l

Python Typing

Agenda

1. Why use typing (or not)
2. Python Type Checkers
3. Strict typing
4. Some typing techniques
5. Pitfalls and oddities
6. Moving to typed python code

Why use typing

- More safety/less bugs
- (Better) type hints in your editor
- Makes it easier to communicate intent and understand code
- Allows making invalid states unrepresentable
- Makes refactoring safer

Why not use typing

- There are edge cases that don't work well
- Takes some effort to add to existing large code-base
- Sometimes a pain fighting to get annotations for code that is obviously correct
- You don't have any strict requirements for safety/your code is trivial

What is typing in Python

```
def encrypt_rsa(public_key, data):
```

What is typing in Python

```
def encrypt_rsa(public_key, data):
```

```
def encrypt_rsa(public_key: rsa.RSAPublicKey, data: bytes) -> bytes:
```

Type checkers

- Mypy (Python)
- Pyright (JS, Microsoft)
- Pyre

Type checkers

- Mypy (Python)
- Pyright (JS, Microsoft)
- Pyre

- Neither of them really checks everything 🙁
- both have edge cases that the other doesn't have 🙁
- Mypy is the reference implementation
  - But the reference leaves out many behaviors as undefined 🙁
- Pyright is faster but not meaningfully so (10s vs 30s, but actually 1s)
- Pyright is used by VSCode (and works as a language server)
- Pyre seems not that mature, very little docs

Strict typing in Renku

There is `mypy --strict`, but we don't use that.

- it does not allow using e.g. just `x: dict`. It requires every generic to be spelled out
- it complains about some valid async code annotations

Strict typing in Renku

But just `mypy` isn't strict enough, so we use:

```
warn_unreachable = true # no unreachable code
warn_redundant_casts = true # no `cast(...)` that's not needed
warn_unused_ignores = true # no unused `# type: ignore`
warn_return_any = true # no ` -> Any ` return
strict_equality = true # equality checks need to be overlapping types
check_untyped_defs = true # check functions that aren't annotated
allow_redefinition = true # allow changing variable type through assignment
disallow_subclassing_any = true # dont allow creating a subclass of `Any`
disallow_untyped_decorators = true # all decorators must be typed
disallow_incomplete_defs = true # don't allow partial annotations
disallow_untyped_defs = true # every function needs to be annotated
```

This allows escape hatches but requires everything to be typed. And for nicer error output:

```
pretty = true
show_column_numbers = true
show_error_codes = true
show_error_context = true
```

Typing Techniques

Overloading

```
def read(path: Path, unicode: bool=False)-> str | bytes:
```

Overloading

```
def read(path: Path, unicode: bool=False)-> str | bytes:
```

```
from typing import overload, Literal

@overload
def read(path: Path, unicode: Literal[False]=False)-> bytes: ...

@overload
def read(path: Path, unicode: Literal[True])-> str: ...

def read(path: Path, unicode: bool=False)-> str | bytes:
```

Overloading

```
def read(path: Path, unicode: bool=False)-> str | bytes:
```

```
from typing import overload, Literal

@overload
def read(path: Path, unicode: Literal[False]=False)-> bytes: ...

@overload
def read(path: Path, unicode: Literal[True])-> str: ...

def read(path: Path, unicode: bool=False)-> str | bytes:
```

```
@overload
def process(data: list[int])-> float: ...

@overload
def process(data: list[str])-> str: ...

def process(data: list[int] | list[str]) -> str | float:
```

Make invalid state unrepresentable

```
class User:
    id: str | None

def save_user(user: User):
    if user.id is not None:
        raise Error("user already saved")

def update_user(user: User):
    if user.id is None:
        raise Error("can't update unsaved user")
```

Make invalid state unrepresentable

```
class User:
    ...

class NewUser(User):
    ...

class SavedUser(User):
    id: str

def save_user(user: NewUser)->SavedUser:
    ...

def update_user(user: SavedUser):
    ...
```

TypeVars (Generics)

```
from typing import TypeVar, Literal, Never, TYPE_CHECKING

New = Literal["New"]
Saved = Literal["Saved"]
UserState = TypeVar("UserState", New, Saved)

class User(Generic[UserState]):
    @classmethod
    def new(cls: type[User[New]]) -> type[User[New]]:  ...

    def __init__(self, user_state: UserState) -> None:
        if TYPE_CHECKING:
            self.__state = user_state # not actually used

    @property
    def id(self: User[New]) -> Never:
       raise Error()

    @property
    def id(self: User[Saved]) -> str: ...

def save_user(user: User[New]) -> User[Saved]:  ...

def update_user(user: User[Saved]) -> None:  ...
```

TypeVars (Generics)

```
U = TypeVar("U") # Could be anything, but needs to be consistent when used

V = TypeVar("V", str, bytes) # must be either str or bytes, exactly

W = TypeVar("W", bound = str | bytes) # can be any combination, subtypes etc

# there is also covariant, contravariant and infer_variance parameters
```

Type aliases

```
type Vector = list[int]

# or pre 3.12
from typing import TypeAlias

Vector: TypeAlias = list[int]

def add(a: Vector, b: Vector) -> Vector: ...
```

`Vector` and `list[int]` are treated as the same thing when checking. Can make code more readable and help shorten long type definitions

```
from typing import NewType

Vector = NewType("Vector", list[int])

v = Vector([1, 6, 2])
```

Vector is a new, separate type and the original type always needs to be wrapped in e.g. `Vector(...)` call. The new type is a subclass of the original, so can still be used in e.g. `def sum(values: list[int]):`

Protocols

```
from typing import Protocol

class CanQuack(Protocol):
    def quack(self) -> None:
        ...

def my_fun(x: CanQuack) -> None:
    x.quack()
```

- Defines interfaces for structural typing
- Does not need to be inherited from
- Anything that has a `quack` method satisfies this
- can also have properties defined instead of/alongside methods
- use `@runtime_checkable` if you want to use this with `isinstance`

Special Types and other things

- Use `typing.Never` or `typing.NoReturn` for things that can't exist
  - the typechecker will fail if code can actually reach them
- `typing.Self` is the type of the enclosing class (no circular reference issues)

Pitfalls

Typing Decorators

They are functions that take a function and return a function

```
from functools import wraps
from typing import ParamSpec, Concatenate, TypeVar

P = ParamSpec("P") # in 3.12 you can use `**P` directly
T = TypeVar("T")

def my_decorator(f: Callable[Concatenate[P], T]) -> Callable[Concatenate[P], T]:
    @wraps(f)
    def wrapper(*args: P.args, **kwargs: P.kwargs) -> T:
        print("my_decorator called")
        return f(*args, **kwargs)
    return wrapper
```

Typing Decorators

Pro:

- Type signature of derocated funtions isn't lost
- Type checks across decorators work (otherwise mypy just doesn't check them…)

Cons:

- Can't type keyword arguments, other than with `P.kwargs`
- Can't use TypeAliases for the `Callable[...]` part to shorten it
- Can't use protocols for wrapped function either in many cases
- Annotating a decorator for class methods is a pain
  - especially a decorator that supports both class methods and regular functions
- Borderline unreadable type signatures
- even worse with decorators that take arguments

Typing Decorators

```
_P = ParamSpec("_P")
_T = TypeVar("_T")
_WithMessageQueue = TypeVar("_WithMessageQueue", bound=WithMessageQueue)


def dispatch_message(
    event_type: type[AvroModel] | type[events.AmbiguousEvent],
) -> Callable[
    [Callable[Concatenate[_WithMessageQueue, _P], Awaitable[_T]]],
    Callable[Concatenate[_WithMessageQueue, _P], Awaitable[_T]],
]:

    def decorator(
        f: Callable[Concatenate[_WithMessageQueue, _P], Awaitable[_T]],
    ) -> Callable[Concatenate[_WithMessageQueue, _P], Awaitable[_T]]:
        @wraps(f)
        async def message_wrapper(self: _WithMessageQueue, *args: _P.args, **kwarg
```

```
def dispatch_message(event_type):
    def decorator(f):
        @wraps(f)
        async def message_wrapper(self, *args, **kwargs):
```

Libraries often aren't typed

- maybe there's an extra library just with types, like `types-urllib3` for `urllib3`
  - Check typeshed: https://github.com/python/typeshed
- You could add annotations yourself (haha)
- Just ignore external libraries without types 😕

```
[[tool.mypy.overrides]]
module = [
  "asyncpg.*",
]
ignore_missing_imports = true
```

Sometimes I just want a dict...

Using

```
x: dict
```

is fine, no need for

```
x: dict[str, Union[str, dict[str, Union[ str, str|int|bool|float]]]]
```

It is python after all... Just don't overdo it.

Use `cast`

- For returns of untyped third party libraries
- if you know it's physically impossible for it to be any other type
  - sometimes mypy just isn't smart enough

```
from typing import cast

x: int
x = cast(int, some_external_call_that_you_know_returns_int())
```

This will shut mypy up

If all else fails, just ignore it

```
x: int = something_annoying() # type: ignore
```

but try ty be more specific to limit the impact of this

```
x: int = something_annoying() # type: ignore[arg-type]
```

Mypy errors will tell you what to put in the brackets

```
 error: Module
"..." does not explicitly export attribute
"..."  [attr-defined]
```

the `[attr-defined]` in this case.

Moving to typed Python code

Moving to typed Python code

- You can gradually add types. Just add ignores for things that aren't typed yet (in pyproject.toml)
- Add types whenever you change code and unignore files as you add types
- Run mypy consistently in e.g. Github Actions and with `pre-commit`
- Annotate critical code first
- Try one of the automated annotations tools
  - Run your app (or tests) with it to gather data
  - Review the gathered data and let the tool apply it to your code, fix any issues that pop up
  - Don't bother with `autotyping`
  - `pyannotate` only works for Python < 3.10, could give it a try.
  - `MonkeyType` works pretty well, but struggles with async functions. It's what I used on Renku

jaxtyping

Typing for your tensors!

```
from jaxtyping import Array, Float

# Accepts floating-point 2D arrays with matching axes
def matrix_multiply(x: Float[Array, "dim1 dim2"],
                    y: Float[Array, "dim2 dim3"]
                   ) -> Float[Array, "dim1 dim3"]:
    ...
```

supports `jax`, `NumPy`, `Tensorflow` and `PyTorch`

https://github.com/patrick-kidger/jaxtyping

Questions/Discussion