7 essential PyPI libraries and how to use them

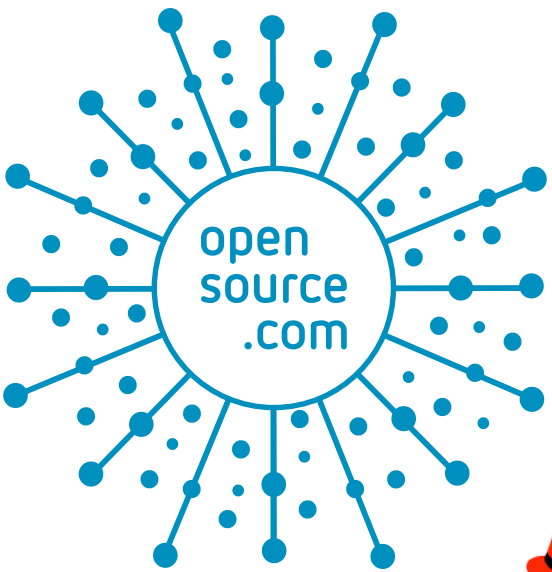
ABOUT OPENSOURCE.COM

What is Opensource.com?

OPENSOURCE.COM publishes stories about creating, adopting, and sharing open source solutions. Visit Opensource.com to learn more about how the open source way is improving technologies, education, business, government, health, law, entertainment, humanitarian efforts, and more.

Submit a story idea: https://opensource.com/story

Email us: open@opensource.com



Supported by
Red Hat

MOSHE ZADKA

MOSHE HAS BEEN INVOLVED IN THE LINUX COMMUNITY since 1998, helping in Linux "installation parties". He has been programming Python since 1999, and has contributed to the core Python interpreter. Moshe has been a DevOps/SRE since before those terms existed, caring deeply about software reliability, build reproducibility and other such things. He has worked in companies as small as three people and as big as tens of thousands — usually some place around where software meets system administration.

FOLLOW MOSHE ZADKA

Twitter: https://twitter.com/moshezadka

CONTENTS

Introduction

PYTHON IS ONE OF THE MOST POPULAR PROGRAMMING LANGUAGES in use today—and for good reasons: it's open source, it has a wide range of uses (such as web programming, business applications, games, scientific programming, and much more), and it has a vibrant and dedicated community supporting it. This community is the reason we have such a large, diverse range of software packages available in the Python Package Index (PyPI) to extend and improve Python and solve the inevitable glitches that crop up.

In this series, we'll look at seven PyPI libraries that can help you solve common Python problems.

Write faster C extensions for Python with **Cython**

Cython is a language that simplifies writing C extensions for Python.

PYTHON IS FUN TO USE, but sometimes, programs written in it can be slow. All the runtime dynamic dispatching comes with a steep price: sometimes it's up to 10-times slower than equivalent code written in a systems language like C or Rust.

Moving pieces of code to a completely new language can have a big cost in both effort and reliability: All that manual rewrite work will inevitably introduce bugs. Can we have our cake and eat it too?

To have something to optimize for this exercise, we need something slow. What can be slower than an accidentally exponential implementation of the Fibonacci sequence?

```
def fib(n):
    if n < 2:
        return 1
    return fib(n-1) + fib(n-2)
```

Since a call to **fib** results in two calls, this beautifully inefficient algorithm takes a long time to execute. For example, on my new laptop, **fib(36)** takes about 4.5 seconds. These 4.5 seconds will be our baseline as we explore how Python's Cython extension [1] can help.

The proper way to use Cython is to integrate it into **setup.py**. However, a quick and easy way to try things out is with **pyximport**. Let's put the **fib** code above in **fib.pyx** and run it using Cython.

```
>>> import pyximport; pyximport.install()
>>> import fib
>>> fib.fib(36)
```

Just using Cython with no code changes reduced the time the algorithm takes on my laptop to around 2.5 seconds. That's a reduction of almost 50% runtime with almost no effort; certainly, a scrumptious cake to eat and have!

Putting in a little more effort, we can make things even faster.

```
cpdef int fib(int n):
    if n < 2:
        return 1
    return fib(n - 1) + fib(n - 2)
```

We moved the code in **fib** to a function defined with **cpdef** and added a couple of type annotations: it takes an integer and returns an integer.

This makes it *much* faster—around 0.05 seconds. It's so fast that I may start suspecting my measurement methods contain noise: previously, this noise was lost in the signal.

So, the next time some of your Python code spends too long on the CPU, maybe spinning up some fans in the process, why not see if Cython can fix things?

Links

[1] https://pypi.org/project/Cython/

Format Python however you like with **Black**

Black formats your Python code consistently for you.

SOMETIMES CREATIVITY can be a wonderful thing. Sometimes it is just a pain. I enjoy solving hard problems creatively, but I want my Python formatted as consistently as possible. Nobody has ever been impressed by code that uses "interesting" indentation.

But even worse than inconsistent formatting is a code review that consists of nothing but formatting nits. It is annoying to the reviewer—and even more annoying to the person whose code is reviewed. It's also infuriating when your linter tells you that your code is indented incorrectly, but gives no hint about the *correct* amount of indentation.

Enter Black [1]. Instead of telling you *what* to do, Black is a good, industrious robot: it will fix your code for you.

To see how it works, feel free to write something beautifully inconsistent like:

```
def add(a, b): return a+b

def mult(a, b):
      return \
        a             *        b
```

Does Black complain? Goodness no, it just fixes it for you!

```
$ black math
reformatted math
All done! ✨ 🍰 ✨
1 file reformatted.
$ cat math
def add(a, b):
    return a + b


def mult(a, b):
    return a * b
```

Black does offer the option of failing instead of fixing and even outputting a **diff**-style edit. These options are great in a continuous integration (CI) system that enforces running Black locally. In addition, if the **diff** output is logged to the CI output, you can directly paste it into **patch** in the rare case that you need to fix your output but cannot install Black locally.

```
$ black --check --diff bad
--- math 2019-04-09 17:24:22.747815 +0000
+++ math 2019-04-09 17:26:04.269451 +0000
@@ -1,7 +1,7 @@
-def add(a, b): return a + b
+def add(a, b):
+    return a + b


 def mult(a, b):
-          return \
-                  a             *             b
+    return a * b

would reformat math
All done! 💥 💔 💥
1 file would be reformatted.
$ echo $?
1
```

Links

[1] https://pypi.org/project/black/

Say goodbye to boilerplate in Python with **attrs**

attrs is a Python package that helps you write concise, correct code quickly.

IF YOU HAVE BEEN USING PYTHON for any length of time, you are probably used to writing code like:

```
class Book(object):

    def __init__(self, isbn, name, author):
        self.isbn = isbn
        self.name = name
        self.author = author
```

Then you write a **__repr__** function; otherwise, it would be hard to log instances of **Book**:

```
    def __repr__(self):
        return f"Book({self.isbn}, {self.name}, {self.author})"
```

Next, you write a nice docstring documenting the expected types. But you notice you forgot to add the **edition** and **published_year** attributes, so you have to modify them in five places.

What if you didn't have to?

```
@attr.s(auto_attribs=True)
class Book(object):
    isbn: str
    name: str
    author: str
    published_year: int
    edition: int
```

Annotating the attributes with types using the new type annotation syntax, **attrs** [1] detects the annotations and creates a class.

ISBNs have a specific format. What if we want to enforce that format?

```
@attr.s(auto_attribs=True)
class Book(object):
    isbn: str = attr.ib()
    @isbn.validator
    def pattern_match(self, attribute, value):
        m = re.match(r"^(\d{3}-)\d{1,3}-\d{2,3}-\d{1,7}-\d$", value)
        if not m:
            raise ValueError("incorrect format for isbn", value)
    name: str
    author: str
    published_year: int
    edition: int
```

The **attrs** library also has great support for immutability-style programming [2]. Changing the first line to **@attr.s(auto_attribs=True, frozen=True)** means that **Book** is now immutable: trying to modify an attribute will raise an exception. Instead, we can get a *new* instance with modification using **attr.evolve(old_book, published_year=old_book.published_year+1)**, for example, if we need to push publication forward by a year.

Links

[1] https://pypi.org/project/attrs/

[2] https://opensource.com/article/18/10/functional-programming-python-immutable-data-structures

Add methods retroactively in Python with **singledispatch**

singledispatch is a library that allows you to add methods to Python libraries retroactively.

IMAGINE YOU HAVE A "SHAPES" LIBRARY with a **Circle** class, a **Square** class, etc.

A **Circle** has a **radius**, a **Square** has a **side**, and a **Rectangle** has **height** and **width**. Our library already exists; we do not want to change it.

However, we do want to add an **area** calculation to our library. If we didn't share this library with anyone else, we could just add an **area** method so we could call **shape.area()** and not worry about what the shape is.

While it is possible to reach into a class and add a method, this is a bad idea: nobody expects their class to grow new methods, and things might break in weird ways.

Instead, the **singledispatch** [1] function in **functools** can come to our rescue.

```
@singledispatch
def get_area(shape):
    raise NotImplementedError("cannot calculate area for
                              unknown shape", shape)
```

The "base" implementation for the **get_area** function fails. This makes sure that if we get a new shape, we will fail cleanly instead of returning a nonsense result.

```
@get_area.register(Square)
def _get_area_square(shape):
    return shape.side ** 2
@get_area.register(Circle)
def _get_area_circle(shape):
    return math.pi * (shape.radius ** 2)
```

One nice thing about doing things this way is that if someone writes a new shape that is intended to play well with our code, they can implement **get_area** themselves.

```
from area_calculator import get_area

@attr.s(auto_attribs=True, frozen=True)
class Ellipse:
    horizontal_axis: float
    vertical_axis: float

@get_area.register(Ellipse)
def _get_area_ellipse(shape):
    return math.pi * shape.horizontal_axis * shape.vertical_axis
```

Calling **get_area** is straightforward.

```
print(get_area(shape))
```

This means we can change a function that has a long **if isintance()/elif isinstance()** chain to work this way, without changing the interface. The next time you are tempted to check **if isinstance**, try using **singledispatch**!

Links

[1] https://pypi.org/project/singledispatch/

Automate your Python code tests with **tox**

tox is a tool for automating tests on Python code.

WHEN WRITING PYTHON CODE, it is good to have automated checks. While you could dump the rules for running the checks directly into the continuous integration (CI) environment, that's seldom the best place for it. Among other things, it is useful to run tests locally, using the same parameters the CI runs, to save CI time..

The tox project [1] is designed to run different checks against different versions of Python and against different versions of dependencies. Very quickly, we find the limiting factor is not the flexibility of tox but the harsh realities of the combinatorial explosions of options!

For example, a simple tox configuration can run the same tests against several versions of Python.

```
[tox]
envlist = py36,py37
[testenv]
deps =
    pytest
commands =
    pytest mylibrary
```

Tox will automatically use the right version of the interpreter, based on the version of the environment, to create the virtual environment. Tox will automatically rebuild the virtual environment if it is missing or if the dependencies change.

It is possible to explicitly indicate the Python version in an environment.

```
[tox]
envlist = py36,py37,docs
[testenv]
deps =
    pytest
commands =
    pytest mylibrary
[testenv:docs]
changedir = docs
deps =
    sphinx
commands =
    sphinx-build -W -b html -d {envtmpdir}/doctrees .
{envtmpdir}/html
basepython = python3.7
```

This example uses Sphinx [2] to build documentation for the library. One nice thing is that the Sphinx library will be installed only in the **docs** virtual environment. If **mylibrary** imports on Sphinx but forgets to indicate an explicit dependency, the tests will, correctly, fail.

We can also use tox to run the tests with different versions of the dependencies.

```
[tox]
envlist = {py36,py37}-{minimum,current}
[testenv]
deps =
  minimum: thirdparty==1.0
  current: thirdparty
  pytest
commands =
    pytest mylibrary
```

This will run *four* different test runs: **py36-minimum**, **py36-current**, **py37-minimum**, and **py37-current**. This is useful in the case where our library depends on **thirdparty>=1.0**: every test run makes sure we are still compatible with the **1.0** version while also making sure the latest version does not break us.

It is also a good idea to run a linter in tox. For example, running Black [3] will do the right thing.

```
[tox]
envlist = py36,py37,py36-black
[testenv]
deps =
    pytest
commands =
    pytest mylibrary
[testenv:py36-black]
deps =
    black
commands =
    black --check --diff mylibrary
```

By default, tox will run all test environments. But you can run just one environment; for example, if you only want to run Black, run **tox -e py36-black**.

If you have a Python library you care about, add **tox.ini** to your workflow to keep its quality high.

Links

[1] https://opensource.com/article/19/5/python-tox
[2] http://www.sphinx-doc.org/en/master/
[3] https://opensource.com/article/19/5/python-black

Ensure consistency in your Python code with **flake8**

flake8 is a linter and linting platform that ensures consistency in Python code.

PYTHON CODE IS MEANT to be easy to read. For this reason, consistency matters. Consistency inside a project matters most of all. How can we enforce such consistency?

Flake8 is really two things: it is both a linter, enforcing some basic rules. Even more important, it is a linting platform that allows plugins to add or change linting rules.

The best thing about flake8 [1] plugins is that you don't need to do anything other than installing them in the virtual environment where you want to run flake8.

Consider the following code:

```
# spew.py
print("Hello world")
# print("Goodbye universe")
```

If we install flake8 in a clean virtual environment and run it, it will say nothing: this file looks fine.

If we install **flake8-print** and run **flake8 spew.py**, we get:

```
spew.py:2:1: T001 print found.
```

If we instead install **flake8-eradicate**, we get:

```
spew.py:1:1: E800: Found commented out code:
```

We can, of course, install both—and get both warnings.

You can also write local, custom plugins. If your team has local conventions that are constantly nit-picked in reviews, why not automate them with a custom flake8 plugin?

Links

[1] https://pypi.org/project/flake8/

Check type annotations in Python with **mypy**

mypy "a Python linter on steroids."

PYTHON IS A "DYNAMICALLY TYPED" language. However, sometimes it is nice to let other beings, both robotic and human, know what types are expected. Traditionally, humans have been prioritized: input and output types of functions were described in docstrings. MyPy [1] allows you to put the robots on equal footing, letting them know what types are intended.

Let's look at the following code:

```
def add_one(input):
    return input + 1

def print_seven():
    five = "5"
    seven = add_one(add_one(five))
    print(seven)
```

Calling **print_seven** raises a **TypeError** informing us we cannot add a string and a number: we cannot add "5" and 1.

However, we cannot know this until we *run* the code. Running the code, if it were correct, would have produced a printout to the screen: a side-effect. A relatively harmless one, as side-effects go, but still, a side-effect. Is it possible to do it without risking any side-effects?

We just have to let the robots know what to expect.

```
def add_one(input: int) -> int:
    return input + 1

def print_seven() -> None:
    five = "5"
    seven = add_one(add_one(five))
    print(seven)
```

We use type annotations to denote that **add_one** expects an integer and returns an integer. This does not change what the code does. However, now we can ask a safe robot to find problems for us.

```
$ mypy typed.py
typed.py:6: error: Argument 1 to "add_one" has incompatible
                   type "str"; expected "int"
```

We have a nice, readable explanation of what we are doing wrong. Let's fix **print_seven**.

```
def print_seven() -> None:
    five = 5
    seven = add_one(add_one(five))
    print(seven)
```

If we run mypy on this, there will not be any complaints; we fixed the bug. This also results, happily, in working code.

The Python type system can get pretty deep, of course. It is not uncommon to encounter signatures like:

```
from typing import Dict, List, Mapping, Sequence

def unify_results(
    results1: Mapping[str, Sequence[int]],
    results2: Mapping[str, Sequence[int]]
) -> Dict[str, List[int]]:
    pass
```

In those cases, remember that everything is an object: yes, even types.

```
ResultsType = Mapping[str, Sequence[int]]
ConcreteResultsType = Dict[str, List[int]]
```

```
def unify_results(results1: ResultsType, results2: ResultsType)
                  -> ConcreteResultsType:
    pass
```

We defined the input types as abstract types (using **Mapping** and **Sequence**). This allows sending in, say, a **defaultdict**, which maps strings to tuples. This is usually the right choice. We also chose to guarantee concrete return types in the signature. This is more controversial: sometimes it is useful to guarantee less in order to allow future changes to change the return type.

MyPy allows *progressive* annotation: not everything has to be annotated at once. Functions without any annotations will not be type-checked.

Go forth and annotate!

Links

[1] https://pypi.org/project/mypy/

WRITE FOR US

In 2010, Red Hat CEO Jim Whitehurst announced the launch of Opensource.com in a post titled *Welcome to the conversation on Opensource.com*. He explained, "This site is one of the ways in which Red Hat gives something back to the open source community. Our desire is to create a connection point for conversations about the broader impact that open source can have—and is having—even beyond the software world." he wrote, adding, "All ideas are welcome, and all participants are welcome. This will not be a site for Red Hat, about Red Hat. Instead, this will be a site for open source, about the future."

By 2013, Opensource.com was publishing an average of 46 articles per month, and in March 2016, Opensource.com surpassed 1-million page views for the first time. In 2019, Opensource.com averages more than 1.5 million page views and 90 articles per month.

More than 60% of our content is contributed by members of open source communities, and additional articles are written by the editorial team and other Red Hat contributors. A small, international team of staff editors and Community Moderators work closely with contributors to curate, polish, publish, and promote open source stories from around the world.

Would you like to write for us? Send pitches and inquiries to open@opensource.com.

To learn more, read 7 big reasons to contribute to Opensource.com.