PEP-484 and PEP-526 Gradual Typing

Or how to make your Python project more sturdy

...and sometimes faster.

About typing

- Big Topic
- Frequently about:
  - Static vs Dynamic (or "Duck Typed")
  - ..and Implicit vs. Explicit (or "Manifest")

Static and Dynamic Typing

- Static Typing: Types are assigned before program execution
- Dynamic Typing: Types are assigned during program execution
- There are degrees of this, EG:
  - Java's "Downcasting"
  - Python's Gradual typing

Explicit vs. Implicit Typing

- Explicit typing is like Java
  - ...where you declare all your types
  - ...except lambdas?
- Implicit typing is like Haskell
  - ...where nearly all your variable types are undeclared and types are inferred from context
  - Despite this, Haskell is statically typed.
  - It's also hard to work with.
- There are degrees of this too. EG:
  - Go
  - Rust
  - PEP-484 / PEP-526
  - Cython
  - MypyC

About Gradual Typing – What is it?

- Allows the developer to manifestly declare *some* variables and not others
- Variables without explicit types have their types inferred
- Can effectively be used just for function signatures and perhaps some collection types
- ...or more
- ...if you feel like it

What is Python Gradual Typing good for?

- It's mostly about making programs more reliable, by checking that callables' inputs and outputs have compatible types.
- It is more appropriate for large projects than for small projects
- It is usually not about making things run faster; CPython, Pypy, Micropython and Tauthon ignore the annotations, treating them as documentation.
  - Cython and mypyc are exceptions – they can use types for speeding things up. More later.

If Python ignores them...

- ...how is reliability improved?
- A static analyzer pass like mypy is run on the code to check the types, *before* the interpreter runs the code.
- This typically becomes part of "the build process."

Motivation: A Quick Example

```
def get_mtime(filename: str) -> float:
    """Return the modification time of filename."""
    stat_buf = os.stat(filename)
    return stat_buf.st_mtime


# Note that we've declared filename
# ...and the return type
# ...but not stat_buf.
# This works fine.
```

What Uses Type Annotations

- The CPython interpreter itself treats type annotations as mere documentation; they have no other meaning to Python
- CPython needs an external tool like mypy or pytype to do gradual type *checking*
- Cython has true type declarations
  - but for now they are on a cdef or cpdef, and use a different syntax
  - Cython is working on PEP-484 and PEP-526 types
  - Pretty mature project
- MypyC:
  - compiles PEP-484 and PEP-526 typed python to C extensions
  - is too young for production use yet.
- IDE's
- Jedi

Implementations

Implementation Name	Origin	Notes
mypy	Dropbox	The original, most common, the one we mostly cover here
pytype	Google	Purportedly fast
pyre-check	Facebook	
pyright	Microsoft	
hug		A REST API framework, not compatible with mypy

Mypy and Python Version

- Type annotations available in Python 3.0 and up (Tauthon 2.8 too)
- The types you declare things with in part come from the typing module
  - typing module comes with Python 3.[56789]
  - typing module available as a backport for 2.7 and 3.[234]
- Can be made to work with Python 2.x, but it requires type comments and/or separate .pyi files for type declarations
- Mypy is under a Google OpenSource copyright

Mypy Sample Invocation

```
python3 \
    -m mypy \
    --disallow-untyped-calls \
    --ignore-missing-imports \
    file1.py file2.py
```

Declaring the Type of a Variable: Python 3.5

- from typing import List, Dict
- size_dict = {} # type: Dict[int, List[str]]
- The comment does it
- Sometimes helps mypy
- Also works on Python 3.6

Declaring the Type of a Variable: Python 3.6 and up

- from typing import List, Dict
- size_dict: Dict[int, List[str]] = {}
- Sometimes helps mypy
- No weird comment involved

A Method with Declared Types in Python 2.7

```
class Example:
    def method(self, lst, opt=0, *args, **kwargs):
        # type: (List[str], int, *str, **bool) -> int
        """Docstring comes after type comment."""
```

A Union Type

```
from typing import Union


def fn(x: Union[int, str]) -> None:
    """x is an int or a string."""
    if instance(x, int):
        print(x + 1)
    else:
        print(x + 'a')
```

An "Optional" type

```
from typing import Optional

def concat(x: Optional[str], y: Optional[str]) -> Optional[str]:
    """
    x and y are None or string.

    This is a common special case of Union types, so it gets its own name.
    """
    if x is not None and y is not None:
        return x + y
    return None
```

A Type Alias

```
# Types can get long!

AliasType = Union[List[Dict[Tuple[int, str], Set[int]]], Tuple[str, List[str]]]

# Now we can use AliasType in place of the full name:

def f() -> AliasType:
    ...
```

An Iterator

```
from typing import Iterator


def squares_forever() -> Iterator[int]:
    value = 1
    odd_number = 3
    while True:
        yield value
        value += odd_number
        odd_number += 2
```

A Variable that can Refer to Anything

```
from typing import Any


a: Any = 5
a = 'spam'


# Nice for some heterogeneous collection types
# Notice that we declared it once and used it twice
```

Declaring Types Used Before They Have Been (Fully) Defined

```
class Fraction:
    def __init__(self):
        self.numerator = 0
        self.denominator = 1
    def __lt__(self, other: ‘Fraction’) -> bool:
        ...
```

Higher Order Functions

```
from typing import Callable

def callback(x: int):
    print(f'In callable, got x: {x}')

def function(fn: Callable[[int], int], value: int):
    fn(value)

function(callback, 2)
```

Overriding mypy type checking

- To turn off mypy checking for just one line, add this to the end of that line:

```
# type: ignore
```

What can Generate Type Annotations for You?

- Adding type annotations to existing code is hard, but that's why they're valuable – because otherwise when modifying that code you'd do almost the same thing.

- These can automatically generate them for existing code:
  - MonkeyType
  - Jedi's pep484transform.py
  - PyAnnotate: Guido Van Rossum contributed to this one

- We focus on MonkeyType here, as it's more popular in Google Search.

MonkeyType

- From their github page:

  MonkeyType collects runtime types of function arguments and return values, and can automatically generate stub files or even add draft type annotations directly to your Python code based on the types collected at runtime.

- Requires Python 3.6 and up to collect the data
- Can output stubs for 2.7
- BSD licensed with a Facebook copyright
- Is by people at Instagram, which is owned by Facebook

Collecting Runtime Types with MonkeyType

- monkeytype run myscript.py
- This collects types used in myscript.py and its dependencies via the sys.setprofile hook (like coverage.py)
- The results are stashed in an SQLite database in the CWD

Applying the Results

- Running "monkeytype stub some.module" will output a stub:

  def add(a: int, b: int) -> int: ...

- Running "monkeytype apply some.module" will modify some/module.py to:

  def add(a: int, b: int) -> int:

      return a + b

- stub and apply appear to only work on imported python modules, not shell-callables.
- Furthermore, they appear to import modules as they add type annotations, so *if __name__ == '__main__':* appears to be required for some modules

How Robust is MonkeyType?

- At last report, Instagram had applied it to about a third of their codebase
- Their codebase is well over one million lines of Python 3.x

IDE's and editors

- PyCharm
- VSCode
- Vim
- Emacs

PyCharm

- Uses intention actions
- https://www.jetbrains.com/help/pycharm/type-hinting-in-product.html
- Sounds pretty complete?
- To install, inside PyCharm:
  - Ctrl-Alt-S > Plugins > search "mypy" > Install
  - There are two mypy plugin options:
    - "Mypy"
      - I had problems with this one, so I tried the other
    - "Mypy (Official)"
      - This one seems to mostly work
  - If you need mypy installed, PyCharm *might* prompt you to install it. The first plugin option above prompted me at least. You could also python3 -m pip install mypy

PyCharm continued

- Using "Mypy (Official)":
  - It gives a "Mypy Console" at the bottom
  - Clicking "Run" in the lower right invokes the type checker
  - It doesn't seem to do much until you have a critical mass of types declared
  - In fact, in a little test program, it ignored a serious error until I added types for *everything* in the script
  - But you can set up stricter checking:
    - In the GUI
      - Right click in the Mypy Terminal subwindow and select "Configure Plugin"
      - Add --disallow-untyped-calls --ignore-missing-imports
      - ...to the dmypy invocation
    - Or in the filesystem (mypy.ini at the root of your project) – this example ignores most errors

```
[mypy]
ignore_missing_imports = True
ignore_errors = True

[mypy-blog.admin.*]
ignore_errors = False
```

Yet more PyCharm

- PyCharm can annotate types for you, though it might be a click-heavy process.
- To enable it:
  - Go to Settings -> Build, Execution, Deployment -> Python Debugger page.
  - There, turn on the option labeled as Collect run-time types information for code insight.
  - Run your code under the debugger, in production and/or with your test suite.
  - Go back to your code, select a def of a function or method with the left mouse button, and hit Alt + Enter (in PyCharm on Linux).
  - Select "Add type hints for…" in the resulting menu.
  - Your single def fn should now have type annotations.
  - If everything comes up 'object', you probably haven't enabled 'Collect run-time types'

VSCode

- Sounds like just another Linter for VSCode
- https://code.visualstudio.com/docs/python/linting

Vim

- I prefer to shell out to `make` for now, and have my Makefile's default rule run mypy, among other things.
- EG this sets up *ma to save the current buffer and invoke make (or setup.py):
  - map *ma :w!^M:!clear; if [-f Makefile]; then make; else python3 setup.py build; fi^M
- Apparently mypy works with Syntastic though.

Emacs

- Sounds like it works with Flycheck

References

- https://breadcrumbscollector.tech/mypy-how-to-use-it-in-my-project/
- https://breadcrumbscollector.tech/mypy-how-to-use-it-in-my-project-part-2-automatically-annotate-code/
- https://breadcrumbscollector.tech/mypy-how-to-use-it-in-my-project-part-3-kick-ass-tools-that-leverage-type-annotations/
- https://github.com/typeddjango/awesome-python-typing

The End

- Questions?
- Comments?