STATIC TYPING FOR PYTHON

TYPE ANNOTATIONS & MYPY

Pawel Stoworowicz

“Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with ***dynamic typing*** *and dynamic binding[…]”*

WHAT IS PYTHON? EXECUTIVE SUMMARY

WHY WE LOVE PYTHON?

Fast development

Nice looking syntax

Requires more attention though…

DUCK TYPING! + MANY OTHER

WHY STATIC TYPING CAN BE COOL?

Faster way to understand already existing code (not always…)

Easier to find bugs

Linters & tools

DUCK TYPING + STATIC TYPES = MYPY

“Mypy is an experimental optional static type checker for Python that aims to combine the benefits of dynamic (or "duck") typing and static typing […]”

WHAT IS MYPY

"This module supports type hints as specified by PEP 484. The most fundamental support consists of the types Any, Union, Tuple, Callable, TypeVar, and Generic. For full specification please see PEP 484.
For a simplified introduction to type hints see PEP 483."

MODULE TYPING - SUPPORT FOR TYPE HINTS PYTHON 3.5+

```
def foo(bar: str = 'bar') -> None:
    print(bar)

print(foo.__annotations__)

>>> {'bar': <class 'str'>, 'return': None}
```

```
from typing import List

def foo(bar: int = 10) -> List[int]:
    _to_return = []
    for elmn in range(bar):
        _to_return.append(elmn)
    return _to_return
```

```
pip install mypy
```

USING MYPY

```
def bar() -> List[str]:
    return foo()
```

```
mypy test_types.py
```

Output:

```
test_types.py:12:4: error: Incompatible return value type (got "List[int]", expected "List[str]")
```

USEFUL TYPES IN TYPING MODULE

- Dict
- Mapping
- List
- Tuple
- **Any**
- **Union**
- **Optional**
- Generator
- Iterator
- Hashable
- Callable
- NamedTuple

+ more

```python
def foo () -> Dict:


def foo () -> Dict[str, Any]:


def foo () -> Dict[str, List[int]]:
```

MULTIPLE TYPES CAN BE RETURNED?

- ANY – allows to return any kind of object
  - (don’t care option)

- Multiple but limited number of types?
  - Use Union

OPTIONAL

```python
class ServiceClient:
    def __init__(self,
                 schema: str = 'http',
                 host: str = 'host_name',
                 port: int = 80) -> None:
        self.schema = schema
        self.host = host
        self.port = port

    def get_url(self) -> str:
        return '{}://{}:{}'.format(self.schema, self.host, self.port)

def get_client() -> ServiceClient:
    return ServiceClient()
```

```python
from typing import Optional

class ServiceAClient(ServiceClient):
    pass

class ServiceBClient(ServiceClient):
    pass

def get_client(conn_id: int) -> Optional[ServiceClient]:
    if conn_id == 1:
        return ServiceAClient()
    if conn_id == 2:
        return ServiceBClient()
    return None            # better raise exception in fact…
```

```
from typing import Optional, Dict
…
    self.data = {}  # type: Dict[str, str]
…
 def get_some_data(self, attr: str) -> Optional[str]:
     return self.data.get(attr)
…
```

OPTIONAL – USEFUL FOR ALL KINDS OF GETS WITH DEFAULT NONE VALUE

CAST FUNCTION

"This returns the value unchanged. To the type checker this signals that the return value has the designated type, but at runtime we intentionally don't check anything (we want this to be as fast as possible)."

```
from typing import cast, Mapping
def bar(data: Mapping) -> str:
    #do something
    ….

data = cast(Mapping, foo)    # foo returns Dict
res = bar(data)
```

CAST(STATIC) VS ASSERT(RUNTIME)?

CUSTOM TYPES

```
from typing import NewType

connection_id = NewType('connection_id', int)
```

```python
def get_connection_id(name: str) -> connection_id:
    if name == 'foo':
        return cast(connection_id, 1)
    else:
        return cast(connection_id, 2)


def get_client(conn_id: connection_id) -> Optional[ServiceClient]:
    if conn_id == 1:
        return ServiceAClient()
    if conn_id == 2:
        return ServiceBClient()
    return None

conn_id = get_connection_id('foo')
client = get_client(conn_id)
client = get_client(1)    # passing int
```

```
>>> test_types.py:47:9: error: Argument 1 to "get_client" has
incompatible type "int"; expected "connection_id"
```

The type checker only needs a *stubs* file to let programs access a Python module. There is no need to port the entire module to mypy. A stubs file is also a good starting point for porting an entire Python module to mypy. They can also highlight potential areas of improvement in the mypy type system.

STUB FILE

You can define types in stub files

e.g. stubs/numpy/__init__.pyi

```
export MYPYPATH=$PWD/stubs
```

Definition example:

```
def f_name(arg: str) -> Union[str, int]:…
```

```python
import numpy as np


def get_ones() -> 'np.ndarray[int]':
    return np.ones(5)
```

CONCLUSIONS, IMPRESSIONS, FEELINGS

Mypy is static checker which means no additional overhead at runtime – does not affect performance.

Ok… let’s ignore memory used for additional definitions in __annotation__ and evaluation of extra frame when cast is called..

In a big project usage of type annotations and mypy can improve delivered quality. There are situations where mypy can prevent us from making mistakes.

OVERALL IMPRESSION

One of key supporters and contributor of mypy is Guido Van Rossum.

However, there are still some issues that make mypy usage a bit tedious.

```
# type: ignore
```

ISSUES

```python
@property   # type: ignore
@lru_cache(1)
def is_passing(self) -> bool:
    ...
```

Mypy is still in early phase of development.

Current version – 0.560 (Feb 2018)

ISSUES

QUESTIONS?

THANK YOU