

 Meta

 omnilib  n7cmdr  n7.gg

click



[omnilib](#) [n7cmdr](#) [n7.gg](#)

Hello, I'm John Reese, and I'm an engineer at Meta. I work on our internal Python Foundation team, building infrastructure and tooling for Python users both inside and outside of the company. That work includes open source projects like `μsort`, an import sorter for Python that is safe and predictable, even on large scale codebases.



Outside of Meta, I also founded the Omnilib Project, a group of MIT licensed libraries with an inclusive code of conduct. This is the home for multiple packages, ranging from asyncio libraries like aiosqlite and aioitertools, to developer tooling like attribution and ufmt. I've been working on open source projects for well over a decade now, and there's one phrase that has always stuck with me:

time is the only resource you can't buy

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

Time is the only resource you can't buy. We only have so much time in a day or week, and we all want to spend more of that time doing things we like, and less of that time doing things we don't. I've spent countless hours working on dozens of open source projects, so it's been worth my time to streamline my development workflow over the years.

Open Source on Easy Mode

A light exploration of tooling, automation, and dealing with humans

omnilib n7cmdr n7.gg

That's why I'm here today, to share what I can about how to min-max our open source experience: to get the maximum impact from our time with a minimum amount of effort. The more time we save on unimportant or boring parts of the process, the more time we can spend on things that actually matter: building features, reviewing pull requests, spending time with our loved ones, and playing video games.

Open Source on Easy Mode

A light exploration of tooling, automation, and dealing with humans

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

The way we'll do that today is by looking at a number of different tools and best practices that together build a cohesive, end-to-end developer experience, where each individual piece provides value by reducing the overall level of effort needed to maintain an open source project. There's a lot to cover, and we're going to go fast, but I'll have a github link for everything I talk about at the end of the presentation.

Foundations

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

So let's jump right in with the foundational pieces that every project needs in order to function within the greater Python ecosystem.

goal: well structured metadata

 omnilib  n7cmdr  n7.gg

First up is project metadata. This is a core requirement to publishing and distributing packages on PyPI, but accessible metadata can also enable developer tooling to better understand our project beyond just the files on disk. The best way to do that is with well structured, and standardized metadata.

setup.py

[omnilib](#) [n7cmdr](#) [n7.gg](#)

For anyone who's been around the block in the Python world, starting a new project means writing a new setup.py...

setup.py

```
with open("README.md") as f:
    readme = f.read()
with open("cursed/__init__.py") as f:
    for line in f:
        if line.startswith("__version__"):
            version = line.split('"')[1]
setup(
    name="cursed",
    description="the shame deep inside us",
    long_description=readme,
    long_description_content_type="text/markdown",
    version=version,
    author="Jane Doe",
    # ...
)
```

 omnilib  n7cmdr  n7.gg

Import setuptools, open the readme file, fetch a long description, or maybe we just find another project and copy paste...

```
setup.py

with open("README.md") as f:
    readme = f.read()
with open("cursed/__init__.py") as f:
    for line in f:
        if line.startswith("__version__"):
            version = line.split("'')[1]

setup(
    name="cursed",
    description="A package that lives inside 's'",
    long_description=readme,
    long_description_content_type="text/markdown",
    version=version,
    author="Jane Doe",
    # ...
)
```



Stop it! That's not metadata; that's arbitrary code that needs to be run every time the package is built from source. Nothing there is standardized beyond the idea of "whatever setuptools does". What we want is a well-defined format, that doesn't involve executing someone else's code.

pyproject.toml

 omnilib  n7cmdr  n7.gg

Thankfully, we have approved standards focused on a single file at the root of Python projects, called `pyproject.toml`. This file acts as a central location for project metadata and configuration for developer tools alike.

PEP 517, 518

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

The original standards for `pyproject.toml` were focused around the selection of build backends, the tools that actually transform your Python sources into installable distributions.

pyproject.toml

```
[build-system]
requires = ["flit-core >= 3.7"]
build-backend = "flit_core.buildapi"
```

 omnilib  n7cmdr  n7.gg

Together, PEP 517 and 518 defined the “build system” table, which any package manager can use to install your preferred build backend, and then actually build and install your package. I’m a fan of the flit backend, so here we need the flit-core package, and the associated build backend for our project.

PEP 621

 omnilib  n7cmdr  n7.gg

But these days, our `pyproject.toml` can also include standardized project metadata, thanks to PEP 621.

pyproject.toml

```
[project]
name = "neat"
description = "A one-line summary"
readme = "README.md"
license = {file="LICENSE"}
authors = [
    {name="Buzz Fizz", email="fizzbuzz@ch"},
]
requires-python = ">= 3.8"
dependencies = [
    "attrs",
    "click",
]
```

 omnilib  n7cmdr  n7.gg

This new format of metadata is in the top level “project” table of our pyproject.toml file, and includes all of the basic package information that would have previously lived in setup.py, or backend-specific tables. This unlocks a great potential for developer tooling to understand and use package metadata without the need to support many different formats.

pyproject.toml

```
[project]
name = "neat"
description = "A one-line summary"
readme = "README.md"
license = {file="LICENSE"}
authors = [
    {name="Buzz Fizz", email="fizzbuzz@ch"},
]
requires-python = ">= 3.8"
dependencies = [
    "attrs",
    "click",
]
```

omnilib n7cmdr n7.gg

At the top here, we have the basic metadata about the package, as you might expect, including the minimum Python version requirement...

pyproject.toml

```
[project]
name = "neat"
description = "A one-line summary"
readme = "README.md"
license = {file="LICENSE"}
authors = [
    {name="Buzz Fizz", email="fizzbuzz@ch"},
]
requires-python = ">= 3.8"
dependencies = [
    "attrs",
    "click",
]
```

omnilib n7cmdr n7.gg

And here we have our list of package dependencies. But just listing them isn't often enough, and can cause a number of bug reports from users who are using your project with older versions of these dependencies, or on platforms where they aren't supported.

goal: well defined dependencies

 omnilib  n7cmdr  n7.gg

So let's look at how we can better define our project's dependencies, and how that can help guide users and their package managers towards optimal choices.

version limits

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

Things change, especially in software. Packages will get new features, or deprecate functionality, or just outright make breaking changes that means your project won't work with some set of versions. That's why we can set version limits on our dependencies, to make sure package managers install compatible versions.

version limits

 omnilib  n7cmdr  n7.gg

If we wanted to absolutely guarantee compatibility, we could pin our dependencies to exact versions. But that's not convenient for users, especially if another package chooses to pin to a *different* version. Even if we expand our limits to anything within a major version number, we are still potentially leaving users stuck on old versions, open to bugs and security vulnerabilities.

future-friendly version limits

 omnilib  n7cmdr  n7.gg

Instead, we can compromise, and specify future-friendly version limits on our dependencies. With this method, we provide only a lower bound on our dependencies, ideally chosen based on the features we use from them, and what versions support those features.

```
dependencies = [  
    "attrs >= 20.1",  
    "click >= 7",  
    "pywin32 >= 301; platform_system == 'Windows'",  
    "typing_extensions >= 4.1; python_version < '3.10'",  
]
```

In our metadata, it could look something like this. By skipping the upper version boundary, we're trusting that our dependencies won't suddenly break compatibility. That might be a leap of faith, but this trade-off allows our users to upgrade to newer and potentially more secure versions, and also means we don't need to release a new version of our own projects every time a dependency releases a new major version. And we can always come back later and set an upper limit if we really need to.

```
dependencies = [  
    "attrs >= 20.1",  
    "click >= 7",  
    "pywin32 >= 301; platform_system == 'Windows'",  
    "typing_extensions >= 4.1; python_version < '3.10'",  
]
```

We can also add environment markers to our dependencies, for packages that only need to be installed on specific platforms, or backports that aren't needed on newer Python versions. But once we set them, how do we know our version limits are correct? The honest answer is that we need to validate them, and the only way to do that is by testing every version. But if we're being generous in our limits, that's a lot of different versions to test with.

pessimist

 omnilib  n7cmdr  n7.gg

Well, my coworker Tim wrote a tool to help with exactly that problem. And because the Python packaging ecosystem makes him so happy, he named the tool pessimist. What it does is quite simple, it looks at your project dependencies, and runs whatever test you prefer on all of the matching versions, then generates a report on whether your limits are valid.

```
(.venv) jreese@mordin ~/workspace/thx main » pessimist --fast -c 'thx test' --requirements= .
Requirement already satisfied: flit_core<4,>=3 in ./venv/lib/python3.10/site-packages (3.7.1)
Summary
=====
Variable ['aiotertools >= 0.10.0b1', 'click >= 8.0', 'packaging >= 21.0', 'rich >= 11.0.0', 'tomli
>= 1.0', 'trailrunner >= 1.1', "typing_extensions >= 4.0; python_version < '3.11'", 'watchdog >= 2
.1']
Fixed []

Versions
=====
aiotertools ['0.10.0b1', '0.10.0']
click ['8.0.0', '8.1.2']
packaging ['21.0', '21.3']
rich ['11.0.0', '12.2.0']
tomli ['1.0.0', '2.0.1']
trailrunner ['1.1.0', '1.1.3']
typing_extensions ['4.0.0', '4.1.1']
watchdog ['2.1.0', '2.1.7']

OK max
OK min
```

There's also a "fast mode", which limits the testing to both the oldest and newest matching versions, specifically for use in local testing or CI, when compute time is otherwise limited. Here, we're running it on a project's test suite, and pessimist validates that it passes with our limits.

goal: reproducible workflow

 omnilib  n7cmdr  n7.gg

Now that we're confident in our metadata and dependencies, let's focus on building a reproducible development workflow. Not only can this simplify the process of developing, testing, and validating our changes, but it will also make it that much easier to replicate our development on new machines or VMs, as well as for new developers to pick up your project and immediately make progress and have confidence that their changes are working.

you need a command runner

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

But just having a list of commands to run isn't really helpful. You need to have a dedicated command runner. This will combine all of your build and testing steps into a single command, including any preparation or setup necessary to get your project running on a new machine.



At a bare minimum, this can be satisfied with make files, and I've personally used them with many of my own projects for the last decade. But it knows nothing about how Python projects work, and requires you to re-engineer every piece of the puzzle.

```
make  
tox  
nox  
thx
```

 omnilib  n7cmdr  n7.gg

Ideally we want something that can do more of that work for us, especially around setting up virtual environments and installing dependencies, and we would like to do that on multiple Python versions with a single command. Tox is one of the most widely used testing tools available, while nox provides an alternative system that lets you define your workflow with actual Python code.

```
make  
tox  
nox  
thx
```

[omnilib](#) [n7cmdr](#) [n7.gg](#)

But today, I'd like to introduce a new tool I created: a rapid development assistant called "Thanks" (t-h-x). It uses many of the same basic concepts, but focuses on optimizing the development workflow for Python projects.

pyproject.toml

```
[tool.thx]
default = ["format", "test", "lint"]

[tool.thx.jobs]
format = "ufmt format neat"
lint = "flake8 neat"
publish = "flit publish"
test = "python -m unittest neat.tests"
```

 omnilib  n7cmdr  n7.gg

Thx is configured entirely in your existing pyproject.toml. Defining simple jobs is straightforward, with a list of job names and associated commands. We can then define a subset of those jobs that should be run by default.


```
jreese@mordin ~/workspace/neat main± >> thx
format OK
test OK
lint OK
```

[omnilib](#) [n7cmdr](#) [n7.gg](#)

Now when we run thx, it executes the default jobs, in virtual environments, and outputs results to the terminal. If everything is successful, we see a bunch of green OK messages.

```
jreese@mordin ~/workspace/neat main: » thx
format OK
test
└─ 3.10.2
  └─ 3.10.2 test> /Users/jreese/workspace/pycon/open-source-easy-mode/.thx/venv/3.10.2/bin/python -m unittest
     neat.tests FAIL

.F
=====
FAIL: test_two (neat.tests.MainTest)
-----
Traceback (most recent call last):
  File "/Users/jreese/workspace/pycon/open-source-easy-mode/neat/tests/__init__.py", line 15, in test_two
    self.assertEqual([1, 2, 3], [1, 2, 3, 4])
AssertionError: Lists differ: [1, 2, 3] != [1, 2, 3, 4]

Second list contains 1 additional elements.
First extra element 3:
4

- [1, 2, 3]
+ [1, 2, 3, 4]
?          +++

-----
Ran 2 tests in 0.000s

FAILED (failures=1)

FAIL
```

But if one of the jobs fails, we get clear error output, along with the exact command that was run. By default, thx will just run jobs using the same Python version that thx is installed with. But if we're trying to support our project on multiple versions of Python, we want our tools to run our development workflow on each of those supported versions. thx can do that, and do it fast.

pyproject.toml

```
[tool.thx]
default = ["format", "test", "lint"]
python_versions = ["3.8", "3.9", "3.10", "3.11"]
```

 omnilib  n7cmdr  n7.gg

We can simply give thx a list of Python versions we care about, and thx will look for these runtimes when executing jobs. Each of the requested jobs will run on each of these Python versions in parallel, with separate virtual environments for each version.

pyproject.toml

```
[tool.thx.jobs.format]  
run = "ufmt format neat"  
once = true
```

 omnilib  n7cmdr  n7.gg

In some cases though, like formatting, we might only need to run a job on a single version. There's no need to format our codebase four times, so we just mark that job with `once = true`.

```
[tool.thx.jobs.lint]
run = [
    "flake8 neat",
    "mypy --install-types --non-interactive -p neat",
    "ufmt check neat",
]
```

For more complex jobs, we might want to run multiple steps. Here, we define the job as a list of steps, each with its own command to be run. thx will run each of these steps in order, stopping early if a step fails. But in this case, none of these commands depend on the results of the previous one. They could all run in parallel just fine.

```
[tool.thx.jobs.lint]
run = [
    "flake8 neat",
    "mypy --install-types --non-interactive -p neat",
    "ufmt check neat",
]
parallel = true
```

So we can just mark the job as parallel, and thx will run all of the steps at the same time. And again, it will run all of these in parallel with each and every Python version we're testing against, taking full advantage of modern, multi-core systems, and making our iteration that much quicker. But thx has one more trick up its sleeve...

```
jreese@mordin ~/scratch/thx main > thx --watch
format OK
docs OK
test OK
lint OK
combine OK
coverage OK
└─ 3.11.0a7 OK
  └─ 3.11.0a7 coverage> /Users/jreese/scratch/thx/.thx/venv/3.11.0a7/bin/python -m coverage report OK
Name Stmts Miss Branch BrPart Cover Missing
-----
thx/cli.py 89 5 43 3 93.9% 83-87, 120->125, 129->132
thx/context.py 135 8 64 8 91.0% 80->86, 82->86, 99, 125->128, 132-133, 161->160, 169-170,
199, 218->226, 238-239
-----
TOTAL 570 13 314 11 97.1%

6 files skipped due to complete coverage.

jreese@mordin ~/scratch/thx main > █

jreese@mordin.local 0 zsh 2022-04-08 12:16P PDT
```

A “watch” mode, where it will wait for modifications to our project, and automatically re-run its jobs every time our files change on disk. It’s perfect for running in an IDE or terminal alongside your code, and gives us immediate feedback when something has broken our test suite or failed coverage thresholds. And its parallel performance provides tremendous leverage for validating our project throughout development.

drink

`::thx.omnilib.dev`

 omnilib  n7cmdr  n7.gg

If you'd like to know more about thx, please check it out at thx.omnilib.dev. There is more documentation and configuration options that you can find there, along with some recipes for common use cases, like code coverage and CI, and how thx can help.

Code Quality

[omnilib](#) [n7cmdr](#) [n7.gg](#)

So now that we have the tools to build out our workflow, we can begin looking at how to improve our development experience. One of the best areas to focus on is code quality, where the side effects of better code are often as important as the code itself. Better code doesn't just mean the project is faster or more reliable, but also that it's easier to read, easier to understand, and easier to debug. Together, this means you'll spend less time maintaining, and more time building. So where do we start?

goal: consistent, predictable code style

omnilib n7cmdr n7.gg

Let's pick the lowest-hanging fruit first: code style. And lets get the flame wars out of the way...

I don't care what your code looks like,
as long as it all looks the same.

omnilib n7cmdr n7.gg

I don't care what your code looks like, as long as it all looks the same... Consistency, and predictability, are the primary factors to making code easy to read. And the easier code is to read, the easier it is to understand what it's actually doing.

I don't care what your code looks like,
as long as it all looks the same.

Just don't make it my problem.

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

Just don't make it my problem. Nobody wants to spend an hour fixing a bug, only to get nitpicky feedback about code style. Choose a code formatter that enforces your preferred style, and make that an integral part of the development workflow, so that nobody, including you, has to ever spend time worrying about code style ever again. The more automated and extensive the better. We want to dump code from our brain, and let the computer worry about making it look right after the fact.

use black

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

But really, you should just use black. It's an excellent tool, with strong safety guarantees, so you can run it on your whole codebase without worrying about breaking anything. And for anyone who cares, it makes your code look like this

style.py

```
def example_function(  
    argument_one: List[str],  
    argument_two: Optional[bool] = False,  
) -> List[str]:  
    path = get_path("something")  
    result = (  
        object()  
        .first_method(path)  
        .second_method(items=argument_one)  
        .third_method(return_exceptions=argument_two)  
    )  
  
    return result
```

 omnilib  n7cmdr  n7.gg

(you shouldn't care)

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

But more importantly, it's configuration looks like this.



When you decide to use black, all the opinions come baked in. There's no more debate; there's only one color you can paint the bike shed. That's the easiest tool you'll ever configure in your life, and everyone will either thank you for it, or be really angry that they have nothing to argue about anymore.

sort your imports

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

So black will make your code look consistent, but it won't move your imports. I know a lot of folks really like their artisanal, hand crafted imports broken down by color and size and favorite musician, but that's a lot of work, and I'm lazy. Like code style, It's an awful lot easier to just let the computer do it for you.

sort your imports
with μ sort

 omnilib  n7cmdr  n7.gg

And this is where I get to shamelessly self promote some more, and say you should all be using usort. It was built from the ground up by Tim and myself to understand and manipulate the syntax tree directly, so that it can recognize and make intelligent choices about when and where it's safe to move imports.

style.py

```
from attr import dataclass
import click
from .data import VALUE
from pathlib import Path
import re
from typing import List, Optional
```

 omnilib  n7cmdr  n7.gg

So in the best case, usort will follow the most common patterns, and sort something that looks like this...

style.py

```
import re
from pathlib import Path
from typing import List, Optional

import click
from attr import dataclass

from .data import VALUE
```

 omnilib  n7cmdr  n7.gg

into this, groups for standard library, third party, and first party imports. *gesture* This is the easy part.

sorting.py

```
import warnings  
warnings.simplefilter("once")
```

```
import asyncio
```


 omnilib  n7cmdr  n7.gg

Now suppose we want to import a module, and change our runtime configuration before importing something else. Unlike other import sorters, usort focuses on making safe changes, and it treats intervening statements as barriers when sorting blocks of imports. So the asyncio import would never move above the simplefilter call, and the warnings import would never move below it, maintaining functionality without the need for ugly or error-prone comment directives.

```
sorting.py

import warnings
warnings.simplefilter("once")

import asyncio
```



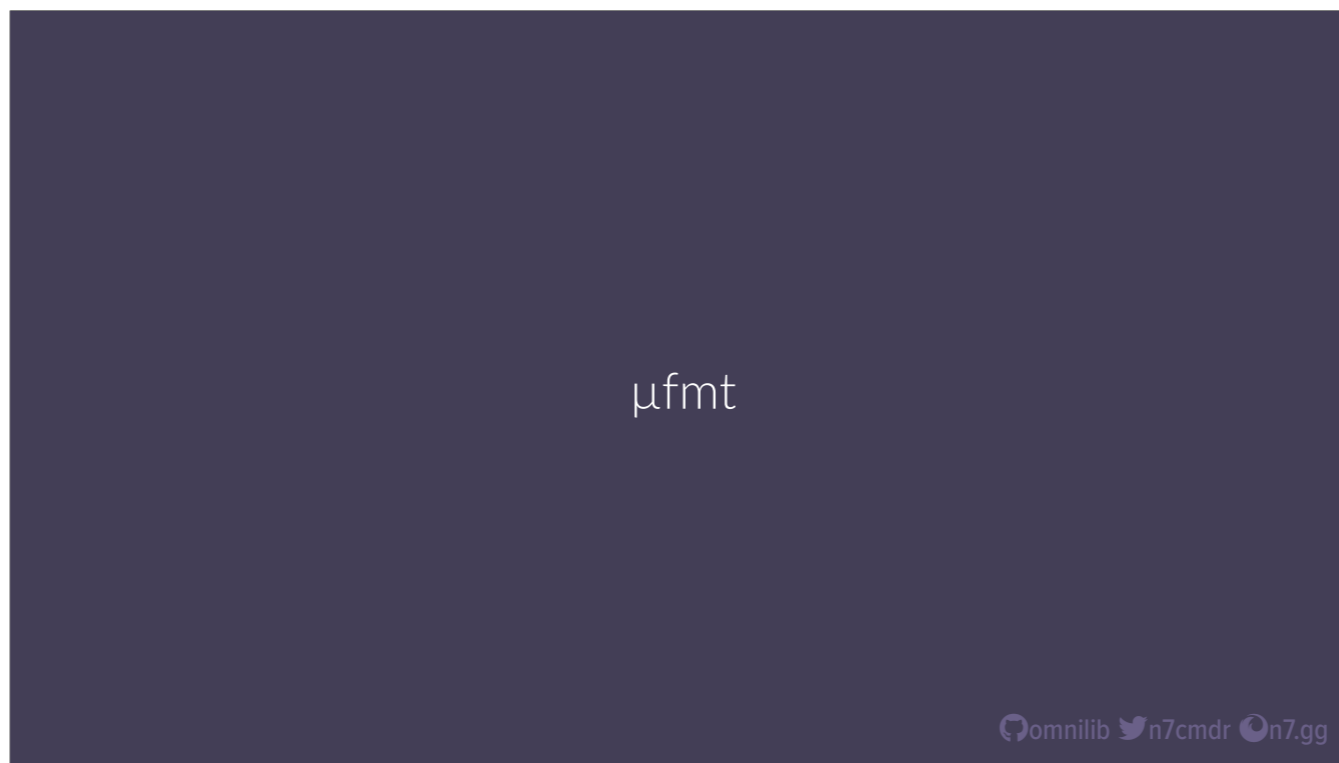
omnilib n7cmdr n7.gg

This level of safety is what allows us to run usort at scale inside Meta. Every Python file covered by our formatter gets sorted by usort, without exception. And we automatically apply “catch-up” formatting for all covered files every morning.

conflict resolution

 omnilib  n7cmdr  n7.gg

Now, its great to have these tools, but running multiple formatters separately is a recipe for minor conflicts, which can cause annoying failures in pre-commit hooks or CI where you just want to validate formatting. What you really need is something that can perform both formatting and import sorting in a single, atomic step.



And that's where ufmt comes in. ufmt is a combined code formatter and sorter...

$\mu\text{fmt} = \text{black} + \mu\text{sort}$

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

Built on top of black and usort. Each file is sorted and then formatted, in memory, as a single, atomic step, guaranteeing consistent results in CI and pre-commit hooks.

```
://usort.rtf.d.io  
://ufmt.omnilib.dev
```

omnilib n7cmdr n7.gg

If any of that sounds interesting, please check out the projects documentation for more details on how they work, and how you can integrate them into your developer workflow.

drink

goal: minimize subtle bugs

omnilib n7cmdr n7.gg

Now that we have formatting taken care of, let's look at how we can apply tooling to actually make the code more reliable by finding and pointing out bad practices, subtle bugs, or edge cases that aren't obvious, no matter how pretty the code looks.



linter

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

The easiest place to start is with a linter. They can help you find potential bugs in your code, or even help prevent you from introducing bugs in the future by steering you away from problematic patterns. There are a number of good options in this space, including pylint and pyflakes, but my personal favorite, and the one we use at Meta, is flake8.

The image shows a dark blue rectangular background with the word "flake8" centered in a white, sans-serif font. In the bottom right corner, there are three small, light blue icons: a GitHub logo, a Twitter logo, and a website icon, followed by the text "omnilib", "n7cmdr", and "n7.gg" respectively.

flake8

[omnilib](#) [n7cmdr](#) [n7.gg](#)

It provides a nice wrapper around pyflakes, has sensible defaults, easy configuration, and most importantly, has a number of excellent plugins available that increase the scope of problems that flake8 can help us find. By default, we'll get warnings that range from pointing out bad practices, like star imports, to bugs waiting to happen, like shadowed imports or asserts with parentheses, or even runtime errors like undefined variables or invalid grammar.

flake8
flake8-breakpoint
flake8-bugbear
flake8-comprehensions

 omnilib  n7cmdr  n7.gg

With some extra plugins, we can also get notified when we accidentally leave debugging breakpoints in our code, or when we use mutable default values for function parameters, or help guide us to better usage of list comprehensions or generator expressions in our code. Preventing bugs before they happen will save future-you an uncountable number of headaches and hours wasted debugging your projects.

.flake8

```
[flake8]
ignore =
    # covered by black/usort
    E1
    E2
    E3
    E4
    W503
max-line-length = 90
per-file-ignores =
    __init__.py: F401
```

 omnilib  n7cmdr  n7.gg

But what you don't want to waste your time with is a linter fighting with your formatter. We recommend turning off most, if not all, lint errors related to code style, and just let the formatters do their job. If it's good enough for the formatter, it should be good enough for your linter.

type checker

[omnilib](#) [n7cmdr](#) [n7.gg](#)

While linters can look for patterns in individual pieces of your codebase, the real meat is in static analysis tools, especially type checkers, which look at your codebase as a whole, and make sure everything is correctly passing and using values. And it can do this based on what it knows about not just your code, but other code as well, including the standard library and any other type annotated libraries used by your project.

bugs.py

```
def get_path(key):  
    ...  
  
def subtle_bug(key):  
    path = get_path(key)  
    with open(path) as f:  
        ...
```

 omnilib  n7cmdr  n7.gg

Let's look at an example of how types can help us find real world bugs, that a human may never notice. Let's assume `get_path` is a foreign API, and we don't really know, or shouldn't need to care, about how it works. Do you see the bug? Do you have an idea of what it could be?

bugs.py

```
def get_path(key: str) -> Optional[Path]:  
    ...  
  
def subtle_bug(key: str) -> None:  
    path = get_path(key)  
    with open(path) as f:  
        ...
```

 omnilib  n7cmdr  n7.gg

What if we add some type annotations? We're adding new information here, that previously we might have needed to read a docstring for.

bugs.py

```
def get_path(key: str) -> Optional[Path]:  
    ...  
  
def subtle_bug(key: str) -> None:  
    path: Optional[Path] = get_path(key)  
    with open(path) as f:  
        ...
```

 omnilib  n7cmdr  n7.gg

Maybe some extra, inline notation helps? What if I told you that `open()` raises an exception when passed `None`? Well, there's some input that can cause `get_path()` to return `None`, and then we're just passing that straight to `open()`, and you end up with an angry bug report because someone crashed their production service.

```
neat/__init__.py:46: error: Argument 1 to "open" has incompatible  
type "Optional[Path]"; expected "Union[Union[str, bytes, PathLike[  
str], PathLike[bytes]], int]"
```

 omnilib  n7cmdr  n7.gg

But our type checker can find and report that before we ever release the code, let alone run in production.

types as documentation

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

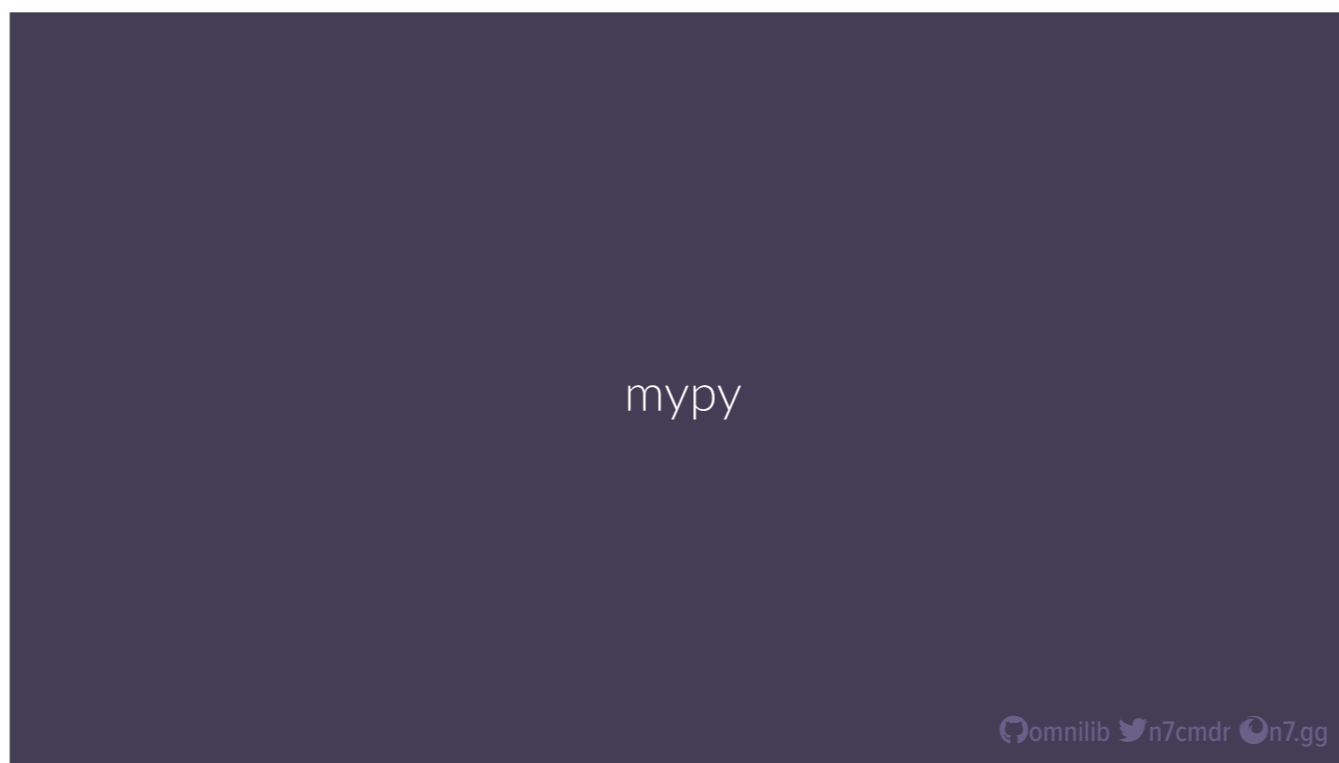
But beyond finding errors, there are benefits just to adding type annotations to your own modules. Consider it as a form of validated documentation for you and everyone else looking at your project.

bugs.py

```
def create_record(id, value):  
    ...  
  
def create_record(id: ID, value: str) -> Record:  
    ...
```

 omnilib  n7cmdr  n7.gg

Given these two definitions for the same function, which one do you find more useful? Which one could you evaluate faster when looking at a foreign API? We don't even need to see a docstring or look at the body of the function to have a good idea of the contract described. THIS IS VALUABLE DOCUMENTATION FOR DEVELOPERS, and unlike a docstring, a good type checker will force you and everyone else to keep this information up to date!



So again, there are multiple good choices here for type checkers, but I recommend the use of mypy for open source projects. It's by far the most popular type checker available, and works really well with minimal setup or configuration needed.

pyproject.toml

```
[tool.mypy]
ignore_missing_imports = true
strict = true
```

 omnilib  n7cmdr  n7.gg

For many projects, this is the only option you need, just because there are still a large number of libraries out there that don't have type stubs, or aren't appropriately marked as annotated, and there's not much we can do about that. You don't even need all of your code to be type annotated to start seeing an impact.

pyproject.toml

```
[tool.mypy]
ignore_missing_imports = true
strict = true
```

 omnilib  n7cmdr  n7.gg

But if you're willing to type annotate all of your code, including your test cases, I highly recommend enabling strict mode. This will let mypy get more pedantic, find more bugs, and also point out any time you fail to document, I mean annotate, your code.

goal: accessible, updated documentation

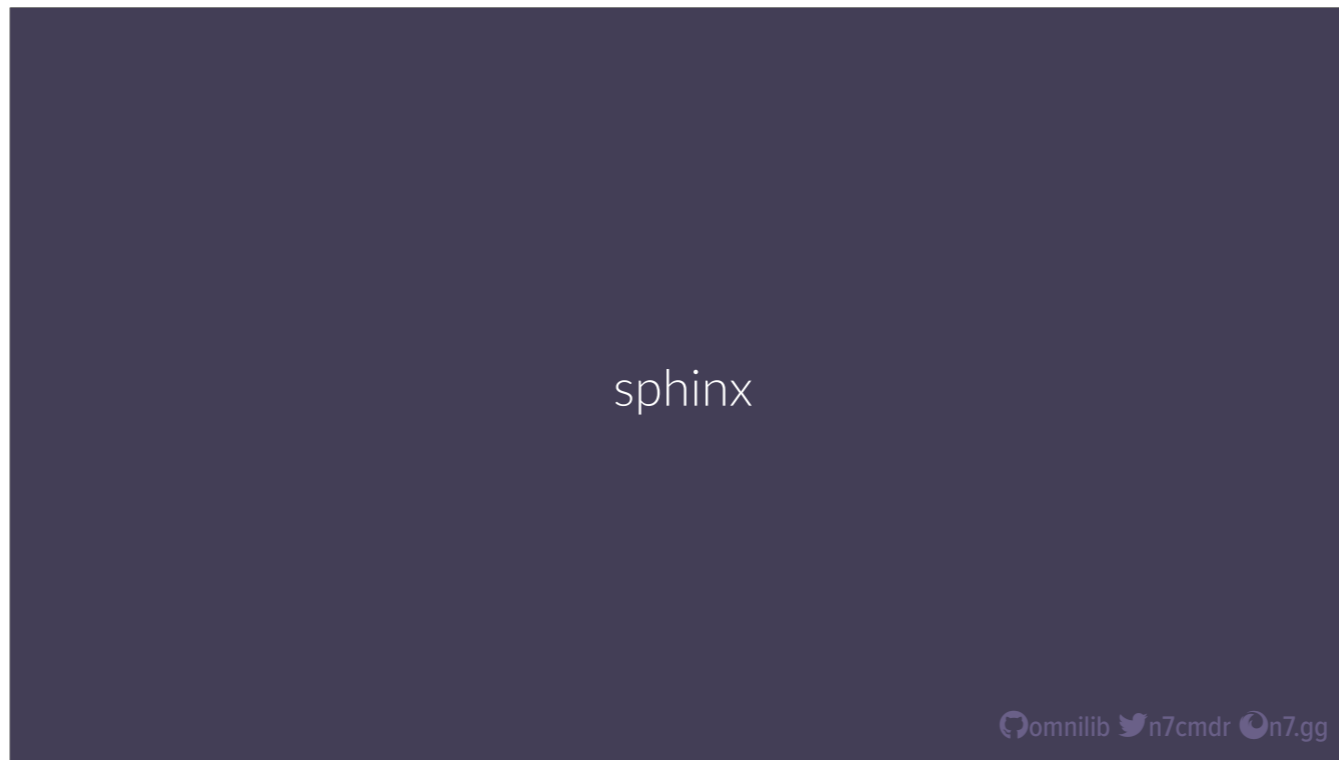
 omnilib  n7cmdr  n7.gg

Speaking of documentation, while looking at the code directly is important for working on your own project, the last thing you want to look at is someone else's code, especially when you just need to know what a function is called, or what various arguments do. Most importantly, there will come a day when even your own code looks foreign to you, and you've forgotten what your own APIs look like. When that happens, you can turn to your own documentation for help.

compiled documentation

 omnilib  n7cmdr  n7.gg

But we don't just want to brain dump info into a text file, tucked away and forgotten, cursed to never be accurate again. What we need is tooling specifically designed to extract as much information as possible from our codebase, including docstrings and type information, and automatically compile that documentation into useful formats.



Now there are many different tools that can do this, but sphinx is designed specifically for Python, and is used by hundreds of open source projects. Sphinx will compile and transform a set of source documents into a full website, including navigation and text search. You can add arbitrary pages as well, allowing you to build a combined project and documentation site, all in one, with relatively little effort required to maintain it.

usort/docs/guide.rst

```

Directives
^^^^^^^^^^

usort will obey simple ``#usort:skip`` directives to prevent moving import statements,
including moving any other statements across the skipped statement::

import math

import important_thing # usort: skip

import difflib

Comment directives must be on the first or last line of multi-line imports::

from side_effect import ( # usort:skip # here
    thing_one,
    thing_two,
) # usort:skip # or here

Directives are also allowed anywhere in a comment, but must include another ``#``
character if they are not the first element::

import side_effect # noqa: F401 # usort:skip

See 'Import Blocks' for details on how skip directives affect sorting behavior.

.. note::
    For compatibility with existing codebases previously using isort, the
    ``#isort:skip`` directive is also supported, with the same behavior as
    ``#usort:skip``.

```

Directives

usort will obey simple `#usort:skip` directives to prevent moving import statements, including moving any other statements across the skipped statement:

```

import math
import important_thing # usort: skip
import difflib

```

Comment directives must be on the first or last line of multi-line imports:

```

from side_effect import ( # usort:skip # here
    thing_one,
    thing_two,
) # usort:skip # or here

```

Directives are also allowed anywhere in a comment, but must include another `#` character if they are not the first element:

```

import side_effect # noqa: F401 # usort:skip

```

See [Import Blocks](#) for details on how skip directives affect sorting behavior.

Note:

For compatibility with existing codebases previously using isort, the `#isort:skip` directive is also supported, with the same behavior as `#usort:skip`.

However, the `#isort:skip_file` directive is **ignored** by usort, and there is no supported equivalent. We believe that usort's behavior is safe enough that all files can be safely sortable, given an appropriate [configuration](#) that includes any known modules with import-time side effects.

If there are files you absolutely don't want sorted; don't run usort on them.

omnilib n7cmdr n7.gg

Sphinx uses reStructuredText format, a plain text format similar to Markdown, but focused on technical documentation. When compiled, Sphinx generates well-formatted HTML—*click*—, and we can include code blocks with syntax highlighting, links to other sections of the documentation, and much more that isn't possible in plain Markdown. And we don't need to sacrifice the ability to read and understand the source text in an editor.

sphinx.ext.autodoc

 omnilib  n7cmdr  n7.gg

But the real magic of Sphinx comes from the autodoc extension. This gives Sphinx the power to look at your project's source code, and automatically extract module, class, and function APIs, including type annotations, docstring contents, and more.

```
async def as_generated(
    iterables: Iterable[AsyncIterable[T]],
    *,
    return_exceptions: bool = False,
) -> AsyncIterable[T]:
    """
    Yield results from one or more async iterables, in the order they are produced.

    Like :func:`as_completed`, but for async iterators or generators instead of futures.
    Creates a separate task to drain each iterable, and a single queue for results.

    If ``return_exceptions`` is ``False``, then any exception will be raised, and
    pending iterables and tasks will be cancelled, and async generators will be closed.
    If ``return_exceptions`` is ``True``, any exceptions will be yielded as results,
    and execution will continue until all iterables have been fully consumed.

    Example::

        async def generator(x):
            for i in range(x):
                yield i

        gen1 = generator(10)
        gen2 = generator(12)

        async for value in as_generated([gen1, gen2]):
            ... # intermixed values yielded from gen1 and gen2
    """
```

With this in mind, we can actually write our docstrings with the Sphinx results in mind, using inline reStructuredText. We can include references to other functions, inline code blocks, and anything else we could put in a normal documentation page.

```
asyncio
```

```
-----
```

```
.. automodule:: aioitertools.asyncio
   :members:
```

```
async aioitertools.asyncio.as_generated(iterables, *,
return_exceptions=False)
```

Yield results from one or more async iterables, in the order they are produced.

Like `as_completed()`, but for async iterators or generators instead of futures. Creates a separate task to drain each iterable, and a single queue for results.

If `return_exceptions` is `False`, then any exception will be raised, and pending iterables and tasks will be cancelled, and async generators will be closed. If `return_exceptions` is `True`, any exceptions will be yielded as results, and execution will continue until all iterables have been fully consumed.

Example:

```
async def generator(x):
    for i in range(x):
        yield i

gen1 = generator(10)
gen2 = generator(12)

async for value in as_generated([gen1, gen2]):
    ... # intermixed values yielded from gen1 and gen2
```

Parameters: • `iterables` (`Iterable[AsyncIterable[aioitertools.types.T]]`) –

• `return_exceptions` (`bool`) –

Return type: `AsyncIterable[aioitertools.types.T]`

Then we just add the appropriate autodoc directives to our documentation, and regenerate our site. *click* And now our compiled documentation includes our function prototype, the formatted contents of our docstring, as well as type information. Now, you might be asking, what do you do with this once you have it? Maybe you've heard of a little site called ...

Read the Docs

 omnilib  n7cmdr  n7.gg

Read the Docs. This an excellent, free service for building and hosting your project documentation sites. It's specifically designed to work with Sphinx, and has excellent integrations for projects on Github, including the ability to automatically rebuild doc sites when pushing commits, as well as building preview sites for each pull request, so you can evaluate any changes to documentation when reviewing PRs.

drink

Community Contributions

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

Now that we have an end-to-end developer and documentation workflow, let's take a look at how we can streamline the process of encouraging and reviewing community contributions, without putting undo burden on your own time.

goal: clear boundaries and expectations

 omnilib  n7cmdr  n7.gg

One of the most important pieces of having positive community interaction starts with setting clear boundaries of expectations, in both directions.

code of conduct

 omnilib  n7cmdr  n7.gg

You really should have a code of conduct; if and when your project gains more attention, clowns will inevitably show up, and the easiest way to deal with them is by having a code of conduct with clear set of rules and consequences that you can follow. I use the Contributor Covenant for the Omnilib Project, and recommend it both for the community standards it includes, as well as for their guidance on how to enforce it.

acceptable contributions

 omnilib  n7cmdr  n7.gg

Beyond a code of conduct, make sure you set clear expectations for what contributions you want from the community. If you only care about bug fixes, say that. If you're happy to take feature requests, say that also. Let folks know ahead of time if they should open an issue or discuss new features before they take up their own time and yours with pull requests that you would never accept.

you do not owe anyone your time

 omnilib  n7cmdr  n7.gg

If you can, document the level of support you will give for each project, but don't over commit yourself. Unless there a paid support contract, you do not owe anyone your time...

you do not owe anyone your energy

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

you do not owe them your energy

you do not owe anyone your sanity

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

and you most certainly do not owe them your sanity. If someone is being too needy, and it's taking up your limited time, or otherwise ruining your enjoyment and willingness to work on the project...

dump ~~him~~ them

 omnilib  n7cmdr  n7.gg

dump them. If they're being nice, apologize and just tell them you can't help them, and maybe point them to an alternative project. If they're being less than nice, just don't engage, and remove them from the community if it continues to escalate. Don't waste your time on unproductive community members.

goal: simple contributor workflow

omnilib n7cmdr n7.gg

On a happier note, let's look at how we can simplify and automate as much of the contributor workflow as possible. We want to remove barriers between community members and their ability to ramp up on your project. The less friction, the more likely you are to receive high quality contributions.

contributor guide

 omnilib  n7cmdr  n7.gg

This starts with, you guessed it, more documentation. A basic contributor's guide starts with steps to set up and build your project from source, as well as how to run the project and test suite. A better guide also includes any extra validation steps needed, like linters or formatters, as well as general guidance on what to include with their changes, like documentation, test cases, or changelog entries.

reproducible workflows

 omnilib  n7cmdr  n7.gg

This is where our reproducible workflow from earlier really comes into play. The closer contributors can get to the workflow you use as the owner of the project, the more trust you will have in their contributions, and the more trust they will have in your project and their own changes during development.

issue templates

 omnilib  n7cmdr  n7.gg

You can further help guide contributors in the right path by adding detailed issue templates and pull request templates to your project. This will pre-populate the text boxes with whatever you want. You can have multiple different templates on github, tailored to specific types of issues or requests. Be sure to include spots for any information you want to see from users for feature requests or bug reports, like steps to reproduce, their environment, or whatever makes sense for your project.

goal: tested and validated contributions

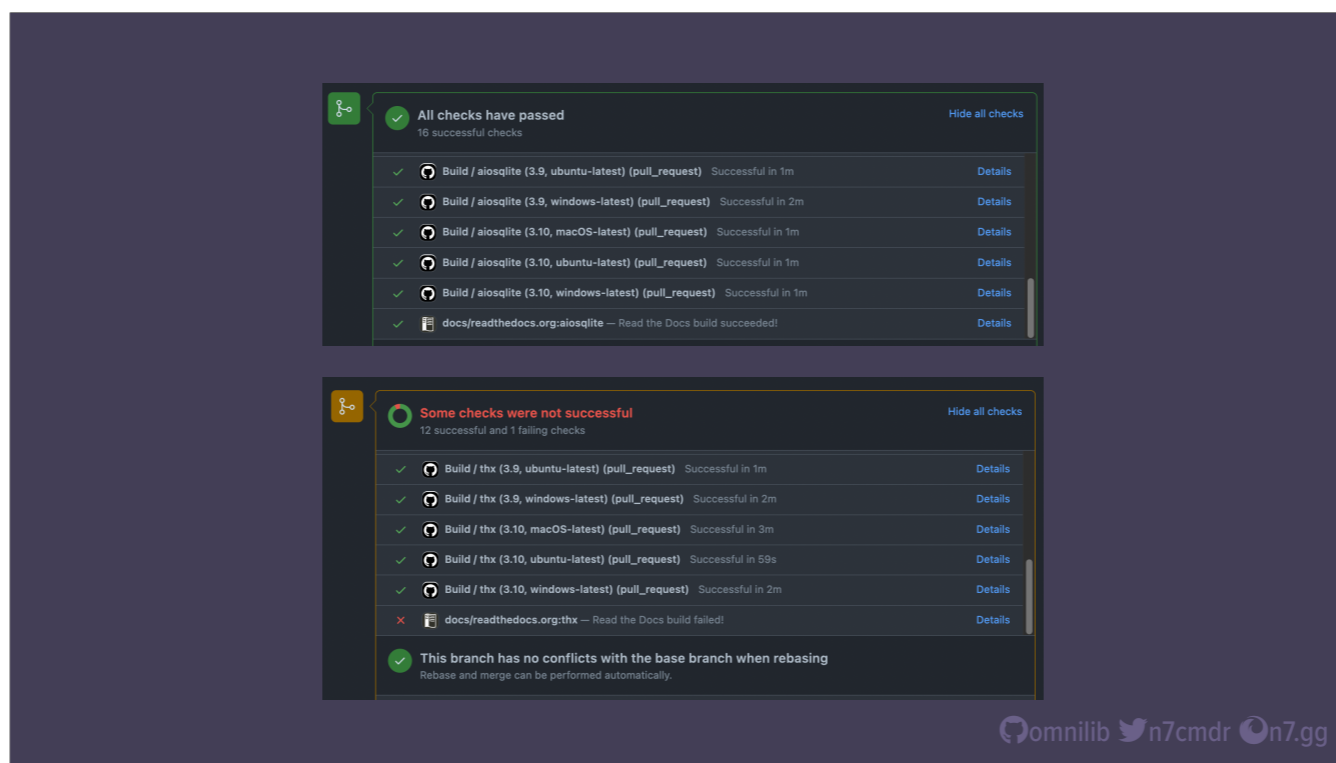
omnilib n7cmdr n7.gg

Once your community members have submitted a pull request, we want to make sure we automate the testing and validation process, in a way that not only reduces the time you spend reviewing their contributions, but also gives immediate, actionable feedback to the contributor for anything that would cause you to request changes or outright reject the PR.

continuous integration

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

Continuous integration systems are one of the easiest and most common ways to make this happen. If your project is on Github, then Github Actions is the easy choice, and runs for free on public repos, with inline display of results on every pull request.



The biggest benefit here is the ability to configure CI to test your project on a variety of operating systems and Python runtime versions simultaneously, and reporting failures separately for each possible combination. This makes it much easier to determine if the error is OS- or version-dependent, or if it's a more basic issue.

REPRODUCIBLE WORKFLOWS

 omnilib  n7cmdr  n7.gg

Again, we'll beat the drum for reproducible workflows. Whatever CI platform or system you choose, it should be configured to use the same build, test, and validation workflow that you run locally. This will help minimize surprises when pushing code or submitting pull requests, and also makes it that much easier to dig in and understand the output for failures, because it will look just like what you experience in your normal development workflow.

github actions / integrations

omnilib n7cmdr n7.gg

Lastly, be aware that Github actions, as well as integrations with other services, like read the docs or various code coverage systems, can also be added to your project, and can affect more than just pull requests. Github Actions can also be triggered by a large variety of events in your project, including new issues or PRs, and generic Github bots can help manage your project and issues as well. Your imagination is the limit.

Release

[omnilib](#) [n7cmdr](#) [n7.gg](#)

The last piece of the puzzle is releasing your project. The easier it is to prepare and release a new version, the faster we can publish our bug fixes. And the more “boring” each individual release is, with targeted fixes or improvements, the more reliable and trustworthy your project will be perceived. So let’s start with the most important part of releasing new versions ...

goal: meaningful versions

 omnilib  n7cmdr  n7.gg

Picking the right version numbers. Ultimately, we want version numbers to convey some amount of meaningful information: some idea of how important, or how “big”, or even how dangerous, any given release is going to be. One of the most well-known schemes is SemVer, or semantic versioning, and I’m here to say

semver is impossible

 omnilib  n7cmdr  n7.gg

semver is impossible. The major version number is supposed to tell us when breaking changes occur, but how do we define “breaking” changes? Is a change to a feature breaking? Is an API change breaking? Is a bugfix breaking?

what happens in CalVer
stays in CalVer

 omnilib  n7cmdr  n7.gg

On the other end of the scale, we can simply base our versions on the year or date of each release. But beware, once your major version is based on the calendar year, you've burned more major versions than Chrome or Messenger. Yes, technically you can increment a version epoch and start over from zero, but literally nobody wants that, and you'd be surprised how many tools that might break.

be consistent

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

So really, just pick something, be consistent with your versioning, and communicate your plan clearly. If you want to try adhering to SemVer, go for it! If you just want to increment an integer every time, go for it. Just be consistent, and set expectations accordingly.

goal: minimal release effort

 omnilib  n7cmdr  n7.gg

But picking a version number is only the first step. There's a lot more that can go into a release than you might expect, so we want to automate as much of that process as we can. And like our development processes from earlier, what do we want?

reproducible workflows 🙄

[omnilib](#) [n7cmdr](#) [n7.gg](#)

That's right, we want another reproducible workflow. You shouldn't have to figure out and piece this all together every time you get a new laptop. At the very least, document the necessary tools and steps to publish a release, so that future-you can just refer to your documentation. But we can do more.

git tags

 omnilib  n7cmdr  n7.gg

The key to reproducible releases is basing them off of published, tagged revisions in your source control system. These tags should already include any changes to documentation, version numbers, release notes, etc. Now, producing those changes might be the hardest part of pushing a release, but there are tools to help.

attribution

 omnilib  n7cmdr  n7.gg

I specifically wrote a tool called “attribution”, which automates the process of generating a changelog from existing source control tags, bumping the version string in your codebase, and then committing and tagging everything to prepare for a new release. All you need to do is pick a version number, fill in your release notes, and you’re ready to publish your release.

goal: desirable release artifacts

omnilib n7cmdr n7.gg

As for building your actual release artifacts, we need to make sure we're building the right pieces. There are two major types of release artifacts, or distributions, in major use today:

wheels

 omnilib  n7cmdr  n7.gg

There are binary wheels, which contain precompiled libraries, ready to be installed. But for projects with compiled extensions, any given wheel can only be installed on a specific OS and Python version. If you want to support multiple platforms or Python versions, you'll need a separate wheel for every possible combination.

sdist

 omnilib  n7cmdr  n7.gg

The second (and original) format is the source distribution, or sdist. This is ideally just like checking out the project repo, and a proper sdist should have everything needed to build a wheel on any OS or Python version.

sdist > wheels

 omnilib  n7cmdr  n7.gg

That last bit is key. An sdist can be used to build and install a package anywhere, including obscure platforms or future Python versions where you haven't already published a wheel. This makes the sdist the single most important release artifact for your project, and you should make sure that your publishing workflow always includes an sdist in your release.

sdist + wheels > sdist

 omnilib  n7cmdr  n7.gg

But if your project includes compiled extensions, then you really should *also* publish wheels, at least for the major platforms and latest Python versions that you support. This will let the majority of your users install your project faster, without needing to build it from source. However, manually building a vast matrix of supported wheels is a daunting challenge, but thankfully, we have help.

cibuildwheel

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

The cibuildwheel project is a well-established tool for building, validating, and publishing wheels as part of your continuous integration.

	macOS Intel	macOS Apple Silicon	Windows 64bit	Windows 32bit	Windows Arm64	manylinux muslinux x86_64
CPython 3.6	✓	N/A	✓	✓	N/A	✓
CPython 3.7	✓	N/A	✓	✓	N/A	✓
CPython 3.8	✓	✓	✓	✓	N/A	✓
CPython 3.9	✓	✓	✓	✓	✓ ²	✓ ²
CPython 3.10	✓	✓	✓	✓	✓ ²	✓

omnilib n7cmdr n7.gg

It supports that giant matrix of platforms and Python versions, and with the right setup, can even publish these wheels automatically with each release. There are examples you can follow for use with various CI services available on the project website.

Relax!

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

Phew, we made it! We've reproduced our workflows, we've improved our code quality, we've fostered our community, and we published our distributions. Now what? Well, now we take time for ourselves. Don't forget the golden rule:

time is the only resource you can't buy

 [omnilib](#)  [n7cmdr](#)  [n7.gg](#)

This is your life, and your project, and ultimately, you're the only one who can choose how to spend your time. I appreciate you spending this time here with me.

[://github.com/jreese/pycon](https://github.com/jreese/pycon)

omnilib n7cmdr n7.gg

You can find links to all of the projects and resources mentioned in this talk on my github repo, [jreese/pycon](https://github.com/jreese/pycon). I also included all of the example code, and a miniature project using many of the tools I recommended throughout the talk. If you have questions, you can find me in the hallway track, or at the PyLounge in the expo hall. Thank you again, and have a wonderful Pycon.