The 22 Most-Used Python Packages in the World

Educational and surprising insights into how Python is used

Erik-Jan van Baaren
Jan 30 · 13 min read ★

Photo by NASA on Unsplash

How is Python being used around the globe and across industries?

This question inspired me to write this piece. I figured a list of the most-used Python packages would give a good indication.

As a starting point, I took a list of the most downloaded Python packages on PyPI over the past 365 days. Let's dive in and find out what they do, how they're related, and why they rank so high!

. . .

1. Urllib3

893M downloads

`Urllib3` is an HTTP client for Python that brings many features that are missing from the Python standard libraries:

- Thread safety.
- Connection pooling.
- Client-side SSL/TLS verification.
- File uploads with multipart encoding.
- Helpers for retrying requests and dealing with HTTP redirects.
- Support for `gzip` and `deflate` encoding.
- Proxy support for HTTP and SOCKS.

Despite its name, `Urllib3` is not a successor of `urllib2`, which is part of Python's core. If you want to use as many core Python features as possible, perhaps because you're limited to what you can install, then take a look at `urlllib.request`.

For end-users, I strongly recommend the requests package (see #6 on this list). This package is #1 because almost 1200 packages depend on `urllib3`, many of them ranking very high on this list as well.

. . .

2. Six

732M downloads

`six` is a Python 2 and 3 compatibility library. The project is intended to support codebases that work on *both* Python 2 and 3.

If offers a number of functions that smooth the differences in syntax between Python 2 and 3. An easy to grasp examples of this is `six.print_()`. In Python 3, printing is done with the `print()` function, while in Python 2, `print` works without the parentheses. So, by using `six.print_()`, you can support both languages with one statement.

Facts:

- The name, `six`, comes from the fact that two times three equals six.
- For a similar library, also check out the `future` package
- If you want to convert your code to Python 3 (and stop supporting 2), check out `2to3`.

Although I understand its popularity, I hope people will start moving away from Python 2 altogether, especially since Python 2 is officially not supported as of January 1, 2020.

Links: the PyPI page and documentation.

. . .

3. botocore, boto3, s3transfer, awscli

I grouped number of related projects here:

- `botocore` (#3, 660M downloads)
- `s3transfer` (#7, 584M downloads)
- `awscli` (#17 with 394M downloads)
- `boto3` (#22 with 329M downloads)

`Botocore` is a low-level interface to Amazon Web Services. `Botocore` serves as the foundation for the `Boto3` (#22) library, which allows you to make use of services like Amazon S3 and Amazon EC2.

Botocore is also the foundation of `AWS-CLI`, which provides a unified command-line interface to Amazon Web Services.

`S3transfer` (#7) is a Python library for managing Amazon S3 transfers. It's under heavy development and its page basically says not to use it, or at least to pin the version down because the API may change, even between minor versions. `Boto3`, `AWS-CLI`, and many other projects have a dependency on `s3transfer`.

It's fascinating to see that these AWS specific libraries rank this high — it says a lot about how prominent AWS is.

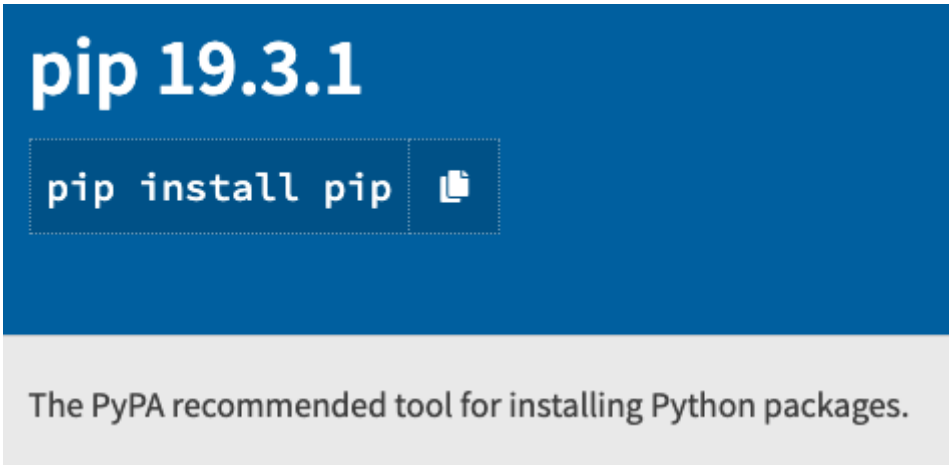
Links:

- `botocore` PyPI page and documentation,
- `Boto3` PyPI page and documentation
- `awscli` PyPI page

. . .

4. Pip

627M downloads



PyPI Screenshot by author

I assume most of you know and love `pip`, the package installer for Python. You can use `pip` to effortlessly install packages from the Python Package Index and other indexes, like a local mirror or custom index with privately-owned software.

Some interesting facts about `pip`:

- **`pip`** is a recursive acronym for "Pip Installs Packages"

- `pip` is very easy to use. Installing a package is as simple as `pip install <package name>` and removing it is accomplished with `pip uninstall <package name>`.
- One of its biggest strengths is that it also takes a list of packages, often in the form of a `requirements.txt` file. This file may optionally include detailed specifications of the required versions. Most Python projects include such a file.
- Using `pip` in combination with `virtualenv` (#57 on the list) allows you to create predictable, isolated environments that won't interfere with your underlying system and vice versa.

. . .

5. Python-dateutil

617M downloads

The `python-dateutil` module provides powerful extensions to the standard `datetime` module. It's my experience that where regular Python `datetime` functionality ends, `python-dateutil` comes in.

You can do so much cool stuff with this library. I'll limit the examples to just one that I found particularly useful: fuzzy parsing of dates from log files and such:

```
from dateutil.parser import parse

logline = 'INFO 2020-01-01T00:00:01 Happy new year, human.'
timestamp = parse(log_line, fuzzy=True)
print(timestamp)
# 2020-01-01 00:00:01
```

dateutil.py hosted with ♡ by GitHub view raw

. . .

6. Requests

611M downloads

`Requests` is built on our #1 library, `urllib3`. It makes web requests *really* simple. Many people prefer it over `urllib3` and it's probably used more by end-users than `urllib3` is. The latter is more low-level and is often a dependency for other projects, because of the level of control over the internals.

Just to show how easy `requests` can be:

```
import requests

r = requests.get('https://api.github.com/user', auth=('user', 'pass'))
r.status_code
# 200
r.headers['content-type']
# 'application/json; charset=utf8'
r.encoding
# 'utf-8'
r.text
# u'{"type":"User"...'
r.json()
# {u'disk_usage': 368627, u'private_gists': 484, ...}
```

requests.py hosted with ♡ by GitHub view raw

Links:

- PyPI page
- Documentation

. . .

7. S3transfer

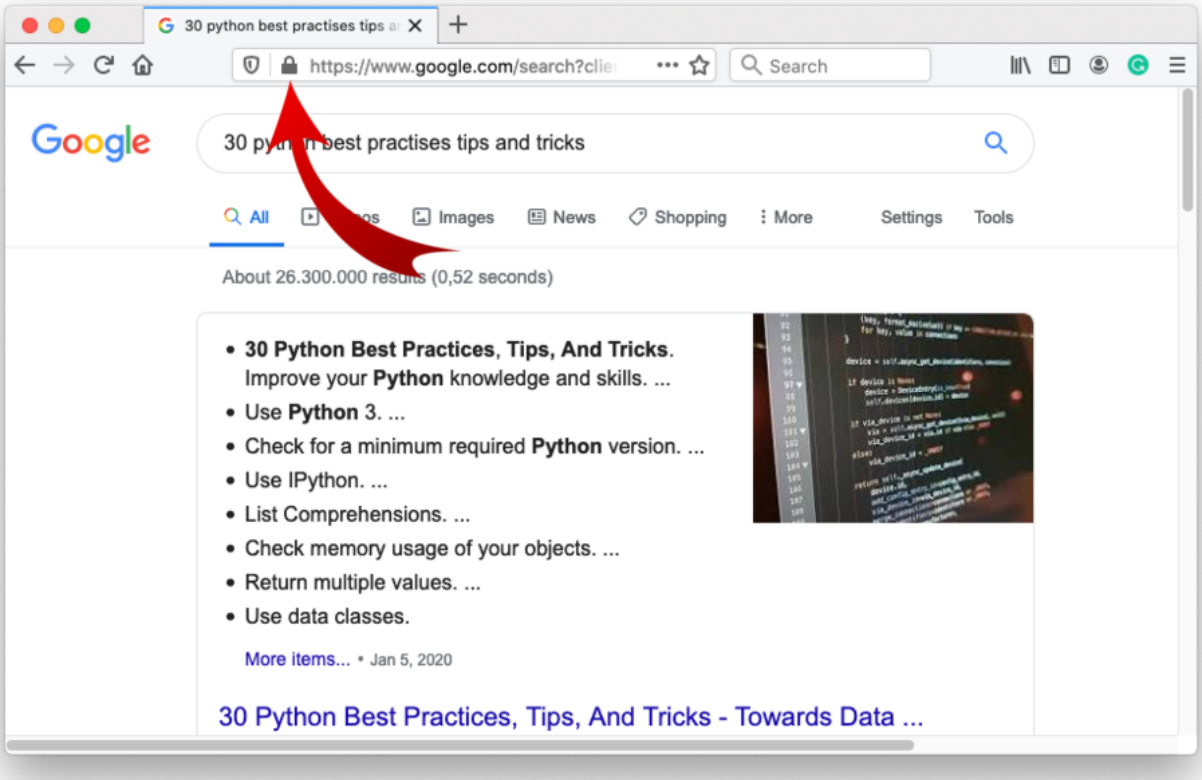
I've combined #3, #7, #17 and #22 since they are all so related. See #3!

. . .

8. Certifi

552M downloads

In recent years, almost all websites moved to SSL, which can be recognized by the little lock symbol in your address bar. It means communication with that site is secure and encrypted, preventing eavesdropping.



The little lock, telling us this site is secured with SSL. Image by author.

The encryption is based on SSL certificates and these SSL certificates are created by trusted companies or non-profits like LetsEncrypt. These organizations digitally sign the certificate with their (intermediary) certificate.

By using the publicly available part of these certificates, your browser is able to verify their signature, so you can be sure you're looking at the real thing and that nobody is snooping on the data.

Python software can do exactly the same. That's where `certifi` comes it. It's not so different from the collection of root certificates that come with web browsers like Chrome, Firefox, and Edge.

`Certifi` ***is a curated collection of root certificates, so your Python code will be able to verify the trustworthiness of SSL certificates.***

Many projects trust and depend on `certifi`, as can be seen here. This is also the reason why this project ranks so high.

Links: certifi PyPI page, documentation, certifi.io

. . .

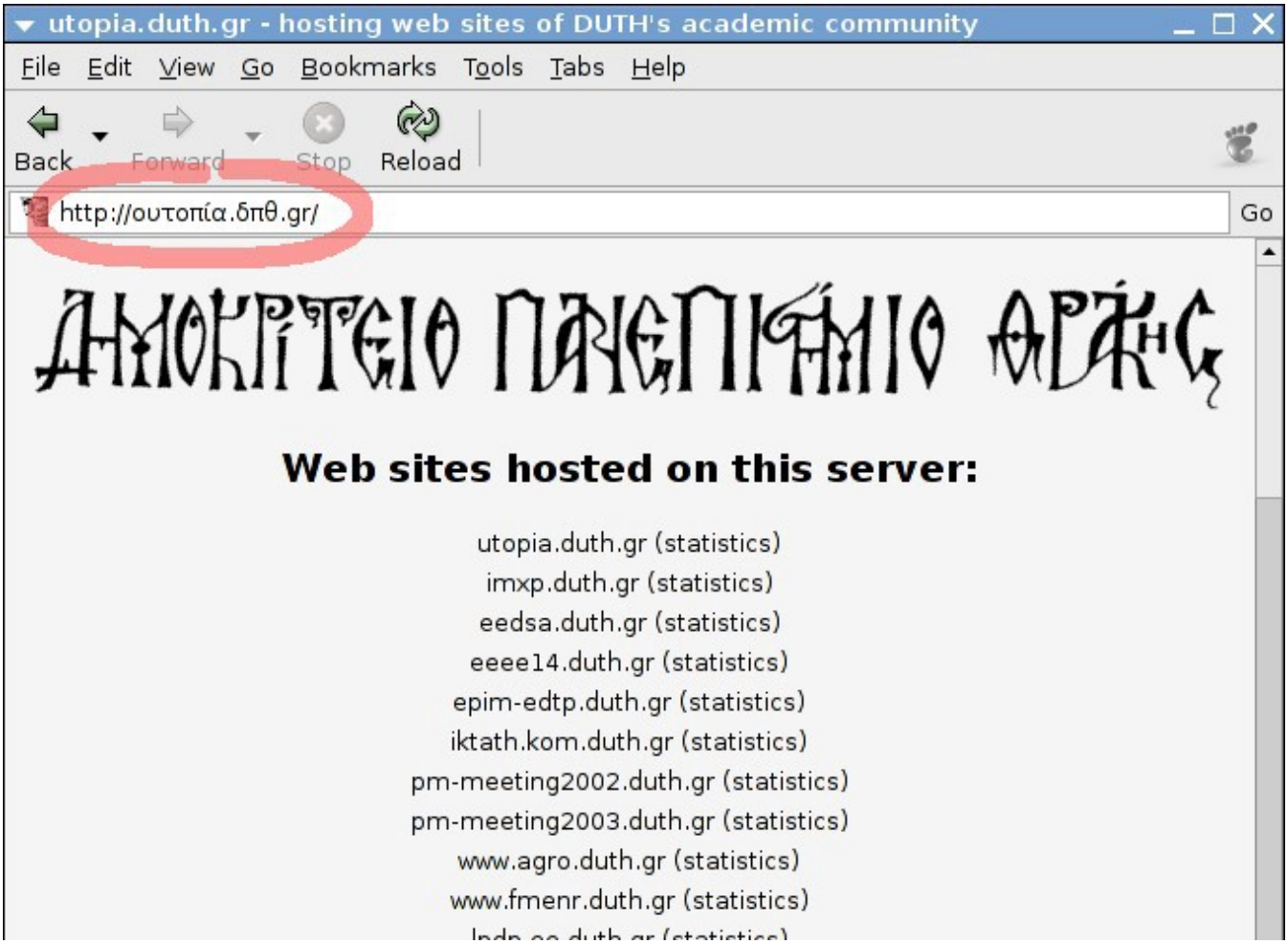
9. Idna

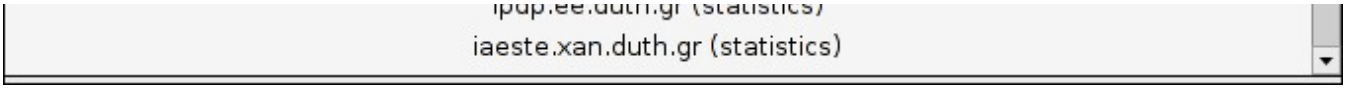
527M downloads

According to the PyPI page, `idna` offers "support for the Internationalised Domain Names in Applications (IDNA) protocol as specified in RFC 5891."

If you're anything like me, you still have no idea what `Idna` is or does! Lucky for you, yours truly did the grunt work of finding it out!

Internationalized Domain Names in Applications (IDNA) is a mechanism for handling domain names containing non-ASCII characters. But the original domain name system already offered support for non-ASCII based domain names. So what's the problem?





By Adamantios — Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1630902

The problem is that *applications*, like e-mail clients and web browsers, do not support non-ASCII characters. Or more specifically, the protocols for email and HTTP don't support these characters.

That was fine for many countries, but a problem for countries like China, Russia, Germany, Greece, Indonesia, etc. So, not entirely coincidentally, a bunch of smart people from these countries came up with `IDNA`.

At the core of `IDNA` are two functions: `ToASCII` and `ToUnicode`. `ToASCII` will translate an international, Unicode domain into an ASCII string. `ToUnicode` will reverse that process. In the `IDNA` package, these functions are called `idna.encode()` and `idna.decode()`, as can be seen in the following snippet:

```
1  import idna
2  idna.encode('ドメイン.テスト')
3  # b'xn--eckwd4c7c.xn--zckzah'
4  print(idna.decode('xn--eckwd4c7c.xn--zckzah'))
5  # ドメイン.テスト
```

idna.py hosted with ♡ by GitHub view raw

Idna at work with Chinese domains

You can read RFC-3490 for the details of this encoding, if you're a masochist.

Links: Idna PyPI page, GitHub page

. . .

10. PyYAML

525M downloads

`YAML` is a data serialization format. It's designed for both human and computer readability — it's easy to read and write for humans but computers can still parse it.

```
repos:
```

```yaml
-   repo: https://github.com/psf/black
    rev: 19.10b0
    hooks:
    -   id: black
        args:
          - --safe
          - --quiet
        files: ^((homeassistant|script|tests)/.+)?[^/]+\.py$
-   repo: https://github.com/PyCQA/flake8
    rev: 3.7.9
    hooks:
    -   id: flake8
        additional_dependencies:
          - flake8-docstrings==1.5.0
          - pydocstyle==5.0.2
        files: ^(homeassistant|script|tests)/.+\.py$
```

Example of YAML, image by author

`PyYAML` is a `YAML` parser and emitter for Python, which means it can read and write `YAML`. It will write any Python object to `YAML`: lists, dictionaries, and even class instances.

Python offers its own config parser, but YAML offers a lot more compared to the basic `.ini` file structure of Python's `ConfigParser`.

For example, `YAML` can store any data type: `boolean`s, `list`s, `float`s, et cetera. `ConfigParser` will store everything as a string internally. If you want to load an integer with `ConfigParser`, you'll need to specify that you want to get an `int` explicitly:

```
config.getint("section", "my_int")
```

While `pyyaml` automatically recognizes the type, so this will return your `int` with `PyYAML`:

```
config["section"]["my_int"]
```

`YAML` also allows arbitrary deep trees, not something every project needs, but it can come in handy.

It's up to you to decide what you prefer, but many projects use `YAML` for their configuration file(s), hence the popularity of this project.

Links: PyYAML PyPI page, documentation.

. . .

11. Pyasn1

512M downloads

Like `IDNA` above, this project also has one of those super helpful descriptions:

> *Pure-Python implementation of ASN.1 types and DER/BER/CER codecs (X.208).*

Fortunately, there's lots of info to be found on this decades-old standard. `ASN.1`, short for Abstract Syntax Notation One, is like the godfather of data serialization. It comes from the telecommunications world. Perhaps you know protocol buffers or Apache Thrift? This is, literally, the 1984 version of those.

ASN.1 describes the cross-platform interface between systems and the data structures that can be sent through this interface.

Remember Certifi (see #8)? ASN.1 is used to define the format of certificates used in the HTTPS protocol, and in many other cryptographic systems. It's also used in SNMP, LDAP, Kerberos, UMTS, LTE, and VOIP protocols.

It's a specification that's very complex and some implementations have proven to be full of vulnerabilities. You may also like this interesting Reddit thread about ASN.1.

I recommend staying away unless you really need it. But, since it's used in so many places, lots of packages are dependent on this one.

. . .

12. Docutils

508M downloads

`Docutils` is a modular system for processing plaintext documentation into useful formats, such as HTML, XML, and LaTeX. `Docutils` is able to read plain text documents in the `reStructuredText` format — an easy-to-read markup syntax similar to MarkDown.

You probably have heard about PEP documents, or even read one. So what is a PEP document? The very first PEP document called PEP-1 explains it well for us:

> *PEP stands for Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. The PEP should provide a concise technical specification of the feature and a rationale for the feature.*

PEP documents are written in a fixed `reStructuredText` template, and converted using `docutils` to nicely formatted documents.

Docutils is also at the core of `Sphinx`. `Sphinx` is used to create documentation projects. If `Docutils` is a machine, `Sphinx` is the factory. It was originally created to build Python documentation but many other projects use it to document their code.

You've probably read documentation on readthedocs.org, right? Most of the documentation on there is created by `Sphinx` and `docutils`.

13. Chardet

501M downloads

You can use the `chardet` module to detect the charset of a file or data stream. This can come in useful when analyzing big piles of random text, for example. But it can also be used when working with remotely downloaded data where you don't know what the charset is.

After installing `chardet`, you also have an extra command-line tool called `chardetect`, which can be used like this:

```
chardetect somefile.txt
somefile.txt: ascii with confidence 1.0
```

You can also use the library programmatically, check out the docs.

`Chardet` is a requirement for `requests` and many other packages. I don't think many people use `chardet` on its own, so its popularity must come from these dependencies.

14. RSA

492M downloads

The `rsa` package is a pure-Python RSA implementation. It supports:

- encryption and decryption,
- signing and verifying signatures,
- key generation according to PKCS#1 version 1.5.

It can be used as a Python library as well as on the command-line.

Some facts:

- The letters in RSA are initial letters of the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman. They described the algorithm in 1977.
- RSA is one of the first public-key cryptosystems and is widely used for secure data transmission. In such a cryptosystem, there are two keys: a public part and a private part. You encrypt data with the public key, which can then only be decrypted with the private key.
- RSA is a slow algorithm. It is less commonly used to directly encrypt user data. Often RSA is used to securely pass a shared key for symmetric key cryptography, which is much faster at encryption and decryption of large amounts of data.

The following code snippet show how RSA can be used for a very simple use-case:

```
import rsa

# Bob creates a key pair:
(bob_pub, bob_priv) = rsa.newkeys(512)

# Alice ecnrypts a message for Bob
# with his public key
crypto = rsa.encrypt('hello Bob!', bob_pub)

# When Bob gets the message, he
# decrypts it with his private key:
message = rsa.decrypt(crypto, bob_priv)
print(message.decode('utf8'))
# hello Bob!
```

rsa.py hosted with ♡ by GitHub view raw

Assuming Bob kept his private key *private*, Alice can be sure that he is the only one who can read the message.

Bob, however, does *not* know for sure that it was Alice that sent the message since anyone can get and use his public key. To prove it was her, Alice could have signed the message with her private key. Bob can verify this signature with her public key, ensuring it was really her sending the message.

Packages like `google-auth` (#37), `oauthlib` (#54), `awscli` (#17) depend on the `rsa` package. Not many people will be using this one as a stand-alone tool since there are faster, more native alternatives.

. . .

15. Jmespath

473M downloads

Using JSON in Python is super easy since JSON maps so well on a Python dictionary. For me, it's one of its best features.

Screenshot by author

I'll be honest here — I never heard of this package, even though I've worked a lot with JSON. I would just use `json.loads()` and get data from the dictionary manually, perhaps with a loop here and there.

`JMESPath`, pronounced "James path", makes JSON in Python even easier. It allows you to declaratively specify how to extract elements from a JSON document. Here are some basic examples to give you a feeling for what it can do:

```python
import jmespath

# Get a specific element
d = {"foo": {"bar": "baz"}}
print(jmespath.search('foo.bar', d))
# baz

# Using a wildcard to get all names
d = {"foo": {"bar": [{"name": "one"}, {"name": "two"}]}}
print(jmespath.search('foo.bar[*].name', d))
# [“one”, “two”]
```

jmespath.py hosted with ♡ by GitHub view raw

This is just touching the surface of all its possibilities. See the documentation and the PyPI page for more.

. . .

16. Setuptools

401M downloads

Setuptools is what you use to create a Python package.

This project is badly documented. It doesn’t describe what it is and it contains dead links in its description. The best source of info is this site: https://packaging.python.org/, and in particular this guide to creating a Python package: https://packaging.python.org/tutorials/packaging-projects/.

. . .

17. `Awscli`

I’ve combined #3, #7, #17 and #22 since they are all so related. See #3!

. . .

18. Pytz

394M downloads

Like `dateutils` (#5), this library helps you to work with dates and times. Working with time zones can be difficult. Luckily, there are packages like these to make it easier.

My experience with time and computers drills down to this: ***always use UTC internally. Convert to local time only when generating output to be read by humans.***

Here's an example `pytz` usage:

```
from datetime import datetime
from pytz import timezone

amsterdam = timezone('Europe/Amsterdam')

ams_time = amsterdam.localize(datetime(2002, 10, 27, 6, 0, 0))
print(ams_time)
# 2002-10-27 06:00:00+01:00

# It will also know when it's Summer Time
# in Amsterdam (similar to Daylight Savings Time):
ams_time = amsterdam.localize(datetime(2002, 6, 27, 6, 0, 0))
print(ams_time)
# 2002-06-27 06:00:00+02:00
```

pytz.py hosted with ♡ by GitHub view raw

Check out the PyPI page for more examples and documentation.

. . .

19. Futures

389M downloads

Since Python 3.2, python has offered the `concurrent.futures` module, which helps you to perform asynchronous execution. The futures packages is a backport of this library, meant for Python 2. ***It is not meant for Python 3 users since Python 3 offers it natively.***

As I mentioned before, Python 2 is officially not supported as of January 1, 2020. When I revisit this piece next year, I truly hope this package won't be in the top 22.

Here's a basic example of `futures` in use:

```python
from concurrent.futures import ThreadPoolExecutor
from time import sleep

def return_after_5_secs(message):
  sleep(5)
  return message

pool = ThreadPoolExecutor(3)

future = pool.submit(return_after_5_secs,
                     ("Hello world"))

print(future.done())
# False
sleep(5)
print(future.done())
# True
print(future.result())
# Hello World
```

futures.py hosted with ♡ by GitHub view raw

As you can see, you can create a pool of threads and submit a function to be executed by one of these threads. In the meantime, your program will continue running in the main thread. It's an easy way to parallelize the execution of your program.

. . .

20. Colorama

370M downloads

With Colorama, you can add some color to your terminal:

Screenshot by Jonathan Hartley from Colorama

To get a feel for how easy this is, here's some example code:

```
from colorama import Fore, Back, Style

print(Fore.RED + 'some red text')
print(Back.GREEN + 'and with a green background')
print(Style.DIM + 'and in dim text')
print(Style.RESET_ALL)
print('back to normal now')
```

colorama.py hosted with ♡ by GitHub view raw

. . .

21. Simplejson

341M downloads

What's wrong with the native `json` module in Python that requires this high ranking alternative? Nothing! In fact, Python's `json` is `simplejson`. But `simplejson` has some advantages:

- It works on more Python versions.
- It is updated more frequently than Python.
- It has (optional) parts that are written in C, making it very fast.

Something you will often see in scripts that work with JSON is this:

```
try:
  import simplejson as json
except ImportError:
  import json
```

Unless you specifically need something that is not in the standard library, I would just use `json`. `Simplejson` can be a lot faster than `json`, because it has some parts implemented in C. This speed will not be an issue for you, unless you are working with thousands of JSON files. Also check out UltraJSON, which is supposed to be even faster because almost all of it is written in C.

. . .

22. Boto3

I've combined #3, #7, #17 and #22 since they are all so related. See #3!

. . .

Final Notes

It's hard to stop writing after 22 packages because many of the ones that follow are some of the most interesting ones for end-users like us. They'll get their chance to shine in another piece I have planned!

Building this list gave me these insights:

- Many of the top-ranking packages offer core functionality of some sort — like working with time, configuration files, encryption, and standardization. They are often a dependency for other projects.
- A common theme is connectivity. Most of these packages allow you to either connect to servers and services or support other packages in doing so.
- The rest are extensions to Python. Tools to create Python packages, tools that help to create documentation, libraries that create compatibility between versions, etc.

I hope you enjoyed this list and perhaps learned something new from it — I sure did!

Python Programming Software Engineering Data Science Software Development

About Help Legal