everything in the module directly, and that's a problem. Imagine the following code...

```
from smart_door import *
from gzip import *
open()
```

What do you suppose will happen? The answer is, `gzip.open()` will be the function that gets called, since that's the last version of `open()` that was imported, and thus defined, in our code. `smart_door.open()` has been **shadowed** - we can't call it as `open()`, which means we effectively can't call it at all.
Of course, since we usually don't know, or at least don't remember, *every single* function, class, and variable in every module that gets imported, we can easily wind up with a whole lot of messes.
The *Zen of Python* addresses this scenario as well...
Explicit is better than implicit.
You should never have to *guess* where a function or variable is coming from. Somewhere in the file should be code that *explicitly* tells us where it comes from. The first two scenarios demonstrate that.
I should also mention that the earlier `foo.bar.baz.whatever.doThing()` scenario is something Python developers do NOT like to see.
Also from the *Zen of Python*...
Flat is better than nested.
Some nesting of packages is okay, but when your project starts looking like an elaborate set of Matryoshka dolls, you've done something wrong. Organize your modules into packages, but keep it reasonably simple.

Importing Within Your Project

That project file structure we created earlier is about to come in *very handy*. Recall my `omission` project...

```
omission-git
├── LICENSE.md
├── omission
│   ├── app.py
│   ├── common
│   │   ├── classproperty.py
│   │   ├── constants.py
│   │   ├── game_enums.py
│   │   └── __init__.py
│   ├── data
│   │   ├── data_loader.py
│   │   ├── game_round_settings.py
│   │   ├── __init__.py
│   │   ├── scoreboard.py
│   │   └── settings.py
│   ├── game
│   │   ├── content_loader.py
│   │   ├── game_item.py
│   │   ├── game_round.py
│   │   ├── __init__.py
│   │   └── timer.py
│   ├── __init__.py
│   ├── __main__.py
│   ├── resources
│   └── tests
│       ├── __init__.py
│       ├── test_game_item.py
│       ├── test_game_round_settings.py
│       ├── test_scoreboard.py
│       ├── test_settings.py
│       ├── test_test.py
│       └── test_timer.py
├── pylintrc
├── README.md
└── .gitignore
```

In my `game_round_settings` module, defined by `omission/data/game_round_settings.py`, I want to use my `GameMode` class. That class is defined in `omission/common/game_enums.py`. How do I get to it?
Because I defined `omission` as a package, and organized my modules into subpackages, it's actually pretty easy. In `game_round_settings.py`, I say...

```
from omission.common.game_enums import GameMode
```

This is called an **absolute import**. It starts at the top-level package, `omission`, and walks down into the `common` package, where it looks for `game_enums.py`.
Some developers come to me with import statements more like `from common.game_enums import GameMode`, and wonder why it doesn't work. Simply put, the `data` package (where `game_round_settings.py` lives) has no knowledge of its sibling packages.
It does, however, know about its parents. Because of this, Python has something called **relative imports** that lets us do the same thing like this instead...

```
from ..common.game_enums import GameMode
```

The `..` means "this package's direct parent package", which in this case, is `omission`. So, the import steps back one level, walks down into `common`, and finds `game_enums.py`.
There's a lot of debate about whether to use absolute or relative imports. Personally, I prefer to use absolute imports whenever possible, because it makes the code a lot more readable. You can make up your own mind, however. The only important part is that the result is *obvious* - there should be no mystery where anything comes from.
(Continued Reading: Real Python - Absolute vs Relative Imports in Python
There is one other lurking gotcha here! In `omission/data/settings.py`, I have this line:

```
from omission.data.game_round_settings import GameRoundSettings
```

Surely, since both these modules are in the same package, we should be able to just say `from game_round_settings import GameRoundSettings`, right?
Wrong! It will actually fail to locate `game_round_settings.py`. This is because we are running the top-level package `omission`, which means the **search path** (where Python looks for modules, and in what order) works differently.
However, we can use a relative import instead:

```
from .game_round_settings import GameRoundSettings
```

In that case, the single `.` means "this package".
If you're familiar with the typical UNIX file system, this should start to make sense. `..` means "back one level", and `.` means "the current location". Of course, Python takes it one step further: `...` means "back two levels", `....` is "back three levels", and so forth.
However, keep in mind that those "levels" aren't just plain directories, here. They're packages. If you have two distinct packages in a plain directory that is NOT a package, you can't use relative imports to jump from one to another. You'll have to work with the Python search path for that, and that's beyond the scope of this guide. (See the docs at the end of this article.)

`__main__.py`

Remember when I mentioned creating a `__main__.py` in our top-level package? That is a special file that is executed when we run the package directly with Python. My `omission` package can be run from the root of my repository with `python -m omission`.
Here's the contents of that file:

```
from omission import app

if __name__ == '__main__':
    app.run()
```

Yep, that's actually it! I'm importing my module `app` from the top-level package `omission`.
Remember, I could also have said `from . import app` instead. Alternatively, if I wanted to just say `run()` instead of `app.run()`, I could have done `from omission.app import run` or `from .app import run`. In the end, it doesn't make much technical difference HOW I do that import, so long as the code is readable.
(Side Note: We could debate whether it's logical for me to have a separate `app.py` for my main `run()` function, but I have my reasons...and they're beyond the scope of this guide.)
The part that confuses most folks at first is the whole `if __name__ == '__main__'` statement. Python doesn't have much **boilerplate** - code that must be used pretty universally with little to no modification - but this is one of those rare bits.
`__name__` is a special string attribute of every Python module. If I were to stick the line `print(__name__)` at the top of `omission/data/settings.py`, when that module got imported (and thus run), we'd see "omission.data.settings" printed out.
When a module is run directly via `python -m some_module`, that module is assigned a special value of `__name__`: "**main**".
Thus, `if __name__ == '__main__':` is actually checking if the module is being executed as the *main* module. If it is, it runs the code under the conditional.
You can see this in action another way. If I added the following to the bottom of `app.py`...

```
if __name__ == '__main__':
    run()
```

...I can then execute that module directly via `python -m omission.app`, and the results are the same as `python -m omission`. Now `__main__.py` is being ignored altogether, and the `__name__` of `omission/app.py` is `"__main__.py"`.
Meanwhile, if I just run `python -m omission`, that special code in `app.py` is ignored, since its `__name__` is now `omission.app` again.
See how that works?

Wrapping Up

Let's review.

- Every project should use a VCS, such as Git. There are plenty of options to choose from.
- Every Python code file (`.py`) is a **module**.

- Organize your modules into **packages**. Each package must contain a special `__init__.py` file.
- Your project should generally consist of one top-level package, usually containing sub-packages. That top-level package usually shares the name of your project, and exists as a directory in the root of your project's repository.
- **NEVER EVER EVER** use `*` in an import statement. Before you entertain a possible exception, the Zen of Python points out "Special cases aren't special enough to break the rules."
- Use absolute or relative imports to refer to other modules in your project.
- Executable projects should have a `__main__.py` in the top-level package. Then, you can directly execute that package with `python -m myproject`.

Of course, there are a lot more advanced concepts and tricks we can employ in structuring a Python project, but we won't be discussing that here. I highly recommend reading the docs:

- Python Reference: the import system
- Python Tutorials: Modules
- PEP 8: Style Guide for Python
- PEP 20: The Zen of Python
- PEP 240: Implicit Namespace Packages

Thank you to `grym`, `deniska` (Freenode IRC `#python`), @cbrintnall , and @rhymes (Dev) for suggested revisions.

Dead Simple Python (13 Part Series)

1 Introducing "Dead Simple Python"

2 Dead Simple Python: Virtual Environments and pip

... 11 more parts...

3 Dead Simple Python: Project Structure and Imports

4 Dead Simple Python: Data Typing and Immutability

5 Dead Simple Python: Classes

6 Dead Simple Python: Errors

7 Dead Simple Python: Loops and Iterators

8 Dead Simple Python: Iteration Power Tools

9 Dead Simple Python: List Comprehensions and Generator Expressions

10 Dead Simple Python (An Announcement)

11 Dead Simple Python: Generators and Coroutines

12 Dead Simple Python: Lambdas, Decorators, and Other Magic

13 Dead Simple Python: Working with Files

Posted on Jan 15 '19 by:

Jason C. McDonald
@codemouse92
Author | Hacker | Speaker | Time Lord