birdseye Documentation

Alex Hall

Feb 26, 2022

Contents

birdseye is a Python debugger which records the values of expressions in a function call and lets you easily view them after the function exits. For example:

You can use birdseye no matter how you run or edit your code. Just `pip install birdseye`, add the `@eye` decorator as seen above, run your function however you like, and view the results in your browser. It's also integrated with some common tools for a smoother experience.

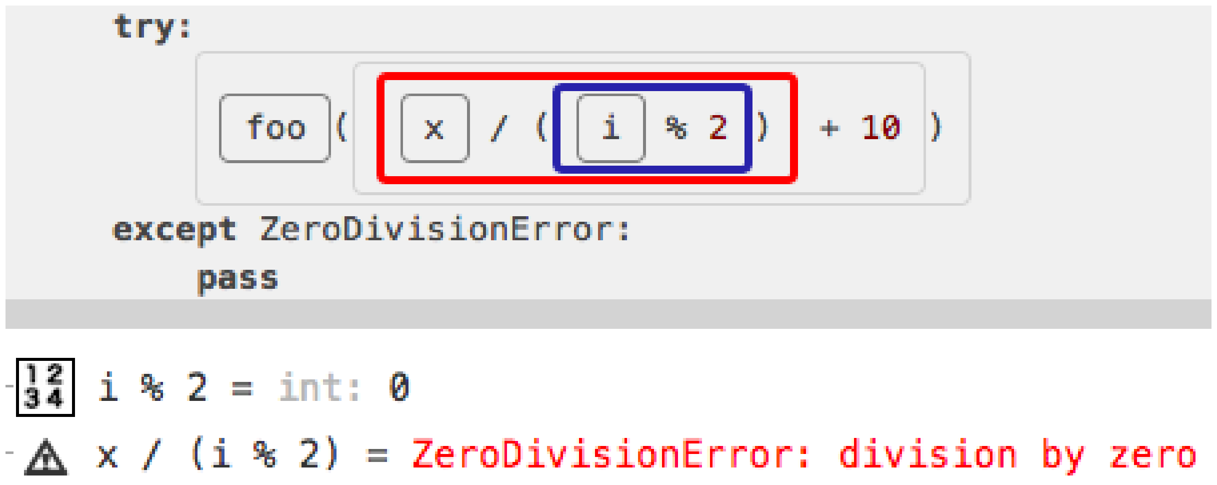
You can try it out **instantly** on futurecoder: enter your code in the editor on the left and click the `birdseye` button to run. No imports or decorators required.

CHAPTER 1

Feature Highlights

Rather than stepping through lines, move back and forth through loop iterations and see how the values of selected expressions change:

See which expressions raise exceptions, even if they're suppressed:

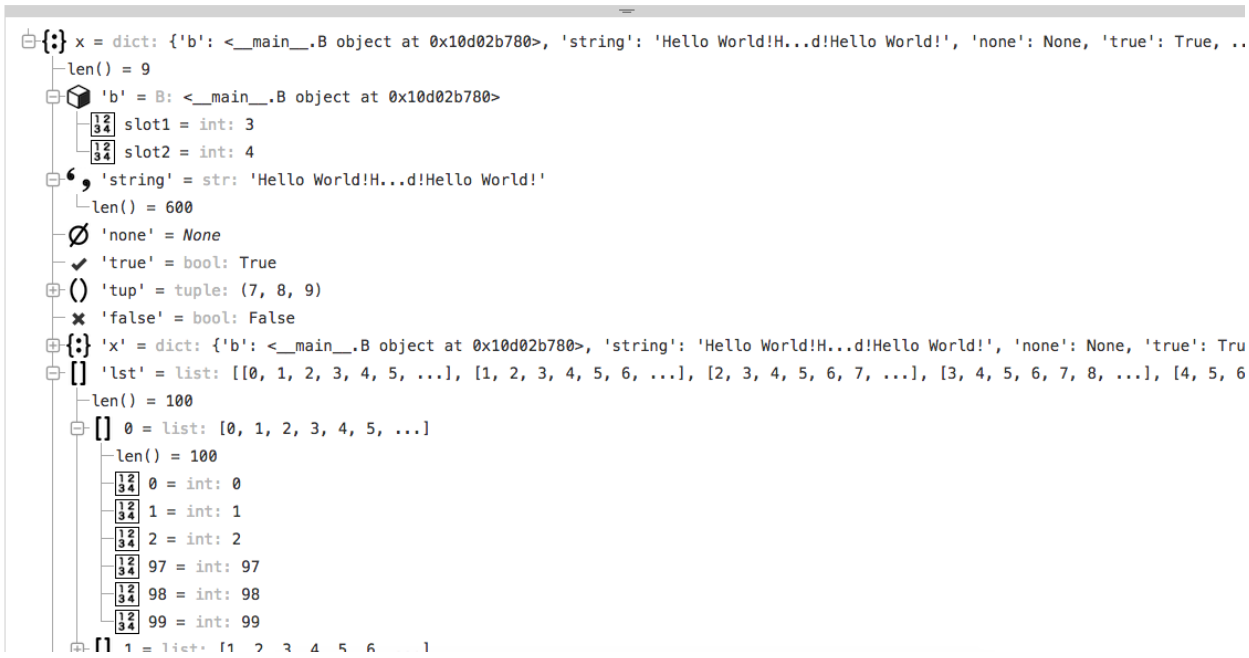


```
try:
    foo(x / (i % 2) + 10)
except ZeroDivisionError:
    pass
```

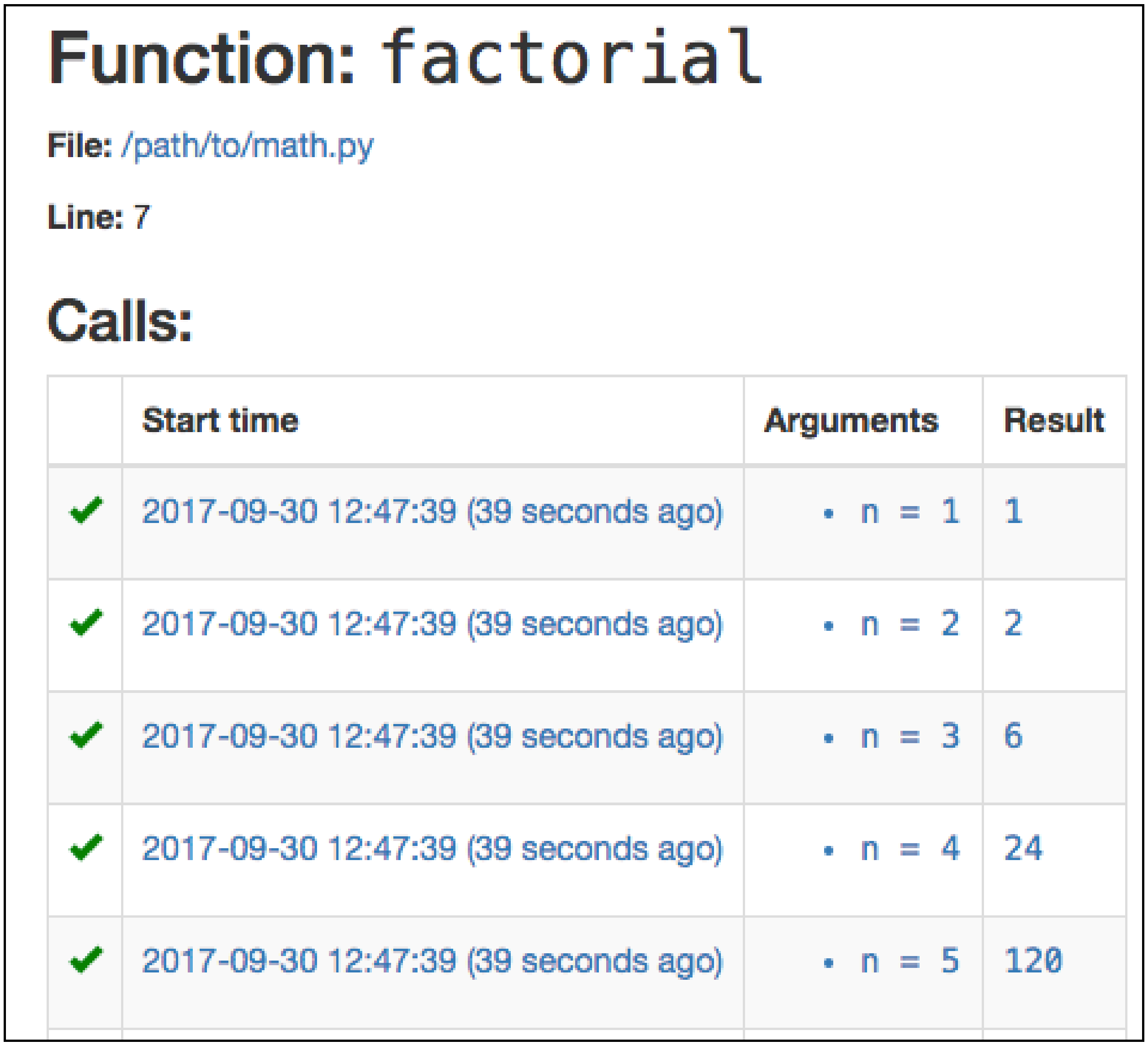
```
i % 2 = int: 0
x / (i % 2) = ZeroDivisionError: division by zero
```

Expand concrete data structures and objects to see their contents. Lengths and depths are limited to avoid an overload of data.

Calls are organised into functions (which are organised into files) and ordered by time, letting you see what happens at a glance:



```
x = dict: {'b': <__main__.B object at 0x10d02b780>, 'string': 'Hello World!H...d!Hello World!', 'none': None, 'true': True, ..
  len() = 9
  'b' = B: <__main__.B object at 0x10d02b780>
    slot1 = int: 3
    slot2 = int: 4
  'string' = str: 'Hello World!H...d!Hello World!'
    len() = 600
  'none' = None
  'true' = bool: True
  'tup' = tuple: (7, 8, 9)
  'false' = bool: False
  'x' = dict: {'b': <__main__.B object at 0x10d02b780>, 'string': 'Hello World!H...d!Hello World!', 'none': None, 'true': Tru
  'lst' = list: [[0, 1, 2, 3, 4, 5, ...], [1, 2, 3, 4, 5, 6, ...], [2, 3, 4, 5, 6, 7, ...], [3, 4, 5, 6, 7, 8, ...], [4, 5, 6
    len() = 100
    0 = list: [0, 1, 2, 3, 4, 5, ...]
      len() = 100
      0 = int: 0
      1 = int: 1
      2 = int: 2
      97 = int: 97
      98 = int: 98
      99 = int: 99
    1 = list: [1, 2, 3, 4, 5, 6, ...]
```



Function: `factorial`

File: /path/to/math.py

Line: 7

Calls:

	Start time	Arguments	Result
✓	2017-09-30 12:47:39 (39 seconds ago)	• n = 1	1
✓	2017-09-30 12:47:39 (39 seconds ago)	• n = 2	2
✓	2017-09-30 12:47:39 (39 seconds ago)	• n = 3	6
✓	2017-09-30 12:47:39 (39 seconds ago)	• n = 4	24
✓	2017-09-30 12:47:39 (39 seconds ago)	• n = 5	120

CHAPTER 2

Contents

2.1 Quick start

First, install birdseye using pip:

```
pip install --user birdseye
```

To debug a function:

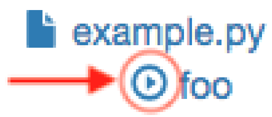
1. Decorate it with `birdseye.eye`, e.g.:

   ```
   from birdseye import eye

   @eye
   def foo():
   ```

   The `eye` decorator must be applied before any other decorators, i.e. at the bottom of the list.

2. Call the function*[0].
3. Run `birdseye` or `python -m birdseye` in a terminal to run the UI server.
4. Open http://localhost:7777 in your browser.
5. Note the instructions at the top for navigating through the UI. Usually you will want to jump straight to the most recent call of the function you're debugging by clicking on the play icon:



When viewing a function call, you can:

[0] You can run the program however you want, as long as the function gets called and completes, whether by a normal return or an exception. The program itself doesn't need to terminate, only the function.

- Hover over an expression to view its value at the bottom of the screen.
- Click on an expression to select it so that it stays in the inspection panel, allowing you to view several values simultaneously and expand objects and data structures. Click again to deselect.
- Hover over an item in the inspection panel and it will be highlighted in the code.
- Drag the bar at the top of the inspection panel to resize it vertically.
- Click on the arrows next to loops to step back and forth through iterations. Click on the number in the middle for a dropdown to jump straight to a particular iteration.
- If the function call you're viewing includes a function call that was also traced, the expression where the call happens will have an arrow () in the corner which you can click on to go to that function call. For generator functions, the arrow will appear where the generator is first iterated over, not just when the function is called, since that is when execution of the function begins.

2.2 Integrations with other tools

birdseye can be used no matter how you write or run your code, requiring only a browser for the interface. But it's also integrated with some common tools for a smoother experience.

2.2.1 snoop

snoop is another fairly similar debugging library by the same author. Typically you decorate a function with `@snoop` and it will log the execution and local variables in the function. You can also use the `@spy` decorator which is a combination of `@snoop` and `@eye` from birdseye so that you get the best of both worlds with no extra effort.

2.2.2 Jupyter/IPython notebooks

First, load the birdseye extension, using either `%load_ext birdseye` in a notebook cell or by adding `'birdseye'` to the list `c.InteractiveShellApp.extensions` in your IPython configuration file, e.g. `~/.ipython/profile_default/ipython_config.py`.

Use the cell magic `%%eye` at the top of a notebook cell to trace that cell. When you run the cell and it finishes executing, a frame should appear underneath with the traced code.

Hovering over an expression should show the value at the bottom of the frame. This requires the bottom of the frame being visible. Sometimes notebooks fold long output (which would include the birdseye frame) into a limited space - if that happens, click the space just left of the output. You can also resize the frame by dragging the bar at the bottom, or click 'Open in new tab' just above the frame.

For convenience, the cell magic automatically starts a birdseye server in the background. You can configure this by settings attributes on `BirdsEyeMagics`, e.g. with:

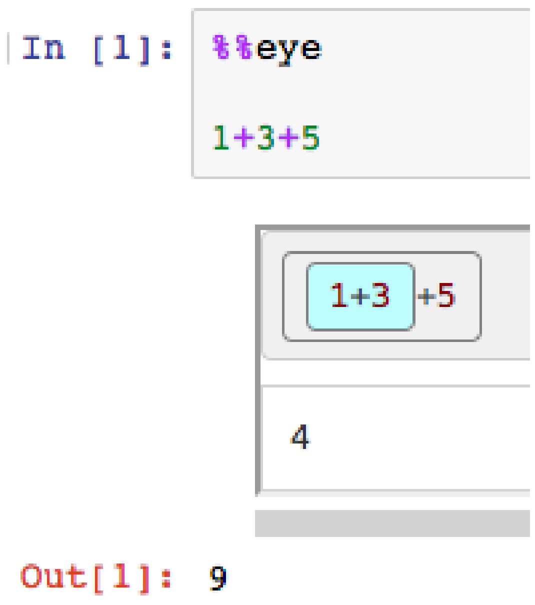
```
%config BirdsEyeMagics.port = 7778
```

in a cell or:

```
c.BirdsEyeMagics.port = 7778
```

in your IPython config file. The available attributes are:

`server_url` If set, a server will not be automatically started by `%%eye`. The iframe containing birdseye output will use this value as the base of its URL.


```
In [1]: %%eye

        1+3+5
```

```
Out[1]: 9
```

`port` Port number for the background server.

`bind_host` Host that the background server listens on. Set to 0.0.0.0 to make it accessible anywhere. Note that birdseye is NOT SECURE and doesn't require any authentication to access, even if the notebook server does. Do not expose birdseye on a remote server unless you have some other form of security preventing HTTP access to the server, e.g. a VPN, or you don't care about exposing your code and data. If you don't know what any of this means, just leave this setting alone and you'll be fine.

`show_server_output` Set to True to show stdout and stderr from the background server.

`db_url` The database URL that the background server reads from. Equivalent to the *environment variable BIRDSEYE_DB*.

2.2.3 Visual Studio Code extension

- Visual Studio Marketplace page
- GitHub repo

Usage is simple: open the Command Palette (F1 or Cmd+Shift+P) and choose 'Show birdseye'. This will start the server and show a browser pane with the UI inside VS Code.

You can also search for birdseye under settings for configuration and possibly troubleshooting.

2.2.4 PythonAnywhere

This isn't really an integration, just some instructions.

The birdseye server needs to run in a web app for you to access it. You can either use a dedicated web app, or if you can't afford to spare one, combine it with an existing app.

To use a dedicated web app, create a new web app, choose any framework you want (manual configuration will do), and in the WSGI configuration file `/var/www/your_domain_com_wsgi.py` put the following code:

```
from birdseye.server import app as application
```

To combine with an existing web app, add this code at the end of the WSGI file:

```
import birdseye.server
from werkzeug.wsgi import DispatcherMiddleware

application = DispatcherMiddleware(application, {
    '/birdseye': birdseye.server.app
})
```

Here `application` should already be defined higher up as the WSGI object for your original web app. Then your existing web app should be unaffected, except that you can also go to `your.domain.com/birdseye` to view the birdseye UI. You can also choose another prefix instead of `'/birdseye'`.

Either way, you should also ensure that your web app is secure, as birdseye will expose your code and data. Under the Security section of your web app configuration, enable Force HTTPS and Password protection, choose a username and password, then reload the web app.

2.2.5 PyCharm plugin

This plugin hasn't worked for a long time and is no longer being maintained.

- JetBrains Plugin Repository page
- GitHub repo

2.3 Tips

2.3.1 Debugging an entire file

Instead of decorating individual functions with `@eye`, you may want to debug *all* the functions in a module, or you may want to debug the top-level execution of the module itself without wrapping it in a function.

To trace every function in the file, as well as the module execution itself, add the line:

```
import birdseye.trace_module_deep
```

To trace only the module execution but none of the functions (to reduce the performance impact), leave out the `_deep`, i.e.:

```
import birdseye.trace_module
```

There are some caveats to note:

1. These import statements must be unindented, not inside a block such as `if` or `try`.
2. **If the module being traced is not the module that is being run directly, i.e. it's being imported by another module, then:**

    1. The module will not be traced in Python 2.
    2. `birdseye` must be imported somewhere before importing the traced module.
    3. The execution of the entire module will be traced, not just the part after the import statement as when the traced module is run directly.

2.3.2 Debugging functions without importing

If you're working on a project with many files and you're tired of writing `from birdseye import eye` every time you want to debug a function, add code such as this to the entrypoint of your project:

```
from birdseye import eye

# If you don't need Python 2/3 compatibility,
# just one of these lines will do
try:
    import __builtin__ as builtins  # Python 2
except ImportError:
    import builtins  # Python 3

builtins.eye = eye
# or builtins.<something else> = eye if you want to use a different name
```

Now you can decorate a function with `@eye` anywhere without importing.

2.3.3 Debugging the middle of a loop

birdseye will always save data from the first and last three iterations of a loop, but sometimes you have a loop with many iterations and you want to know about a specific iteration in the middle. If you were using a traditional debugger, you might do something like:

```
for item in long_list_of_items:
    if has_specific_property(item):
        print(item)  # <-- put a breakpoint here
    ...
```

You can actually use the same technique in birdseye, and you don't even need anything like a breakpoint. For every statement/expression node in a loop block, birdseye will ensure that at least two iterations where that node was evaluated are saved, assuming they exist. That means that if a statement/expression is only evaluated in the middle of the loop, those iterations will still be saved. Use a specific `if` or `try/except` statement to track down the iterations you need.

You can also try wrapping the contents of the loop in a function and debugging that function. Then every call to the function will be saved and you can find the call you want by looking at the arguments and return values in the calls table. However this does come with a performance cost.

2.4 Configuration

2.4.1 Server

The server provides the user interface which can be accessed in the browser. You can run it using the `birdseye` command in a terminal. The command has a couple of options which can be viewed using `--help`:

```
$ birdseye --help
usage: birdseye [-h] [-p PORT] [--host HOST]

optional arguments:
  -h, --help            show this help message and exit
  -p PORT, --port PORT  HTTP port, default is 7777
  --host HOST           HTTP host, default is 'localhost'
```

To run a remote server accessible from anywhere, run `birdseye --host 0.0.0.0`.

The `birdseye` command uses the Flask development server, which is fine for local debugging but doesn't scale very well. You may want to use a proper WSGI server, especially if you host it remotely. Here are some options. The WSGI application is named `app` in the `birdseye.server` module. For example, you could use `gunicorn` as follows:

```
gunicorn -b 0.0.0.0:7777 birdseye.server:app
```

2.4.2 Database

Data is kept in a SQL database. You can configure this by setting the environment variable `BIRDSEYE_DB` to a database URL used by SQLAlchemy, or just a path to a file for a simple sqlite database. The default is `.birdseye.db` under the home directory. The variable is checked by both the server and the tracing by the `@eye` decorator.

If environment variables are inconvenient, you can do this instead:

```python
from birdseye import BirdsEye

eye = BirdsEye('<insert URL here>')
```

You can conveniently empty the database by running:

```
python -m birdseye.clear_db
```

2.4.3 Making tracing optional

Sometimes you may want to only trace certain calls based on a condition, e.g. to increase performance or reduce database clutter. In this case, decorate your function with `@eye(optional=True)` instead of just `@eye`. Then your function will have an additional optional parameter `trace_call`, default False. When calling the decorated function, if `trace_call` is false, the underlying untraced function is used. If true, the traced version is used.

2.4.4 Collecting more or less data

Only pieces of objects are recorded, e.g. the first and last 3 items of a list. The number depends on the type of object and the context, and it can be configured according to the `num_samples` attribute of a `BirdsEye` instance. This can be set directly when constructing the instance, e.g.:

```python
from birdseye import BirdsEye

eye = BirdsEye(num_samples=dict(...))
```

or modify the dict of an existing instance:

```python
from birdseye import eye

eye.num_samples['big']['list'] = 100
```

The default value is this:

```python
dict(
    big=dict(
        attributes=50,
```

(continues on next page)

(continued from previous page)

```
        dict=50,
        list=30,
        set=30,
        pandas_rows=20,
        pandas_cols=100,
    ),
    small=dict(
        attributes=50,
        dict=10,
        list=6,
        set=6,
        pandas_rows=6,
        pandas_cols=10,
    ),
)
```

Any value of `num_samples` must have this structure.

The values of the `big` dict are used when recording an expression directly (as opposed to recording a piece of an expression, e.g. an item of a list, which is just part of the tree that is viewed in the UI) outside of any loop or in the first iteration of all current loops. In these cases more data is collected because using too much time or space is less of a concern. Otherwise, the `small` values are used. The inner keys correspond to different types:

- `attributes`: (e.g. `x.y`) collected from the `__dict__`. This applies to any type of object.
- `dict` (or any instance of `Mapping`)
- `list` (or any `Sequence`, such as tuples, or numpy arrays)
- `set` (or any instance of `Set`)
- `pandas_rows`: the number of rows of a `pandas` `DataFrame` or `Series`.
- `pandas_cols`: the number of columns of a `pandas` `DataFrame`.

2.5 Performance and limitations

Every function call is recorded, and every nontrivial expression is traced. This means that:

- Programs are greatly slowed down, and you should be wary of tracing functions that are called many times or that run through many loop iterations. Note that function calls are not visible in the interface until they have been completed.
- A large amount of data may be collected for every function call, especially for functions with many loop iterations and large nested objects and data structures. This may be a problem for memory both when running the program and viewing results in your browser.
- To limit the amount of data saved, only a sample is stored. Specifically:
  - The first and last 3 iterations of loops, except if an expression or statement is only evaluated at some point in the middle of a loop, in which case up to two iterations where it was evaluated will also be included (see *Debugging the middle of a loop*).
  - A limited version of the `repr()` of values is used, provided by the cheap_repr package.
  - Nested data structures and objects can only be expanded by up to 3 levels. Inside loops this is decreased, except when all current loops are in their first iteration.
  - Only pieces of objects are recorded - see *Collecting more or less data*.

In IPython shells and notebooks, `shell.ast_transformers` is ignored in decorated functions.

2.6 How it works

The source file of a decorated function is parsed into the standard Python Abstract Syntax Tree. The tree is then modified so that every statement is wrapped in its own `with` statement and every expression is wrapped in a function call. The modified tree is compiled and the resulting code object is used to directly construct a brand new function. This is why the `eye` decorator must be applied first: it's not a wrapper like most decorators, so other decorators applied first would almost certainly either have no effect or bypass the tracing. The AST modifications notify the tracer both before and after every expression and statement.

Here is a talk going into more detail.

See the *Source code overview* for an even closer look.

2.7 Contributing

Here's how you can get started if you want to help:

1. Fork the repository, and clone your fork.
2. Run

   ```
   pip install -e .
   ```

   in the root of the repo. This will install it using a symlink such that changes to the code immediately affect the installed library. In other words, you can edit a `.py` file in your copy of birdseye, then debug a separate program, and the results of your edit will be visible. This makes development and testing straightforward.

   If you have one or more other projects that you're working on where birdseye might be useful for development and debugging, install birdseye into the interpreter (so the virtualenv if there is one) used for that project.
3. Try using birdseye for a bit, ideally in a real scenario. Get a feel for what using it is like. Note any bugs it has or features you'd like added. Create an issue where appropriate or ask questions on the gitter chatroom.
4. Pick an issue that interests you and that you'd like to work on, either one that you created or an existing one. An issue with the easy label might be a good place to start.
5. Read through the source code overview below to get an idea of how it all works.
6. *Run the tests* before making any changes just to verify that it all works on your computer.
7. Dive in and start coding! I've tried to make the code readable and well documented. Don't hesitate to ask any questions on gitter. If you installed correctly, you should find that changes you make to the code are reflected immediately when you run it.
8. Once you're happy with your changes, make a pull request.

2.7.1 Source code overview

This is a brief and rough overview of how the core of this library works, to assist anyone reading the source code for the first time.

See also '*How it works*' for a higher level view of the concepts apart from the actual source code.

Useful background knowledge

1. The `ast` module of the Python standard library, for parsing, traversing, and modifying source code. You don't need to know the details of this in advance, but you should know that this is a great resource for learning about it if necessary, as the official documentation is not very helpful.
2. **Code objects**: every function in Python has a `__code__` attribute pointing to a special internal code object. This contains the raw instructions for executing the function. A locally defined function (i.e. a `def` inside a `def`) can have multiple separate instances, but they all share the same code object, so this is the key used for storing/finding metadata for functions.
3. **Frame objects**: sometimes referred to as the frame of execution, this is another special python object that exists for every function call that is currently running. It contains local variables, the code object that is being run, a pointer to the previous frame on the stack, and more. It's used as the key for data for the current call.

When a function is decorated [`BirdsEye.trace_function`]

1. [`TracedFile.__init__`] The entire file is parsed using the standard `ast` module. The tree is modified so that every expression is wrapped in two function calls [`_NodeVisitor.visit_expr`] and every statement is wrapped in a `with` block [`_NodeVisitor.visit_stmt`].
2. [`BirdsEye.compile`] An `ASTTokens` object is created so that the positions of AST nodes in the source code are known.
3. The modified tree is compiled into a code object. Inside this we find the code object corresponding to the function being traced.
4. The `__globals__` of the function are updated to contain references to the functions that were inserted into the tree in step 1.
5. A new function object is created that's a copy of the original function except with the new code object.
6. An HTML document is constructed where the expressions and statements of the source are wrapped in `<span>`s.
7. A `Function` row is stored in the database containing the HTML and other metadata about the function.
8. A `CodeInfo` object is kept in memory to keep track of metadata about the function.

When a function runs

1. When the first statement of the function runs, the tracer notices that it's the first statement and calls `TreeTracerBase._enter_call`. A new `FrameInfo` is created and associated with the current frame.
2. [`BirdsEye.enter_call`] The arguments to the function are noted and stored in the `FrameInfo`. If the parent frame is also being traced, this is noted as an inner call of the parent call.
3. A `_StmtContext` is created for every statement in the function. These lead to calling `BirdsEye.before_stmt` and `BirdsEye.after_stmt`.
4. For every expression in the function call, `BirdsEye.before_expr` and `BirdsEye.after_expr` are called. The values of expressions are expanded in `NodeValue.expression` and stored in an `Iteration`, belonging either directly to the current `FrameInfo` (if this is at the top level of the function) or indirectly via an `IterationList` (if this is inside a loop).
5. When the function call ends, `BirdsEye.exit_call` is called. The data from the current `FrameInfo` is gathered and stored in the database in a new `Call` row.

2.7.2 Testing

Run `python setup.py test` to install test requirements and run all tests with a single Python interpreter. You will need to have phantomjs installed, e.g. via:

```
npm install --global phantomjs
```

Run tox (`pip install tox`) to run tests on all supported versions of Python: 2.7, 3.5, and 3.6. You must install the interpreters separately yourself.

Pushes to GitHub will trigger a build on Travis to run tests automatically. This will run `misc/travis_test.sh`.

`test_against_files`

One of the tests involves comparing data produced by the debugger to the contents of golden JSON files. This produces massive diffs when the tests fail. To read these I suggest redirecting or copying the output to a file and then doing a regex search for `^[+-]` to find the actual differences.

If you're satisfied that the code is doing the correct thing and the golden files need to be updated, set the environment variable `FIX_TESTS=1`, then rerun the tests. This will write to the files instead of comparing to them. Since there are files for each version of python, you will need to run the tests on all supported interpreters, so tox is recommended.

Browser screenshots for test failures

`test_interface.py` runs a test using selenium and phantomjs. If it fails, it produces a file `error_screenshot.png` which is helpful for debugging the failure locally. If the test only fails on travis, you can use the `misc/travis_screenshot.py` script to obtain the screenshot. See the module docstring for details.

2.7.3 Linting

None of this is strictly required, but may help spot errors to improve the development process.

Linting Python using mypy (type warnings)

The code has type hints so that `mypy` can be used on it, but there are many false warnings for various reasons. To ignore these, use the `misc/mypy_filter.py` script. The docstring explains in more detail.

Linting JavaScript using gulp and eslint

1. Install `npm`
2. Change to the `gulp` directory.
3. Run `install-deps.sh`.
4. Run `gulp`. This will lint the JavaScript continuously, checking every time the files change.