DPI

De Programmatica *Ipsum*

DE PROGRAMMATICA IPSUM

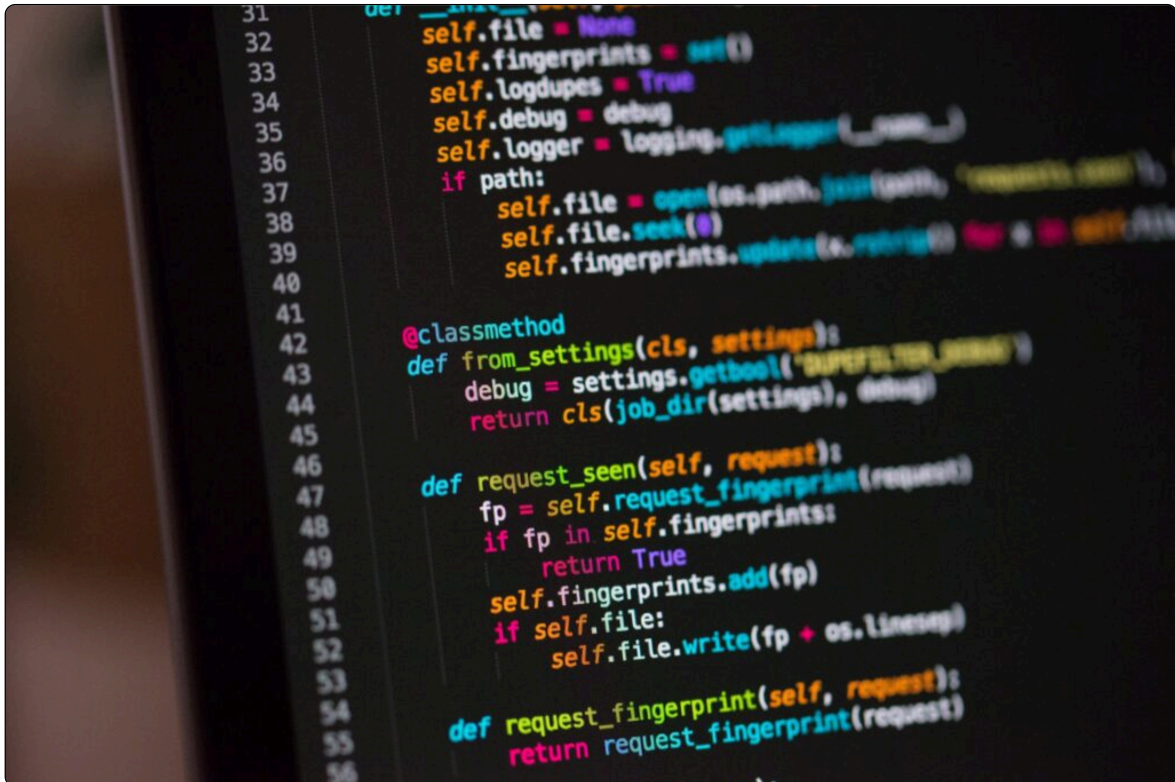
Issue 035: Python

August 2nd, 2021

Table of Contents

Issue 035: Python 5
Majoring In Versions 9
The State Of Python In 2021 15
Tim Peters 25

Issue 035: Python



August 2nd, 2021

Welcome to the thirty-fifth issue of *De Programmatica Ipsum*, dedicated to the subject of *Python*. In this edition:

- Graham describes the pains caused by the decade-long migration from Python 2 to 3[1].
- Adrian describes the preeminence of Python[2] in the software industry of 2021.
- In the Library section[3], Graham types `import this` in the REPL and recites the Zen of Python[4] by Tim Peters.

Enjoy this issue! Please subscribe to our free newsletter[5] to stay updated about new releases, share the articles on social media, or contribute[6] if you would like to support our work.

Cover photo by Chris Ried[7] on Unsplash[8].

References

[1] https://deprogrammaticaipsum.com/majoring-in-versions/

[2] https://deprogrammaticaipsum.com/the-state-of-python-in-2021/

[3] https://deprogrammaticaipsum.com/category/library/

[4] https://deprogrammaticaipsum.com/tim-peters/

[5] https://deprogrammaticaipsum.com/newsletter/

[6] https://deprogrammaticaipsum.com/contribute/

[7] https://unsplash.com/@cdr6934?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[8] https://unsplash.com/s/photos/python?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Majoring In Versions

By Graham Lee

For many people, both inside and outside the Python community, even mentioning the Python language brings to mind the Python 2-3 transition. Let us see what happened.

I first got involved with Python in a project to "port" my Physics Department's undergraduate computing teaching to a new language, back in 2003. The computing practicals were taught in Pascal using the

Free Pascal Compiler[1] up until then. There were various choices for alternative languages, some more modern, some more useful, some both: eventually a bake-off was organised between Python (more modern, arguably useful in 2003) and C (less modern, definitely useful).

At the time, Python was thought of as a "scripting" language along with the Bourne shell, C shell, Perl, Tcl, and others. "Scripting language" does not actually mean anything. It is said by people who want to imply that a programming language is less worthy somehow because it is easier to use. LAMP was already in common parlance back in 2003 so it was already clear that real web applications could be, and were being, developed in Perl, Python and the like.

It is a tangent at this point but Python won out over C in that Physics teaching bake-off. For the kinds of problems we covered in undergraduate computing for Physicists, each language was as easy as the other to write, and students got just as far. But Python was way easier to read, as the enforced whitespace organisation unified, to some extent, the presentation of the code. Demonstrators reading over the shoulders of 20 different undergrads trying to debug their programs found it way easier to spot problems in the Python.

Anyway, all of this means that when I was new to Python, version 1 of the language was well-established, version 2 was fairly broadly adopted, and talk was turning to the future. The community had a name for this future: Python 3000.

Python 3000 was a project that took a number of different concerns into account. Firstly, that Python was broken. Not egregiously broken, but broken-ish. Various choices, made deliberately or accidentally, were seen to be suboptimal and Guido and the community saw that if they were allowed to break everything, they would do things differently.

Secondly, that even though things are currently broken, quite a few people seem to enjoy using them thank you very much. And that is why it was set in the future, version 3000. None of these breaking changes would come in version 2, and discussion of what would come down the line was happening in the open, via the Python Enhancement Process.

In fact, third, not only were the breaking changes discussed in the open, but they were implemented in Python 2 so that you could test and migrate. If new syntax was added that was simply erroneous for Python 2, it just went into Python 2 directly (the `with` keyword for context managers is such a case). You could carry on without knowing it was there, or change and get the modern stuff. For changes that made the language work differently, you could `from **future** import breaking_change` and live tomorrow's Python, today.

This whole Python 3000 discussion probably started around the year 2001, when it was "intentionally vapourware" as Mark Lutz described it. Five years later, around the beginning of 2006, PEP-3000[2] was published documenting the process for implementing Python 3000, the fact that it would be synonymous with Python 3, and finally kickstarting the implementation phase. The future was becoming real!

Python 2 was still maintained alongside the development of Python 3, and even after it had stabilised. The last release of the 2.7 lineage was in April 2020, two decades after the first release, two decades after the discussions of py3k started, 14 years after the migration path was published, and 11 years after the release of version 3.

And people still felt that they had not had enough warning.

In fairness, some of them had not. They were not Pythonistas per se, they were computer users who happened to engage with Python when using a computer. Climate scientists, perhaps, who relied on their library vendors and their site administrators to keep everything ticking over. But they did not realise that their library vendors did not have funding for maintenance, and that their site administrators were relying on the operating system package maintainers.

The operating system package maintainers did not dare upgrade the default Python package, because that would break people's scripts. Better to ship version 2 tomorrow so that everybody's programs from yesterday carry on running.

Nobody was responsible for migrating to Python 3, so nobody did. It was not until 2.7, when people were finally told that this was the final release of Python 2, that these Pythonistas noticed the corner that they were painted into. They were not happy.

Of course, this story is not necessarily over. Somebody, or various somebodies, may well fork Python 2 and carry on releasing updates, becoming the Mate to Python 3's GNOME.

This model has worked well for Perl. The Perl language, currently on version 5.34.0, is a different thing from the Raku language (formerly Perl 6), on version 6.d. Some people use both, some prefer one over the other, and they coexist peacefully. Raku started out as the successor to Perl, and gradually evolved into her sister. But still other people are turned off of Perl/Raku by confusion over whether they are "supposed" to use 5 or 6 (the answer, of course, being "you are not *supposed* to use either": and besides which I expect they are put off Perl/Raku because it is not faddish enough and the version thing is a convenient smokescreen).

The model that has worked for C++ is never (or rarely, anyway) taking away people's toys. They might get warnings on, saying "this toy is now known to be a choking hazard and you should never give it to your children", but they never issue a manufacturer's recall. You are not supposed to use `new` or `delete` any more, you are supposed to use smart pointers, but nobody turned off `new` and `delete`.

For a counter example, the model that has worked for Swift is YOLO. The language got changed once a year, people guessed what some of the changes were going to be based on reading the community proposals tea leaves. An automated tool rewrote some of your existing Swift in the new version of the syntax, and people were *still* excited to rewrite all the bits that the tool missed.

You can never give people too much warning about a change. Giving a fixed time notice of a change, however long that time, paints people into a corner. Prefer that the lead time be infinite, except where that paints people into a corner.

Cover photo by Manasvita S[3] on Unsplash[4].

References

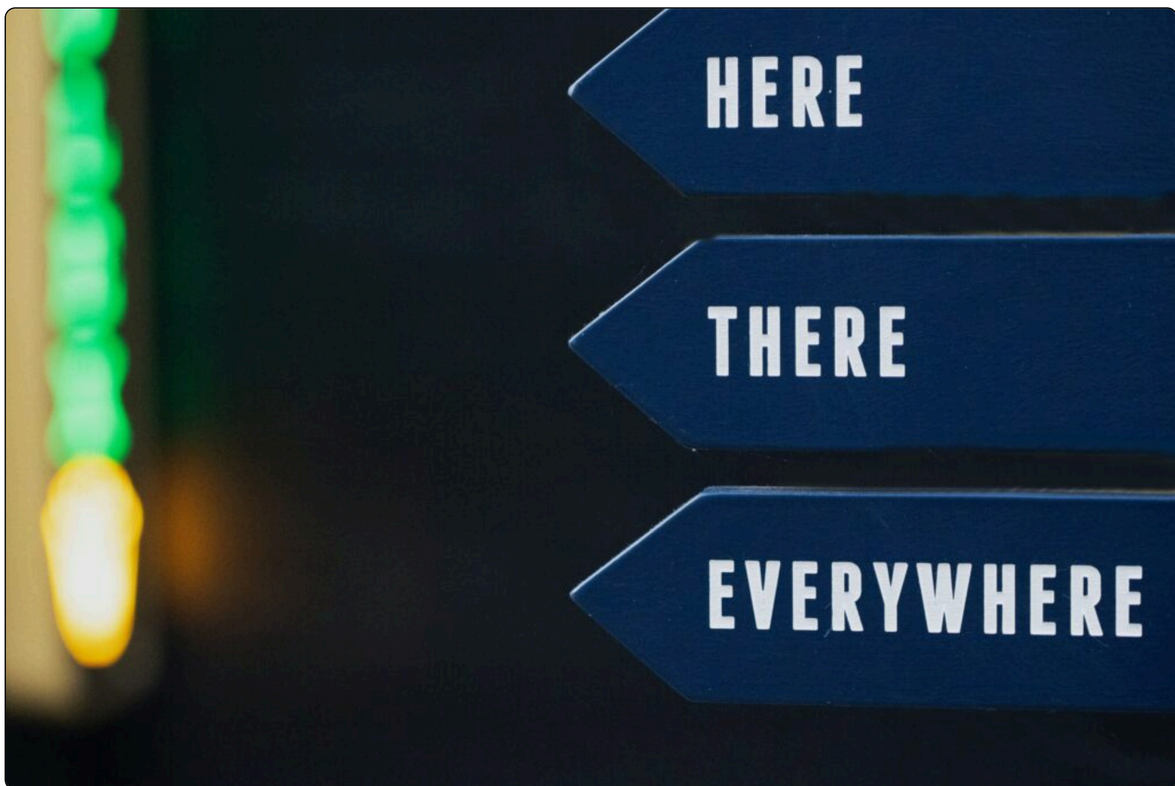
[1] https://www.freepascal.org

[2] https://www.python.org/dev/peps/pep-3000/

[3] https://unsplash.com/@manasvita?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[4] https://unsplash.com/s/photos/calendar?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

The State Of Python In 2021



By Adrian Kosmaczewski

At the risk of alienating most of the readership of this magazine, here is a confession. I hated Python for a very long time. My issue was not with the language *per se*, even though the indentation rules have put me off for a while. No, the reason I kept myself away from Python was the unfortunate contact with some (too many for my taste) hubris-filled Python developers. Well, that, and the *no man's land* that was the transition between Python 2 and 3.

That was, of course, unfortunate. But after a few years, pragmatism naturally brought me back to Python again; not so much the language *per se*, which I now neither like nor dislike, but the sheer amount of libraries available (and their excellent quality). Another big contributing factor to this renewed interest in Python was the long awaited announcement of the deprecation of Python 2; this event signals a major point of maturity in the language, and one that was badly required.

The ecosystem around Python is outstanding, and the rise of Machine Learning cemented its status as the new Fortran, while at the same time it became the new pseudocode, or even, the new BASIC.

These days it is not only hard, but unwise for a developer to ignore Python. This article is the product of six months of exploration in online and offline media, to find out how much Python has permeated our craft. The result was a surprise even to this author. Python is not only everywhere, it has become a "default" programming language for the computer industry.

The Ecosystem

As stated above, the first thing that brought me into Python was its libraries. I needed to get stuff done (who does not?) and Python had precisely the libraries I needed. Its standard csv[1] package was the first one I explored and used with intensity, almost 20 years ago; next came argparse[2], doctest[3], and shutil[4], all part of the standard library.

Manipulating data in CSV format took me to export it to other formats; Python-OpenXML[5] provided a first solution, later enhanced with Pandas[6] to manipulate Excel files.

Little by little, I started to use Python more and more. At some point I got a job as a Django[7] application developer, a framework pompously marketed "for perfectionists with deadlines"; a punchline filled with hubris and undeserved merit. Thankfully these days Cloud Native Python apps are built around less monolithic options such as Flask[8], Pyramid[9], or Tornado[10].

In terms of IDEs, PyCharm[11], Spyder[12], or Wing[13], are fighting for the attention of developers, many of whom are migrating to Visual Studio Code with Microsoft's

own Python extension[14] (who by the way are very interested in the Python ecosystem lately[15]).

Developers might be interested to use mypy[16], Flake8[17] or Pylint[18] to check the quality of their code.

Most Python scripts start and end their life in the command line. But for those interested in providing a GUI, choices abound: Kivy[19] for touchscreen apps, the venerable wxPython[20], the powerful Qt for Python[21], and its often overlooked sibling PySide2[22]. In the experimental part of the spectrum, REMi[23], and Gooey[24] deserve special mentions; the former turning scripts into fully-fledged web apps, and the latter transforming any CLI script into a desktop GUI application.

Python has also strong interoperability with other languages and runtimes. Suffice to mention SWIG[25] and Boost.Python[26], enabling interoperability with C and C++; and Jython[27], enabling its use in the JVM–still lacking support for Python 3 at the time of this writing.

Speaking about alternative implementations of the language, one must also mention PyPy[28], recommended by Guido van Rossum for its speed, and IronPython[29] for .NET, whose Python 3 alpha implementation was recently released[30].

Python is a fertile ground for package managers and build systems: Conan[31], Poetry[32], Wheels[33], redo[34] (a recursive build system), and of course the venerable pip[35], together with the more recent pipx[36] (an equivalent to JavaScript's `npx` command).

Many ground-breaking ideas have been powered by Python lately: Ansible[37]; asciinema[38]; the Atheris[39] code coverage tool; authentik[40] (replacement for Keycloak); Beautiful Soup[41] to parse HTML; bump2version[42]; Celery[43] for distributed queue processing; Click[44] and Typer[45] to replace `argparse`; Cookiecutter[46]; Depix[47] to recover passwords from pixelized screenshots; Diagrams[48]; Ecco[49] to explore NLP models; Glances[50] to monitor your servers; GOMP[51] to compare Git branches; IceCream[52]; jrnl[53] to write your journals; lorem[54] to create placeholder text; Outrun[55] to run commands in a separate machine; Pillow[56] to manipulate images; pdoc[57] to generate API docs; PyGithub[58]; PyYAML[59]; qrcode[60]; rainbowstream[61]; Ramanujan Machine[62]; Requests[63] for your HTTP needs;

Rich[64] for formatting text in the console; TOML[65] because there are not enough configuration languages; Vulture[66] to find dead code; youtube-dl[67] to download videos online; Weechat[68] Matrix protocol client; whereami[69]; the Whoosh[70] search library; and so much more.

If Rust seems to be the language of the great rewrite[71], Python is definitely the language of the first release.

Machine Learning and Scientific Computing

In the past decade, Python single-handledy replaced Fortran as the programming language of choice for scientific computing. It all started with Numpy[72], and then followed with Jupyter[73] notebooks, a concept loosely based on Knuth's idea of literate programming[74]. Followed Anaconda[75], SciPy[76], PyTorch[77], and the rest is history. In particular, the rise in popularity of Machine Learning[78] owes a great deal to the pre-existence of these tools.

Getting Python

Installing Python is still a non trivial task, and usually the biggest hurdle for developers new to the ecosystem, particularly given the sheer number of contradicting advice online. May this section remove some of the confusion, through proven techniques this author has used to install Python in different environments.

Windows users can simply use Chocolatey[79] to get their Python environment installed, and ready to use. Mac users are better served by not using Homebrew[80] for it; the author of this text heartily recommends using pyenv[81] instead for both Mac and Linux users.

No matter which technique you use to install Python, always use a virtual environment[82] per project. This ensures they are self contained, with well-defined dependencies, and are guaranteed to work in other people's computers.

```
$ python3 -m venv .venv
$ source .venv/bin/activate
$ pip install PyYAML
```

```
$ pip install PyGithub
$ pip freeze > requirements.txt

# Later on...
$ pip install -r requirements.txt
```

Virtual environments also help developers get their code inside Docker containers easily; remember to pass the `--no-cache-dir` option to `pip install` in your Dockerfiles. Never forget to add a proper .gitignore file[83] to your project.

Learning Python

There is no shortage of material to learn Python. The following books might serve as a starting point, in the order specified:

- Python Programming: An Introduction to Computer Science[84] by John M. Zelle, Ph.D.
- The Python Guide for Beginners[85] by Renan Moura.
- Effective Python[86] by Brett Slatkin.
- Beej's Guide to Python Programming[87] by Brian "Beej Jorgensen" Hall.
- Business Python[88] by Theodore Deden.
- Create GUI Applications with Python & Qt5–PyQt5 Edition[89] and PySide2 Edition[90] by Martin Fitzpatrick.

If you are interested in online tutorials, I will only mention here Python basics[91] and Real Python[92], but a short DuckDuckGo search will bring many more resources.

Guido van Rossum

The creator of Python[93], also known as its "Benevolent Dictator for Life", has had quite an illustrious career, closely monitored by the press. In the past 15 years he spent his time between Google and Dropbox. In the former he contributed to the creation of the Mondrian[94] code review tool, written in 2006 to replace[95] a mail-based workflow. Later on, he arguably kicked-off the first popular PaaS, Google AppEngine[96], whose first supported runtime was, of course, Python[97].

A curious fact: Python had not yet been featured in the HOPL conferences[98] created by Jean Sammet[99]. Interestingly enough, Mr. Van Rossum started working[100] in such paper in 2006, to be presented in HOPL III[101], but after much back and forth, he used the material to start a new blog about the history of Python[102] instead.

After Google, Mr. Van Rossum went to Dropbox, from 2013[103] to his retirement[104] in October 2019. At the time of this writing he is said to be working in the mypy[105] static type checker–or at least, that is what the project blog[106] says.

More

Of course, not everything made with Python was successful. Suffice to mention the GadflyB5[107] SQL relational database, or Mercurial[108]; excellent tools in their own right, but shadowed by much more popular options. reStructuredText[109] also is generally dismissed, developers usually prefering Markdown or Asciidoc.

Still, Python became endless material for memes[110] that make[111] developers[112] laugh[113]. There is a Zen of Python[114]. Python made developers sing[115]. The Texas Instrument TI-84 Plus CE-T Python Edition[116] calculator can edit and run Python code. People use Python inside Excel[117] spreadsheets instead of VBA. They create Gimp plugins[118] or render movies[119] with Python. They write serverless[120] applications with it to process 558K transactions in 5 minutes[121]. It was named the TIOBE Programming Language of the Year[122] in 2007, 2010, 2018 and 2020. Its syntax has been translated from English to German[123], Chinese, Javanese, Lithuanian, Spanish, Swedish, and Russian[124], among others[125]. People ask themselves WTF Python[126].

Even though I do not hate Python anymore, I still facepalm every time I type `quit` in the REPL and get the tone-deaf `"Use quit() or Ctrl-D (i.e. EOF) to exit"` message instead.

Cover photo by Nick Fewings[127] on Unsplash[128].

References

[1] https://docs.python.org/3/library/csv.html
[2] https://docs.python.org/3/library/argparse.html
[3] https://docs.python.org/3/library/doctest.html
[4] https://docs.python.org/3/library/shutil.html
[5] https://github.com/python-openxml
[6] https://pandas.pydata.org/
[7] https://www.djangoproject.com/
[8] https://flask.palletsprojects.com/
[9] https://trypyramid.com/
[10] https://www.tornadoweb.org/
[11] https://www.jetbrains.com/pycharm/
[12] https://www.spyder-ide.org/
[13] https://wingware.com/
[14] https://marketplace.visualstudio.com/items?itemName=ms-python.python
[15] https://www.techrepublic.com/article/microsoft-is-boosting-its-support-for-the-python-programming-ecosystem/
[16] http://mypy-lang.org/
[17] https://flake8.pycqa.org/en/latest/
[18] https://www.pylint.org/
[19] https://kivy.org/
[20] https://wxpython.org/
[21] https://www.qt.io/qt-for-python
[22] https://coderslegacy.com/pyside-vs-pyqt-difference/
[23] https://github.com/dddomodossola/remi
[24] https://github.com/chriskiehl/Gooey
[25] http://swig.org/
[26] https://www.boost.org/doc/libs/1_75_0/libs/python/doc/html/index.html
[27] https://www.jython.org/
[28] https://www.pypy.org/
[29] https://ironpython.net/
[30] https://github.com/IronLanguages/ironpython3/releases/tag/v3.4.0-alpha1
[31] https://conan.io/
[32] https://python-poetry.org/
[33] https://pythonwheels.com/
[34] https://github.com/apenwarr/redo/
[35] https://pypi.org/project/pip/
[36] https://github.com/pipxproject/pipx
[37] https://www.ansible.com/
[38] https://asciinema.org/

[39] https://github.com/google/atheris
[40] https://github.com/beryju/authentik
[41] https://www.crummy.com/software/BeautifulSoup/
[42] https://pypi.org/project/bump2version/
[43] https://pypi.org/project/celery/
[44] https://click.palletsprojects.com/en/7.x/
[45] https://github.com/tiangolo/typer
[46] https://cookiecutter.readthedocs.io
[47] https://github.com/beurtschipper/Depix
[48] https://diagrams.mingrammer.com/
[49] https://www.eccox.io/
[50] https://nicolargo.github.io/glances/
[51] https://github.com/MarkForged/GOMP
[52] https://github.com/gruns/icecream
[53] https://github.com/jrnl-org/jrnl
[54] https://github.com/per9000/lorem
[55] https://github.com/Overv/outrun
[56] https://python-pillow.org/
[57] https://pdoc.dev/
[58] https://pygithub.readthedocs.io/
[59] https://pyyaml.org/wiki/PyYAMLDocumentation
[60] https://pypi.org/project/qrcode/
[61] https://github.com/orakaro/rainbowstream
[62] https://github.com/ShaharGottlieb/MasseyRamanujan/
[63] https://requests.readthedocs.io/en/master/
[64] https://github.com/willmcgugan/rich
[65] https://github.com/toml-lang/toml/
[66] https://github.com/jendrikseipp/vulture
[67] https://youtube-dl.org/
[68] https://github.com/poljar/weechat-matrix
[69] https://github.com/kootenpv/whereami
[70] https://github.com/whoosh-community/whoosh
[71] https://deprogrammaticaipsum.com/the-great-rewriting-in-rust/
[72] https://numpy.org/
[73] https://jupyter.org/
[74] https://www-cs-faculty.stanford.edu/~knuth/lp.html
[75] https://www.anaconda.com/
[76] https://www.scipy.org/
[77] https://pytorch.org/
[78] https://deprogrammaticaipsum.com/issue-11-artificial-intelligence/
[79] https://community.chocolatey.org/packages/python

[80] https://justinmayer.com/posts/homebrew-python-is-not-for-you/
[81] https://github.com/pyenv/pyenv
[82] https://docs.python.org/3/tutorial/venv.html
[83] https://www.toptal.com/developers/gitignore/api/python
[84] https://mcsp.wartburg.edu/zelle/python/
[85] https://renanmf.com/the-python-guide-for-beginners/
[86] https://effectivepython.com/
[87] https://beej.us/guide/bgpython/
[88] https://business-python.com/
[89] https://www.learnpyqt.com/pyqt5-book/
[90] https://www.learnpyqt.com/pyside2-book/
[91] https://pythonbasics.org/
[92] https://realpython.com/
[93] https://gvanrossum.github.io/
[94] https://m.youtube.com/watch?v=IHHAL1pqsPk
[95] https://www.niallkennedy.com/blog/2006/11/google-mondrian.html
[96] https://cloud.google.com/appengine
[97] https://cloud.google.com/appengine/docs/standard/python3
[98] https://hopl4.sigplan.org/
[99] https://deprogrammaticaipsum.com/jean-sammet/
[100] https://www.artima.com/weblogs/viewpost.jsp?thread=161207
[101] https://neopythonic.blogspot.com/2009/01/history-of-python-introduction.html
[102] https://python-history.blogspot.com/
[103] https://techcrunch.com/2012/12/07/dropbox-guido-van-rossum-python/
[104] https://blog.dropbox.com/topics/company/thank-you--guido
[105] http://www.mypy-lang.org/
[106] https://mypy-lang.blogspot.com/
[107] http://gadfly.sourceforge.net/
[108] https://www.mercurial-scm.org/
[109] https://www.writethedocs.org/guide/writing/reStructuredText/
[110] https://twitter.com/nuclearkatie/status/1367673827157667844
[111] https://www.monkeyuser.com/2021/adoption/
[112] https://xkcd.com/1987/
[113] https://twitter.com/thejenkinscomic/status/1413996755126001675
[114] https://deprogrammaticaipsum.com/tim-peters/
[115] https://www.youtube.com/watch?v=hgI0p1zf31k
[116] https://education.ti.com/en-gb/products/calculators/graphing-calculators/ti-84-plus-ce-t-python
[117] https://www.xlwings.org/
[118] https://www.gimp.org/docs/python/structure-of-plugin.html
[119] https://www.gfx.dev/python-for-feature-film

[120] https://dylananthony.com/posts/best-supported-serverless-languages

[121] https://aws.plainenglish.io/how-did-i-processed-half-a-million-transactions-in-aws-lambda-within-minutes-120c69d37ce5

[122] https://www.tiobe.com/tiobe-index/

[123] http://www.fiber-space.de/EasyExtend/doc/teuton/teuton.htm

[124] https://sourceforge.net/projects/perunis/

[125] https://en.wikipedia.org/wiki/Non-English-based_programming_languages

[126] https://github.com/satwikkansal/wtfpython

[127] https://unsplash.com/@jannerboy62?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

[128] https://unsplash.com/s/photos/python-everywhere?utm_source=unsplash&utm_medium=referral&utm_content=creditCopyText

Tim Peters

```
[Clang 12.0.5 (clang-1205.0.22.9)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```

By Graham Lee

Not everything that is worth reading is a book. A good programmer's library (I will let you decide whether that is a good library owned by a programmer, or a library belonging to a good programmer) includes essays, scholarly articles, videos, magazines, blog posts, podcast episodes, and more. This month, we are going to read an Easter egg in a programming language.

Type `import this` into the Python interpreter and you will get a short collection of aphorisms that each summarise a design principle of the Python language (or

really of Guido van Rossum). This document is published as PEP 20 — The Zen of Python[1].

This sort of information is priceless: knowing how your programming language was designed will help you understand how to read code written in it, and write your own code. There are only 19 lines, we may as well go through each one!

> *Beautiful is better than ugly.*

Well, this is clearly subjective, but still important: *we are taking subjective principles into account*. We are considering not just the practicality of programming with this tool, but the *experience* of using the tool, too.

> *Explicit is better than implicit.*

This can be read as a preference for configuration over convention: do things I say, and only things I say. In fact it is more likely to be a dig at a design choice in Perl, where unary functions (one argument) also have a nullary (zero-argument) form: `chomp $foo;` strips trailing whitespace from `$foo`, while `chomp;` acts on the `$_` variable. Other operations implicitly *set* `$_`, so knowing the behaviour of a Perl script can mean knowing the behaviour of Perl, not understanding the instructions in the script.

> *Simple is better than complex.*

Simple to implement, or simple to use? Yes.

> *Complex is better than complicated.*

When something is not straightforward it should be because what you are trying to do is not straightforward; not because the tool makes it harder.

> *Flat is better than nested.*

Think of the Law of Demeter[2]: do not make people reach into the innards of modules or objects to find things that are useful. If they are useful, they belong at the surface.

Sparse is better than dense.

Various programming languages are great at "code golf", providing incredibly terse ways to get a lot of functionality. Python is not. Python would prefer that you write out your program in a way that is easy to read.

Readability counts.

This seems to be coming up a lot! Python is a programming language that was not just designed for *you*, but for everyone you work with.

Special cases aren't special enough to break the rules.

The law of minimal surprise: both the common things and the edge cases should work in the same way, no matter how much you think it would be nicer to do it differently. Although...

Although practicality beats purity.

Indeed. If that uniformity gets in the way, abandon it.

Errors should never pass silently.

This is an important part of the user experience! If your program does not do what you think it does, you should find out quickly and loudly. This leads to design choices like `KeyError` for missing entries in a dictionary, which can be frustrating but ultimately lead to more robust code. Unless...

Unless explicitly silenced.

Of course, if you want to ignore an error, fill your boots.

In the face of ambiguity, refuse the temptation to guess.

Again, your program should do what you expect. If you have not told the computer enough information to understand what you expect, then you have not written your program yet.

There should be one– and preferably only one –obvious way to do it.

This is broadly interpreted as a swing at Perl, which has the Tim Toady (TMTOWTDI—"there is more than one way to do it") design maxim. And indeed it is. But it is also a broadside at all C flavoured languages. Look at the spacing around those hyphens: they are the prefix and postfix subtraction operators! Those definitely give us plenty more than one way to do subtraction. Resolving their effects, and eliding the parenthetical statement, we find a hidden twentieth aphorism, a little bonus piece of Python design knowledge:

There should be zero less-than-obvious ways to do it.

Although that way may not be obvious at first unless you're Dutch.

Guess where Guido van Rossum comes from.

Now is better than never.

Also known as "done is better than perfect": get it out rather than getting paralysed by all these design choices.

Although never is often better than right *now.*

See Python 3000, which was still intentional vapourware at the time PEP-20 was written. If you know you are trying to fix core design problems, and *you introduced those problems*, you may still need to ruminate on the solution.

If the implementation is hard to explain, it's a bad idea.

If this thing is going to be easy to use, it needs to be easy to understand.

If the implementation is easy to explain, it may be a good idea.

Even if this thing is easy to understand, that understanding had better be correct!

Namespaces are one honking great idea – let's do more of those!

While it is true that "flat is better than nested", it is also true that qualified is better than global. My maths library and my data types library may well both have a set, and I should be allowed to call both of them "set".

Cover photo by the author.

References

[1] https://www.python.org/dev/peps/pep-0020/
[2] https://en.wikipedia.org/wiki/Law_of_Demeter