typin Documentation

Release 0.1.0

Paul Ross

Dec 11, 2017

Contents

typin is a *Type Inferencer* for understanding what types of objects are flowing through your Python code. It observes your code dynamically and can record all the types that each function sees, returns or raises. typin can then use this information to create Python's type annotations or __doc__ strings to insert into your code. Contents:

CHAPTER 1

typin README

`typin` is a *Type Inferencer* for understanding what types of objects are flowing through your Python code. It observes your code dynamically and can record all the types that each function sees, returns or raises. `typin` can then use this information to create Python's type annotations or `__doc__` strings to insert into your code.

`typin` is currently proof-of-concept and a very early prototype. It is Python 3 only at the moment. There is a forthcoming project https://github.com/paulross/pytest-typin which turns `typin` into a pytest plugin so that your unit tests can generate type annotations and documentation strings.

1.1 Example

Lets say you have a function that creates a repeated string, like this:

```
def function(s, num):
    if num < 1:
        raise ValueError('Value must be > 0, not {:d}'.format(num))
    lst = []
    while num:
        lst.append(s)
        num -= 1
    return ' '.join(lst)
```

You can exercise this under the watchful gaze of `typin`:

```
from typin import type_inferencer

with type_inferencer.TypeInferencer() as ti:
    assert function('Hi', 2) == 'Hi Hi'
```

You can then get the types that `typin` has observed as a string suitable for a stub file:

```
ti.stub_file_str(__file__, '', 'function')
# returns: 'def function(s: str, num: int) -> str: ...'
```

Then adding code that provokes the exception we can track that as well:

```
from typin import type_inferencer

with type_inferencer.TypeInferencer() as ti:
    assert function('Hi', 2) == 'Hi Hi' # As before
    try:
        function('Hi', 0)
    except ValueError:
        pass
```

Exception specifications are not part of Python's type annotation but they are part of of the Sphinx documentation string standard and `typin` can provide that, and the line number where it should be inserted:

```
line_number, docstring = ti.docstring(__file__, '', 'function', style='sphinx')
docstring
"""
<insert documentation for function>

:param s: <insert documentation for argument>
:type s: ``str``

:param num: <insert documentation for argument>
:type num: ``int``

:returns: ``str`` -- <insert documentation for return values>

:raises: ``ValueError``
"""
# Insert template docstrings into the source code.
new_src = ti.insert_docstrings(__file__, style='sphinx')
with open(__file__, 'w') as f:
    for line in new_src:
        f.write(line)
```

Sadly `typin` is not smart enough to write the documentation text for you :-)

There is a CLI interface `typin_cli` that is an entry point to `typin/src/typin/typin_cli.py`. This executes arbitrary python code using `compile()` and `exec()` like the following example. Note use of `--` followed by Python script then the arguments for that script surrounded by quotes:

```
$ python typin_cli.py --stubs=stubs/ --write-docstrings=docstrings/ -- example.py
↪'foo bar baz'
```

This will `compile()/exec()` `example.py` with the arguments `foo bar baz` write the stub files (`'.pyi'` files) to `stubs/` and the source code with the docstrings inserted to `docstrings/`.

`typin_cli.py` help:

```
$ python typin_cli.py --help
usage: typin_cli.py [-h] [-l LOGLEVEL] [-d] [-t] [-e EVENTS_TO_TRACE]
                    [-s STUBS] [-w WRITE_DOCSTRINGS]
                    [--docstring-style DOCSTRING_STYLE] [-r ROOT]
                    program argstring

typin_cli - Infer types of Python functions.
  Created by Paul Ross on 2017-10-25. Copyright 2017. All rights reserved.
  Version: v0.1.0 Licensed under MIT License
USAGE
```

```
positional arguments:
  program               Python target file to be compiled and executed.
  argstring             Argument as a string to give to the target. Prefix
                        this with '--' to avoid them getting consumed by
                        typin_cli.py

optional arguments:
  -h, --help            show this help message and exit
  -l LOGLEVEL, --loglevel LOGLEVEL
                        Log Level (debug=10, info=20, warning=30, error=40,
                        critical=50) [default: 30]
  -d, --dump            Dump results on stdout after processing. [default:
                        False]
  -t, --trace-frame-events
                        Very verbose trace output, one line per frame event.
                        [default: False]
  -e EVENTS_TO_TRACE, --events-to-trace EVENTS_TO_TRACE
                        Events to trace (additive). [default: []] i.e. every
                        event.
  -s STUBS, --stubs STUBS
                        Directory to write stubs files. [default: ]
  -w WRITE_DOCSTRINGS, --write-docstrings WRITE_DOCSTRINGS
                        Directory to write source code with docstrings.
                        [default: ]
  --docstring-style DOCSTRING_STYLE
                        Style of docstrings, can be: 'google', 'sphinx'.
                        [default: sphinx]
  -r ROOT, --root ROOT  Root path of the Python packages to generate stub
                        files for. [default: .]
```

Python type inferencing.

- Free software: MIT license
- Documentation: https://typin.readthedocs.io.

1.2 Features

- TODO

1.3 Credits

This package was created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.

CHAPTER 2

Installation

First make a virtual environment in your *<PYTHONVENVS>*, say *~/pyvenvs*:

```
$ python3 -m venv <PYTHONVENVS>/typin
$ . <PYTHONVENVS>/typin/bin/activate
(typin) $
```

2.1 Stable release

To install typin, run this command in your terminal:

```
$ pip install typin
```

This is the preferred method to install typin, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

2.2 From sources

The sources for typin can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/paulross/typin
```

Or download the tarball:

```
$ curl  -OL https://github.com/paulross/typin/tarball/master
```

Once you have a copy of the source, you can install it with:

```
$ python setup.py install
```

Install the test dependencies and run typin's tests:

```
(typin) $ pip install pytest
(typin) $ pip install pytest-runner
(typin) $ python setup.py test
```

2.3 Developing with typin

If you are developing with typin you need test coverage and documentation tools.

2.3.1 Test Coverage

Install `pytest-cov`:

```
(typin) $ pip install pytest-cov
```

The most meaningful invocation that elimates the top level tools is:

```
(typin) $ pytest --cov=typin --cov-report html tests/
```

2.3.2 Documentation

If you want to build the documentation you need to:

```
(typin) $ pip install Sphinx
(typin) $ cd docs
(typin) $ make html
```

The landing page is *docs/_build/html/index.html*.

CHAPTER 3

Usage

To use typin in a project:

```
import typin
```

CHAPTER 4

Example

There is a CLI interface `typin/src/typin/typin_cli.py` to execute arbitrary python code using `compile()` and `exec()` like this:

```
python typin_cli.py --stubs=stubs -- example.py 'foo bar baz'
```

This will `compile()/exec()` `example.py` with the arguments `foo bar baz` and dump out the results. These include the docstrings for the functions in `example.py` which have been inserted in that source code to produce this:

4.1 example.py

class `typin.example.`**`BaseClass`**
Example base class to explore inheritance.

class `typin.example.`**`ExampleClass`**(*first_name*, *last_name*)
An example class with a couple of methods that we exercise.

`name`()
Returns the last name, first name.

Returns `str` – Formatted name.

class `typin.example.`**`InnerClass`**(*value*)
Same named inner class to explore inner/outer namespaces.

`value`()
Returns the value.

Returns `bytes` – The value.

class `typin.example.`**`MyNT`**(*a*, *b*, *c*)

`a`
Alias for field number 0

b
Alias for field number 1

c
Alias for field number 2

class typin.example.**OuterClass**(*value*)
Example outer class to explore inner/outer issues.

class **InnerClass**(*value*)
Example inner class to explore inner/outer issues.

value()
Returns the value.

Returns str – The value.

value()
Returns the value.

Returns str – The value.

typin.example.**example_function**(*x*)
Example function.

Parameters **x** (int) – Example argument.

Returns int – Doubles the argument.

typin.example.**main**()
Main entry point.

Returns int – status code, 0 is success.

CHAPTER 5

Reference

5.1 typin_cli

5.2 types

Created on 17 Jul 2017

@author: paulross

class `typin.types.`**`FunctionTypes`**(*signature=None*)

Class that accumulate function call data such as call arguments, return values and exceptions raised.

`add_call`(*arg_info*, *file_path*, *line_number*)

Adds a function call from the frame.

`add_exception`(*exception*, *line_number*)

Add an exception.

`add_return`(*return_value*, *line_number*)

Records a return value at a particular line number. If the return_value is None and we have previously seen an exception at this line then this is a phantom return value and must be ignored. See `TypeInferencer.__enter__` for a description of this.

`argument_type_strings`

A `collections.OrderedDict` of `{argument_name : set(types, ...), ...}` where the types are strings.

`docstring`(*include_returns*, *style='sphinx'*)

Returns a pair (line_number, docstring) for this function. The docstring is the __doc__ for the function and the line_number is the docstring position (function declaration + 1). So to insert into a list of lines called `src`:

```
src[:line_number] + docstring.split('\n') + src[line_number:]
```

style can be: 'sphinx', 'google'.

Raises `TypesExceptionBase` or derived class.

exception_type_strings
A dict of `{line_number : set(types, ...), ...}` for any exceptions raised where the return types are strings. There should only be one type in the set.

filtered_arguments()
A `collections.OrderedDict` of `{argument_name : set(types, ...), ...}` where the types are strings. This removes the 'self' argument if it is the first argument.

has_self_first_arg()
Returns True if 'self' is the first argument i.e. I am a method.

line_decl
Line number of the function declaration as an integer.

Returns `int` – Function declaration line.

Raises `FunctionTypesExceptionNoData` If there is no entry points recorded.

line_range
A pair of line numbers of the span of the function as integers. The first is the declaration of the function, the last is the extreme return point or exception.

num_entry_points
The number of entry points, 1 for normal functions >1 for generators. 0 Something wrong.

return_type_strings
A dict of `{line_number : set(types, ...), ...}` for the return values where the return types are strings. There should only be one type in the set.

stub_file_str()
A string suitable for writing to a stub file. Example:

```
def encodebytes(s: bytes) -> bytes: ...
```

types_of_self()
Returns the set of types (as strings) as seen for the type of 'self'. Returns None if 'self' is not the first argument i.e. I am not a method.

exception `typin.types.`**FunctionTypesExceptionNoData**
Exception thrown when no call date has been added to a FunctionTypes object.

class `typin.types.`**Type**(*obj*, *_Type__ids=None*)
This class holds type information extracted from a single object. For sequences and so on this will contain a sequence of types.

exception `typin.types.`**TypesExceptionBase**
Base class for exceptions thrown by the types module.

5.3 type_inferencer

CHAPTER 6

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

6.1 Types of Contributions

6.1.1 Report Bugs

Report bugs at https://github.com/paulross/typin/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

6.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

6.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

6.1.4 Write Documentation

typin could always use more documentation, whether as part of the official typin docs, in docstrings, or even on the web in blog posts, articles, and such.

6.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/paulross/typin/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

6.2 Get Started!

Ready to contribute? Here's how to set up *typin* for local development.

1. Fork the *typin* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/typin.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv typin
$ cd typin/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 typin tests
$ python setup.py test or py.test
$ tox
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

6.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python 2.6, 2.7, 3.3, 3.4 and 3.5, and for PyPy. Check https://travis-ci.org/paulross/typin/pull_requests and make sure that the tests pass for all supported Python versions.

6.4 Tips

To run a subset of tests:

```
$ py.test tests.test_typin
```

CHAPTER 7

Credits

7.1 Development Lead

- Paul Ross <apaulross@gmail.com>

7.2 Contributors

None yet. Why not be the first?

CHAPTER 8

History

8.1 0.1.0 (2017-06-22)

- First commit on GitHub.

CHAPTER 9

Various Research notes

9.1 Tracing functions

This applies to both 2.7 and 3.6.

sys.setprofile() only sees call and return. If an exception is raised the function appears to return None. *sys.setprofile()* can return None as the return value is ignored.

sys.settrace() is more fine grained and gets exception and line events as well. *sys.settrace()* can return itself, None or some other function and this will be respected.

Both are thread specific so it doesn't make much sense to use these in a multithreaded environment.

9.1.1 Tracing Exceptions

If we have this code:

```
def a(arg):              # Line 29
    b('calling b()')     # Line 30
    return 'A'           # Line 31

def b(arg):                  # Line 33
    try:
        c('calling c()')     # Line 35
    except ValueError:
        pass
    return 'B'               # Line 38

def c(arg):                  # Line 40
    raise ValueError()       # Line 41
    return 'C'
```

sys.settrace() sees the exception:

```
/Users/paulross/Documents/workspace/typin/src/typin/research.py 29 a call None
/Users/paulross/Documents/workspace/typin/src/typin/research.py 33 b call None
/Users/paulross/Documents/workspace/typin/src/typin/research.py 40 c call None
# c() raises. We can see this as an exception event is followed by a return None with␣
↪the same lineno.
# Return None on its own is not enough as that might happen in the normal course of␣
↪events.
/Users/paulross/Documents/workspace/typin/src/typin/research.py 41 c exception (
↪<class 'ValueError'>, ValueError(), <traceback object at 0x102365c08>)
/Users/paulross/Documents/workspace/typin/src/typin/research.py 41 c return None
# b() reports the exception at the point that the call to c() is made.
# b() handles the exception, this can be detected by the exception and return events␣
↪being on different lines.
/Users/paulross/Documents/workspace/typin/src/typin/research.py 35 b exception (
↪<class 'ValueError'>, ValueError(), <traceback object at 0x102365c48>)
/Users/paulross/Documents/workspace/typin/src/typin/research.py 38 b return 'B'
/Users/paulross/Documents/workspace/typin/src/typin/research.py 31 a return 'A'
```

sys.setprofile() does not see the exception:

```
/Users/paulross/Documents/workspace/typin/src/typin/research.py 29 a call None
/Users/paulross/Documents/workspace/typin/src/typin/research.py 33 b call None
/Users/paulross/Documents/workspace/typin/src/typin/research.py 40 c call None
/Users/paulross/Documents/workspace/typin/src/typin/research.py 41 c return None
/Users/paulross/Documents/workspace/typin/src/typin/research.py 38 b return 'B'
/Users/paulross/Documents/workspace/typin/src/typin/research.py 31 a return 'A'
/Users/paulross/Documents/workspace/typin/src/typin/research.py 53 main c_call <built-
↪in function setprofile>
```

I think at this stage that we should ignore exception specifications as static typing does not accomodate them interesting though they are. So we use *sys.setprofile()* for now.

9.2 2017-07-22

We neeed to use *sys.settrace()* because if the function ever raises then *sys.setprofile()* will only see it returning None which it might never actually do (implicitly or explicitly). So we would then record any function that raises as possibly returning None. With *sys.settrace* we can eliminate the false returns None by identifying, and ignoring, it as above.

9.3 2017-11-16 and 18

Raising and catching exceptions.

Given this code:

This function is defined on line 45 def exception_propogates(): # Line 45

raise ValueError('Error message') # Line 46 return 'OK'

This function is defined on line 50 def exception_caught(): # line 50

try: raise ValueError('Bad value') # line 52

except ValueError as _err: # line 53 pass

try: raise KeyError('Bad key.') # line 56

except KeyError as _err: # line 57 pass

return 'OK' # line 59

try: exception_propogates()

except ValueError: pass

exception_caught()

We get:

Event: research.py 45 exception_propogates call None Event: research.py 46 exception_propogates line None Event: research.py 46 exception_propogates exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x101031108>) Event: research.py 46 exception_propogates return None

And:

Event: research.py 49 exception_caught call None Event: research.py 50 exception_caught line None Event: research.py 51 exception_caught line None Event: research.py 51 exception_caught exception (<class 'ValueError'>, ValueError('Bad value',), <traceback object at 0x101a31188>) Event: research.py 54 exception_caught line None Event: research.py 55 exception_caught line None Event: research.py 56 exception_caught line None Event: research.py 57 exception_caught line None Event: research.py 57 exception_caught exception (<class 'KeyError'>, KeyError('Bad key.',), <traceback object at 0x101a310c8>) Event: research.py 60 exception_caught line None Event: research.py 61 exception_caught line None Event: research.py 62 exception_caught line None Event: research.py 62 exception_caught return 'OK'

So exception propagation can be detected by the appearance of a return None at the same line number as the exception.

Caught exceptions have a line event following the exception where the line number is greater than that of the exception.

So when we see an exception event we need to defer judgement and wait until the next event to decide if it is propagated or not.

Exception propagates out of function:

Event: research.py 46 exception_propogates exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x101031108>) Event: research.py 46 exception_propogates return None

Exception does not propagate out of function:

Event: research.py 51 exception_caught exception (<class 'ValueError'>, ValueError('Bad value',), <traceback object at 0x101a31188>) Event: research.py 54 exception_caught line None

So if the event following the exception is the same line number, event == 'return' and arg (return value) is None then ignore the return value and record the exception.

If the next event is event == line event at a line greater than the Exception event then the exception has been caught internally.

In both cases the event following the exception must have the same file and function and the arg must be None.

9.4 2017-11-17

sys.settrace() and sys.setprofile():

sys.settrace() creates 'call', 'line', 'return', 'exception' events. sys.setprofile() creates 'call', 'c_call', 'return', 'c_return', 'exception' events.

Both sys.settrace() and sys.setprofile() can be set to the same function but then you get duplicates:

Event: research.py 30 func_a call None Event: research.py 30 func_a call None Event: research.py 31 func_a line None Event: research.py 34 func_b call None Event: research.py 34 func_b call None Event: research.py 35 func_b

line None Event: research.py 36 func_b line None Event: research.py 41 func_c call None Event: research.py 41 func_c call None Event: research.py 42 func_c line None Event: research.py 42 func_c exception (<class 'ValueError'>, ValueError(), <traceback object at 0x1022150c8>) Event: research.py 42 func_c return None Event: research.py 42 func_c return None

9.5 2017-11-20

Revisiting order of events with exceptions:

(typin_00) Pauls-MacBook-Pro-2:typin paulross$ python research.py Event: research.py 81 func_that_catches_import call None Event: research.py 82 func_that_catches_import line None Event: research.py 83 func_that_catches_import line None Event: /Users/paulross/Documents/workspace/typin/src/typin/research_import.py 5 func_no_catch call None Event: /Users/paulross/Documents/workspace/typin/src/typin/research_import.py 6 func_no_catch line None Event: /Users/paulross/Documents/workspace/typin/src/typin/research_import.py 2 func_that_raises call None Event: /Users/paulross/Documents/workspace/typin/src/typin/research_import.py 3 func_that_raises line None Event: /Users/paulross/Documents/workspace/typin/src/typin/research_import.py 3 func_that_raises exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x102333348>) Event: /Users/paulross/Documents/workspace/typin/src/typin/research_import.py 3 func_that_raises return None Event: /Users/paulross/Documents/workspace/typin/src/typin/research_import.py 6 func_no_catch exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x1023332c8>) Event: /Users/paulross/Documents/workspace/typin/src/typin/research_import.py 6 func_no_catch return None Event: research.py 83 func_that_catches_import exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x1023333c8>) Event: research.py 84 func_that_catches_import line None Event: research.py 85 func_that_catches_import line None Event: research.py 85 func_that_catches_import return None (typin_00) Pauls-MacBook-Pro-2:typin paulross$

Simplifying file names:

(typin_00) Pauls-MacBook-Pro-2:typin paulross$ python research.py Event: research.py 81 func_that_catches_import call None Event: research.py 82 func_that_catches_import line None Event: research.py 83 func_that_catches_import line None Event: research_import.py 5 func_no_catch call None Event: research_import.py 6 func_no_catch line None Event: research_import.py 2 func_that_raises call None Event: research_import.py 3 func_that_raises line None Event: research_import.py 3 func_that_raises exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x102333348>) Event: research_import.py 3 func_that_raises return None Event: research_import.py 6 func_no_catch exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x1023332c8>) Event: research_import.py 6 func_no_catch return None Event: research.py 83 func_that_catches_import exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x1023333c8>) Event: research.py 84 func_that_catches_import line None Event: research.py 85 func_that_catches_import line None Event: research.py 85 func_that_catches_import return None (typin_00) Pauls-MacBook-Pro-2:typin paulross$

Exception raised, not caught:

Event: research_import.py 3 func_that_raises exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x102333348>) Event: research_import.py 3 func_that_raises return None

self.exception_in_progress is created with:

filename: research_import.py function: func_that_raises lineno: 3 exception_value: ValueError('Error message',) eventno: X

Next event has same filename, function, lineno, returning None with event X+1 and this means the exception is propogated. So add the exception to the func_types.add_exception and set self.exception_in_progress to None.

Exception propogated: Event: research_import.py 6 func_no_catch exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x1023332c8>) Event: research_import.py 6 func_no_catch return None

self.exception_in_progress is created with:

filename: research_import.py function: func_no_catch lineno: 6 exception_value: ValueError('Error message',) eventno: X

This is as above. Next event has same filename, function, lineno, returning None with event X+1 and this means the exception is propogated. So add the exception to the func_types.add_exception and set self.exception_in_progress to None.

Event: research.py 83 func_that_catches_import exception (<class 'ValueError'>, ValueError('Error message',), <traceback object at 0x1023333c8>) Event: research.py 84 func_that_catches_import line None

self.exception_in_progress is created with:

filename: research.py function: func_that_catches_import lineno: 83 exception_value: ValueError('Error message',) eventno: X

Next event is 'line' event has same filename, function with event X+1 and line > 83. This means the exception is caught. So do not add the exception to the func_types.add_exception but set self.exception_in_progress to None.

So the original analysis (above) is correct even when the exception is thrown across modules.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index