Static typing: beyond the basics of

def foo(x: int) -> str:

Vita Smid | EuroPython 2019

July 10, 2019
Vita

QUANTLANE

Prague

python™
Static typing is still quite new in Python.
Static typing is sometimes difficult.
Static typing helps prevent errors early.
1. Strategy
How to approach a large codebase

2. Tactics
Dealing with complex code
How to approach a large codebase
Try to start with strict configuration

1. Ensure full coverage

```ini
mypy.ini
disallow_untyped_calls = True
disallow_untyped_defs = True
disallow_incomplete_defs = True
disallow_untyped_decorators = True
```

2. Restrict dynamic typing (a little)

```ini
mypy.ini
disallow_any_generics = True  # e.g. `x: List[Any]` or `x: List`
disallow_subclassing_any = True
warn_return_any = True        # From functions not declared
                            # to return Any.
```

3. Know exactly what you're doing

```ini
mypy.ini
warn_redundant_casts = True
warn_unused_ignores = True
warn_unused_configs = True
```
Gradual coverage

Begin with *opt-in*: only explicitly listed modules are checked.

```
$ mypy models/ lib/cache/ dev/tools/manage.py
```

Add this command to your CI pipeline and gradually grow that list.

Tip: try an internal hackathon.
Opt-in and imports

```ini
mypy.ini
ignore_missing_imports = True
follow_imports = silent
```

We used `follow_imports = skip` before. Terrible idea.
Getting to opt-out

$ mypy

mypy.ini

[mypy-lib.math.*]
ignore_errors = True
[mypy-controllers.utils]
ignore_errors = True
...

Now you work to gradually reduce that list.
Tests

1. Cut yourself some slack

```ini
[mypy-*.tests.*]
disallow_untyped_decorators = True  # pytest decorators are untyped.
disallow_untyped_defs = False        # Properly typing *all* fixtures
disallow_incomplete_defs = False     # and tests is hard and noisy.
```

2. `# type: ignore` your way around mocks and monkey patching

   * [mypy#2427](https://github.com/python/mypy/issues/2427) Unable to assign a function to a method
   * [mypy#1188](https://github.com/python/mypy/issues/1188) Need a way to specify types for mock objects
   * [mypy#6713](https://github.com/python/mypy/issues/6713) MyPy throws errors when mocking a method

3. Don't give up on test code completely.
Your own packages

Inline type annotations in packages are *not checked* by default.

You need to add a `py.typed` marker file (PEP 561):

```bash
$ touch your_package/py.typed
```

```python
setup(  
    ...,  
    package_data = {  
        'your_package': ['py.typed'],  
    },  
    ...,  
)
```
Third-party packages

- You might have to write stubs for third-party packages
- You might want to ignore them completely

```ini
mypy.ini
ignore_missing_imports = True
follow_imports = silent
```

- You might want to ignore just some of them

```ini
mypy.ini
[mypy-package.to.ignore]
ignore_missing_imports = True
follow_imports = silent
```
Dealing with complex code
Generics and type variables
WeightedAverage = \frac{value_0 \cdot weight_0 + value_1 \cdot weight_1 + \ldots}{weight_0 + weight_1 + \ldots}

```python
class WeightedAverage:
    def __init__(self) -> None:
        self._premultiplied_values = 0.0
        self._total_weight = 0.0

    def add(self, value: float, weight: float) -> None:
        self._premultiplied_values += value * weight
        self._total_weight += weight

    def get(self) -> float:
        if not self._total_weight:
            return 0.0
        return self._premultiplied_values / self._total_weight
```
This of course works...

```python
global vars

avg = WeightedAverage()
avg.add(3.2, 1)
avg.add(7.1, 0.1)
reveal_type(avg.get())  # Revealed type is 'builtins.float'
```

...and this, of course, does not:

```python
from decimal import Decimal
avg = WeightedAverage()
avg.add(Decimal('3.2'), Decimal(1))
# error: Argument 1 to "add" of "WeightedAverage"
# has incompatible type "Decimal"; expected "float"
# error: Argument 2 to "add" of "WeightedAverage"
# has incompatible type "Decimal"; expected "float"
```
from typing import cast, Generic, TypeVar
from decimal import Decimal

AlgebraType = TypeVar('AlgebraType', float, Decimal)

class WeightedAverage(Generic[AlgebraType]):
    _ZERO = cast(AlgebraType, 0)

    def __init__(self) -> None:
        self._premultiplied_values: AlgebraType = self._ZERO
        self._total_weight: AlgebraType = self._ZERO

    def add(self, value: AlgebraType, weight: AlgebraType) -> None:
        self._premultiplied_values += value * weight
        self._total_weight += weight

    def get(self) -> AlgebraType:
        if not self._total_weight:
            return self._ZERO
        return self._premultiplied_values / self._total_weight
avg1 = WeightedAverage[float]()
avg1.add(3.2, 1)
avg1.add(7.1, 0.1)
reveal_type(avg1.get())  # Revealed type is 'builtins.float*'  

avg2 = WeightedAverage[Decimal]()
avg2.add(Decimal('3.2'), Decimal(1))
avg2.add(Decimal('7.1'), Decimal('0.1'))
reveal_type(avg2.get())  # Revealed type is 'decimal.Decimal'*

Types cannot be mixed 😊

avg3 = WeightedAverage[Decimal]()
avg3.add(Decimal('3.2'), 1.1)
# error: Argument 2 to "add" of "WeightedAverage"
# has incompatible type "float"; expected "Decimal"
Using a *bounded* type variable would be even nicer…

```python
AlgebraType = TypeVar('AlgebraType', bound=numbers.Real)
```

Unfortunately, abstract number types do not play well with typing yet.

[mypy#3186 int is not a Number?](https://mypy.readthedocs.io/en/stable/errors.html#mypy-error-3186-int-is-not-a-number)
Protocols: nominal typing vs. *structural* typing
Nominal typing: class inheritance as usual

class Animal:
    pass

class Duck(Animal):
    def quack(self) -> None:
        print('Quack!')

make_it_quack(Duck()) # ✔

def make_it_quack(animal: Duck) -> None:
    animal.quack()

make_it_quack(Penguin()) # error: Argument 1 to "make_it_quack" has incompatible type "Penguin"; expected "Duck"
Structural typing: describe capabilities, not ancestry

```python
from typing_extensions import Protocol

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

def make_it_quack(animal: CanQuack) -> None:
    animal.quack()

make_it_quack(Duck())  # ✓
make_it_quack(Penguin())  # ✓
```

Note that we didn't even have to inherit from CanQuack!
Defining your own types
The case for custom types

```python
def place_order(price: Decimal, quantity: Decimal) -> None:
    ...
```

If we could differentiate between a 'price decimal' and 'quantity decimal'...

```python
def place_order(price: Price, quantity: Quantity) -> None:
    ...
```

1. More readable code
2. Hard to accidentally mix them up
Option 1: Type aliases

```
from decimal import Decimal
Price = Decimal
p = Price('12.3')
reveal_type(p)  # Revealed type is 'decimal.Decimal'
```

Aliases save typing and make for easier reading, but do not really create new types.
Option 2: NewType

from typing import NewType
from decimal import Decimal

Price = NewType('Price', Decimal)
Quantity = NewType('Quantity', Decimal)

p = Price(Decimal('12.3'))
reveal_type(p)  # Revealed type is 'module.Price'

def f(price: Price) -> None:
    pass

f(Decimal('12.3'))  # Argument 1 to "f" has incompatible type "Decimal";
                    # expected "Price"

f(Quantity(Decimal('12.3')))  # Argument 1 to "f" has incompatible
                              # type "Quantity"; expected "Price"

NewType works as long as you don't modify the values:

reveal_type(p * 3)  # Revealed type is 'decimal.Decimal'
reveal_type(p + p)  # Revealed type is 'decimal.Decimal'
reveal_type(p / 1)  # Revealed type is 'decimal.Decimal'
reveal_type(p + Decimal('0.1'))  # Revealed type is 'decimal.Decimal'
Writing your own `mypy` plugins
Here be dragons

Documentation and working examples are scarce

Check out our plugin: 170 lines of code and 350 lines of comments

github.com/qntln/fastenum/blob/master/fastenum/mypy_plugin.py
Overloading functions
```python
s = Series[int]([2, 6, 8, 1, -7])
s[0] + 5  # ✓
sum(s[2:4])  # ✓

from typing import Generic, overload, Sequence, TypeVar, Union

ValueType = TypeVar('ValueType')
class Series(Generic[ValueType]):
    def __init__(self, data: Sequence[ValueType]):
        self._data = data

    @overload
    def __getitem__(self, index: int) -> ValueType:
        ...

    @overload
    def __getitem__(self, index: slice) -> Sequence[ValueType]:
        ...

    def __getitem__(self, index: Union[int, slice]) -> Union[ValueTypes, Sequence[ValueType]]:
        return self._data[index]
```

The code defines a `Series` class that takes a sequence of `ValueType` and provides methods for accessing its elements. It uses type hints from the `typing` module to specify the type of elements and the return types of its methods. The class constructor initializes the sequence, and the `__getitem__` method is overloaded to handle both integer and slice indices, returning the appropriate type (`ValueType` or `Sequence[ValueType]`) based on the type of the index.
1. Try to use strict(er) configuration
2. Cover your code gradually
3. Learn to work with generics
4. Use protocols for duck typing
5. NewType can add semantics
6. Writing plugins will surely get easier over time
7. Overloading is verbose but makes sense
Thank you

twitter.com/quantlane

tita@quantlane.com