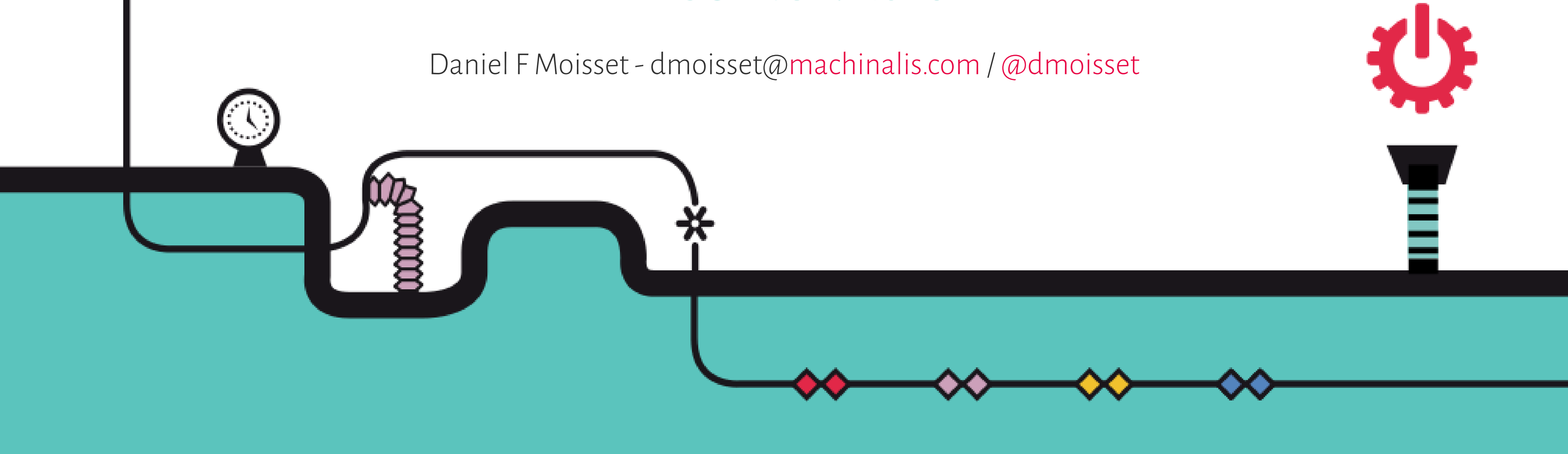
PYTHON AND STATIC TYPES: LET'S USE MYPY!

PYCON.UK 2016

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



ABOUT ME

Hi! I'm Daniel Moisset!

dmoisset@machinalis.com / @dmoisset

Python devel since 1.5.2 times

Also have worked in other statically-typed languages (including an Eiffel compiler)

Currently a minor contributor to mypy

I work at Machinalis

If you have a project and are looking for awesome devs to build stuff for you in Python/Django, let me know!

SHORT STORY: WHAT IS MYPY

A static *type-checker* for Python:

1. You add "type annotations" on the code (not everywhere, many can be inferred)
2. You run the mypy tool on your source code
3. You get warning messages if the annotations do not match the code

SHORT STORY: MINI-DEMO

```
ONE = "1"

def add_one(x: int) -> int:
    return x + ONE
```

Make sure to `pip install mypy-lang`

```
$ mypy basic_demo.py
basic_demo.py: note: In function "add_one":
basic_demo.py:4: error: Unsupported operand types for + ("int" and "str")
```

Note: mypy requires python 3 to *run*, but it can run *on* both python 2 and 3 code

ALREADY CONVINCED? YAY!

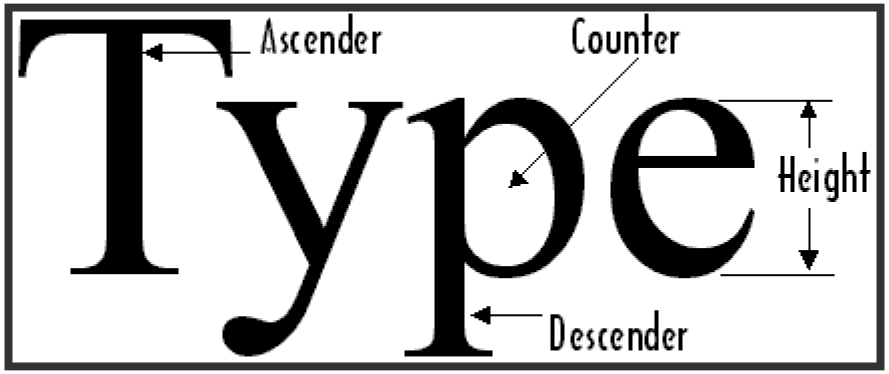
You can learn more reading:

- PEP 484
- The mypy documentation
- PEP 483 (for those who like more "theory")

THANK YOU

Oh, wait, you're not convinced yet?

WHY?



STRUCTURES VS LABELS

“Type” can mean (at least) 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.

WHY: FINDING BUGS?

- Python is “lazy” about reporting obvious type errors.
- Most type errors are obvious, a few aren't.
- You have good tests, right?
- Most usefulness at development time

SOME NOTES ABOUT TESTS

- Most tests are not written specifically to detect type errors
- The error messages do not always point directly at the problem
- You *could* add tests for every function, but it would be cumbersome!

You can also think of this as a very specific form of test.

WHY: READABILITY COUNTS!

- 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*

WHY: EXPLORING LEGACY CODE

For me, this was an accidental discovery:
http://www.machinalis.com/blog/a-day-with-mypy-part-1/

- It's easier to learn by explaining.
- But even better if your assumptions are checked.

WHY: TOOL INTEGRATION

- IDE integration (completions, go to definition)
- Assisted refactoring
- Documentation generators
- Optimization?

WHY NOW?

- The idea has been discussed on record since at least 1999
- Annotations (PEP 3107) Introduced in 2006, included in 2008 3.0 release
- MyPy appears around 2013
- PEP-484 standardizes in 2015 (some stdlib support in python 3.5)
- mypy got semiofficial and huge push from dropbox+guido in 2016

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

MANY PEOPLE

- have used very bad statically typed languages
- have used dynamically typed languages very badly

what's different in mypy?

MISCONCEPTIONS TO AVOID

- Static typing ≠ Explicit typing
- Static typing does not have to be mandatory

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)

A LABEL FOR PYTHON OBJECTS

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

```python
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

```python
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.

POSSIBLE PROBLEMS

- If your code is too dynamic, don't use it
- Duck typing has some ad-hoc support, fixes are planned
- 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/

SUMMARY

- Various good reasons to use it (mainly readability)
- Makes your code better
- You can choose how much or how little to use with a lot of granularity
- More useful if your libraries support it (little support now)
- Very dynamic code might get smaller benefit

THANKS

QUESTIONS?

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

CAN I USE IT IN PRODUCTION?

- Yes! it doesn't affect runtime
- Checking is quite fast (more using incremental mode), so good for CI

ANY VERSUS OBJECT

- Everything is an object
- Everything is an Any
- You can do anything on an Any
- You can do (almost) nothing on an object

LABELING VARS

```
# 'a' is automatically inferred as List[int]
a = [1, 2, 3]
# b needs an explicit annotation
b = []  # type: List[Employee]
# c is inferred as List[str] because of the append later
c = []
for item in a:
    c.append("item %d" % item)
```

- Most of them are inferred.
- It's nice to be explicit about instance/class vars
- (PEP 526) provides some nicer syntax support for this. But comments work if you don't have python 3.6 yet

PYTHON 2 SUPPORT

- PEP-484 gives a comment based syntax which works in python 2
- But `__anotations__` won't be available at runtime
- mypy itself runs on python 3, but `mypy --py2` understands python 2 correctly
- typeshed has different python versions (even minor) support.
- Still not clear how to support different library versions with typeshed

FLEXIBILITY OF TYPE SYSTEMS

- The type system is quite flexible, it supports
  - Generic functions/classes
  - overloaded signatures
  - Union types
  - Some Duck typing support