Keep Those Ducks in (Type) Check!

Francesco Pierfederici

Hi, I am Francesco.
Python trainer (hire me!)
Engineering director @ IRAM
Loooove Python!

Introductory Stuff

- Standard EuroPython training (three hours with a break)
- Ask questions any time (trainings are not recorded)
- Be mindful of others when leaving & coming back
- I will be using Python 3.8 & mypy 0.711+
- Code on github.com/pythoninside/europython2019
- Follow along and have fun!

Types in Python???

Do not panic

But Why?

- Documentation
- Detect some bugs
- Performance? Maybe some day?

Saving Grace

- Code behaviour not affected
- Gradual typing

Epiphany

- Type checkers to a LOT of work
  - Maybe “static code safety checkers”?
- They are like linters on steroids (and much more)

Type Checkers

- mypy (Dropbox)
  - `shell> mypy [OPTIONS] ./some_dir/ /path/to/some/code.py`
- pyre (Facebook)
- pyright (Microsoft)
- pytype (Google)

Type Annotations

(Python 3 Syntax)

Functions

```python
from typing import Callable

# Single argument
def square_root(x: float) -> float:
    return x ** .5

# Default values
def shift_left(x: int, places: int = 1) -> int:
    return x << places

# No return/return None
def greetings(name: str) -> None:
    print(f'Hello {name}')

# The type of a function/callable
fn: Callable[[int, int], int] = shift_left
```

Variables

```
from typing import Tuple

def fast_fib(n: int) -> int:
    assert n >= 0, 'Expecting a non-negative integer'

    seq: Tuple[int, int, int]
    seq = (0, 1, 1)
    if n < 3:
        return seq[n]

    nminustwo: int = 1
    nminusone: int = 1
    for i in range(3, n + 1, 1):
        nminusone, nminustwo = nminustwo + nminusone, nminusone
    return nminusone
```

Forward References

- Python 3.7+ import *annotations* (from *__future__*)
- Use the type name in quotes
- Use type comments (see next slides)

Type Annotations

(Python 2 Syntax)

Use Comments!

```python
from typing import List, Union, Optional


a_complex_list = [1, '2', 3, 4, '5']  # type: List[Union[int, str]]


# Sometimes we return something, other times nothing: Optionals!
# Optional[int] = Union[int, None]
def find_element_index(el, elements):
    # type: (int, List[int]) -> Optional[int]
    if el in elements:
        return elements.index(el)
    return None     # required by the type checker
```

Type Annotations

(Python 3 Syntax Continued)

Builtin Types

```python
from typing import Tuple, List, Set, Dict

# Built-in types
an_integer: int = 1
a_float: float = 1.2
a_string: str = 'Hello there!'
some_bytes: bytes = b'Hello there!'
a_boolean: bool = False

# Simple collections
a_list: List[int] = [1, 2, 3]
a_set: Set[int] = {1, 2, 3}

# Tuples can be heterogeneous
a_tuple: Tuple[int, str, bool] = (1, 'foo', True)
another_tuple: Tuple[int, ...] = (1, 2, 3, 4)

# Dictionaries need types for keys and values
a_dict: Dict[str, float] = {'one': 1.0, 'two': 2.0}
```

None

```python
import traceback
from typing import Optional, TypeVar

# Use None as the return type of functions which do not return a value
def greet(name: str) -> None:
    print(f'Hello {name}')
    # No need for an explicit return here

# Do not assign the result of greet to a variable
foo = greet('Francesco')

AnyException = TypeVar('AnyException', bound=Exception)

# Beware when using Optionals (which can be None)
def print_traceback(ex: Optional[AnyException]) -> None:
    # traceback.print_tb(ex.__traceback__)  # type error: ex could be None!
    if ex:                                  # type checker understands this
        traceback.print_tb(ex.__traceback__)
```

Unions & Optionals

```python
from typing import List, Union, Optional


a_complex_list: List[Union[int, str]]
a_complex_list = [1, '2', 3, 4, '5']

# Sometimes we return something, other times nothing: Optionals!
# Optional[int] = Union[int, None]
def find_element_index(el: int, elements: List[int]) -> Optional[int]:
    if el in elements:
        return elements.index(el)
    return None     # required by the type checker
```

Callables

```python
from typing import Callable

# The type of a function/callable
# Types in lambdas are usually inferred and not annotated
fn: Callable[[int, int], int] = lambda x, y: x + y

# Callable object with any number and type of argument
decorator: Callable[..., int]
```

Coroutines & Generators

```python
from asyncio import AbstractEventLoop
from socket import socket
from typing import Optional, Iterator

def my_range(n: int) -> Iterator[int]:
    # while i:=0 < n:  <- assignment expressions not supported :-(
    i = 0
    while i < n:
        yield i
        i += 1
    return 'foo'  # <- this is embed in the StopIteration exception

async def connection_handler(client: socket, loop: AbstractEventLoop) -> None:
    while True:
        data: Optional[bytes] = await loop.sock_recv(client, 10000)
        if not data:
            break
        await loop.sock_sendall(client, data)
    print('Connection closed')
    client.close()
```

Coroutines II

```python
from typing import Generator

#                            Generator[YieldType, SendType, ReturnType]
def echo_n_times(n: int) -> Generator[str, str, int]:
    value = 'Please type something'
    orig_n = n
    while n >= 0:
        value = yield value
        n -= 1
    return orig_n
```

Classes

```
from typing import ClassVar

class Point:
    x: int       # considered an instance variable
    y: int       # considered as instance variable
    num_points: ClassVar[int] = 0       # class variable

    def __init__(self, x: int, y: int) -> None:  # Do not annotate self
        self.x = x
        self.y = y
        Point.num_points += 1

class Point3D(Point):
    z: int

    def __init__(self, x: int, y: int, z: int) -> None:
        super().__init__(x, y)
        self.z = z

p = Point(1, 2)
# p.x = 1.2               # type error
# p.num_points += 1       # p cannot write to a class variable
print(p.num_points)      # OK
p3 = Point3D(1, 2, 3)
# p3 = p                  # error: cannot use a Point in place of a Point3D
p = p3                   # OK: Point3D upgraded to the super type
```

Leave Me Alone!

```python
from typing import Any, List, Dict, cast

a: List     # equivalent to List[Any]
b: Dict     # equivalent to Dict[Any, Any]

a = [1, 'foo']
# a = 123   # this would fail
b = {'a': 1, 'b': 'foo'}
c = cast(str, a)    # mypy belives us
c << 2      # mypy error as it assumes c to be a string
c += 3      # type: ignore

def foo(x: Any) -> str:
    print(x + 1)    # not type-checked
    return x        # not type-checked, but return necessary
```

Advanced Topics

Optionals Can Be a Pain

```
from typing import Optional, List

def find_element_index(el: int, elements: List[int]) -> Optional[int]:
    if el in elements:
        return elements.index(el)
    return None     # required by the type checker

x = 3
xs = [1, 2, 3, 4, 5, 6]
i = find_element_index(x, xs)
print(f'{x} is element number {i + 1} of {xs!r}')  # mypy error
```

Overloaded Functions

```python
from typing import Optional, overload

# Example: the create_user function could be defined with Optionals only but a
# better solution could be this:
@overload
def create_user(user_id: None) -> None:
    ...         # <- note the ellipsis

@overload
def create_user(user_id: int) -> User:
    ...         # <- note the ellipsis

# Implementation (User class defined somewhere)
def create_user(user_id: Optional[int]) -> Optional[User]:
    if user_id is None:
        return None
    return User.mkuser(user_id)

user = create_user(123)
_ = create_user(None)
```

Type Variables

```python
from typing import MutableSequence, TypeVar

# Define an unbound type variable
T = TypeVar('T')        # <- can be any type

# Now a bound type variable (it is actually already in the typing module, btw)
AnyStr = TypeVar('AnyStr', str, bytes)  # <- can be either str or bytes

# And finally a type variable with an upper bound
AnyAnimal = TypeVar('AnyAnimal', bound=Animal)

def append(x: T, xs: MutableSequence[T]) -> None:
    return xs.append(x)

def concatenate(s1: AnyStr, s2: AnyStr) -> AnyStr:
    return s1 + s2

def greet(animal: AnyAnimal) -> None:
    print(f'Hello {animal.__class__.__name__.lower()}')
```

Type Variables II

- Placeholders for a type
- Can be read as “is a type of” or “is an instance of”
- NOT the same as Union
  - Once bound, a type variable is always the same type

Generics

```python
# We can use type variables to create generic types ourselves. We have already
# seen the use of type variables in generic types in the typing module like
# e.g., List[T] or Dict[T, S]
from typing import Generic, List, TypeVar

T = TypeVar('T')

class Vector(Generic[T]):
    def __init__(self, elements: List[T]) -> None:
        self.elements = elements

    def pop(self) -> T:
        return self.elements.pop()

# We can also define generic functions
def head(v: Vector[T]) -> T:
    # return v[0]   # error: Vector does not define __getitem__
    return v.elements[0]
```

Where Can I Use Generics?

- Things get complicated when we throw sub/super types in the mix
  - Can I use List[int] where List[float] is expected? Vice-versa?
  - What about Tuple[int, …] and Tuple[float, …]?
  - What about Callables?
  - …

Variance

- Generic types are called
  - Covariant if they preserve ordering of types
  - Contravariant if they reverse that order
  - Invariant if are neither of the two forms

Variance Examples

- **Covariant**
  - Union
  - Most inmutable containers
  - Callable (only in the return type)
- **Contravariant**
  - Callable (in the argument types)
- **Invariant**
  - Most mutable containers and mappings

Covariance

```python
from typing import Callable

def mkint() -> int:
    return 42

def mkfloat() -> float:
    return 3.14

def process_float(fn: Callable[[], float]) -> None:
    x = fn()
    res = x ** .5
    print(f'{x} -> {res}')

def process_int(fn: Callable[[], int]) -> None:
    x = fn()
    res = x << 1
    print(f'{x} -> {res}')

# Callables are covariant in their return types. This means that we should be
# able to use a function that return an int where one that returns a float is
# expected (assuming that the arguments are the same).
process_float(mkint)

# The reverse is not true.
process_int(mkfloat)        # error!
```

Contravariance

```python
from typing import Callable, TypeVar

T = TypeVar('T')

def proc_int(x: int) -> None:
    res = x << 1
    print(f'{x} -> {res}')

def proc_float(x: float) -> None:
    res = x ** .5
    print(f'{x} -> {res}')

def pipeline(x: T, fn: Callable[[T], None]) -> None:
    fn(x)

x: int = 42
pipeline(x, proc_float)     # OK
y: float = 3.14
pipeline(y, proc_int)       # Error
```

Invariance

```python
from typing import TypeVar, Callable, List

T = TypeVar('T')

def pipeline(data: List[T], data_processor: Callable[[T], T]) -> None:
    """A simple data pipeline. """
    for el in data:
        res = data_processor(el)
        print(f'{el} -> {res}')

def int_proc(n: int) -> int:
    """Some operation not available to floats."""
    return n << 1

def float_proc(x: float) -> float:
    return x ** .5

# Can we use List[int] where List[float] is expected?
ints: List[int] = [1, 2, 3, 4, 5]
floats: List[float] = [1., 2., 3., 4., 5.]
pipeline(floats, int_proc)         # Error
pipeline(ints, float_proc)         # Error
```

Variance: Custom Generics

- User-defined Generics are **invariant** by default
  - Can specify variance by hand (in their type variables)

User-Defined Generics

```python
from typing import TypeVar, Generic

T_co = TypeVar('T_co', covariant=True)

class Foo(Generic[T_co]):
    def __init__(self, element: T_co) -> None:
        self._x = element

    def bar(self) -> None:
        print(f'self._x = {self._x}')

x: Foo[int] = Foo(42)
y: Foo[float] = Foo(3.14)

x = y           # Error: I cannot simply replace a Foo[int] by a Foo[float]
y = x           # But I can replace a Foo[float] by its subtype
# Similarly
tx = (1, 2, 3)
ty = (1., 2., 3.)
tx = ty         # Error: Tuple is covariant in its arguments
ty = tx         # OK
```

Protocols

- Structural subtyping / duck typing support
- Types defined by attributes / methods
- Subclasses of *typing.Protocol* (Python 3.8+) or *typing_extensions.Protocol*

Pre-Defined

`Iterable[T]`	`def __iter__(self) -> Iterator[T]`
`Iterator[T]`	`def __next__(self) -> T` `def __iter__(self) -> Iterator[T]`
`Sized`	`def __len__(self) -> int`
`Container[T]`	`def __contains__(self, x: object) -> bool`

`Collection[T]`	`def __len__(self) -> int`
	`def __iter__(self) -> Iterator[T]`
	`def __contains__(self, x: object) -> bool`

`Awaitable[T]`	`def __await__(self) -> Generator[Any, None, T]`
`AsyncIterable[T]`	`def __aiter__(self) -> AsyncIterator[T]`
`AsyncIterator[T]`	`def __anext__(self) -> Awaitable[T]` `def __aiter__(self) -> AsyncIterator[T]`

`ContextManager[T]`	`def __enter__(self) -> T`
	`def __exit__(self,`
	`exc_type: Optional[Type[BaseException]],`
	`exc_value: Optional[BaseException],`
	`traceback: Optional[TracebackType]) -> Optional[bool]`

`AsyncContextManager[T]`	`def __aenter__(self) -> Awaitable[T]`
	`def __aexit__(self,`
	`exc_type: Optional[Type[BaseException]],`
	`exc_value: Optional[BaseException],`
	`traceback: Optional[TracebackType]) -> Awaitable[Optional[bool]]`

Custom Protocols

```python
from typing import Protocol

class Event: pass

class AppDelegate(Protocol):
    def finished_launching(self, event: Event) -> None: ...

    def should_terminate(self, event: Event) -> bool: ...

class NamedAppDelegate(AppDelegate, Protocol):
    name: str

class Application:
    delegate: AppDelegate

class Delegate:
    def __init__(self, name: str) -> None:
        self.name = name

    def finished_launching(self, event: Event) -> None:
        print(f'{self.name}: yippy!')

    def should_terminate(self, event: Event) -> bool:
        print(f'{self.name}: bye')
        return True

app = Application()
app.delegate = Delegate('foo')
```

Custom Protocols

```python
from typing import Protocol

class Event: pass

class AppDelegate(Protocol):
    def finished_launching(self, event: Event) -> None: ...

    def should_terminate(self, event: Event) -> bool: ...

class NamedAppDelegate(AppDelegate, Protocol)
    name: str

class Application:
    delegate: AppDelegate

class Delegate:
    def __init__(self, name: str) -> None:
        self.name = name

    def finished_launching(self, event: Event) -> None:
        print(f'{self.name}: yippy!')

    def should_terminate(self, event: Event) -> bool:
        print(f'{self.name}: bye')
        return True

app = Application()
app.delegate = Delegate('foo')
```

Protocol Tricks

- No need to subclass a Protocol to adhere to it
- Sub-protocols: inherit from Protocol *explicitly*
  - As opposed to just subclassing the parent = implementing it
- Protocols can define std method implementations
  - But need to inherit from the protocol to get them
- Protocols can be recursive
- *isinstance()* sort of works (use *@runtime decorator*)
- Implement __*call*__ to describe *Callables*!

Odds & Ends

```python
# Some tricks and random convenience stuff
from typing import Iterator

# Positional-only args in callables with __
def irange(__n: int) -> Iterator[int]:
    i = 0
    while i < __n:
        yield i
        i += 1

print(list(irange(__n=10)))     # error


# Convenience shorthand for *args and **kwargs
def foo(*args: int, **kwargs: str) -> None:
    print('Hi there')

foo(1, 2, 3, a='bar', b='baz')
```

Odds & Ends II

- MyType = int is a type alias
- MyType = NewType('MyType', int) is a completely new type

Type Annotations

(Stubs)

Stub Files

- Same name as the corresponding Python file, with .pyi extension
- Use Python 3.6+ syntax (even for Python 2 code)
- Only include declarations, no logic
- Logic is replaced by "…" (ellipsis)
- mypy ships with many stub files for stdlib etc.
  - https://github.com/python/typeshed

Strategy

- Always run checker in the same way
  - Pin checker & version
  - Freeze options & config
- Start checking code as is
  - I.e. check top-level declarations only
- Add type annotations
  - Start small (only critical & new code at first)
  - mypy: check_untyped_defs = True ?

Strategy II

- Concentrate on callables
- Annotate variables only if asked to
- Think carefully before using Union/Optional
    - You have to check them for None

Get Some Help

- Add type annotations automatically
    - Mypy’s stubgen (static)
    - MonkeyType (runtime)
    - PyAnnotate (runtime)
    - pytype (static)