Practical uses for

Function Annotations

Manuel Ceron
manuel.ceron@booking.com

About me...

- Colombian developer living in the Netherlands
- First time at EuroPython
- Programming Python for 10 years
- Currently working at

Booking.com

Currently hiring...
www.booking.com/jobs

What are Function Annotations?

Syntactic sugar for adding metadata to function definitions

```
def compile(source: "something compilable",
            filename: "where the compilable thing comes from",
            mode: "is this a single statement or a suite?"):
```

```
def greet(name: str, age: int) -> str:
    return 'Hello {0}, you are {1} years old'.format(name, age)
```

PEP 3107

Python 3.x only

First draft since 2006

Annotations are just syntactic sugar:

```
>>> def distance(p1: Point, p2: Point) -> float:
...     return math.sqrt( (p2.x - p1.x)**2 + (p2.y - p1.y)**2 )
...

>>> distance.__annotations__
{'p2': <class '__main__.Point'>,
 'p1': <class '__main__.Point'>,
 'return': <class 'float'>}
```

Why this talk

- Converging static and dynamic typing
- Potential of optional static typing
- Why nobody uses this feature

```
1
2 function findTitle(title: string) {
3     var titleElement = document.getElementById('title-' + title);
4     return titleElement;
5 }
6
7 var t = findTitle('mytitle');
8
```

Adding Typing Information

```
def open(filename: str,
         mode: str,
         encoding: str = None,
         buffering: int = -1,
         closedfd: bool = True) -> IOBase:
    ...
```

Why:

- Help tools to make life easier.
- Some auto documentation.

Helping tools (IDE, Editor)

```
def authenticate(self, request):
    request.
```

```
def authenticate(self, user, password):
    return user.check_pasword(password):
```

```
class User:
    def check_password(password):
        ...
```

Helping tools (IDE, Editor)

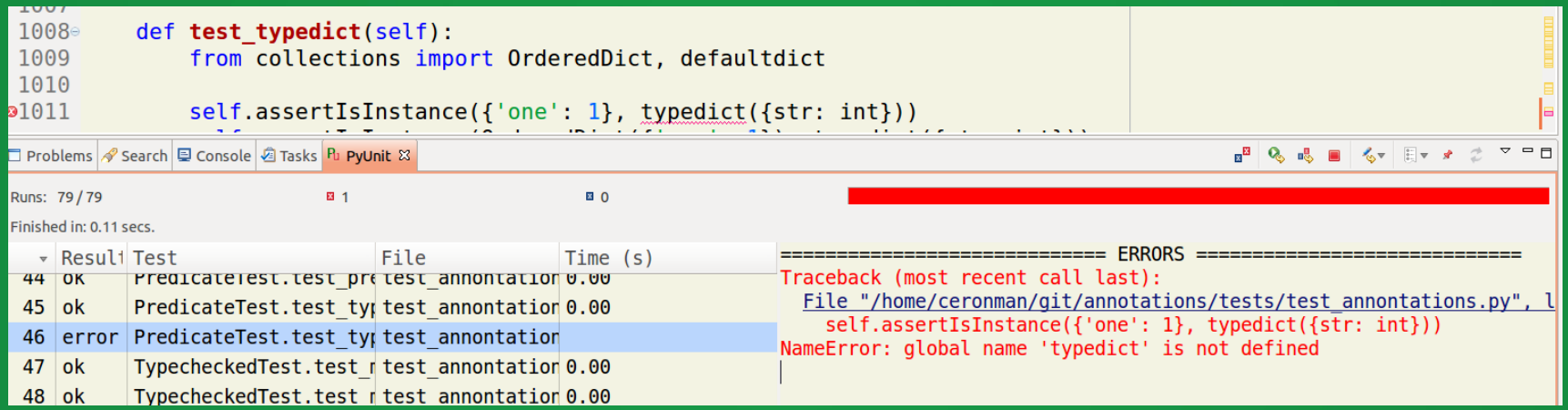
```
def authenticate(self, request: HttpRequest):
    request.
```

```
def authenticate(self, user: auth.User, password):
    return user.check_pasword(password):
```

```
class User:
    def check_password(password):
        ...
```

Wait a sec!

You don't need this if you write Unit Tests



It's not about getting the error,
but when and how you get it.

Wait a sec!

You are missing the point of Duck Typing

Duck typing is not incompatible with type annotations

More on this later

Wait a sec!

Writing types is too verbose

- Remember: annotations are optional
- In practice dynamic languages can be very verbose

Libraries usually do this

```
def attach_volume(self, volume_id, instance_id, device):
    """
    Attach an EBS volume to an EC2 instance.

    :type volume_id: str
    :param volume_id: The ID of the EBS volume to be attached.

    :type instance_id: str
    :param instance_id: The ID of the EC2 instance to which it will
                        be attached.

    :type device: str
    :param device: The device on the instance through which the
                   volume will be exposted (e.g. /dev/sdh)

    :rtype: bool
    :return: True if successful
    """

    params = {'InstanceId': instance_id,
              'VolumeId': volume_id,
              'Device': device}
    return self.get_status('AttachVolume', params, verb='POST')
```

Experiment: Runtime checking

- Probably a very bad idea...
  - It's slow... (up to 200x slow)
  - Not very useful
- Testbed for concepts though
- Might become useful if we expand the notion of type

Runtime checking: how it looks

```
>>> @typechecked
... def test(a: int) -> int:
...     return a
...
>>> test(1)
1
>>> test('string')
Traceback (most recent call last):
  ...
TypeError: Incorrect type for "a"
```

Warning: slow code

- The @typechecked decorator checks arguments and uses isinstance to determine if the they match the annotation

But isinstance is killing duck typing!

Or not...:

```
 >>> class IterableWithLength(Interface):
...     def __iter__():
...             pass
...     def __len__():
...             pass
...
>>> isinstance([1, 2, 3], IterableWithLength)
True

>>> isinstance({'one': 1, 'two': 2}, IterableWithLength)
True

>>> isinstance((x for x in range(10)), IterableWithLength)
False

>>> isinstance(1, IterableWithLength)
False
```

Another example

```
>>> class Person(Interface):
...     name = str
...     age = int
...     def say_hello(name: str) -> str:
...             pass


>>> class Developer:
...     def __init__(self, name, age):
...             self.name = name
...             self.age = age
...     def say_hello(self, name: str) -> str:
...             return 'hello ' + name
...
>>> isinstance(Developer('bill', 20), Person)
True
```

Implementation of structural interfaces:

- A little bit of python black magic:
  - Metaclasess
  - __instancecheck__
  - __subclasscheck__

Types + behaviour = predicates

```
>>> Positive = predicate(lambda x: x > 0)
>>> isinstance(1, Positive)
True
>>> isinstance(0, Positive)
False

>>> @typechecked
... def sqrt(n: Positive):
...     ...
>>> sqrt(-1)
Traceback (most recent call last):
    ...
TypeError: Incorrect type for "n"
```

More fun with predicates

```
 >>> FileMode = options('r', 'w', 'a', 'r+', 'w+', 'a+')
>>> isinstance('w', FileMode)
True
>>> isinstance('x', FileMode)
False


>>> isinstance(True, only(bool))
True
>>> isinstance(1, only(bool))
False


>>> isinstance(1, optional(int))
True
>>> isinstance(None, optional(int))
True


>> @predicate
... def Even(object):
...     return object % 2 == 0
>>> EvenAndPositive = Even & Positive
```

The open function annotated again

```
def open(filename: FilenameString,
         mode: options('r', 'w', 'a', 'r+', 'w+', 'a+'),
         encoding: optional(str) = None,
         buffering: int = -1,
         closedfd: bool = True) -> FileInterface:
  ...
```

Unions

```
 >>> NumberOrString = union(int, str)
>>> isinstance(1, NumberOrString)
True
>>> isinstance('string', NumberOrString)
True
>>> issubclass(int, NumberOrString)
True
>>> issubclass(str, NumberOrString)
True


def isinstance(object, class_: union(type, tuple)) -> bool:
    ...
```

Function overloading

```
@overload
def isinstance(object, t: type):
    ...

@isinstance.add
def isinstance(object, t: tuple):
    ...


# PEP 443

>>> from functools import singledispatch
>>> @singledispatch
... def fun(arg):
...     print(arg)

>>> @fun.register(collections.MutableMapping)
... def _(arg):
...     for k, v in arg.items():
...         print(k, '->', 'v')
```

More kinds of types

typedefs:

```
 >>> @typedef
...  def EventHandler(event: Event) -> bool:
...     pass
```

```
>>> def handler(event: MouseEvent) -> bool:
...     print('click')
...     return True
...
>>> isinstance(handler, callback)
True
>>> isinstance(lambda: True, callback)
False
```

Parameterized types

```
typedict({str: int})
typeseq([int])
typeseq(set(int))
typeseq((int,))
```

A different approach to type annotations: rightarrow

```
Named types: int, long, float, complex, str, YourClassNameHere, ...
Lists: [int], [[long]], ...
Tuples: (int, long), (float, (int, Regex)), ...
Dictionaries: {string: float}, { (str, str) : [complex] }, ...
Unions: int|long|float, str|file, ...
"Anything goes": ??

Functions:
str -> int
(int) -> int
(int, int) -> int
( (int, int) ) -> int
( str|file ) -> SomeClass
(int, *[str]) -> [(str, int)]
(int, *[int], **{int: str}) -> str

Objects:

object(self_type, field1: int, field2: str, ...)
```

Using rightarrow

```
 >>> @guard('unicode -> str')
... def safe_encode(s):
...     return s.encode('utf-8')

>>> safe_encode(u'hello')
'hello'
>>> safe_encode('\xa3')
TypeError: Type check failed: ? does not have type unicode
```

Or better:

```
>>> def safe_encode(s: str) -> bytes:
...     return s.encode('utf-8')
```

Other uses for annotations

Documentation:

```
def attach_volume(self,
                  volume_id: "The ID of the EBS volume "
                             "to be attached",
                  instance_id: "EC2 Instance to which it "
                               "will be attached",
                  device: "The device on the instance "
                          "through which the "
                          "volume will be exposed "
                          "(e.g. /dev/sdh) ") -> "True if successful":
    ...
```

Combining different uses of annotations:

```
def compile(source: (str, "something compilable"),
            filename: (str, "where the compilable thing comes from"),
            mode: (int, "is this a single statement or a suite?")):
```

Annotations for language bridges

```
@ctypes_import('libc.strchr')
def strchr(s: 'char *', pos: 'short') -> 'char *':
    pass
```

```
@sql_insert
def strchr(s: 'VARCHAR', pos: 'INTEGER'):
    ...
```

Why annotations are not used?

- Not a wide know feature
- Python 3.x only
- Too much flexibility
- Some use cases not complete
- Not immediate benefit

Questions?

Meanwhile:

typeannotations:
https://github.com/ceronman/typeannotations

rightarrow:
https://github.com/kennknowles/python-rightarrow

PEP 3107:
http://www.python.org/dev/peps/pep-3107/

Thanks!