Python Typology

Matthias Kramm <kramm@google.com>

PyCon 2016

```
def make_announcement(emails):
    for addr in emails:
        send_email(addr)

make_announcement("users@goo.gl")
```

```
def make_announcement(emails):
    for addr in emails:
        send_email(addr)

make_announcement("users@goo.gl")
```

```
send_email("u")
send_email("s")
send_email("e")
send_email("r")
send_email("s")
…
```

```
from typing import List

def make_announcement(emails: List[str]):
    for addr in emails:
        send_email(addr)
```

```
from typing import List

def make_announcement(emails: List[str]) -> None:
    for addr in emails:
        send_email(addr)
```

```
from typing import List, Union

def make_announcement(
      emails: List[Union[str, bytes]]
) -> None:
    for addr in emails:
        send_email(addr)
```

```
from typing import Iterable, Unicode

def make_announcement(
      emails: Iterable[Union[str, bytes]]
) -> None:
    for addr in emails:
        send_email(addr)
```

Annotations don't change how Python works!

```
$ python3
Python 3.5.1 (default, Apr 12 2016, 11:08:00)
[GCC 4.8.4] on linux
Type "help", "copyright", "credits" or "license" for more
information.
>>> import typing
>>> def make_announcement(emails: typing.List[str]) -> None:
...     return emails
...
>>> make_announcement("foo")
'foo'
```

does **not** throw an error!

The "typing" module

"anything" type	`Any`
function types	`Callable[[p1, p2, …], ret]`
container types	`Dict[k, v], List[t], Set[t], Tuple[t], …`
abstract types	`Hashable, Iterable[t], Mapping[k, v], …`
"noneable" types	`Optional[t]`
unions	`Union[t1, t2, …]`

.pyi files

Python now has "header files":

.pyi is to **.py** what **.h** is to **.c**.

This is e.g. used to annotate the Python standard library, or any other file you can't edit directly.

sys.pyi (simplified extract)

(see: http://github.com/typeshed)

```
def abort() -> None: ...
def access(path: str, mode: int) -> bool: ...
def chdir(path: str) -> None: ...
def chflags(path: str, flags: int) -> None: ...
def chmod(path: str, mode: int) -> None: ...
def chown(path: str, uid: int, gid: int) -> None: ...
def chroot(path: str) -> None: ...
def close(fd: int) -> None: ...
def closerange(fd_low: int, fd_high: int) -> None: ...
def confstr(name: str) -> str: ...
def ctermid() -> str: ...
…
```

Tools for checking types

- **mypy**
  http://github.com/python/mypy
- **pytype**
  http://github.com/google/pytype
- **pycharm**
  https://www.jetbrains.com/pycharm/
- **pylint (soon!)**
  https://www.pylint.org/

```
def f(x: int):
    return x
f("foo")
```

mypy

```
def f(x: int):
    return x
f("foo")
```

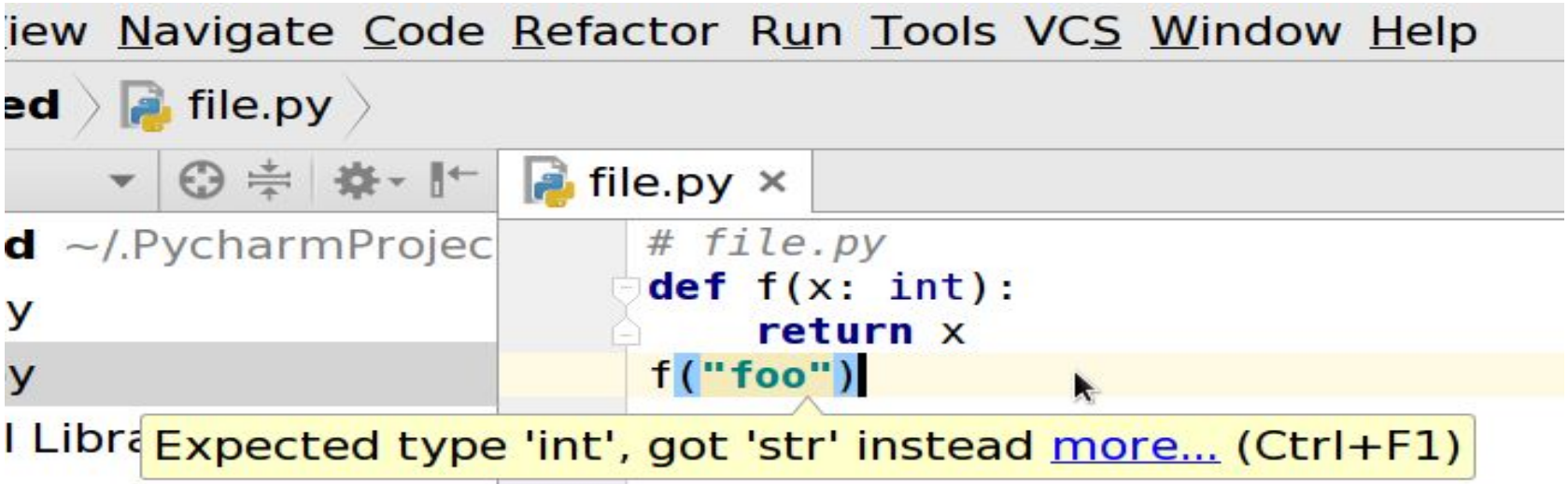
```
$ mypy file.py
file.py:3: error: Argument 1 to "f" has
incompatible type "str"; expected "int"
```

pytype

```
def f(x: int):
    return x
f("foo")
```

```
$ pytype file.py
File "file.py", line 3, in <module>:
  Function f was called with the wrong arguments
  Expected: (x: int)
  Actually passed: (x: str)
```

pycharm


iew Navigate Code Refactor Run Tools VCS Window Help
ed
file.py
file.py ×
d ~/.PycharmProjec
y
y
l Libra
# file.py
def f(x: int):
return x
f("foo")
Expected type 'int', got 'str' instead more... (Ctrl+F1)

What can type checkers detect?

- Bad return types
- Incorrect function calls
  - too many arguments
  - too few arguments
  - invalid keyword arguments
  - invalid argument types
- Invalid attribute access
- Missing modules
- Unsupported operands for operators
- etc.

```
def avg(x: Iterable[float]):
    return sum(x) / len(x)

avg(["1", "2", "3"])
```

```
File "file.py", line 4: Function avg was
called with the wrong arguments
  Expected: (x: Iterable[float])
  Actually passed: (x: List[str])
```

```
def f() -> int:
    return "hello world"
```

```
file.py: note: In function "f":
file.py:2: error: Incompatible return value
type: expected builtins.int, got builtins.str
```

```
def f(x: str):
    x = 42
```

```
file.py: note: In function "f":
file.py:2: error: Incompatible types in
  assignment (expression has type "int",
  variable has type "str")
```

```
def foo(i: int):
   m = []
   x = m[i]
```

```
File "file.py", line 3, in foo:
    Can't retrieve item out of list. Empty?
```

```
def foo(i: int):
    m = []   # type: list
    x = m[i]
```

The Transition to Type Annotations: Part 1: Checking unannotated code

Already have Python code?

- **pytype** allows you to run type-checking on Python code that's completely unannotated. (So does **mypy**, e.g. with `--check-untyped-defs`)

Invalid attribute access

```
import sys
sys.foobar()
```

```
file.py:3: error: "module" has no attribute
"foobar"
```

Unsupported operands

```
def f():
  return "foo" + 42
```

```
File "file.py", line 2, in f:
  unsupported operand type(s) for '+':
  'str' and 'int'
```

Missing parameters

```
import warnings

warnings.formatwarning("out of foobar",
    filename="foobar_factory.py", lineno=42)
```

```
File "file.py", line 3, in <module>:
  Missing parameter 'category' in call to
  function warnings.formatwarning
```

Incorrect calls to builtins

```
import math
math.sqrt(3j)
```

```
File "t.py", line 3: Function math.sqrt was
called with the wrong arguments
  Expected: (x: float)
  Actually passed: (x: complex)
```

The Transition to Type Annotations: Part 2: Generating Annotations

You can automate the process of annotating your Python code, using e.g. pytype and merge_pyi: http://github.com/google/merge_pyi

Automatic annotation: pytype

```
$ pytype file.py -o file.pyi
```

⇒ Now, `file.pyi` contains the (inferred) types of `file.py`.

```
$ merge_pyi file.py file.pyi -o file.py
```

⇒ Now, `file.py` is type-annotated.

https://github.com/edreamleo/make-stub-files

```
$ make_stub_files file.py
```

⇒ Now, `file.pyi` contains the (inferred) types of `file.py`.

```
$ merge_pyi file.py file.pyi -o file.py
```

⇒ Now, `file.py` is type-annotated.

```
def get_label(self):
  return self.name.capitalize()
```

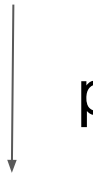
```
def get_label(self):
  return self.name.capitalize()
```


pytype + merge_pyi

```
def get_label(self) -> str:
  return self.name.capitalize()
```

```
def dict_subset(d):
  return {key: d[key] for key in d.keys()
          if key.startswith(PREFIX)}
```

```
def dict_subset(d):
  return {key: d[key] for key in d.keys()
          if key.startswith(PREFIX)}
```




```
def dict_subset(d: Dict[str, Any])
            -> Dict[str, Any]:
  return {key: d[key] for key in d.keys()
          if key.startswith(PREFIX)}
```

```
import StringIO

def read_byte(f):
  return f.read(1)
```

```
import StringIO

def read_byte(f):
  return f.read(1)
```

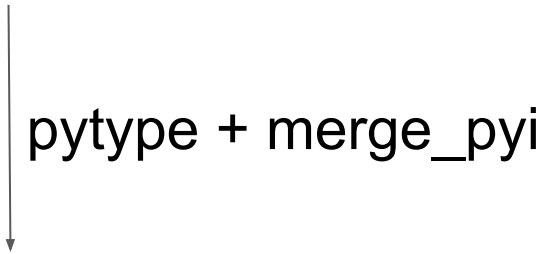

pytype + merge_pyi

```
import StringIO

def read_byte(f: Union[
    file, StringIO.StringIO]) -> str:
  return f.read(1)
```

```
def f(i):
  array = [1, 2, 3]
  return array[i]
```

```
def f(i):
  array = [1, 2, 3]
  return array[i]
```




```
def f(i: Union[int, slice])
        -> Union[int, List[int]]:
  array = [1, 2, 3]
  return array[i]
```

```
def miles_to_kilometers(m):
  return m * 1.6
```

```
def miles_to_kilometers(m):
  return m * 1.6
```


pytype + merge_pyi

```
def miles_to_kilometers(
    m: Union[int, complex, float])
        -> Union[complex, float]:
  return m * 1.6
```

future features

duck-typing

```
def compute_height(pressure: SupportsFloat):
  return 10711.9 - float(pressure) * 10.57
```

duck-typing

```
class PressureSensor(object):
    …
```

duck-typing

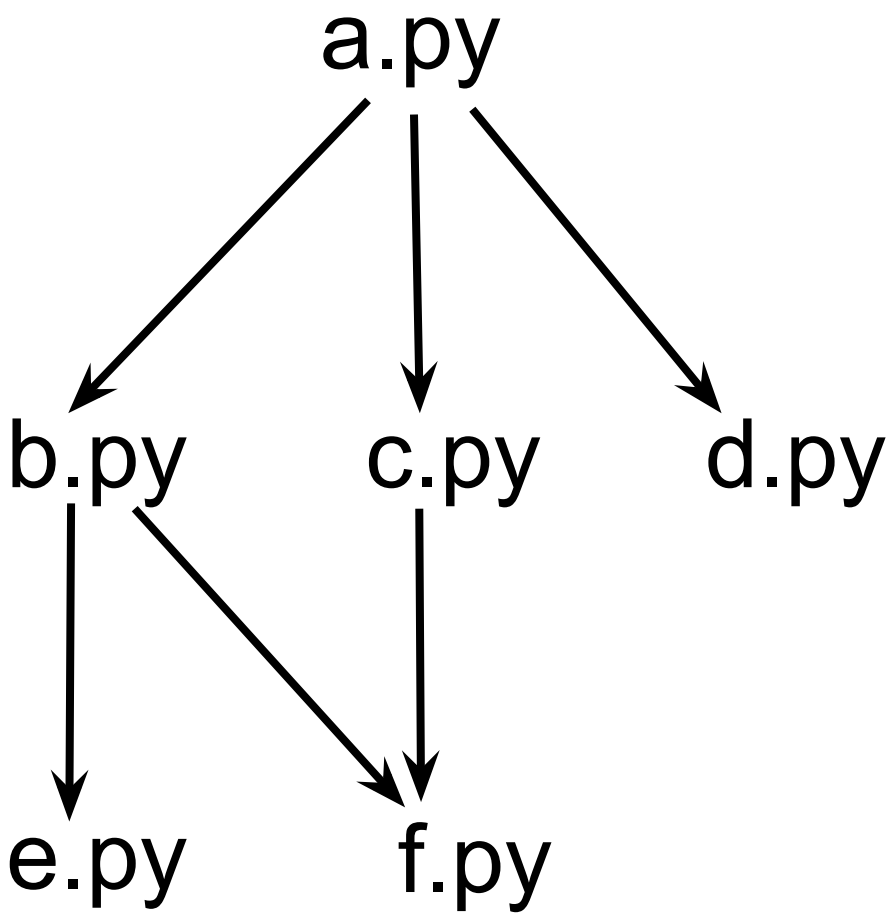
```
class PressureSensor(object):
    def __float__(self):
        return 910.3 - self.voltage * 223.11

compute_height(PressureSensor())   # ok!
```

Imports

Suppose we have the two files `a.py` and `b.py`, and `a.py` imports `b.py`.

```
$ pytype a.py
File "a.py", line 1:
  Can't find module 'b'. Did you
  generate the .pyi?
```

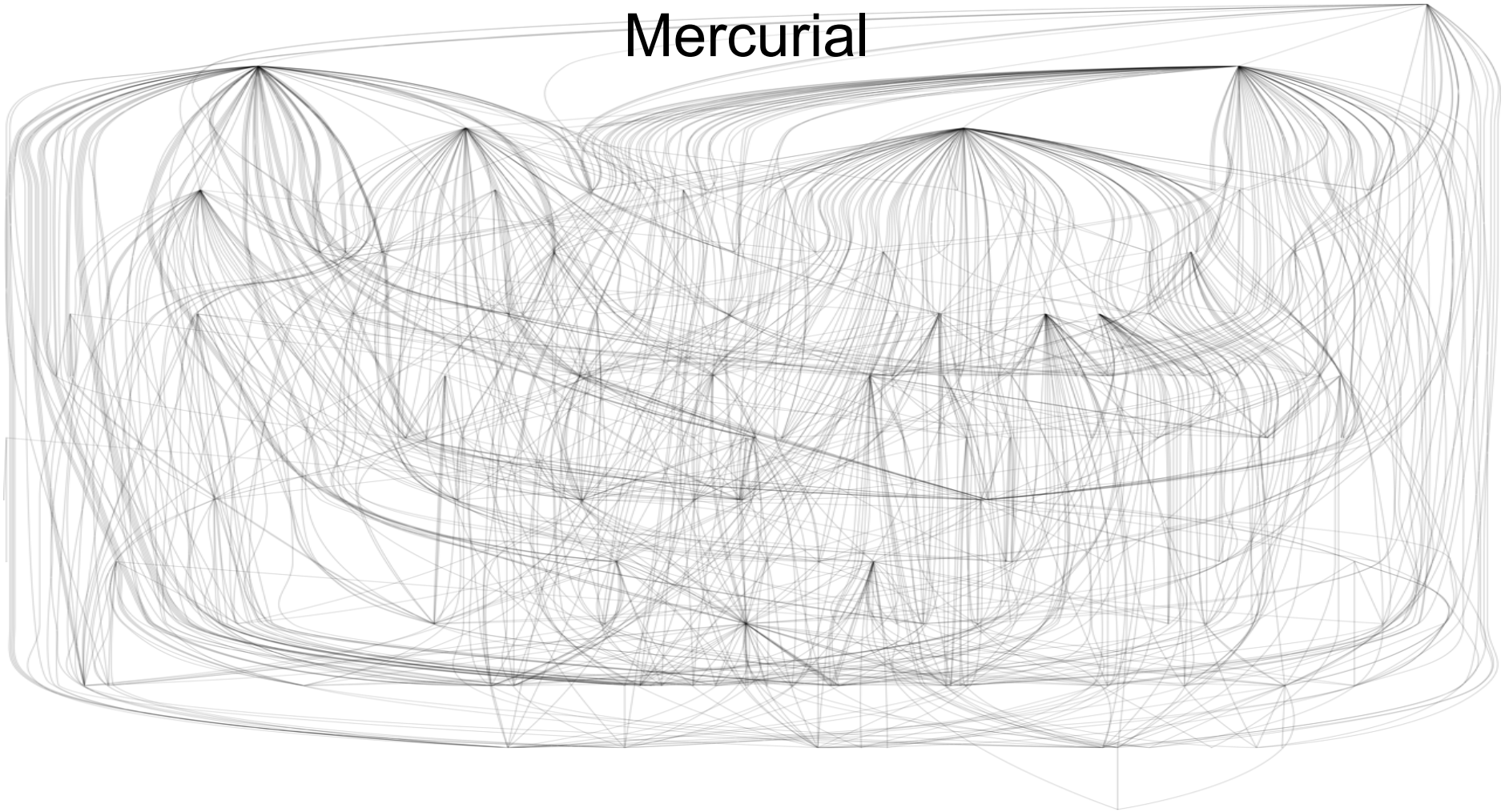

a.py
b.py
c.py
d.py
e.py
f.py

automatic dependency analysis

(Think: "gcc -M")

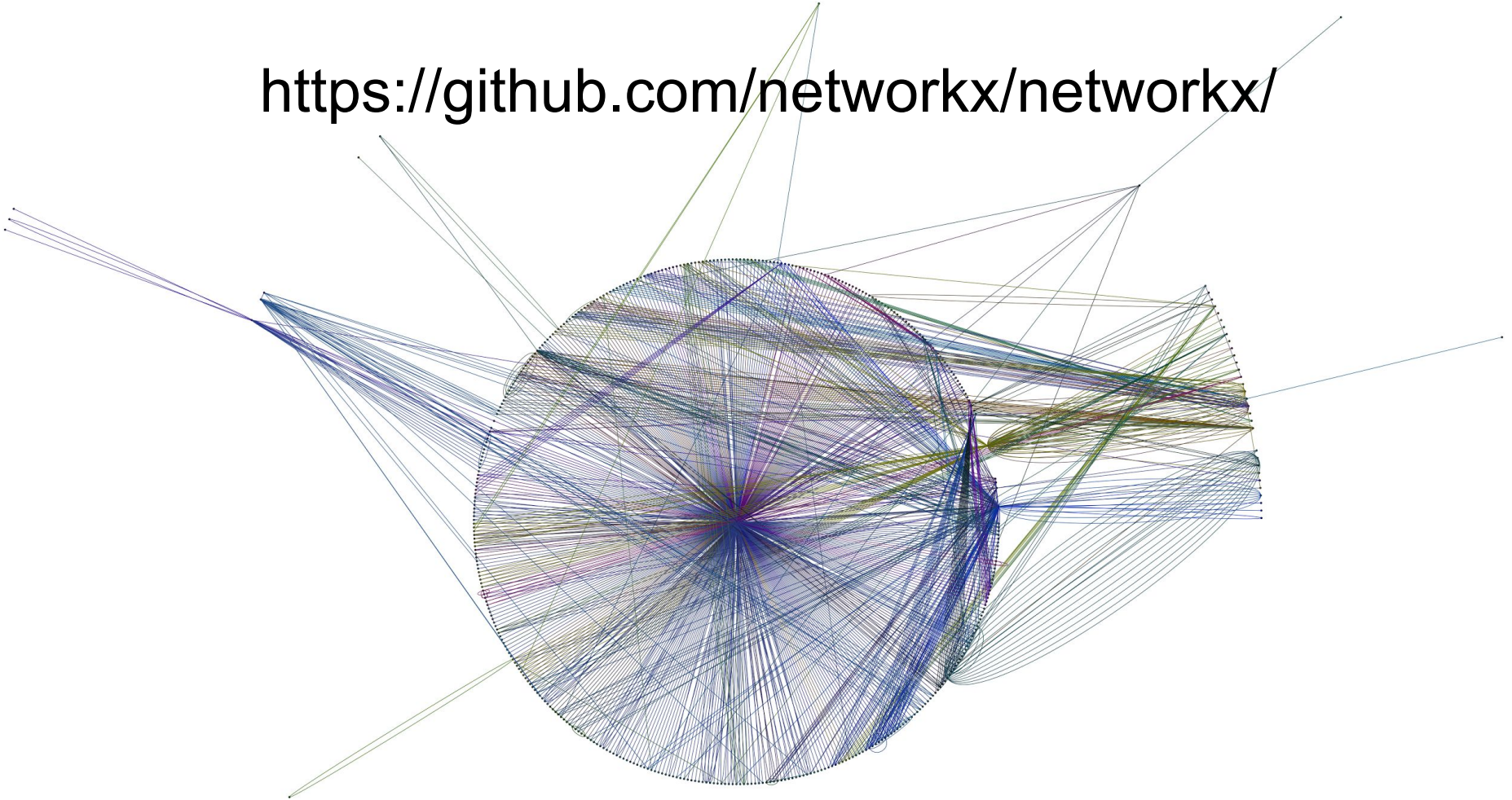
```
importlab --tool pytype ./your_project
```

Mercurial



Python 2.7 standard library

https://github.com/networkx/networkx/



Thanks for listening!

Questions?