Typed Python with PEP 484 and MyPy

(Or, How To Get Your Ducks In A Row)

Pi Delport <pjdelport@gmail.com>
at CTPUG, 2016-08-20

Static types? *In Python?*

Why check types?

- Safer code
- Faster interactive feedback
- Easier, more confident refactoring
- Machine-verified documentation

What is a type system?

1. A **semantic model** of program evaluation
    - Simplest model: constrain a variable's possible types
2. A way to **annotate** code with types
    - Optionally: a type inference mechanism
3. Tooling to **check** a program’s declared types against the model, and **report errors**

How does this relate to Python?

Semantic model:

PEP 483 — The Theory of Type Hints

Type annotation syntax:

PEP 484 — Type Hints

Checking & reporting tool:

MyPy

A Taste of Typing

Installing MyPy

$ pip install mypy-lang

Note: *“mypy-lang”, not “mypy”!*

$ mypy src/program.py …

Hello World, Typed

```
def greeting(name: str) -> str:
    return 'Hello ' + name
```

Hello World, Checked

```
greeting('world')      # OK

greeting(5)            # error: Argument 1 to
                       # "greeting" has
                       # incompatible type "int";
                       # expected "str"
```

Something a bit more complex

```
d = ToyDeferred()  # type: ToyDeferred[int, int]

d = (d
    .addCallback(lambda x: x*2)
    .addCallback(str)
    .addCallback(lambda x: x - 5)                # error: Unsupported operand types for - ("str" and "int")
    .addCallback(lambda x: len(x) + 1)
    .addCallback(print)
    .addCallback(lambda x: x*2))                 # error: "print" does not return a value

d.callback(5)
d.callback('5')                                  # error: Argument 1 to "callback" of "ToyDeferred" has incompatible
                                                 type "str"; expected "int"
```

A Tour of the Type System

Function annotations (PEP 3107)

```
def len(o: Sized) -> int: ...

def repr(o: object) -> str: ...

def hasattr(
    o: Any,
    name: str,
) -> bool:
    ...
```

```
def print(
    *values: Any,
    sep: str = ' ',
    end: str = '\n',
    file: IO[str] = None,
    flush: bool = False,
) -> None:
    ...
```

Type comments

```
x = 'spam'  # type: str

nums = []   # type: List[int]

q = Query(...)  # type: Query[User]
```

Coming soon in Python 3.6:
PEP 526 — Variable and Attribute Annotations

```
x: str = 'spam'

nums: List[int] = []

q: Query[User] = Query(...)
```

Supported types

- Classes (built-in and user-defined)
- Abstract base classes (ABCs)
- Special types from the **typing** module
- Generic (parameterised) types
- Type aliases, type variables

Special type: **Any**

- The most general type possible
  - The *top type*, or ⊤
- All other types are subtypes of **Any**
- Every type that’s not annotated or inferred will default to **Any**

Base types

Tuple[*A*, *B*, ..., *N*]

(1, 2.0, 'three'): Tuple[int, float, str]

Callable[[*A*, *B*, ..., *N*], *R*]

len: Callable[[Sized], int]

More base types

- **List[*E*]**
- **Iterable[*E*]**
- **Dict[*K*, *V*]**
- **Set[*E*]**

Special type: **Union**

Union[*A*, *B*, *C*] represents either *A*, *B*, or *C*

Optional[*T*] is an alias for **Union[*T*, None]**

Type aliases

```
Url = str

IntList = List[int]

Point = Tuple[float, float]

Predicate = Callable[[Any], bool]
```

Distinct types: **NewType**

```
Kg = NewType('Kg', float)
Pound = NewType('Pound', float)

def kilogram_to_pound(kg: Kg) -> Pound:
    return Pound(kg * 2.2)
```

Forward references

```
def draw_line(start: 'Point', stop: 'Point'):
    ...

class Point:
    def distance(self, other: 'Point'):
        ...
```

Type casting

```
value = ...   # value can have any type
t = cast(T, value)  # t has type T
```

Note: Does nothing at run-time!
Purely a compile-time type specification.

Type variables (a.k.a. parametric polymorphism)

Variables *parameterise* and *constrain* multiple instances of a type in some scope:

```
E = TypeVar('E')
def repeat(elem: E) -> Iterable[E]:
    ...
```

Type variables
(a.k.a. parametric polymorphism)

```
E = TypeVar('E')
def repeat(elem: E) -> Iterable[E]: ...

repeat(5)   # Iterable[int]
repeat('x')  # Iterable[str]
```

Generic (parameterised) types

```
T = TypeVar('T')

class MyStructure(Generic[T]):
    def get(self) -> T: ...
    def set(self, T) -> None: ...
```

Generic (parameterised) types

```
t = MyStructure(...)  # type: MyStructure[int]

t.set(5)      # OK
t.get()       # type: int
t.set('x')    # error: type "str"; expected "int"
```

Bounded type variables

```
K = TypeVar('K', bound=Hashable)
V = TypeVar('V')
class CustomHash(Generic[K, V]):
...

CustomHash[str, set]      # OK
CustomHash[set, str]      # Error
```

Thanks!

Any questions?

Discussion?

- Types versus tests
- What PEP 484 isn’t
  - Performance-enhancing
  - Mandatory