Using Type Annotations in Python

by Philippe Fremy / IDEMIA

Python code can be obscure

PyParis

```python
def validate(form, data):
    """Validates the input data"""
    return form.validate(data)
```

- You do not know the types of the arguments
- The function may accept multiple types and you don’t know it
- Docstrings (when present) may not be accurate or useful
- You may break production code just be providing an unexpected type and you will only notice it at run-time.

Very brief history of type annotations

- Function argument annotations introduced for Python 3.0 in 2006 by Guido van Rossum
- Type annotations introduced in Python 3.5 (2015)
- Further improved in Python 3.6 (2016) and Python 3.7 (2018)

Syntax for type annotation

```python
# a function
def my_func(a : int, b : str = "") -> bool:
    # ...


# a method
class A:
    def my_method(self, a : bool, b : int = 0) -> None:
        # ...
```

Syntax for type annotation

```python
# Variables (only with python 3.6 and later)
a: int = 0
b: str

class MyClass:

    c: float # type of the instance variable
             # (only with python 3.6 and later)

    def __init__(self) -> None:
        self.c = 33.17
        self.d: str = "I am a string"
```

Available types for annotations

```python
# List defines a general content type
my_list_int: List[int] = [1,2,3]

# multiple types content requires Union
my_multi_type_list: List[ Union[bool, int] ] = [ True, 33 ]

# Tuple usually define precisely all their members
my_tuple: Tuple[int, str, float] = (1, "abc", 3.14)

# Tuple can also declare a general content type
my_float_tuple: Tuple[float, ...] = (11.14, 20.18, 0.1)
```

Available types for annotations

```
# Dict defines keys and content type
my_dict: Dict[str, int] = { "33": 17 }

# Containers may be combined
school_coords: Dict[ str, Tuple[int, int] ]
school_coords = {"Epita": (10, 20)}
```

Available types for annotations

```
# None is a valid type annotation
def f(a: None) -> int:
    ...

# None is always used in a Union:
def f(a: Union[None, int]) -> int:
    ...

# Union[None, int] may be spelled as Optional[int]
def f(a: Optional[int] = None) -> int:
    ...
```

And there is more…

The *typing* module also offers :

- Duck typing with types such as *Sequence*, *Mapping*, *Iterable*, *Sized*, ...
- Type aliasing, type generics, subtyping, typing joker with *Any*, …
- Conversion between types with *cast*

Please check the *typing* module documentation and the *Mypy* tool

How does Python handle type annotations ?

- Annotations are valid expressions evaluated during module loading
- Result is stored in the function object
- And then … they are totally ignored by Python

Type annotations are verified by <u>external</u> tools : *Mypy*, *Pyre*, …

Type Annotations verification tools

Tools to verify static type information:

- *PyCharm* IDE along with inspection mode

- *Mypy* : Open Source, written in Python, maintained by Dropbox team on GitHub

- *Pyre* : Open Source, written in OCaml, maintained by Facebook team on GitHub, only for Linux and MacOs X

How to get started with annotations

- On a new codebase set the rule of having annotations and be strict about it.
- On an existing codebase, start small, one module at a time.

  Then improve gradually.

  All the annotation tools are designed for gradual improvements.
- Put static type verification in your Continuous Integration / Nightly builds / non regression tests.

Proceed one module at a time

Step 1: add annotations to *my_module.py* and verify them

```
$ mypy --strict my_module.py
my_module.py:11: error: Function is missing a return type annotation
```

Mypy in strict mode complains about every missing annotation.

Proceed one module at a time

Step 1: add annotations to *my_module.py* and verify them

```
$ mypy --strict my_module.py
my_module.py:11: error: Function is missing a return type annotation
```

Mypy in strict mode complains about every missing annotation.

Step 2: when the module is fully annotated, check the whole codebase.

```
$ mypy *.py
mod2.py:5: error: Argument 1 to "my_func" has incompatible type
"float"; expected "int"
```

Mypy reports every misuse of *my_module* (only in annotated code).

Proceed one module at a time

Step 1: add annotations to *my_module.py* and verify them

```
$ mypy --strict my_module.py
my_module.py:11: error: Function is missing a return type annotation
```

Mypy in strict mode complains about every missing annotation.

Step 2: when the module is fully annotated, check the whole codebase.

```
$ mypy *.py
mod2.py:5: error: Argument 1 to "my_func" has incompatible type
"float"; expected "int"
```

Mypy reports every misuse of *my_module* (only in annotated code).

Step 3: run your non-regression tests

Where to add type annotation

```python
# annotate all your functions and methods

# variable with value do not need type annotation
vat_rate = 20      # OK, vat_rate is an int


# unless the value type is not correct…
if reduced_vat:
    vat_rate = 5.5 # Error from mypy, vat_rate does not accept float


vat_rate: float = 20 # OK for float and int values
```

Where to add type annotations

```
# All empty containers need annotations

names = []    # Mypy can not figure out the content type

names: List[str] = [] # OK

# Dict and other empty containers need annotations
birth_dates: Dict[str, Date]
birth_dates = {}
```

Let's practice

Example 1

```python
class A:
    def use_another_a(self, a: A) -> None:
        pass

    def use_b(self, b: Optional[B]) -> None:
        pass

class B:
    pass
```

```python
class A:
    def use_another_a(self, a: A) -> None:
        pass

    def use_b(self, b: Optional[B]) -> None:
        pass

class B:
    pass
```

```
$ mypy --strict ab.py
$
$ python ab.py
File “ab.py", line 4, in A
    def use_another_a( self, a: A ) -> None:
NameError: name 'A' is not defined
 File “ab.py", line 7, in A
    def use_b( self, b: Optional[B] ) -> None:
NameError: name 'B' is not defined
```

```python
from __future__ import annotations # python 3.7 only

class A:
    def use_another_a(self, a: A) -> None:
        pass

    def use_b(self, b: Optional[B]) -> None:
        pass

class B:
    pass
```

```
$ mypy --strict ab.py
$
$ python ab.py
$
```

```python
# Other solution: put annotations inside quotes

class A:
    def use_another_a(self, a: "A") -> None:
        pass

    def use_b(self, b: Optional["B"]) -> None:
        pass

class B:
    pass
```

```
$ mypy --strict ab.py
$
$ python ab.py
$
```

Let’s practice

Example 2

```python
class A:
    def __init__(self, step_init: Optional[int] = None) -> None:
        self.step = step_init

    def get_step(self) -> int:
        return self.step + 1
```

```python
class A:
    def __init__(self, step_init: Optional[int] = None) -> None:
        self.step = step_init

    def get_step(self) -> int:
        return self.step + 1
```

```
$ mypy --strict a.py
a.py:6: error: Unsupported operand types for + ("Optional[int]" and
"int")
```

```python
class A:
    def __init__(self, step_init: Optional[int] = None) -> None:
        self.step = step_init

    def get_step(self) -> int:
        return self.step + 1
```

Mypy found a bug !

```
$ mypy --strict a.py
a.py:6: error: Unsupported operand types for + ("Optional[int]" and
"int")
```

```python
# Solution 1: prepend a check for None

class A:
    def __init__(self, step_init: Optional[int] = None) -> None:
        self.step = step_init

    def get_step(self) -> int:
        # deal with self.step being None
        if self.step is None: return 0

        # now we can proceed
        return self.step + 1
```

```
$ mypy --strict a.py
$
```

```python
# Solution 2: default initialise with the right type

class A:
    def __init__(self, step_init: Optional[int] = None) -> None:
        self.step = step_init or 0    # self.step type is always int

    def use_step(self) -> int:
        return self.step + 1
```

```
$ mypy --strict a.py
$
```

```python
# Solution 3: do not use Optional, have better default

class A:
    def __init__(self, step_init: int = 0) -> None:
        self.step = step_init

    def get_step(self) -> int:
        return self.step + 1
```

```
$ mypy --strict a.py
$
```

```python
# Solution 4: disable None checking in Mypy

class A:
    def __init__(self, step_init: Optional[int] = None) -> None:
        self.step = step_init

    def get_step(self) -> int:
        return self.step + 1
```

```
$ mypy --strict --no-strict-optional a.py
$
```

```python
# Solution 5: silence the error (not a good practice)

class A:
    def __init__(self, step_init: Optional[int] = None) -> None:
        self.step = step_init

    def get_step(self) -> int:
        return self.step + 1 # type: ignore
```

```
$ mypy --strict a.py
$
```

Let’s practice

Example 3

```python
# Dealing with multiple types

def upper(thing: Union[str, bytes, List[str]]) -> str:
    if type(thing) == list:
        thing = "".join(thing)

    return thing.upper()
```

```
$ mypy --strict upper.py
upper.py:5: error: Argument 1 to "join" of "str" has incompatible
type "Union[str, bytes, List[str]]"; expected "Iterable[str]"
upper.py:8: error: Incompatible return value type (got "Union[str,
bytes, List[str]]", expected "str")
```

```python
# Dealing with multiple types

def upper(thing: Union[str, bytes, List[str]]) -> str:
    if type(thing) == list:
        thing = "".join(thing)

    return thing.upper()
```

```python
# Dealing with multiple types

def upper(thing: Union[str, bytes, List[str]]) -> str:
    if type(thing) == list:
        thing = "".join(thing)

    return thing.upper()
```

```
$ mypy --strict upper.py
upper.py:5: error: Argument 1 to "join" of "str" has incompatible type "Union[str, bytes, List[str]]"; expected "Iterable[str]"
upper.py:8: error: Incompatible return value type (got "Union[str, bytes, List[str]]", expected "str")
```

```python
# Solution: use isinstance()

def upper(thing: Union[str, bytes, List[str]]) -> str:
    if isinstance(thing, list): # mypy understand isinstance()
        thing = "".join(thing)  # so now, join() passes fine

    return thing.upper()
```

```python
# Solution: use isinstance()

def upper(thing: Union[str, bytes, List[str]]) -> str:
    if isinstance(thing, list): # mypy understand isinstance()
        thing = "".join(thing)  # so now, join() passes fine

    return thing.upper()
```

Mypy found a bug !
I forgot to deal with bytes

```
$ mypy --strict upper.py
upper.py:7: error: Incompatible return value type (got "Union[str,
bytes]", expected "str")
```

```python
# Solution: use isinstance()

def upper(thing: Union[str, bytes, List[str]]) -> str:
    if isinstance(thing, list): # mypy understand isinstance()
        thing = "".join(thing)  # so now, join() passes fine

    return thing.upper()
```

Mypy found a bug !
I forgot to deal with bytes

```
$ mypy --strict upper.py
upper.py:7: error: Incompatible return value type (got "Union[str, bytes]", expected "str")
```

```python
# Solution: use isinstance() and catch all types

def upper(thing: Union[str, bytes, List[str]]) -> str:
    if isinstance(thing, list):
        thing = "".join(thing)

    elif isinstance(thing, bytes): # we also check for bytes
        thing = thing.decode("UTF8")

    # now, all paths make thing a string
    return thing.upper() # OK, returning a str
```

```
$ mypy --strict upper.py
$
```

```python
# Solution: use cast and catch all types

def upper(thing: Union[str, bytes, List[str]]) -> str:
    if type(thing) == list:
        thing = cast(List[str], thing)
        thing = "".join(thing)

    elif type(thing) == bytes:
        thing = cast(bytes, thing)
        thing = thing.decode("UTF8")

    thing = cast(str, thing)
    return thing.upper()
```

```
$ mypy --strict upper.py
$
```

Let’s practice

Example 4

```python
# file form_validator.py

def validate(form, data):
    # ... (do some pre-validation stuff)
    return form.validate(data)

class UserForm:
    def validate(self, data):
        """Validates the data. Data must be a list of int"""
        data[4] = data[1] * data[2] % data[3]
        return data[4] > 21
```

```python
# file production_code.py

def production_code():
    userForm = UserForm()
    data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

    # ...
    return validate(userForm, data)
```

```python
# file production_code.py

def production_code():
    userForm = UserForm()
    # data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    data = range(10)
    # ...
    return validate(userForm, data)
```

```python
# file production_code.py

def production_code():
    userForm = UserForm()
    # data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    data = range(10)
    # ...
    return validate(userForm, data)
```

```
$ python production_code.py
Traceback (most recent call last):
  File "production_code.py", line 7, in <module>
    production_code()
  File "form_validator.py", line 4, in validate
    return form.validate(data)
  File "form_validator.py", line 9, in validate
    data[4] = data[1] * data[2] % data[3]
TypeError: 'range' object does not support item assignment
```

```python
# file form_validator.py

def validate(form: UserForm, data: List[int]):
    # ... (do some pre-validation stuff)
    return form.validate(data)

class UserForm:
    def validate(self, data: List[int]):
        """Validates the data. Data must be a list of int"""
        data[4] = data[1] * data[2] % data[3]
        return data[4] > 21
```

```python
# file production_code.py

def production_code() -> bool:
    userForm = UserForm()
    # data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    data = range(10)
    # ...
    return validate(userForm, data)
```

```python
# file production_code.py

def production_code() -> bool:
    userForm = UserForm()
    # data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    data = range(10)
    # ...
    return validate(userForm, data)
```

```
$ mypy production_code.py
production_code.py:7: error: Argument 1 to "validate" has
incompatible type "range"; expected "List[int]“
```

```python
# file production_code.py

def some_production_code() -> bool:
    userForm = UserForm()
    # data = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
    data = list(range(10))
    # ...
    return validate(userForm, data)
```

```
$ mypy production_code.py
$
$ python production_code.py
…
```

Let’s practice

Monkeytype

Let the monkey find the types for you

```python
def validate(form, data):
    """Validates the input data"""
    return form.validate(data)
```

Let the monkey find the types for you

```python
def validate(form, data):
    """Validates the input data"""
    return form.validate(data)
```

```
$ monkeytype run all_unit_tests.py
$ monkeytype run end_to_end_tests.py
$ monkeytype run production_code.py
$ monkeytype apply validate
```

Let the monkey find the types for you

```python
def validate(form, data):
    """Validates the input data"""
    return form.validate(data)
```

```
$ monkeytype run all_unit_tests.py
$ monkeytype run end_to_end_tests.py
$ monkeytype run production_code.py
$ monkeytype apply validate
```

```python
def validate(form: Union[UserForm, AdminForm],
             data: List[int]) -> bool:
    """Validates the input data"""
    return form.validate(data)
```

You can also use *PyAnnotate* which does the same thing.

Conclusion

- Type annotation is powerful to bug finder. Use it !
- Type annotation is also good way of documenting your code
- Feedback from developers using type annotation: “It rocks !”
- Some Python dynamic constructs are difficult to verify statically
  - That’s why you should go step-by-step when adding annotations
  - *Mypy* has excellent documentation to complement this presentation
- Tools like *MonkeyType* or *PyAnnotate* can really help.

Philippe Fremy / IDEMIA in Bordeaux

(IDEMIA is recruiting)

philippe.fremy@idemia.com

Slides are online at PyParis website
and at:
https://github.com/bluebird75/whoiam

Time for questions