✨ Member-only story

How to Package Your Python Code

Learn how Python code should be packaged for PyPI

James Briggs · Follow

Published in Towards Data Science · 7 min read · Apr 2, 2021

Photo by Lawless Capture on Unsplash

The most powerful feature of Python is its community. Almost every use-case out there has a package built specifically for it.

Need to send mobile/email alerts? `pip install knockknock` — Build ML apps? `pip install streamlit` — Bored of your terminal? `pip install colorama` — It's too easy!

I know this is obvious, but those libraries didn't magically appear. For each package, there is a person, or many persons — that actively developed and deployed that package.

Every single one.

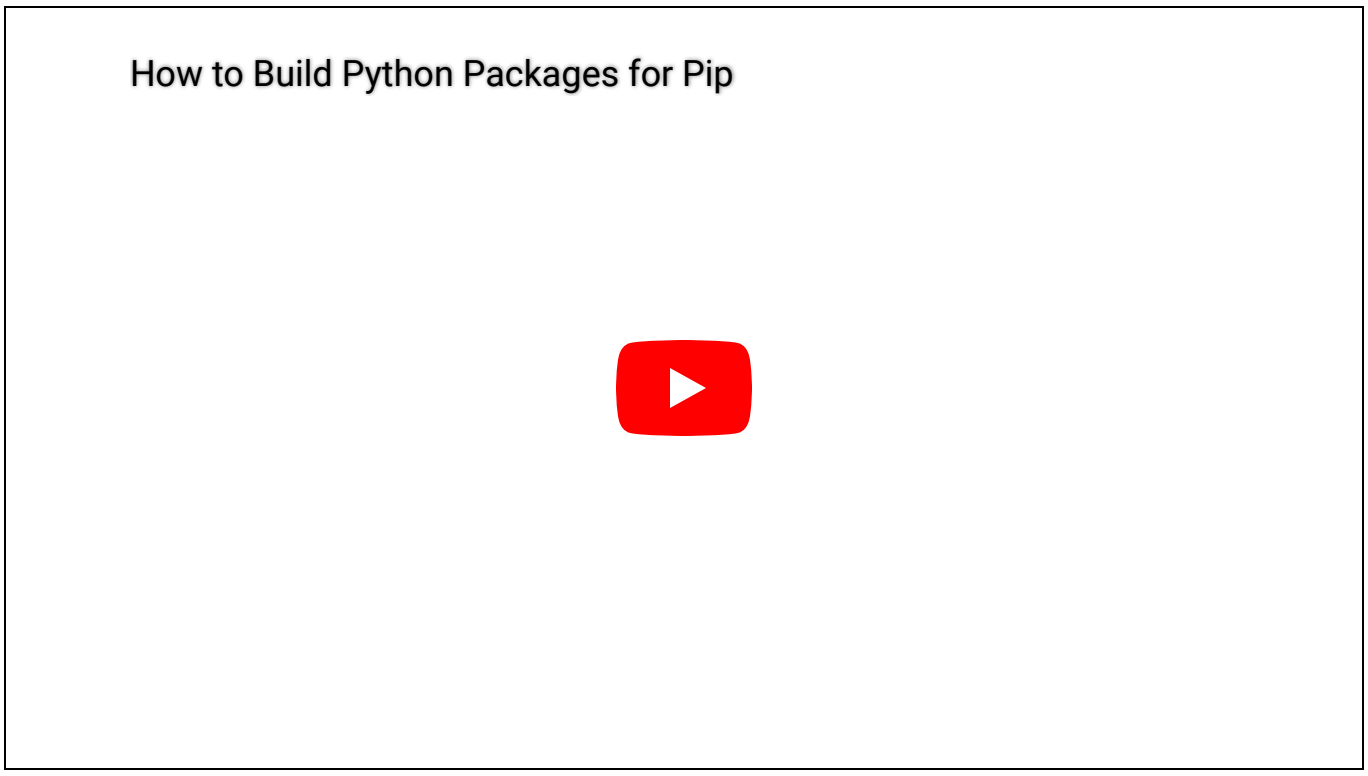
All 300K+ of them.

That is why Python is Python, the level of support is phenomenal — *mindblowing*.

In this article, we will learn how to build our own packages. And add them to the Python Package Index (PyPI). Afterward, we will be able to install our packages using `pip install`! Let's get started.

What to Build?

The first step is to figure out what we should build. Once we have an idea, head over to PyPI and search for the same package, does it exist? If so, does your idea offer anything new or better?



Once we've got something unique, we need to double-check that our package name isn't already taken.

In this article, we're going to be building a package called `aesthetic_ascii` — which will produce random synthwave-inspired ASCII art. A fairly unique idea, although perhaps not *too* useful.

Directory Setup

We need to create a new directory that will contain our package and act as our package name (*aesthetic_ascii*).

Inside this directory, we have our `__init__.py` file which initializes our directory as a Python module— and becomes the top-level script for the module.

We can add sub-modules by add subdirectories inside the package directory and adding `__init__.py` files inside those too.

For example, we know that we can write `from os import path`. The directory structure for this would look something like this:

```
os/
    LICENSE
    MANIFEST.in
    pyproject.toml
    README.md
    setup.cfg
    ...
    os/
        __init__.py
        ...
        path/
```

```
__init.py__
...
```

We'll cover the other files in here soon. But first, we need to understand what `__init__.py` is actually doing.

__init__.py

At its core `__init__.py`, marks a directory as a Python package. So, we include the top-level `__init__.py` file which marks the directory as our 'package'.

We then include the actual code files (or *modules*) alongside our `__init__.py` files. So, if we had this structure:

```
aesthetic_ascii/
    *all config files*
    aesthetic_ascii/
        __init__.py
        synthesize.py
```



```
from aesthetic_ascii import synthesize
```

We'll use this structure throughout the article.

Configuration Files

setup.cfg

Alongside our code files, we need several configuration files. The most important for configuring our package is the `setup.cfg` file.

This file is critical, it is our *setup configuration* file. During the `pip install`, this file tells *pip* how to install our package.

To declare that we are using **setuptools** to package our project, we first create a `pyproject.toml` file in our top-level directory containing:

```
[build-system]
requires = [
    "setuptools>=54",
    "wheel"
]
build-backend = "setuptools.build_meta"
```

pyproject.toml hosted with ❤ by **GitHub** **view raw**

And then we create our `setup.cfg` file. For *aesthetic_ascii*, it looks like this:

```
[metadata]
name = aesthetic_ascii
version = 0.0.1
author = James Briggs
author_email = jamescalam94@gmail.com
description = For generating A E S T H E T I C ascii art
long_description = file: README.md
long_description_content_type = text/markdown
url = https://github.com/jamescalam/aesthetic_ascii
classifiers =
    Programming Language :: Python :: 3
    License :: OSI Approved :: MIT License
    Operating System :: OS Independent

[options]
packages = find:
python_requires = >=3.7
include_package_data = True
```

setup.cfg hosted with ❤ by GitHub view raw

There's a lot going on here, but each item is quite descriptive — so I won't describe each, however, there are a few that are less clear.

- **long_description** — the parameter name tells us what this is, and we can provide different file formats here. Many packages stick with markdown though, and that is what we will be using — as specified in **long_description_content_type**.
- **classifiers** — a set of identifiers that PyPI will use to categorize our package, we can find a full list of valid classifiers here.
- **packages** — a list of dependency packages required by our package to run.
- **python_requires** — Python version requirements for the library.

- **include_package_data** — whether to include additional files as defined in `MANIFEST.in` — more on this later.

LICENSE

Next up is our license, a very important part of the package — even if you don't care who uses it, why they're using it, and whether they're making money from it.

```
MIT License

Copyright (c) 2021 James Calam Briggs

Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
copies of the Software, and to permit persons to whom the Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```

LICENSE hosted with ❤ by **GitHub** **view raw**

We include a license to remove any shred of doubt for users of the package. Choosing a license can be very easy, just copy and paste the most suitable text from here — and remember to update any custom parts!

README.md

This file acts as the `long_description` of our package — which we set up in the `setup.cfg` file.

Aesthetic ASCII

Generate **A E S T H E T I C** ASCII art.

README.md hosted with ❤ by **GitHub** **view raw**

Additionally, back in our `setup.cfg` — we specified in `long_description_content_type` that we would be using a `text/markdown` file — so, we use a markdown text file!

Adding Resources

We've set up our configuration files, but we're missing one thing — our images that will be used when generating random ASCII art.

These images could be any type of file that our package relies on. From Python 3.7 we deal with this by using the `importlib.resources` module.

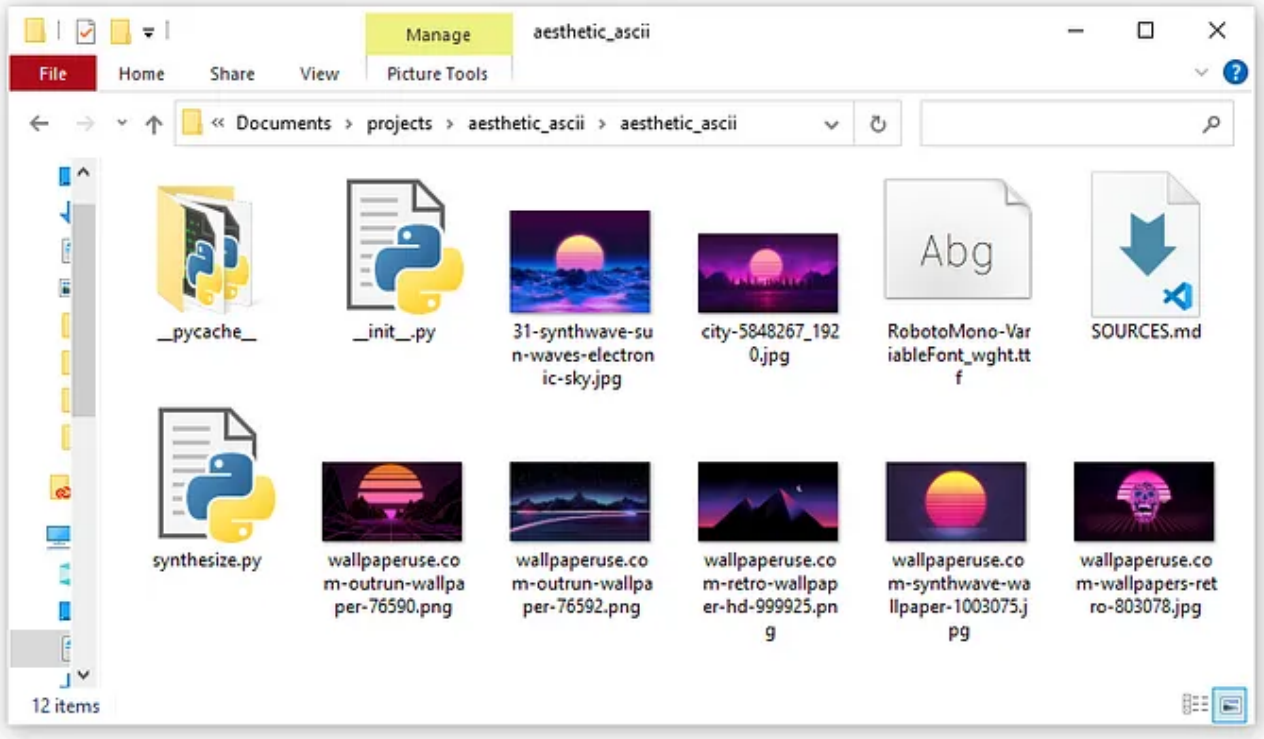
First, we need to import this into our code and update the part of our code reading these files to use `importlib.resources`:

```python
from importlib import resources
...
# read text file
with resources.open_text('aesthetic_ascii', 'somefile.txt') as fp:
    txt = fp.read()
...
# or binary
with resources.open_binary('aesthetic_ascii', 'somepic.png') as fp:
    img = fp.read()
```

importlib_usage.py hosted with ❤ by **GitHub** **view raw**

The first argument of both functions refers to the directory that the files are stored within. Additionally, any resources **must** be stored in a directory that contains a `__init__.py` file.

So, in our case, we need to store all image files inside the `aesthetic_ascii` **sub**directory:



The **aesthetic_ascii/aesthetic_ascii** directory includes all of our image files and our font file — alongside the **__init__.py** and **synthesize.py** module

Once this is all in place, we must make sure our image files will be included in the package — which we do via the `MANIFEST.in` file:

```
include aesthetic_ascii/*.png
include aesthetic_ascii/*.jpg
include aesthetic_ascii/*.ttf
```

MANIFEST.in hosted with ❤ by GitHub view raw

And then we make sure the `MANIFEST.in` file is added to `setup.cfg`:

```
...
[options]
include_package_data = True
...
```

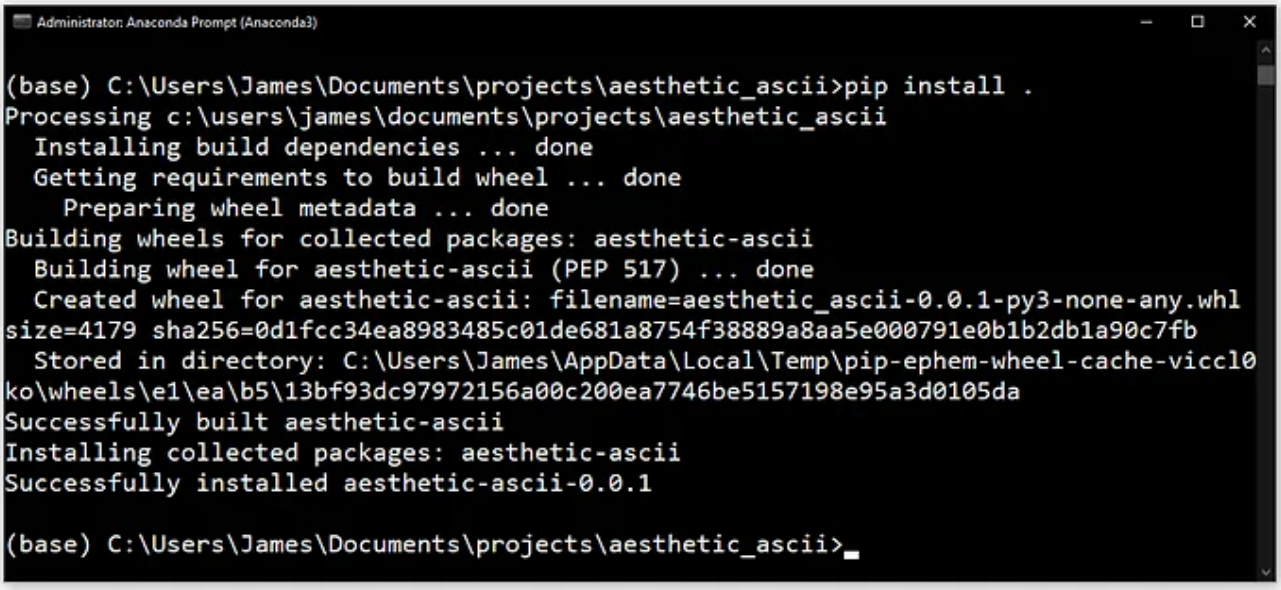
And we're good to go!

Local Install

Before uploading our package to PyPI we can confirm that our package can be installed via `pip install` by navigating to our package directory and entering:

```
pip install .
```

This should then install our package like any other package install via `pip`:



Screenshot of the local **pip install .** test

And we can then `import` our package like we would any other:

```
from aesthetic_ascii import synthesize

drive = synthesize.Drive()

drive.generate(dark_mode=True)

drive.to_png('test.png')
```

Example usage of the **aesthetic_ascii** package, now installed

Build

Once we've written our code files, setup configuration, and tested the install — we're ready to build our package distribution.

The build process creates a new directory `dist` which will contain a `.tar.gz` and `.whl` file — this is what we need to publish our package to PyPI.

To build our `dist` files, we use a tool creatively named `build`. First, we `pip install build`, then, while in our package directory — type:

```
python -m build
```

Once this is complete, we should find a new `/dist` directory inside our package directory.

Publish to TestPyPI

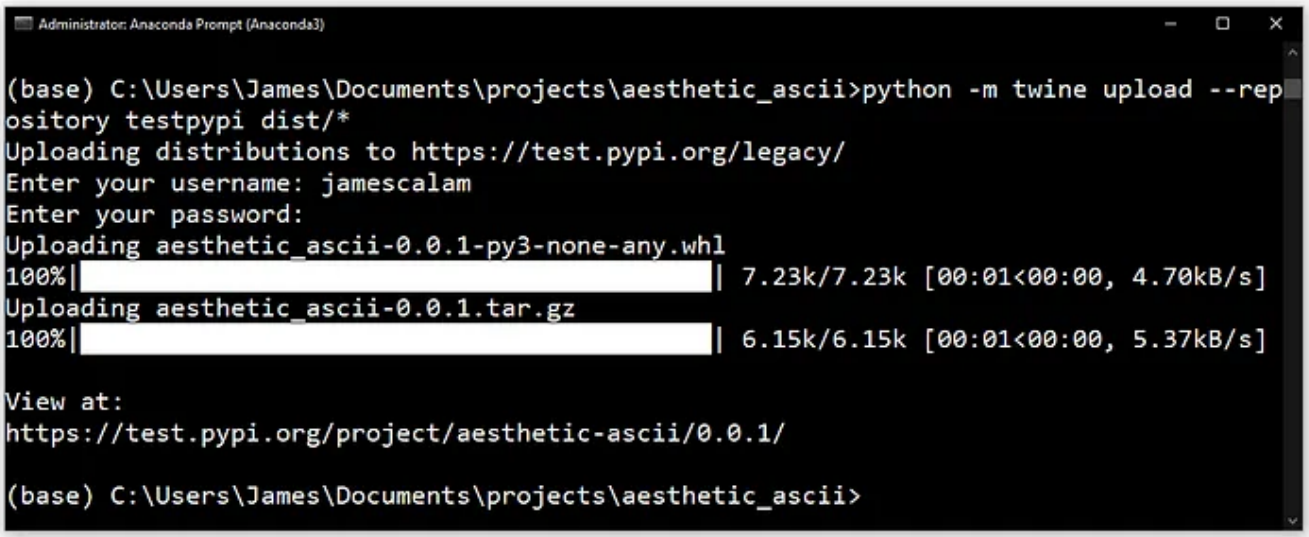
Finally, we are ready to publish our new Python package! Again, we make use of another package called `twine`. We install this with:

```
pip install twine
```

Once installed, we upload to *TestPyPI* — a 'test' version of PyPI so that we can double-check that we have set everything up correctly. We do this by typing:

```
python -m twine upload --repository testpypi dist/*
```

At this point, we'll need to login to TestPyPI — if you don't have an account, sign up for one here. If everything is set up correctly, our package will be uploaded:

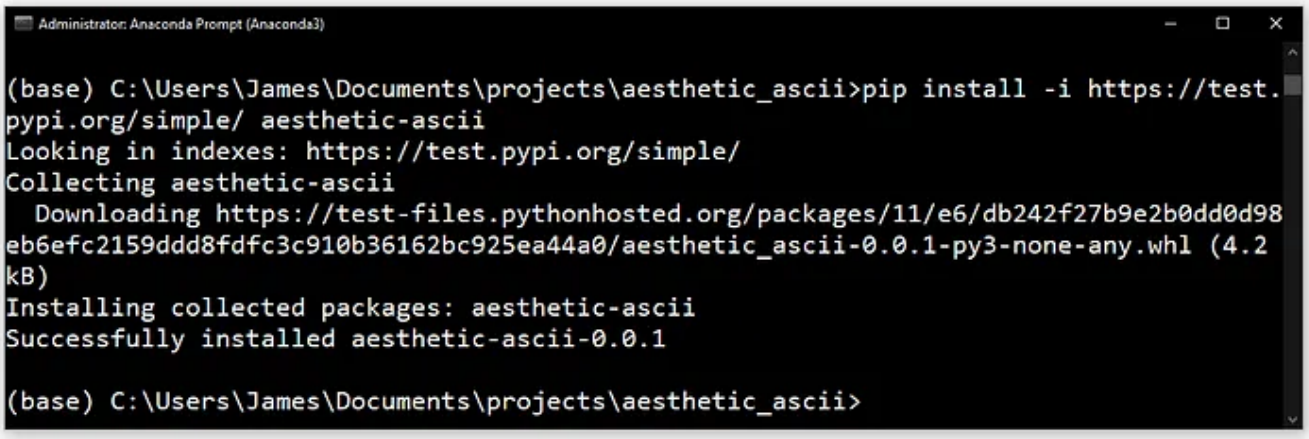


Successful upload process for **aesthetic_ascii** on test PyPI

Now, we can test that our new package works through another `pip install` — but this time from TestPyPI:

```
pip install -i https://test.pypi.org/simple/ aesthetic-ascii
```

(If you find that the package is already installed — just `pip uninstall aesthetic-ascii`).



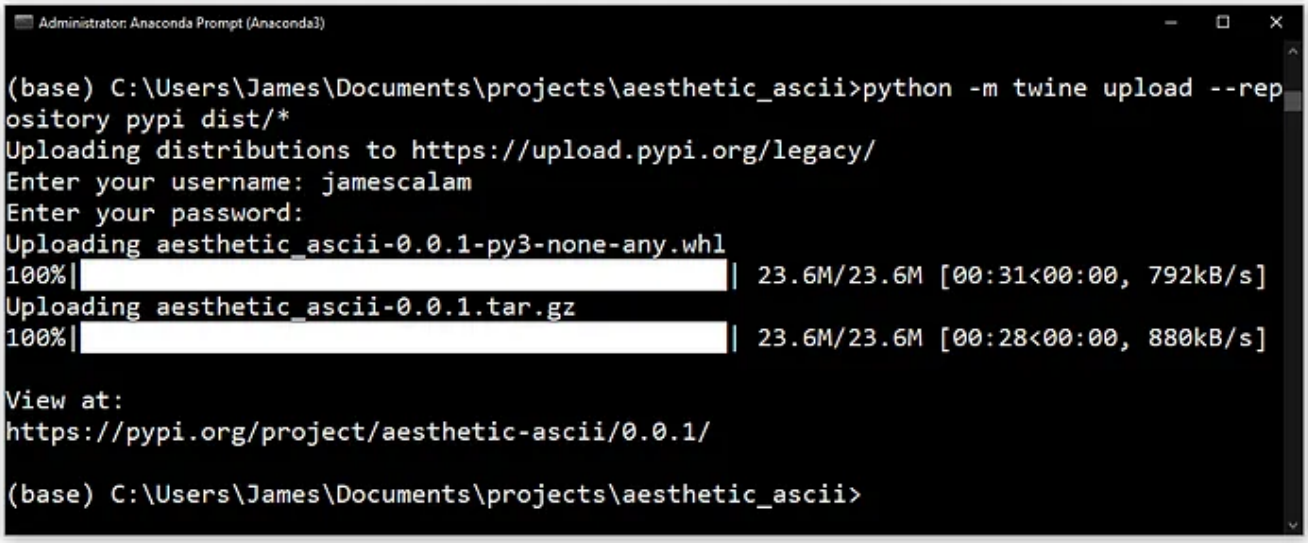
Installing **aesthetic-ascii** via TestPyPI

PyPI

Once we've confirmed that the package works — we take our final step, publishing to PyPI. Again, you'll need to register here.

Next, we upload our package to PyPI with:

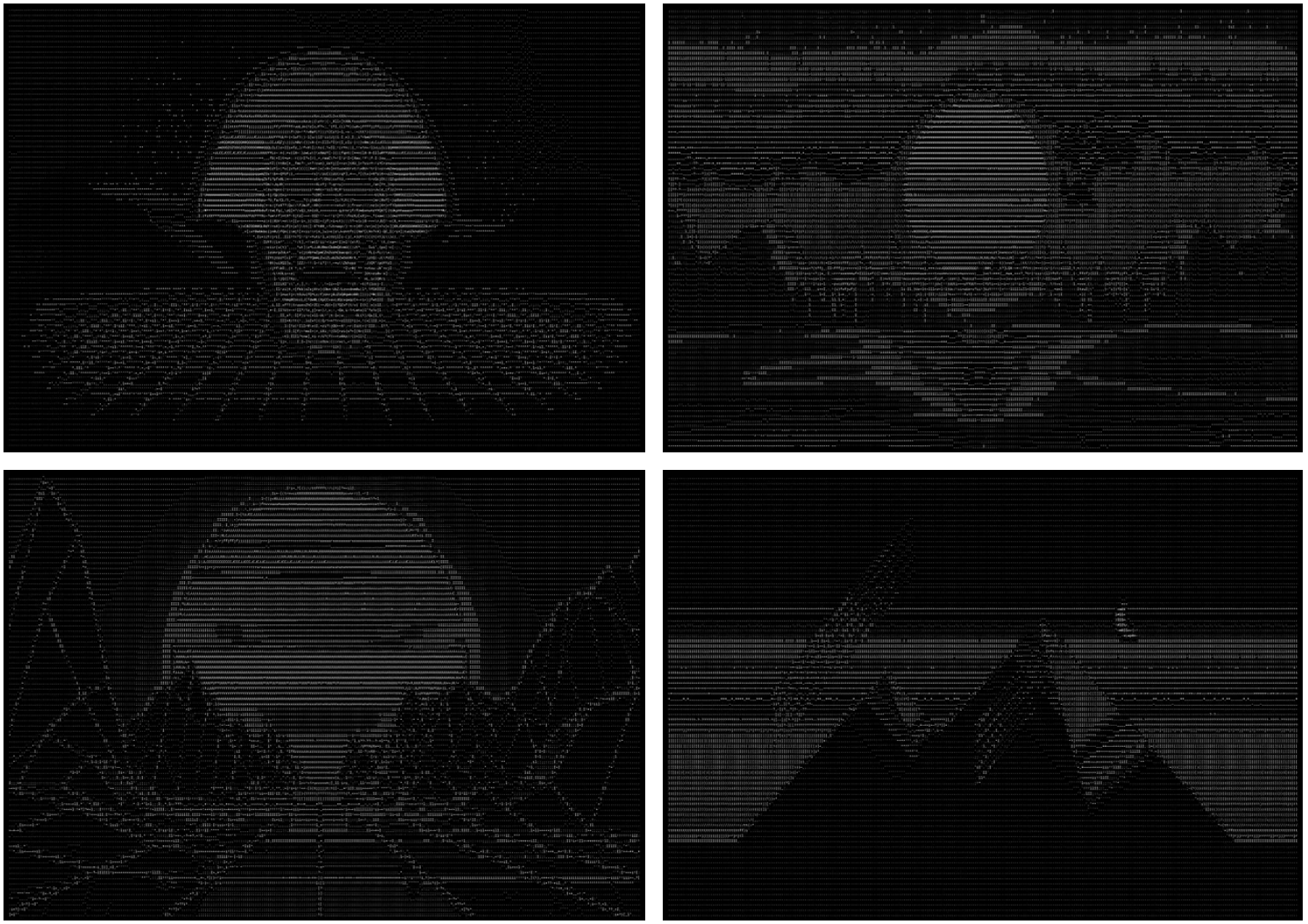
```
python -m twine upload --repository pypi dist/*
```



Uploading our package to PyPI

And we're done!

That's our first Python package deployment — it's surprisingly straightforward. We can go ahead and `pip install aesthetic_ascii` to use the package.



A few of our **AESTHETIC** ascii masterpieces

The full code including setup/configuration files can be found on GitHub here.

I hope you've enjoyed the article. Let me know if you have any questions or suggestions via Twitter or in the comments below. If you're interested in more content like this, I post on YouTube too.

Thanks for reading!

Further Reading

How To Package Your Python Code, Python Packaging

Packaging Projects, Python Packaging

Kite, Turn any image into ASCII art! (Easy Python PIL Tutorial), YouTube

P. Bourke, Character representation of grey scale images (1997), paulbourke.net

**All images are by the author except where stated otherwise*

Python Programming Software Development Technology Software Engineering

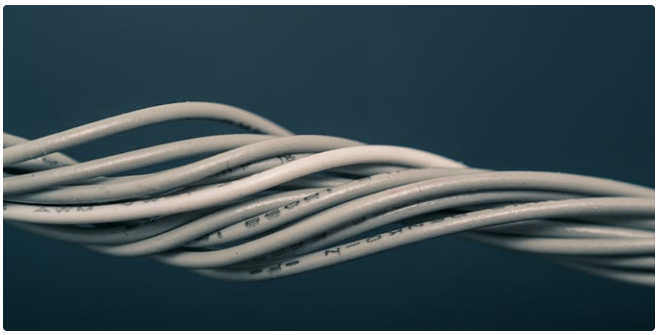
Written by James Briggs

Follow

11.1K Followers · Writer for Towards Data Science

Freelance ML engineer learning and writing about everything. I post a lot on YT
https://www.youtube.com/c/jamesbriggs

More from James Briggs and Towards Data Science



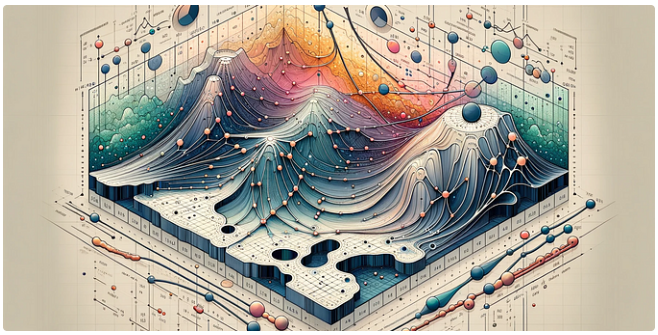
James Briggs in Towards Data Science

The Right Way to Build an API with Python

All you need to know on API development in Flask

✨ · 7 min read · Sep 11, 2020

1.2K 13



Cristian Leo in Towards Data Science

The Math behind Adam Optimizer

Why is Adam the most popular optimizer in Deep Learning? Let's understand it by diving...

16 min read · Jan 30, 2024

1.5K 9

Sheila Teo in Towards Data Science

How I Won Singapore's GPT-4 Prompt Engineering Competition

A deep dive into the strategies I learned for harnessing the power of Large Language...

✦ · 23 min read · Dec 28, 2023

11.2K 126

James Briggs in Towards Data Science

How to Train a BERT Model From Scratch

Meet BERT's Italian cousin, FiliBERTo

✦ · 7 min read · Jul 6, 2021

460 11

See all from James Briggs

See all from Towards Data Science

Recommended from Medium

Mike Shakhomirov in Towards Data Science

Pandas for Data Engineers

Advanced techniques to process and load data efficiently

9 min read · 3 days ago

317 5

Anmol Tomar in CodeX

Say Goodbye to Loops in Python, and Welcome Vectorization!

Use Vectorization—a super-fast alternative to loops in Python

5 min read · Dec 27, 2023

4.1K 46

Lists

General Coding Knowledge

20 stories · 912 saves

Coding & Development

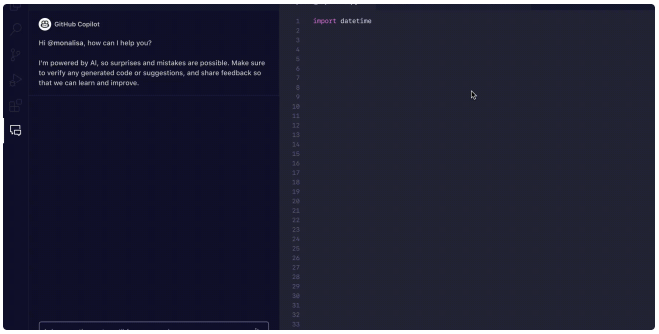
11 stories · 445 saves

Stories to Help You Grow as a Software Developer

19 stories · 805 saves

ChatGPT

21 stories · 460 saves

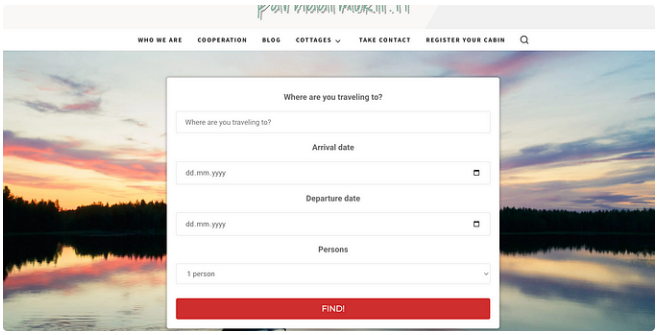


Jacob Bennett in Level Up Coding

The 5 paid subscriptions I actually use in 2024 as a software engineer

Tools I use that are cheaper than Netflix

5 min read · Jan 4, 2024



Artturi Jalli

I Built an App in 6 Hours that Makes $1,500/Mo

Copy my strategy!

3 min read · Jan 23, 2024

8.5K 99

Vaishnav Manoj in DataX Journal

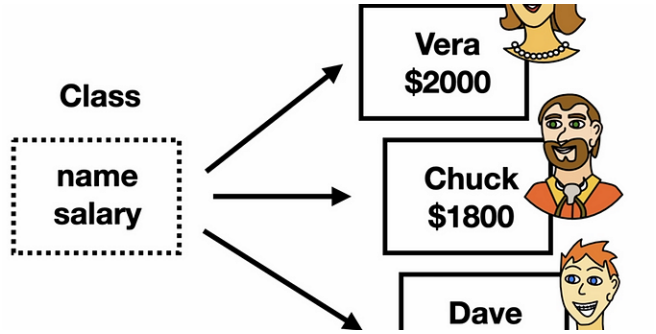
JSON is incredibly slow: Here's What's Faster!

Unlocking the Need for Speed: Optimizing JSON Performance for Lightning-Fast Apps...

16 min read · Sep 28, 2023

13.1K 155

6K 88



Aserdargun

Advanced OOP in Python

Classes and objects: Class instantiation, self, data attributes, UML, methods, __str__,...

72 min read · Jan 14, 2024

686 7

See more recommendations