Statically Checking Python Code

Nat Knight, May 2018

I’m Nat, I do bioinformatics at the BC CfE

What to expect

I will talk about:

My experience with typed Python

What I've found it useful/not useful for

Comparisons to other typed languages

I will not talk about:

Details of the type system

How to be more like Haskell

Monads

Python’s type annotations:

Are applied to function arguments and return values

Don’t do anything at run-time

Can be checked by type checkers (e.g. mypy)

```python
def add(x: int, y:int) -> int:
    …
```

Add takes two integers and returns an integer.

```python
class User(object):
    …


def fetch_user(user_id: int) -> User:
    …
```

Fetch_user takes a user id and returns an instance of the user-defined class User.

```python
import typing

def sum(numbers: typing.List[int]) -> int:
    …
```

Sum takes a list of integers and returns an integer.

Pretty straightforward.

```python
import typing

def save(arg: typing.Union[int, str]) -> None:
    …
```

Save takes either an integer or a string and returns nothing.

Not totally clear what's going on; types aren't everything.

```python
import typing

T = typing.TypeVar("T")

def batches(xs: typing.Iterator[T]) -> typing.List[T]:
    …
```

New Idea: type variables (generics, type parameters)

Batches splits an iterator into lists.

Iterator and lists are homogeneous (all items have same type).

The mypy type checker is:

Static (doesn't run your code)

Incremental (mix annotated and un-annotated code)

Configurable (tune strictness to your needs)

A lot like a linter (pylint, flake8, etc.)

Static:
- highly dynamic code (e.g. Django ORM) can confuse it
- Requires some annotation to work

Incremental:
- Adding type annotation is "extra work"
- Need to use judgement, decide when returns are diminishing
- Can co-exist in "legacy" codebases
- Can be non-obvious when code is being checked

Good Use: Verified Documentation

```
def register(u: User) -> None:
    …

def register(user_name: str) -> RegistrationOutcome:
    …

def register(
    account_id: typing.Union[str, uuid.UUID]):
    …
```

Can know about these functions without reading their bodies.

Unlike docstrings, these can be checked automatically.

Good Use: Design Thinking Tool

```
def process_trn(arg: typing.Union[int, str, TrnId]):
    …

def fetch_user(
    user_id: uuid.UUID) -> typing.Optional[User]:
    …

def parse_message(input: typing.Any) -> Message:
    …
```

process_trn:
- Very polymorphic, no return value
- Possible candidate for a re-factor: what happens if a transaction fails?
- Possibly confusing semantics.
- Bad name, too.

fetch_user:
- user_id is *only* a UUID, and returns either a user or None.
- Easy to track which functions are “reliable” and which ones may fail.

parse_message:
- Takes in unstructured data and returns parsed data.
- Probably an important piece of code that should be thoroughly tested.

Bad Use: Replacing Tests

```python
def add(x: int, y: int) -> int:
    return x - y


def main():
    add("not a number", None)
    # No error. `main` isn't checked
```

Types a poor replacement for tests:

- Some code isn’t checked
- Types may not capture behaviour

Different kinds of type system:

Elm/Haskell/Scala/Rust etc.	**mypy**
Part of compilation, mandatory part of execution	Optional, separate from execution
Provides strict, semantically consistent guarantees	Provides ad-hoc, inconsistent guarantees
Flexible, expressive, modern features	Flexible, expressive, modern features

With stricter type systems:

- can’t run code that’s not checked
- must devise types for every part of every program
- gain assurance for your trouble
- While different, Python checker is very **capable**

In summary:

- Python can have static types now, but it’s not Haskell
- Types make good design tools and checkable documentation, but aren’t a replacement for tests

Further Reading:

- mypy docs
- Łukasz Langa: Gradual Typing of Production Applications
- Static types in Python, oh my(py)!

The End

Questions?