What (not) to do when type hinting Python?

16.01.2025

PyWaw #117

<div class="to-right">

by Bartosz Sławecki (@bswck)

Junior Python Engineer @ Printbox

</div>

Check-in

<center>

</center>

Special thanks

Shoutout to:

- Carl Meyer, for helping me understand the type theory
- a user named `decorator-factory`, for helping me understand the true purpose of this presentation
- Jelle Zijlstra, for participating in the discussion about my talk
- everyone else involved who encouraged me in this endeavor

Who is this talk for?

Everyone!

And *especially* for you, if you:

- are interested in using typing and have no prior practice (<code class="beginner"></code>)
- occasionally use typing, but not a lot (<code class="intermediate"></code>)
- already use typing extensively and maybe like it (<code class="advanced"></code>)

What is typing in Python all about?

<div class="labels"> <code class="beginner"></code> </div>

It's about describing what sets of runtime values can reside in particular variables.

What is type hinting in Python all about?

<div class="labels"> <code class="beginner"></code> </div>

Type hinting is as simple as turning

<div class="flex">

```python
def cube_area(e):
    return f"Cube area: {6 * e ** 2}."
```

into

```python
def cube_area(e: float) -> str:
    return f"Cube area: {6 * e ** 2}."
```

</div>

If you're starting out

<div class="labels"> <code class="beginner"></code> </div>

Let's learn about two main kinds of types really quickly.

Don't forget about these useful *go-tos*

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> <code class="advanced"></code> </div>

- Python type system specification
- `typing` standard library docs
- relevant PEPs
- typeshed
- docs of particular type checkers (mypy, pyright, et alia)
- Python docs
- YouTube videos from Anthony Sottile, James Murphy, me et alia

Check out various different type checkers

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> <code class="advanced"></code> </div>

- mypy (esp. recommended for <code class="beginner"></code>s)
- pyright (esp. recommended for <code class="beginner"></code>s)
- pyre
- pytype

When would you use mypy?

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> <code class="advanced"></code> </div>

- You want to stick with the most popular option
- You want to compile your code with `mypyc` to C extensions (~2.5x speedup)

Docs: https://mypy.readthedocs.io/en/stable/

When would you use pyright?

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> <code class="advanced"></code> </div>

- You use Pylance in VS Code
- You like pyright's approach, design decisions and behaviors that differ from mypy's

Docs: https://microsoft.github.io/pyright/, comparison with mypy

When would you use Pyre?

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

- You want to check out the type checker used for linting Instagram
- You've heard about Pysa and want to test it too

Docs: https://github.com/facebook/pyre-check

Some background: https://news.ycombinator.com/item?id=17048682

When would you use pytype?

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

- You prefer lenient type checking
- You want to rely more on type inference than on explicit annotations (no *gradual typing*)

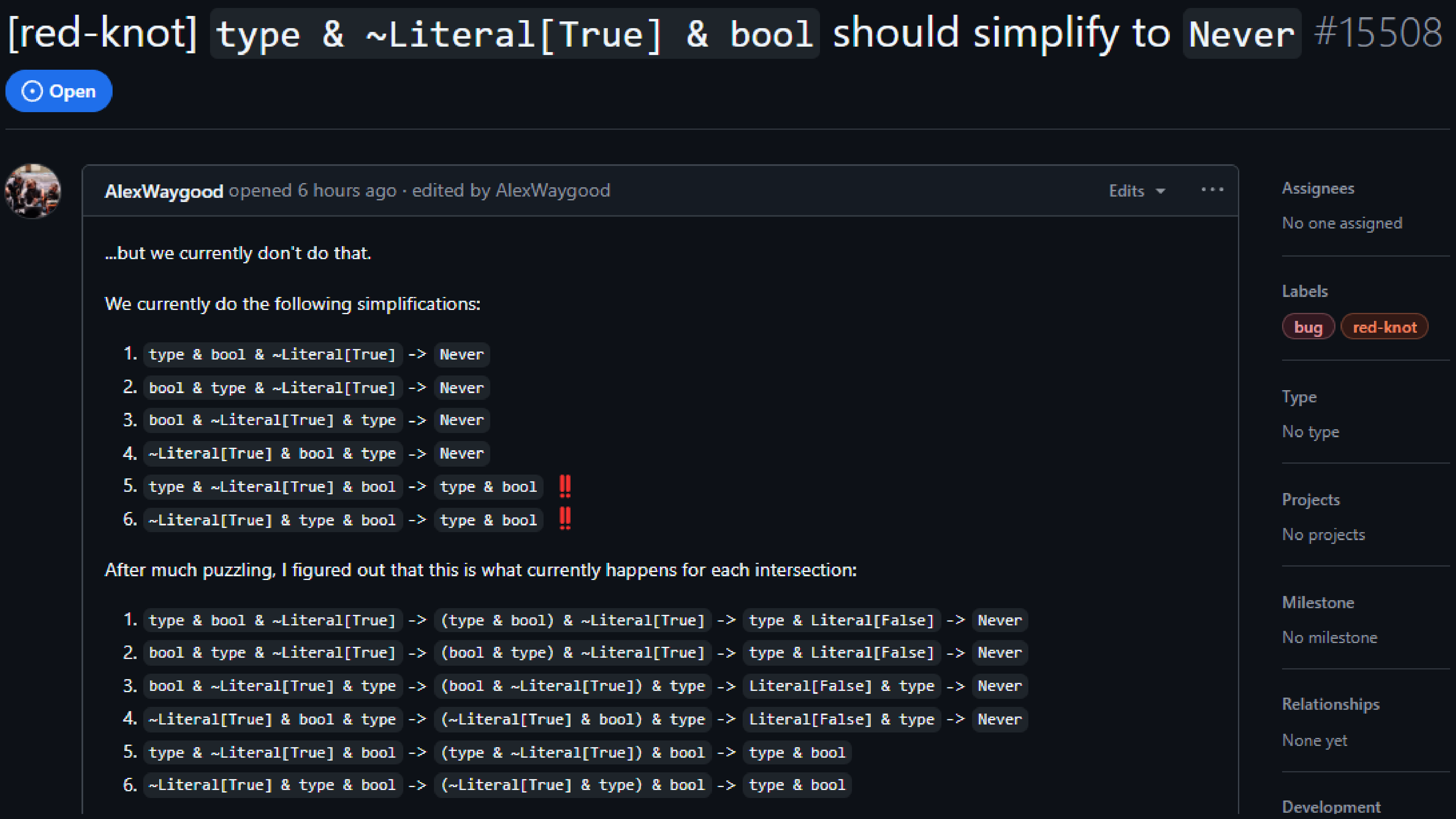
Docs: https://google.github.io/pytype/, comparison with mypy

It's not everything...

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> <code class="advanced"></code> </div> <div class="small">

From Astral, the team behind Ruff and uv

</div> <center>



[red-knot] `type & ~Literal[True] & bool` should simplify to `Never` #15508

Open

AlexWaygood opened 6 hours ago · edited by AlexWaygood

Edits ···

...but we currently don't do that.

We currently do the following simplifications:

1. `type & bool & ~Literal[True]` -> `Never`
2. `bool & type & ~Literal[True]` -> `Never`
3. `bool & ~Literal[True] & type` -> `Never`
4. `~Literal[True] & bool & type` -> `Never`
5. `type & ~Literal[True] & bool` -> `type & bool` ‼️
6. `~Literal[True] & type & bool` -> `type & bool` ‼️

After much puzzling, I figured out that this is what currently happens for each intersection:

1. `type & bool & ~Literal[True]` -> `(type & bool) & ~Literal[True]` -> `type & Literal[False]` -> `Never`
2. `bool & type & ~Literal[True]` -> `(bool & type) & ~Literal[True]` -> `type & Literal[False]` -> `Never`
3. `bool & ~Literal[True] & type` -> `(bool & ~Literal[True]) & type` -> `Literal[False] & type` -> `Never`
4. `~Literal[True] & bool & type` -> `(~Literal[True] & bool) & type` -> `Literal[False] & type` -> `Never`
5. `type & ~Literal[True] & bool` -> `(type & ~Literal[True]) & bool` -> `type & bool`
6. `~Literal[True] & type & bool` -> `(~Literal[True] & type) & bool` -> `type & bool`

Assignees
No one assigned

Labels
bug red-knot

Type
No type

Projects
No projects

Milestone
No milestone

Relationships
None yet

Development

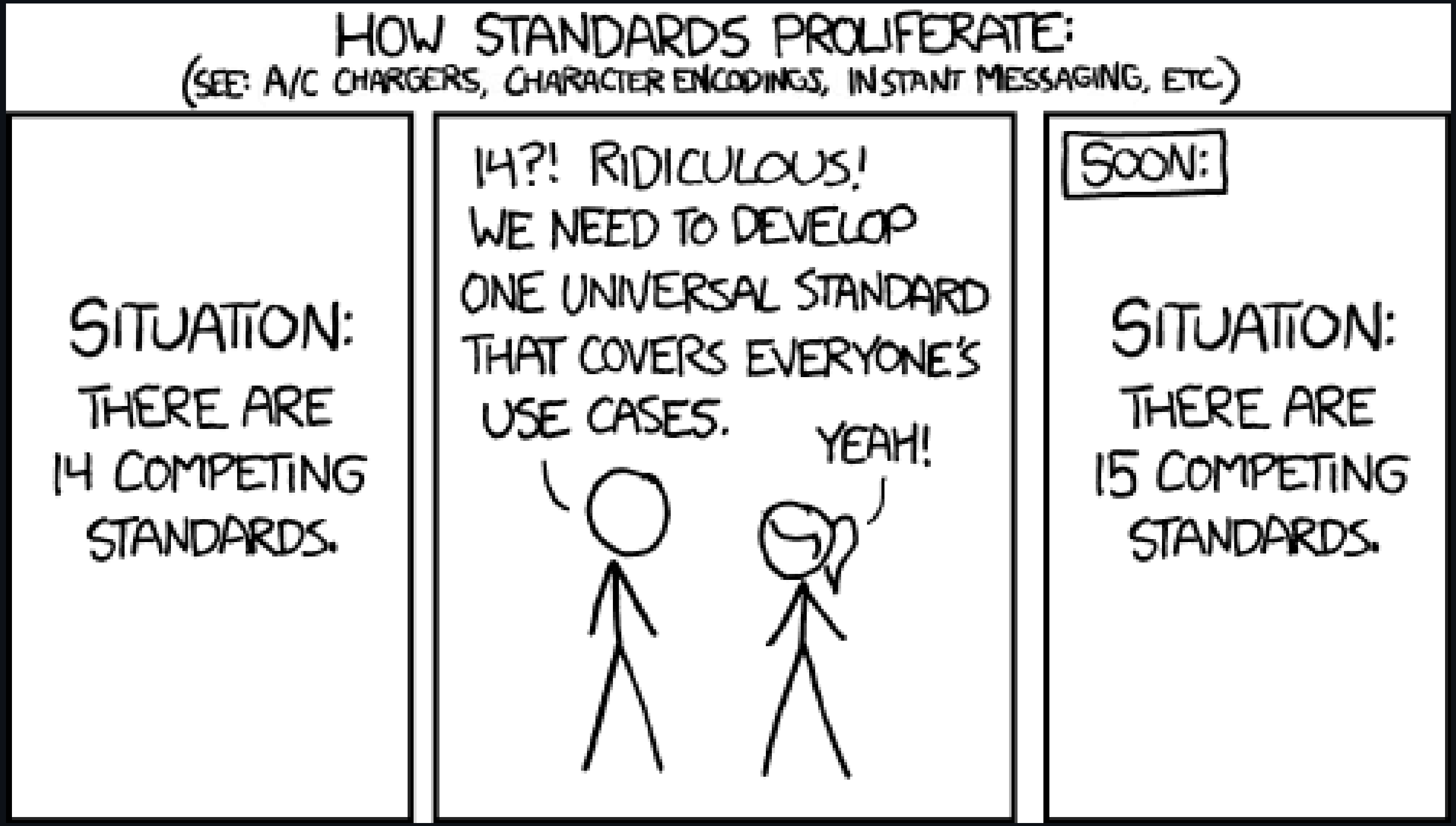
...and if you like rabbit holes,

<div class="labels"> <code class="advanced"></code> </div>

check out those: basedmypy, basedpyright, pyanalyze

To conclude,

<center>


HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)
SITUATION: THERE ARE 14 COMPETING STANDARDS.
14?! RIDICULOUS! WE NEED TO DEVELOP ONE UNIVERSAL STANDARD THAT COVERS EVERYONE'S USE CASES.
YEAH!
SOON:
SITUATION: THERE ARE 15 COMPETING STANDARDS.

</center>

Typing: a strategy that works

<div class="labels"> <code class="beginner"></code> <code class="intermediate"> </code> </div>

Typing: a strategy that works*

<div class="labels"> <code class="beginner"></code> <code class="intermediate"> </code> </div>

Typing: a strategy that works*

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> </div>

*on my machine

Typing: a strategy that works

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> </div>

For every typing feature, do the following:

0. Learn about it
1. Gradual introduction (remember about chunking and aliasing)
2. Troubleshooting (optionally, *trouble-shouting*) / Getting it right
3. Staying up to date (but not up late)

Learn about it

<div class="labels"> <code class="beginner"></code> <code class="intermediate">
</code> </div>

Example: From a linter

Gradual introduction

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> </div> Example: Adding types to my code from 4 years ago

Troubleshooting / Getting it right

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> </div>

- Work through the errors reported by your type checker
- Don't be afraid to google things
- Suggest improvements to type checkers / File bug reports
- Ask questions in Python Discord's `#type-hinting`

Staying up to date

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> <code class="advanced"></code> </div>

- Follow the changelogs (or videos about them)
- Subscribe to Codezarre (https://codezarre.com—try to print this website)

Want to dabble even more?

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

- Read the new typing PEPs
- Participate in typing discussions!
- Contribute to typing-related projects or create them!
    - https://github.com/bswck/class_singledispatch
    - https://github.com/bswck/runtime_generics
    - https://github.com/alexmojaki/eval_type_backport

To avoid common pitfalls...

<div class="labels"> <code class="intermediate"></code> </div>

In the end, it's the runtime that matters.

<div class="examples">

```python
# passes type checking
x: complex = True

# fails at runtime
assert isinstance(x, complex)

# passes type checking
message: str = NotImplemented

# fails at runtime
assert isinstance(message, str)
```

```python
from typing import TYPE_CHECKING

if TYPE_CHECKING:
    # analyzed by type checkers
    # never executed at runtime
    from circular import something
    from costly import just_for_types
else:
```

Think about types, not classes

<div class="labels"> <code class="intermediate"></code> </div>

> In Python, classes are object factories defined by the `class` statement, and returned by the `type(obj)` built-in function. Class is a dynamic, runtime concept.

Classes are commonly used to create:

- Nominal types (e.g. `str`)
- Structural types (`TypedDict` constructs, `Protocol`s)

Besides that, there are...

- Special forms (e.g. `Never`, `Literal`, `Generic`, `TypedDict`)
- *Weird* types (e.g. `None`, `NotImplemented`, `NewType`)

Assignability relation – the Liskov substitution principle

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

We have a function `compute_salary(e: Employee)`.
It accepts an argument of type `Employee`.
Does it also accept an argument of type `Manager`,
given that `Manager` is a subtype of `Employee`?

Practice the S from SOLID

<div class="labels"> <code class="advanced"></code> </div>

Don't make others narrow down your types.

Have an async function? Create a separate coroutine function.

Have a sync function? Create a separate function.

Not both at the same time ;)

What I like about strict static typing in Python

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

Opinionated section.

Autocompletions are very helpful

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> <code class="advanced"></code> </div>

- Easier learning of available functionality
- Thinking about developer experience => Faster development in the long run

Explicit type hinting feels PEP 20-ish

<div class="labels"> <code class="beginner"></code> <code class="intermediate"></code> <code class="advanced"></code> </div>

```python
import this
```

```python
def foo() -> int:
    return 5
```

writing the `-> int` ensures you always return `int`, and not a supertype or other incompatible type.

Hacking and golfing are costly

<div class="labels"> <code class="intermediate"></code> <code class="advanced"> </code> </div>

- You are encouraged to rely on single-purpose and statically known constructs
- The code structure can be fairly simpler to be understood by the type checker

You need a good reason to lie

<div class="labels"> <code class="advanced"></code> </div>
...otherwise don't.

Good reasons:

- the impossibility of expressing a type in the current type system
- lack of ergonomicity to specifying the correct type
- DX—see Werkzeug proxies! (`flask.g`, `flask.request`)

Some great projects do lie, so just be sure to have reasons if you need to.

Things to study

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

That's your homework assignment!

https://github.com/bswck (pinned repo)

New features

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

- Use `typing.Self` (3.11+) or `typing_extensions.Self` (3.9+) for methods returning `cls`/`self`.
- Leverage PEP 696 (`Generator[int, None, None]` -> `Generator[int]`) to reduce redundancy.
- Type hint using generic built-in types (3.9+, PEP 585). E.g. use `list[str]`, not `typing.List[str]`.
- Use `X | Y` instead of `typing.Union[X, Y]` in 3.10+ codebases (PEP 604). This applies to `typing.Optional`, too!
- Don't use `dict[str, Any]` for annotating fixed-structure data (use `TypedDict`, dataclasses, or other models instead).

New features

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

- Review your `TypeGuard`s that could be `TypeIs`s. See (TypeIs vs TypeGuard and PEP 742).
- Don't bother using `TYPE_CHECKING` in a module where any of your types are evaluated at runtime (e.g. in Pydantic models).
- Be pragmatic about `TYPE_CHECKING`. Use it to optimize import times, avoid circular imports and import symbols from stubs.

General

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

- Don't confuse `Any` with `object` (check this).
- Don't use `Any` as an easy way to type a hard-to-annotate interface. Read how to move away from `Any`.
- Don't use the deprecated aliases from `typing`.
- Try not to have to use overloads, but use them to logically associate call conventions, especially when unions are involved.
- Use stub files for annotating extension modules.

Opinionated

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

- DON'T use bare `# type: ignore` (this applies also to `# noqa`).
- DON'T skip annotating return values (here's a writeup).
- DO allow yourself to use PEP 563 despite future deprecation.
- DO prefer `typing.Never` to `typing.NoReturn` (no difference).
- Avoid `T | Awaitable[T]` (`T | Coroutine[T, None, None]`). Single responsibility and interface segregation.

Opinionated (for libraries)

<div class="labels"> <code class="intermediate"></code> <code class="advanced"></code> </div>

- DO type-check at tail (minimum supported version) or all supported versions (to cover `if sys.version_info` branches).
- DO use `__all__` to control re-exports.
- DO minimize runtime overhead if using inlined types. E.g. this PR

Wrapping up

Share your feedback

<center>

</center>