U.P.B. Sci. Bull., Series C, Vol. 81, Iss. 4, 2019 ISSN 2286-3540

INSTRUMENTING PYTHON CODE TO IMPROVE THE DEVELOPMENT PROCESS

Mihnea Dobrescu-Balaur[1] and Lorina Negreanu[2]

Python is one of the most popular programming languages today. This is mainly thanks to its dynamic character, allowing fast development times and short feedback loops while building prototypes. However, the dynamic nature of Python becomes a disadvantage when deploying it in production, making it hard to catch errors ahead of time. Thankfully, the Python developers have implemented various tools to help with this. We will examine the state of the art in Python tooling, as well as propose a new debugging library that developers could use to examine the state within any Python function.

Keywords: tracing, function state, python, programming languages

1. Introduction

Python is one of the most popular dynamic programming languages. It is used in production at companies such as Google, Dropbox and Yahoo. Its main advantages are the clean, straightforward syntax and vast standard library. Thanks to this, developing in Python is fast and productive [1]. Despite being a dynamic language, its strong typing gives programmers more trust when it comes to their programs behavior. To give an example, while you can add a number and a string in PHP and JavaScript, you cannot do so in Python. Another component of the Python ecosystem that makes it really easy for developers is the Python interpreter. One can fire up a Python shell just as they would open a terminal and start experimenting right away. Even more, a breakpoint can be set from within the code and at runtime the programmer gets a fully enabled Python shell within the execution of the program. This, together with the powerful introspection tools that come included with the language, makes Python a very developer-friendly language. This is important because when we think about successful projects, more time will be spent maintaining them and trying to find and fix bugs rather than the amount of time spent implementing them in the first place, so good debugging capabilities are crucial.

[1]Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, email `mihnea@linux.com`

[2]Faculty of Automatic Control and Computers, University POLITEHNICA of Bucharest, email `lorina.negreanu@cs.pub.ro`

This leads to the core of this article - tooling. Developer tools are distinct programs that help developers in their work. The main categories are debuggers, static checkers, profilers and tracers. Debuggers help with executing code step by step in order to find a bug as it happens. They require plenty manual intervention, since the programmer controls the process entirely. Static checkers are tools that scan the codebase of a project and detect errors based on the type system (if the programming language is typed) or on user-defined rules (*a function shouldn't have more than 5 arguments*); in the latter case, the checkers are also called *linters*. Profilers are useful when performance is in question. The programmer might want to find out which function is spending the most CPU cycles, or how much memory is the program using over time. Finally, tracers are useful in following the flow of execution through a program. Especially in dynamic languages such as Python where one can build function names dynamically, based on user input, and then call them, understanding the exact path that a program's execution followed is not always trivial.

We will dive into more details as follows. First, we will examine the current state of the art for developer tools in the Python ecosystem. Then, we will compare these tools with the latest advances in developer tools from other programming communities. Finally, we will introduce *execution-trace* [2], a new developer tool for Python based on what we've learned from other languages, and draw some conclusions.

2. **Related Work**

2.1. **Python Tooling**

We will distinguish between two types of analysis that developer tools perform: static and dynamic. Naturally, we only discuss about what is available within the Python ecosystem.

2.**1**.1. *Static Analysis.* Static checkers [3] are called so because they do not execute the target code. This is a great advantage, because any side effects that might be performed by executing arbitrary code from a project are avoided.

Implementations range from simple tools such as linters that can check for style errors to more complex ones such as type checkers that can use the code in order to determine the type correctness of a program. In the Python ecosystem, Pylit [4] is the most common linter, while Mypy [5] is the leading static checker. Mypy relies on optional type annotations [6].

2.**1**.2. *Dynamic Analysis.* Opposed to static analysis we have dynamic analysis [7]. This implies running the target program and capturing the relevant information at runtime. These tools *instrument* the target code; that is, they modify the code before running it such that internal functions needed for data acquisition are called at relevant points in a program's execution. Such tools are used in static validation research [8] in order to better understand existing programs. Similar approaches have been used even in statically typed languages such as Java in order to improve the context around runtime exceptions when debugging [9][10].

The main types of dynamic analysis tools one can find, particularly for Python, are profilers and tracers. To name a few: *cProfile*, *line_profiler*, *memory_profiler*, *trace*.

2.1.3. *Interactive Development.* Building on dynamic analysis tools, others have implemented entire interactive development environments. One such example is Light Table [11], an integrated development environment (IDE) where the user can interact with their code in ways that were previously only possible in an interpreter. The developer can evaluate any line of code from the file and see the result right next to it, as a comment. This all happens within a sandboxed interpreter, invisible to the developer. Using this concept, Light Table's developers have implemented other plugins, allowing for even more interesting interactions, such as changing the parameters of a WebGL animation while it was rendered.

3. **Problem Description**

3.1. **Tracking State in Python Functions**

Bugs and errors are expected when it comes to live, production systems. Developers try their best to guard against errors by using `catch` blocks and putting logging in place so that they know as much as possible about the state of the program when an exception was raised. However, this is not perfect all the time - one cannot correctly guess ahead of time all the information that would be needed for investigating an exception. When bugs occur, it is worse, because there is no code in place to guard against them. We found that in high-volume systems, such a situation usually happens multiple times, in a deterministic manner. Thus, it would be useful to have a way in which a developer could get access to the complete execution flow of a function, including the state of all accessible variables.

Usually, to achieve something similar, a developer would either start a debugger or insert multiple logging statements. These approaches are not ideal. First of all, it might be impossible to reach the live process in order to start a debugger. Then, even if it were possible, the developer would have to go through a lot of steps by hand until the error is reproduced. As for logging statements, it involves writing multiple lines of code that could still miss some relevant state.

To solve this problem we took inspiration from interactive IDEs such as LightTable - while they show the value of every evaluated expression, this only happens during the development process. We suggest taking this idea and extending it to be applicable within a live deployment of the target program.

We are aiming towards a solution through which a developer is able to inspect the complete state of a function, after each execution step, within

```python
def add_if_positive(a, b):
    if a >= 0 and b >= 0:
        s = a + b
    else:
        s = 0
    return s

print add_if_positive(2, 3)  # 5
print add_if_positive(-1, 2) # 0
```

Listing 1. Our hello.py file.

a production system. A possible user interface would allow the developer to scroll through separate function executions. Within each execution they would be able to follow the actual code path that was taken during runtime, while inspecting the values of all the local variables of the given function.

3.2. Python Internals

Our solution heavily relies on the inner workings of the Python interpreter and its execution flow. Thus, before being able to explain our implementation, we should first introduce the concepts and behaviors we are relying on.

3.2.1. *Execution Model.*

Python is an interpreted language, making use of a simple, stack-based virtual machine. We will now look at the events that take place when a developer executes a Python program.

Let us assume we have a file named `hello.py`, as shown in the code listing below. Running `python hello.py` will produce the following two outcomes:

- `5` and `0` will be printed to the screen
- A new file, `hello.pyc` will be present in the current working directory

Here are the steps that have happened in order to produce the two outcomes mentioned above:

(1) The `hello.py` file was loaded.
(2) The code was parsed and checked for syntactic errors.
(3) An abstract syntax tree (AST) representation was built.
(4) The AST was converted into interpreter bytecode.
(5) The bytecode was saved as `hello.pyc`.
(6) The bytecode was executed by the Python interpreter.

There are a few aspects that are worth detailing from the steps above. First of all, the abstract syntax tree representation (AST). ASTs are useful for applying generic transformations and optimizations to code, without having to take into account coding style conventions or even syntax. In Python's case, no optimizations are performed, but the AST representation is still extremely important because it is used to generate the needed bytecode. Python allows the developers to interact with AST representations using the `ast` module.

```
Module(body=[
    FunctionDef(name='add_if_positive', args=arguments(...),
    Print(dest=None, values=[Call(func=Name(id='add_if_positive', ..),
    Print(dest=None, values=[Call(func=Name(id='add_if_positive', ..)
])
```

LISTING 2. AST representation of `hello.py`.

Being an interpreted programming language, Python makes use of a small virtual machine which executes custom instructions, namely the bytecode. Python allows developers to view the bytecode corresponding to a function through the `dis` module.

It is important to mention that all the steps outlined above are performed at runtime. This means that if one is able to change the bytecode of an already loaded function, they can make the Python interpreter execute something else than what was originally intended. It turns out that this is possible, and we will examine just how this can be achieved in the following section.

3.3. Executing Arbitrary Code

We have previously shown what happens whenever a Python script is being executed. Now, we will outline how a developer can access all the internal representation data at runtime, as well as how they can change this data in order to alter the execution of a program.

Let us say we want the `hello` module to print the value of `add_if_positive(-1, 2)` twice. To achieve this, here is what we need to do:

(1) Get the source code of `hello`.
(2) Parse it into the AST representation.
(3) Modify the body of the module such that it has one more print statement with the value of `add_if_positive(-1, 2)`.
(4) Compile the modified AST into a new code object (the type of objects that store bytecode), using the `exec` mode. This is the equivalent of executing the code within the object in the root of the module. That means that if the object defines a function, that function will be defined in the environment as soon as the compiled code is executed.
(5) Get a handle to the global environment of the program.
(6) Execute the code object in the global environment, essentially redefining the imported module.

Now, the globally available `hello` module is not the one we have defined in `hello.py`, but a changed version which has one more print statement at the end.

Having the ability to execute arbitrary code besides any imported module or function allows us to implement the solution described in Section 3.1.

3.4. **Changing Functions to Track State**

The problem we are trying to solve is the following: given a Python function object (which we receive thanks to the decorator syntactic sugar), how can we track the state of all its local variables over time?

We know from Section 3.**3** that we can change a function's body as we wish and then inject it back into the environment. Based on this, if we had a way that takes a snapshot of the local variables within a function at a given time, we could do this after every statement that is in the function.

Python's built-in `locals()` function does exactly this - it returns a dictionary with all the local variables and their values at evaluation time. However, calling `locals()` is not enough, because that information is bound within the stack frame of the function. We need to store this information outside the function, such that we can later dump it to a file.

To achieve this, we define a helper function that takes a state snapshot and a line number (we need this for populating the data in the user interface) and stores the relevant information.

Having the helper function, what we have to do is insert calls to it after each statement of the recorded function. In addition to this, there is one more thing to do. The helper function that we have defined is available in the environment of our library, but not in the environment of the recorded function. Thus, we need to inject one more reference in the target environment.

3.4.1. *Nested Blocks.* Statements such as `if` and `while` introduce nested blocks of code that are represented as a subtree in the AST. To properly handle this, we need a recursive algorithm that knows how to walk tree-like structures and recurse on nested bodies. Furthermore, due to the internal representation of such blocks, we also need to prepend a `record_state` call in order to observe the local state when evaluating an `if` or `while` statement's predicate.

`Try` blocks represent a special case of this, having multiple exception handlers (e.g. every `except` block) that need to be annotated. The same pattern occurs with `elif`, Python's version of the `switch` statement.

3.4.2. *Return Statements.* We cannot inject a `record_state` call *after* a return statement, because it would never be evaluated. Thus, we need to detect return statements and handle them separately by introducing an auxiliary variable where we capture the last value before it is returned. Of course, we have to be careful not to evaluate the returned expression twice, because it might have side effects.

3.4.3. *Storing the Traced Data.* Recording the trace is only one part of the problem at hand. As a function is finished executing, we still need to expose the valuable data somehow. The approach we have chosen is to dump it into a file on the user's filesystem. This way, the data is persisted and it can be visualized later by the programmer.

Although writing to a file is trivial, calling a procedure that does this from someone else's code is not. The first approach that we took was to augment the `return` statement behavior by also inserting a call to the dump function, and this worked for functions with explicit `return` statements.

However, not all Python functions use explicit `return` statements, which made the mentioned approach insufficient. To solve this problem, we returned to the decorator pattern[12]. Instead of simply replacing the user's function with our annotated one, we wrap it into a function that calls the dump data handler after calling the original. This way, we cover both functions that use `return` statements and the ones that do not.

3.**4**.4. *Recursion.* Recursive functions are a special category that needs our attention. Our first implementation had no issues with regard to them. However, after we have added the function wrapper described in Subsection 3.**4**.3, our tests uncovered that recursive functions would trigger infinite loops.

The reason for this is subtle and has to do with the way we called the annotated function in our wrapper. To get a reference to the called function, we would look into the user's environment and get the function object by its name. This worked fine for non-recursive functions. However, when a function ended up calling itself, the Python interpreter would do the same thing. Because of the way decorators work, this time the function object behind that name was our wrapper. The actual function could no longer be reached, and this led to infinite recursion.

To address this issue, we experimented with a few unsuccessful approaches. First, we tried storing the annotated function under a different name, but that would not work for the original recursive calls that used the old name. Then, we attempted altering the AST nodes by changing the function name, and even a simpler approach of just performing text replacement on the source code. However, this approach still did not cover the dynamic calls that Python allows at runtime.

Finally, we found the proper approach. By leveraging the fact that Python decorators replace the original function with the wrapped one, we are certain that any call will point to our wrapper. Then, to avoid the infinite recursion, we store a reference to the annotated original function under a different name in the user's environment. This way, our wrapper never calls itself anymore.

3.**4**.5. *Complete Syntax Support.* Our current implementation supports all of Python's syntax. In a following section, on testing, we discuss how we make sure that every syntax feature is supported and maintained as development advances.

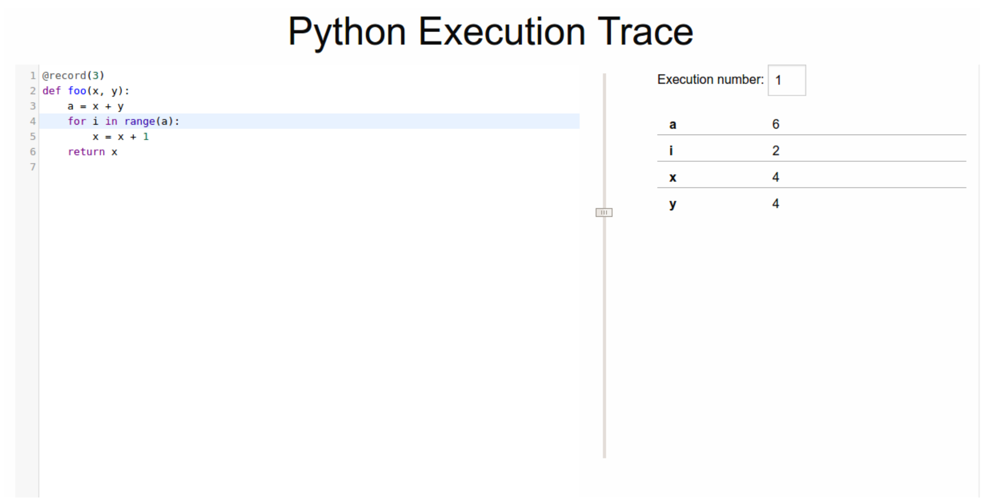
3.**4**.6. *Multiple Executions Suport.* Because most functions are executed more than once, and some of them are even run multiple times per second, our library needs to be able to only record a certain number of executions. This is necessary both from a user friendliness perspective, since the user does not want to be overwhelmed by the quantity of data, but also from a performance perspective, as memory usage increases both in RAM and on disk as more executions are recorded. Therefore, the user can specify a maximum number of executions that should be recorded. After that number is reached, no additional data is stored.

3.4.7. *Reverting to the Original Function.* While considering the performance overhead of our library, we came to realize that there is no need to keep calling the instrumented version of the function after the required number of executions was recorded.

Thanks to Python's decorators, we still have a reference to the original function within our wrapper. This, combined with the *number of executions that were recorded* logic allows us to conditionally call either the instrumented version of the function or the original one. Doing this, we are avoiding the overhead introduced by the record calls, as we will show in the Evaluation chapter.

3.5. **Visualizing the Traced Data**

Recording large amounts of data is useless if there is no convenient way of examining it. Therefore, we also provide an interactive, Web-based visualization tool. This allows the user to explore the recorded traces. On one hand, the user can select a particular trace from the recorded set. On the other hand, they can go through the function's execution using a scroll axis, replaying the way in which the code ran. For every step, all the variables' values are displayed.



```
@record(3)
def foo(x, y):
    a = x + y
    for i in range(a):
        x = x + 1
    return x
```

Execution number: 1

a	6
i	2
x	4
y	4

FIG. 1. Visualization of recorded data.

```python
from execution_trace.record import record

@record()
def repeat(x, count):
    r = []
    for i in range(count):
        r.append(x)
    return r
```

Listing 3. Using `@record`.

3.6. Packaging and Release

Both the decorator and the visualizer are packaged as a standard Python package - *execution-trace* [2] - available through Pypi, the central Python packages index. This makes our tool easily available to all developers.

The install script also places an executable in the user's `PATH`, making the viewer available with a simple command: `view_trace`. Running `view_trace /path/to/dump/file` will start the Web server, making the visualization available at `http://localhost:5000`.

4. Evaluation

Putting it all together, we get the `@record` decorator, which we can use to obtain a step-by-step evolution of the state of all the variables within any function, with minimal setup. Listing 4 shows an example.

As we can see, the only setup needed is to install the record library, which can easily be done using Python's package manager, and then decorate the target function.

Thanks to the way we record the state, by using injected calls after every relevant statement, we get to see the program's evolution, just as we would using a debugger. In the example presented in Listing 13, we can see how `r` grows with every step through the loop.

Then, using the built-in graphical viewer, the programmer can step through the execution of the program in a user friendly way, going through time back and forward as he needs.

4.1. Testing

Our library aims to support all possible Python functions. Even with Python's simple syntax, this still amounts to a large number of possibilities. Thus, to avoid bugs as much as possible, we have created a test suite that covers every case of syntax that we know we support. Furthermore, development on the `record` module happens in a test-driven manner. This makes it easier because first we outline a test case in which we specify what kind of function we are targeting and what calls we are expecting to see, and only after this we start developing the necessary support in the library, having a clear picture about what is needed.

Our tests only check the first argument of the recording function (i.e. the line number) and not also the state snapshot because the value of `lineno` is controlled by our implementation, while `locals()` is a built-in Python function that we safely assume that it works correctly.

In addition to testing the recorded calls, we have also developed a set of schemas for validating the dump file's contents, and we check those as well in every test.

4.2. Performance

While our solution gathers plenty of data in order to provide value to the programmer, we cannot be oblivious to any effects it might have when run in production. High traffic solutions are usually tuned and stretched to the limit in order to provide as much performance as possible, and while useful, our library would not be embeddable in a production system if it meant decreasing the performance of the system being instrumented.

To measure the performance impact of our library, we have set up a naive implementation of finding the prime numbers up to a number `N`, having a helper function that checks (innefficiently, as well) if a number is prime. We chose the innefficient approach because we wanted a CPU-intensive function that is called multiple times, and that is exactly the case with our `is_prime` helper.

We have set up an experiment where we run `find_first_primes` for increasing values of `N` and we measure the time the function calls take.

First, we measure the *baseline* version, with no instrumentation whatsoever, to establish the expected performance. Then, we run the *instrumented* version which records 100 executions but does not have the *revert to original function* optimization (Subsection 3.4.7). Fig. 2 shows that as soon as the instrumented function gets called more often and with larger iterations, performance decreases considerably.

Including the aforementioned optimization, we can observe the overhead diminishing. As Fig. 3 shows, performance is no longer an issue.

4.3. Using the Tool

These are the steps of using *execution-trace*:

(1) install the library: `pip install execution-trace`
(2) annotate the function that should be traced
(3) run the program
(4) execute the logged command: `view_trace /path/to/file.json`

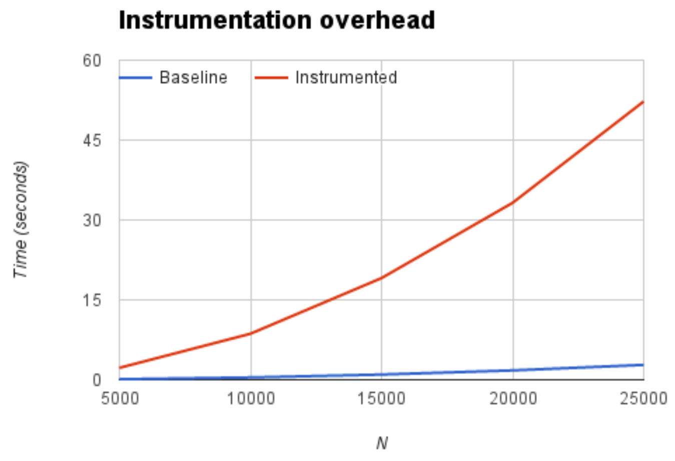


Fig. 2. Always running the instrumented function decreases performance.

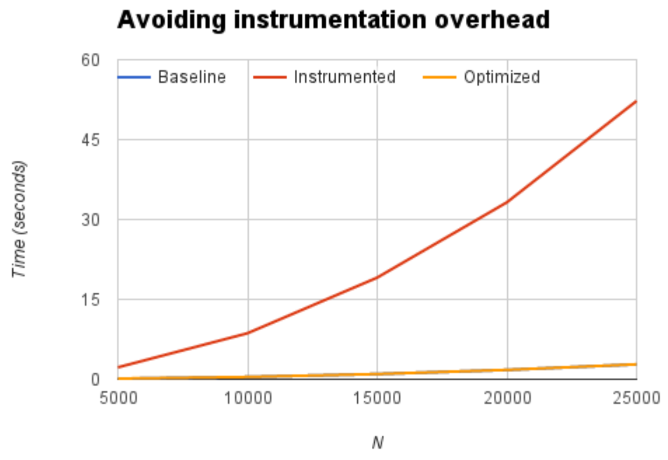


Fig. 3. Reverting to the original function avoids the overhead.

5. Conclusions

We have examined the available developer tools for Python. As we have shown, most needs are already covered, be it memory profiling or even static checking. However, recent developments in other programming languages' communities have produced development experiences with a higher level of interactivity. These interactive IDEs have led us to propose a new approach in debugging - recording a function's execution in production deployments.

Our implementation provides the developer with a simple way to trace the state of any given function, as well as an intuitive graphical interface to explore the traced data. When compared to the usual methods developers have when inspecting unexpected behavior in live deployments, our approach has several advantages:

- All the variables' state is captured, so the programmer does not have to worry about what should be logged and what should not.
- There is no need to reach the live process in order to attach a debugger - the regular deploy flow (no matter how complicated) can be reused.
- Ease of use - the developer only has to add an extra library and annotate a function.
- Visual way of stepping through the execution, being able to both observe the code path taken, as well as the state of the program.
- Performance is not affected - the library can safely be run in production.

REFERENCES

[1] M. F. Sanner *et al.*, "Python: a programming language for software integration and development," *J Mol Graph Model*, vol. 17, no. 1, pp. 57–61, 1999.

[2] "execution-trace." https://github.com/mihneadb/python-execution-trace, 2017.

[3] K. R. M. Leino, "Extended static checking: A ten-year perspective," in *Informatics*, pp. 157–175, Springer, 2001.

[4] "Pylint - code analysis for python." http://pylint.org/, 2017.

[5] "Mypy - optional static typing for python." http://mypy-lang.org/, 2017.

[6] "Pep 0484 - type hints." https://www.python.org/dev/peps/pep-0484/, 2014.

[7] M. Leucker and C. Schallhart, "A brief account of runtime verification," *The Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[8] B. Akerblom, J. Stendahl, M. Tumlin, and T. Wrigstad, "Tracing dynamic features in python programs," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, (New York, NY, USA), pp. 292–295, ACM, 2014.

[9] P. M. Pietrek, "Arbitrary runtime function call tracing," Sept. 4 2012. US Patent 8,261,244.

[10] B. Alsallakh, P. Bodesinsky, A. Gruber, and S. Miksch, "Visual tracing for the eclipse java debugger," in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 545–548, March 2012.

[11] "Light table - a new ide concept." http://www.chris-granger.com/2012/04/12/light-table-a-new-ide-concept/, 2012.

[12] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, "Design patterns: Elements of reusable object-oriented software," *Reading: Addison-Wesley*, vol. 49, no. 120, p. 11, 1995.