# USING TYPE CHECKING IN DJANGO PROJECTS WITH MYPY

## LONDON DJANGO - DJANGO SEPTEMBER 2016 MEETUP

Daniel F Moisset - dmoisset@machinalis.com / @dmoisset

# ABOUT ME

Hi! I'm Daniel Moisset!

dmoisset@machinalis.com / @dmoisset

I've worked with Django for more than 10 years

I work at Machinalis

Looking for awesome devs to build stuff for you in Python/Django? let me know!

# LET'S START WITH A STORY

Writing a django view, which sometimes redirects

```
def my_view(request, pk):
    obj = get_object_or_404(SomeModel, pk=pk)
    if obj.should_redirect():
        response = HttpResponseRedirect(obj.redirect_url())
    else:
        response = render(request, 'some_template.html',
                          context={'object': obj})
    return
```

Oops! I forgot the return expression!

# BUT I HAVE TESTS!

So I'm lucky to get a nice ValueError

```
# Complain if the view returned None (a common error).
if response is None:
    if isinstance(callback, types.FunctionType):    # FBV
        view_name = callback.__name__
    else:                                           # CBV
        view_name = callback.__class__.__name__ + '.__call__'

    raise ValueError(
        "The view %s.%s didn't return an HttpResponse object. It "
        "returned None instead." % (callback.__module__, view_name)
    )
```

# SO, WHAT'S WRONG?

- My tests are not written to detect this!
- The error message does not always point me directly at the problem
- I could add tests for every view, but it would be cumbersome!

*But if I had some kind of tooling to help me with this...*

# A BRIEF HISTORY OF TYPES IN PYTHON

# THE AGE OF THE ANCIENTS...

- 1999: Python devs in types-sig (GvR, Jim Fulton). `zope.interface` was born here.
- 2004: Artima posts

> *Optional static typing has long been requested as a Python feature. It's been studied in depth before (e.g. on the type-sig) but has proven too hard for even a PEP to appear. In this post I'm putting together my latest thoughts on some issues, without necessarily hoping to solve all problems. — Guido van Rossum*

# PYTHON 3

- 2006: PEP 3107, function annotations
- 2008: Python 3.0 released

```
def gcd(a: int, b: int) -> int:
    while a:
        a, b = b%a, a
    return b
```

```
def gcd(a: list.append, b: 'bananas') -> (42,):
    while a:
        a, b = b%a, a
    return b
```

# THE MIDDLE AGES

2010:I presented a talk about typing... The summary about annotations was "*Nobody* is using it*". The important question is "*Why?*"

- A mechanism for describing interfaces.
- Benefits from network effects.
- But no standard.
- (and only available in Python 3)

# A NEW HOPE

- 2012: Jukka Lehtosalo starts mypy as a python-like compiled language with static types
- 2013: Mypy presented in PyCon.US 2013. GvR, Armin Rigo gave feedback
  - No one will use it if it has a different syntax
  - The new VM isn't that exciting, the ability to do checking is!
- So mypy started losing its code generation abilities.
- And making decisions on how to use PEP3107 annotations

# GOING OFFICIAL

- 2015 (Jan): First Draft of PEP-484 made public
- 2015 (May): PEP-484 accepted. Some 3rd parties already adopting it (pyCharm, pytype)
- 2015 (Sep): Python 3.5 released. Includes provisional `typing` module in the stdlib.

---

*PEP 3107 introduced syntax for function annotations, but the semantics were deliberately left undefined. There has now been enough 3rd party usage for static type analysis that the community would benefit from a standard vocabulary and baseline tools within the standard library*

---

# CURRENT STATUS

- 2016 (Jan):
  - GvR (and more dropbox engineers) directly involved
  - python 2 support
  - repo moved to the python organization
  - typesheds shared between projects
- 2016 (Aug): MyPy 0.4.4 released
- 2016 (Sep): ← YOU ARE HERE

# THIS IS *NOT* JAVA

*Finally, if you are worried that this will make Python ugly and turn it into some sort of inferior Java, then I share you concerns, but I would like to remind you of another potential ugliness; operator overloading. C++, Perl and Haskell have operator overloading and it gets abused something rotten to produce "concise" (a.k.a. line noise) code. Python also has operator overloading and it is used sensibly, as it should be. Why? It's a cultural issue; readability matters.* — Mark Shannon, 2015

# PYTHON IS ALREADY TYPED

Everything is an object!

But Python (CPython) is written in C…

Everything is a `PyObject *`

(not just a silly technicality, bear with me)

# STRUCTURES VS TYPES

"Type" can mean two different things:

- The *structure* or memory layout for an object on runtime
- A *label* (implicit or explicit) over some piece of source denoting an object.
- Sometimes, a name for an interface (as in API) of several objects.

# A LABEL FOR PYTHON OBJECTS

PEP-484 gives a label for objects called “Any”:

```
from typing import Any

def gcd(a: Any, b: Any) -> Any:
    while a:
        a, b = b%a, a
    return b
```

# A LABEL FOR PYTHON OBJECTS

Which is actually the default, so equivalent to

```
def gcd(a, b):
    while a:
        a, b = b%a, a
    return b
```

That looks a lot like, umm, Python?

# GRADUAL TYPING

- Use type only annotations only if you want.
- Use type only annotations only where it makes sense.
- You can mix up static and dynamic code freely
- Type checker is separate, independent of run-time semantics

# GRADUAL TYPING: AN EXAMPLE

```
def dynamic_gcd(a, b):
    while a:
        a, b = b%a, a
    return b

def static_gcd(a: int, b: int) -> int:
    while a:
        a, b = b%a, a
    return b

assert static_gcd(21, 14) == 7   # Allowed, and ok
assert dynamic_gcd(21, 14) == 7  # Allowed, and ok

d = dynamic_gcd("foo", "bar")  # OK in typechecker, TypeError in runtime
d = dynamic_gcd("", "%s")      # This is allowed, it takes Any

static_gcd(d, d)       # OK! d is Any, can be used as int. OK in runtime
static_gcd("", "%s")   # Error on typecheck, ok in runtime
static_gcd(10, "foo")  # Error on typecheck, TypeError in runtime
```

# IF EVERYTHING IS ANNOTATED "ANY"...

... then you're not doing any checking

- **Any** tends to be contagious
- So it's better if libraries are annotated. And most aren't...

# TYPESHEDS

- Python files (with “pyi” extensions) in a tree mimicing a library.
- Only type annotations.
- Official "typeshed" repo with most stdlib, some 3rd party.
- Possible to add more with MYPYPATH env var.
- Working on django support at github.com/machinalis/mypy-django

# OOPS, THIS WAS A DJANGO MEETUP RIGHT?

What's available now:

- Request and response objects
- Generic views
- URL resolver
- Other minor bits and pieces (django.core.files, timezone, django.utils.datastructures, ...)

Coming up:

- django.shortcuts
- admin

# SO BACK TO THE STORY

- Install mypy (pip install mypy-lang), clone the django typesheds
- Set the env var MYPATH=/path/to/django/typeshed
- The current code passes ok with `mypy views.py`... Everything is an Any

```
$ mypy views.py
views.py:2: error: No library stub file for module 'django.shortcuts'
```

So I prefer to use

```
$ mypy --disallow-untyped-defs --strict-optional views.py
views.py:2: error: No library stub file for module 'django.shortcuts'
views.py:2: note: (Stub files are from https://github.com/python/typeshed)
views.py: note: In function "my_view":
views.py:8: error: Function is missing a type annotation
```

# NOW WITH ANNOTATIONS

```
def my_view(request: HttpRequest, pk: int) -> HttpResponse:
    obj = get_object_or_404(SomeModel, pk=pk)
    if obj.should_redirect():
        response = HttpResponseRedirect(obj.redirect_url())
    else:
        response = render(request, 'some_template.html',
                          context={'object': obj})
    return
```

And then…

```
$ mypy --disallow-untyped-defs --strict-optional views.py
views.py:2: error: No library stub file for module 'django.shortcuts'
views.py:2: note: (Stub files are from https://github.com/python/typeshed)
views.py: note: In function "my_view":
views.py:14: error: Return value expected
```

# REMOVING WARNINGS

```python
from django.http import HttpRequest, HttpResponse, HttpResponseRedirect
from django.shortcuts import render, get_object_or_404  # type: ignore

def my_view(request: HttpRequest, pk: int) -> HttpResponse:
    ...
    return response
```

All ~~tests~~ type checks passing!

# WHY?

# AVOIDING BUGS?

- No. Your tests are better. But…
- Typecheck is fast, so you can run it all the time. Even from your editor.
- So you save time in the write/edit/test cycle.
- Very useful when refactoring.

# DOCUMENTATION

- YES! Annotations are labels specifying author intent
- Like a docstring, without the code rot.
- Like a doctest, but compact.
- Very useful when refactoring.

Better than

---

*If all went well, a file-like object is returned. This supports the following methods: read(), readline(), readlines(), fileno(), close(), info(), getcode() and geturl(). It also has proper support for the iterator protocol. One caveat: the read() method, if the size argument is omitted or negative, may not read until the end of the data stream; there is no good way to determine that the entire stream from a socket has been read in the general case. — Python 2 documentation for* `urllib` *module*

---

# DJANGO ALREADY DOES IT! (AD-HOC)

- Checks like the one I shown
- manage.py check (ImproperlyConfigured)
- models are essentially typed

# POSSIBLE PROBLEMS

- If your code is too dynamic, don't use it
- If you depend an unsupported library, you may use it anyway
  - But you don't have checking, so your annotations may be inconsistent.
  - And you might have to adjust things if the library is updated later.
- Some details still needing fix (but it's moving quickly). See http://www.machinalis.com/blog/a-day-with-mypy-part-1/

# SOME OTHER RANDOM DETAILS

- The type system is quite flexible, it supports
  - Generic functions/classes
  - overloaded signatures
  - Union types
  - Some Duck typing support
- Sometimes you need to label vars, but usually inferred.
- Project is quite open.
- Help with typesheds welcome!

# THANKS

## QUESTIONS?

Daniel F Moisset - dmoisset@machinalis.com / @dmoisset