Hot Code Patching in CPython

Supporting Edit-and-Continue Debugging in CPython with Less Than 300 Lines of Code

Johannes Henning
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
johannes.henning@hpi.de

David Stangl
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
david.stangl@student.hpi.de

Fabio Niephaus
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.de

Bastian Kruck
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
bastian.kruck@student.hpi.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
robert.hirschfeld@hpi.de

ABSTRACT

Some language runtimes such as Java Hotspot or the virtual machine for Squeak/Smalltalk support edit-and-continue debugging, which allows developers to make changes to a program while it is running. This capability is especially useful for recovering from errors in a program. However, it is not supported by CPython, the reference interpreter for Python.

In this paper, we demonstrate how edit-and-continue debugging can be integrated into CPython in just under 300 lines of code. We evaluate performance implications, demonstrate how this style of debugging improves the programming experience, and discuss limitations of our approach.

CCS CONCEPTS

• **Software and its engineering → Virtual machines**; **Software testing and debugging**.

KEYWORDS

edit and continue, debugging, virtual machines, tail recursion

ACM Reference Format:
Johannes Henning, David Stangl, Fabio Niephaus, Bastian Kruck, and Robert Hirschfeld. 2019. Hot Code Patching in CPython: Supporting Edit-and-Continue Debugging in CPython with Less Than 300 Lines of Code. In *14th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'19), July 19, 2019, London, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3340670.3342424

1 INTRODUCTION

Debugging, the process of finding and fixing faults in computer programs, is an ever-present part of programming. There is a multitude of different techniques and tools that aim to improve the debugging workflow or to offer novel ways of reasoning about the program code. Back-in-time debugging [7], for example, allows the programmer to observe when and where a variable was modified and to go back in time to the moment an erroneous modification was made.

Many of the theoretically available debugging methods are unfortunately only available in a very select pool of programming languages. The actual choice a programmer has in choosing novel debugging techniques is thereby limited in practice. The rudimentary debugging support implemented by most languages are *step-and-continue-debugging*, *printf-debugging*, and *post-mortem debugging* [5, 9].

Some interactive debuggers support a more powerful technique: edit-and-continue (E'N'C) debugging (also known as *edit-and-resume debugging*, *online-debugging*, or *fix-and-continue debugging*). E'N'C debugging is, for example, available in Squeak/Smalltalk, Java Virtual Machine (JVM), and in JavaScript by browsers supporting the Chrome debugging protocol. It refers to the ability to halt a program, exchange parts of the program code (hot code patching), and continue the execution without restarting the entire application. Reducing the number of restarts necessary can save developers a lot of time, particularly when debugging problems that require cumbersome setup or time extensive computations before they occur. We refer to this scenario as *recovering* from an error. In addition, E'N'C debugging can also be helpful in use cases like programming against a novel API and where mistakes occur frequently. If the regular workflow would be to restart repeatedly making small modifications each time, E'N'C can be a faster option. We call this scenario *exploratory programming* [12, 13].

Python is among the languages that do not support E'N'C debugging. In previous work [8] we added support for E'N'C debugging to PyPy, an alternative interpreter for Python. The modifications made to PyPy in order to support E'N'C were minor, which raised the question of how difficult it would be in other Python implementations. In this paper, we will answer this question for the reference implementation of Python, by presenting our light-weight implementation of E'ɴ'C debugging in CPython. Our implementation allows for functions to be exchanged at run-time and for the execution to restart at the start of the modified function. Our implementation passes the CPython test suite and has performance penalty of at most 23% in the Python Performance Benchmark Suite.

The remainder of the paper is structured as follows: We introduce relevant technologies and concepts in Section 2, describe our approach in Section 3 and our implementation of E'ɴ'C in CPython in Section 4, evaluate our approach in Section 5, compare our work to related work in Section 6, discuss future work in Section 7, and close in Section 8.

2 BACKGROUND

CPython. CPython is the reference implementation of Python and is the most widely used Python runtime. CPython is written in C and shipped with many Linux distributions. If you do not know which Python runtime you are using, you are most likely using CPython.

As CPython is the reference implementation, most other implementations follow its evolution. IronPython written in C#, Jython written in Java, or PyPy written in RPython are some of the other important Python implementations that aim to achieve full compatibility with CPython. Thus, features introduced to CPython have a rippling effect on most Python runtime environments and if E'ɴ'C debugging was added to CPython it would likely be adopted by the other implementations as well.

Debugging in Python. Python includes the `PDB` package, which implements a debugger with a command line interface (CLI). `PDB` implements stepping, breaking, variable inspection as well as assignment, and can evaluate Python code just like the regular Python read-eval-print loop (REPL). To use `PDB`, programmers import the package and can, for example, set a breakpoint via the `breakpoint()` builtin in the code and then start the program with the regular shell command. The program then runs as it would regularly until it runs into the breakpoint where it halts and presents a REPL. Users can modify variable values at will and evaluate statements, but if they want to change any of the code of the program they subsequently need to restart this process from the beginning. This cumbersome interaction, as well as the rudimentary tool support (no auto completion or graphical interface), is one of the reasons why *printf debugging* is still one of the most common ways to debug Python programs[1].

Debugging with extended tool support. The runtimes of other languages like Squeak/Smalltalk or Java Hotspot give programmers more options when debugging. In Java, the program is started in a separate debugging mode (which is typically slower) during which stepping, variable inspection, and variable modification can be used just like in Python. Java is typically not developed via command-line, but instead through sophisticated integrated development environments (IDEs) such as Eclipse or IntelliJ IDEA. When function code is modified during debugging and saved, the IDEs automatically recompiles the modified code and patches it into the currently executing program, restarting the modified function if it was currently being executed. For most modifications, restarting the entire application is not necessary.

Squeak/Smalltalk goes even further. Any uncaught run-time error opens its debugger, which is capable of changing class and method definitions on the fly. A possible workflow in Squeak/Smalltalk is *debugger driven development* in which parts of the implementation are left unfinished and are then implemented during program run-time as the debugger pops up because, for example, a method is missing.

These more interactive development styles give the programmer more flexibility during development and debugging.

Resumption model. E'ɴ'C in Squeak/Smalltalk relies on the resumption model of exception handling [8]. But Python uses a different approach:

> "Python uses the "termination" model of error handling: an exception handler can find out what happened and continue execution at an outer level, but it cannot repair the cause of the error and retry the failing operation (except by re-entering the offending piece of code from the top). When an exception is not handled at all, the interpreter terminates execution of the program, or returns to its interactive main loop" [14].

In Squeak/Smalltalk, any uncaught run-time error opens the debugger where the problem can be fixed and execution be resumed at the start of the method. Whereas in Python, unless an exception was previously anticipated, the state we are interested in is already gone when the debugger is invoked and thus, we cannot recover from the error.

The termination model of exception handling is one reason for the limitations of our prototype, which we discuss further in Section 5.5.

3 APPROACH

For supporting E'ɴ'C debugging, the CPython interpreter needs to be extended in the following ways:

First, a prerequisite for E'ɴ'C is the ability to reset an execution context or *frame* in the interpreter and to patch in a new version of the function provided by the user. This can be done by either modifying existing frames or by creating new copies of them.

Second, a new control-flow exception is needed, which can be used to signal that a specific frame should be restarted in the interpreter. It may hold further information such as source code in case the frame should also be patched with the new code.

Third, the interpreter loop needs to know how to handle these kinds of exceptions and needs to facilitate the actual frame restarting and patching.

Lastly, new tooling is required to make E'ɴ'C available to the user. Since Python already comes with its own debugger, we suggest adding commands to it through which the new E'ɴ'C capabilities can be accessed.

[1]"[O]ften the quickest way to debug a program is to add a few print statements to the source: the fast edit-test-debug cycle makes this simple approach very effective"[15]

```
PyObject *
PyEval_EvalFrameEx(PyFrameObject *f, int throwflag)
{
    PyThreadState *tstate = PyThreadState_GET();
    return tstate->interp->eval_frame(f, throwflag);
}

PyObject *
PyEval_EvalFrameRestartable(PyFrameObject *f,
                            int throwflag)
{
    PyThreadState *tstate = PyThreadState_GET();
    PyFrameObject *backup;
    PyObject *retval;
    PyCodeObject *new_code;
    // ...
    while (1) {
        backup = make_frame_backup(f);
        retval = tstate->interp->eval_frame(f, throwflag);
        new_code = NULL;
        // ...
        if (retval != NULL ||
            !PyErr_ExceptionMatches(PyExc_RestartFrame) ||
            !is_matching_restart_frame(f, &new_code))
        {
            // ...
            return retval; // common case
        }
        if (new_code != NULL) {
            // ...
            backup->f_code = new_code; // patch backup
        }
        f = backup; // try again with backup
    }
}
```

Listing 1: CPython's PyEval_EvalFrameEx and our replacement function PyEval_EvalFrameRestartable. Code for reference counting omitted for brevity.

4 IMPLEMENTATION

We have implemented our approach in 293 lines of code (LoC) (according to git diff) in a branch of CPython 3.7.0 (commit 1bf9cc5, tag v3.7.0). All sources including our modifications are available on GitHub[2].

The main part of our modifications is the `PyEval_EvalFrameRestartable` function (see Listing 1). We introduced this function as a replacement for `PyEval_EvalFrameEx`, which is the key function responsible for executing frames in CPython. The function only takes a preconstructed frame and a flag as arguments, has two lines of code, and is therefore a good location for adding E'N'C capabilities at minimal implementation cost.

Our replacement function, on the other hand, also manages a backup, a return value, as well as a code object and executes the frame in a while-true loop. Before evaluating the frame, a backup copy of the frame is created. Although this backup is always allocated and consequently negatively affects performance, working with a copy of the frame was easier than resetting the existing frame object, which proved difficult. This approach has the added advantage of allowing us to recover the original arguments which could have been modified during the execution of the frame. As a consequence of these copies, however, our function also needs to consider the reference counting GC.

[2]https://github.com/hpi-swa-lab/cpython/tree/edit-and-continue-debugging

In the common case, `eval_frame` returns a value which is the returned by our function. If the return value is `NULL`, we check for a `PyExc_RestartFrame` and whether the current frame is the target of the exception. Only then, a new code object is installed in the backup frame if any and finally, the current frame is replaced by the copy before the next iteration of the loop is executed.

With this infrastructure, it is possible to throw a new `RestartFrame` exception in Python code. This exception requires a target frame object as argument which will then be restarted. To also patch this target frame, a code object or function can be provided as an optional argument.

To enable user interaction, we extended `PDB` with a new `restartf` command. This command restarts the current frame of the debugging session by default. Instead of having to find the correct frame object in the call stack, a parent frame can be restarted by providing the number of frames in between that frame and the current frame (e.g., "2" is the parent of the parent frame). `restartf` also supports a second argument (a code object or function) for patching the target frame. Internally, the new `PDB` command throws an appropriate `RestartFrame` exception.

5 EVALUATION AND EXPERIENCE REPORT

Our prototype adds E'N'C debugging capabilities to CPython. In this section, we quantify the performance penalty imposed by our implementation and show the experience of using E'N'C debugging in its current form. Finally, we demonstrate how our modifications can also be used for tail call optimization in Python.

5.1 Benchmarks

For assessing the performance impact of our modifications to CPython, we use the *Python Performance Benchmark Suite* version 0.7.0[3] with `--rigorous` and `--track_memory` for the memory consumption benchmark. We compare trunk CPython 3.7.0 (commit 1bf9cc5, tag v3.7.0) with E'N'CPython[4] (commit ff918a2, branch: edit-and-continue-debugging) which contains the changes discussed in Section 4. Both versions were compiled with the following flags:

```
--enable-optimizations
--with-lto
--with-computed-gotos
```

Benchmarks were run on Ubuntu 18.04.2 LTS (GNU/Linux 4.15.0-47-generic x86_64) with an Intel(R) Core(TM) i5-4690 CPU with microcode version 0x25 and 2x8GB of DDR3L-SDRAM running at 1333 MHz.

Run-time comparison. The performance overhead added by our modifications has a significant[5] impact on 42 of the 60 benchmarks, shown in Table 1. As we add overhead to each function call noticeable differences in performance were to be expected. Depending on the number of function calls made in each benchmark, the performance penalty varies between +23% and -3%. We should point out that the 3% improvement in run-time performance in the `scimark_monte_carlo` benchmark is within the margin of error. As our

[3]https://github.com/python/performance
[4]https://github.com/hpi-swa-lab/cpython
[5]The Python Performance Benchmark Suite classifies results as "significant" using a Student's two-sample, two-tailed t-test with $\alpha = 0.95$ (see https://git.io/fj1p1).

modifications only add overhead, an increase in performance is improbable.

While the impact on performance with up to 23% is significant, it is lower than we expected. We believe that in the context of debugging it is an acceptable trade-off for the functionality we are adding. However, it is too large of an overhead to add to every program running in CPython. Accordingly, we have further ideas on how to optimize our approach, as well as feasible ways to confine this performance impact to the context of debugging, which we will discuss in Section 7.

Memory comparison. The memory consumption added by our prototype is insignificant for 55 of the 60 benchmark algorithms. The other five are presented in Table 2. The highest memory consumption increase happens in benchmark *xml_etree_process*, with a 3% increase. The **`float`** benchmark has reportedly 24% less memory consumption when executed in our prototype. This does not make sense, as our modifications only add overhead when compared to unmodified CPython. We believe this to be a measuring error, as the standard deviation for this benchmark is particularly high (7-12%). Furthermore, we ran the benchmark suite multiple times and observed fluctuation in particular when tracking memory consumption, which might suggest a systemic error in this part of the benchmark design.

Overall the memory consumption of our prototype is very close to that of CPython, imposing at most 3% increased memory consumption during debugging which should be inconsequential in practice.

5.2 Python integration

Our prototype passes the CPython test suite, with only small modifications. As discussed in Section 4, we add the `RestartFrame` exception to CPython. The test suite contains two tests (`test_baseexception.py` and `test_pickle.py`), that depend on the list of built-in exceptions to which we added ours. Thus, we also had to add our new exception to the exception hierarchy in `exception_hierarchy.txt` for `test_baseexception.py` and add it to the list of exceptions that have no reverse mapping in `test_pickle.py`.

No further modifications were necessary.

5.3 Experience Report

Scenario: Recover. To illustrate how the programmer can recover from an error without restarting potentially expensive computation, consider the example in Listing 2. After spending the majority of execution time inside the `mandelbrot` function, the program tries to save the computed values within the `save` function. This will open up `PDB`, because the image files are not opened in binary mode, see Listing 3. Through `interact`, programmers can enter the interactive mode in Listing 4, where they can define a new function `save_fixed`, in which the file is opened in binary mode, see Listing 5. By using the newly added `restartf` command, they can restart the current frame with the newly defined function, after which the save operation succeeds in Listing 6.

In this example, the programmer was able to recover from his error and fix the save function from inside the debugger, without having to recompute the mandelbrot data. While this usability can be improved, see Section 7, we believe this to be a useful improvement upon the regular CPython debugging experience and a successful demonstration of the capability of our prototype.

Table 1: Mean run-time benchmark (abbreviated for space) results with standard deviation, as produced by `pyperformance compare`, sorted by most performance overhead to least.

Benchmark	CPython	E'N'CPython	±%
genshi_text	44.5 ms±0.6 ms	54.7 ms±0.9 ms	+23%
genshi_xml	90.3 ms±0.8 ms	109.7 ms±1.5 ms	+21%
logging_silent	266 ns±8 ns	317 ns±10 ns	+19%
xml...process	120 ms±2 ms	137 ms±2 ms	+14%
hexiom	14.1 ms±0.1 ms	16.0 ms±0.1 ms	+13%
raytrace	767 ms±6 ms	865 ms±8 ms	+13%
richards	111 ms±1 ms	126 ms±2 ms	+13%
logging_simple	13.1 us±0.2 us	14.7 us±0.2 us	+12%
sympy_sum	135 ms±1 ms	151 ms±2 ms	+12%
deltablue	10.9 ms±0.2 ms	12.2 ms±0.2 ms	+11%
logging_format	14.5 us±0.2 us	16.1 us±0.2 us	+11%
nqueens	136 ms±1 ms	151 ms±1 ms	+11%
pick...python	702 us±10 us	775 us±9 us	+11%
scimark_sor	270 ms±7 ms	300 ms±3 ms	+11%
sympy_expand	562 ms±4 ms	622 ms±10 ms	+11%
scimark_lu	249 ms±3 ms	275 ms±6 ms	+10%
sympy_str	266 ms±2 ms	292 ms±3 ms	+10%
2to3	432 ms±3 ms	473 ms±3 ms	+9%
go	366 ms±3 ms	400 ms±3 ms	+9%
sympy_int...	26.9 ms±0.2 ms	29.3 ms±0.2 ms	+9%
tornado_http	245 ms±3 ms	267 ms±4 ms	+9%
unpick...python	546 us±4 us	593 us±8 us	+9%
xml...generate	146 ms±2 ms	160 ms±3 ms	+9%
xml...iterparse	141 ms±4 ms	154 ms±2 ms	+9%
django_t...	177 ms±2 ms	192 ms±2 ms	+8%
regex_compile	255 ms±2 ms	275 ms±2 ms	+8%
spectral_norm	185 ms±2 ms	199 ms±2 ms	+8%
html5lib	126 ms±5 ms	134 ms±5 ms	+7%
sql...declarative	204 ms±3 ms	218 ms±3 ms	+7%
sql...imperative	39.7 ms±0.9 ms	42.6 ms±0.8 ms	+7%
dulwich_log	103 ms±1 ms	109 ms±1 ms	+6%
pathlib	27.2 ms±0.5 ms	28.8 ms±0.7 ms	+6%
chameleon	14.6 ms±0.2 ms	15.2 ms±0.3 ms	+4%
chaos	169 ms±2 ms	175 ms±2 ms	+4%
float	155 ms±2 ms	162 ms±2 ms	+4%
mako	27.6 ms±0.2 ms	28.7 ms±0.5 ms	+4%
json_dumps	16.7 ms±0.4 ms	17.2 ms±0.5 ms	+3%
pickle	13.0 us±0.2 us	13.3 us±0.2 us	+2%
python_startup	9.83 ms±0.18 ms	10.04 ms±0.20 ms	+2%
p...no_site	6.97 ms±0.22 ms	7.13 ms±0.22 ms	+2%
unpickle_list	4.70 us±0.10 us	4.80 us±0.02 us	+2%
scimark...carlo	162 ms±4 ms	157 ms±6 ms	-3%

Scenario: Exploratory programming. Continuously modifying the current function in order to explore the right way to, for example, program against a novel API, could in theory be done in the same

Table 2: Mean memory consumption with standard deviation in benchmarks (abbreviated for space) run with `--track-memory`, as produced by `pyperformance compare`.

Benchmark	CPython	E'n'CPython	±%
xml...process	11.6 MB±260.2 kB	11.9 MB±438.7 kB	+3%
genshi_text	9.9 MB±120.2 kB	9.6 MB±147.2 kB	-2%
xml...generate	12.2 MB±592.7 kB	11.9 MB±274.9 kB	-2%
genshi_xml	10.1 MB±0.1 MB	9.8 MB±42.1 kB	-3%
float	19.9 MB±1.4 MB	16.0 MB±2.0 MB	-24%

```
import numpy as np
import matplotlib.pyplot as plt
from pathlib import Path

def mandelbrot(h, w, maxit=20, yield_each=5):
    # omitted for brevity

def save(images, directory):
    print('Saving_images...')
    try:
        dir_path = Path(directory)
        dir_path.mkdir(exist_ok=True)
        for it, img in images:
            img_path = dir_path / f'mandelbrot{it:03}.png'
            with img_path.open('w') as f:
                plt.imsave(f, img, format='png')
    except Exception:
        breakpoint()
    print('Finished!')


images = list(mandelbrot(5000, 5000,
                        maxit=100, yield_each=10))
save(images, 'mandelbrot')
```

Listing 2: Saving mandelbrot image[6] with wrong argument in open().

```
> python mandelbrot.py
Generating mandebrot images...
Finished!
Saving images...
>~/mandelbrot.py(37)save() -> print('Finished!')
(Pdb)
```

Listing 3: Exception on saving

```
(Pdb) interact
*interactive*
>>>
```

Listing 4: Entering interactive mode

way as in our *recover* example above. However, the usability of the REPL to implement functions and incrementally modify existing source code is subpar, as explained in Section 2. We believe that a workflow similar to Eclipse or Chrome could be achieved on top of our prototype, which we discuss in Section 7.

[6]Source of mandelbrot function: https://docs.scipy.org/doc/numpy/user/quickstart.html

```
>>> def save_fixed(images, directory):
...     print('Saving images (fixed)...')
...     try:
...         dir_path = Path(directory)
...         dir_path.mkdir(exist_ok=True)
...         for it, img in images:
...             img_path = dir_path / f'mandelbrot{it:03}.png'
...             with img_path.open('wb') as f:
...                 plt.imsave(f, img, format='png')
...     except Exception:
...         breakpoint()
...     print('Finished!')
...
```

Listing 5: Patching the save function

```
>>> restartf(0, save_fixed.__code__)
Saving images (fixed)...
Finished!
>
```

Listing 6: Restarting the fixed function, after which saving succeeds

```
import inspect

class State:
  def __init__(self, n):
    self.n = n
    self.result = 1

def factorial(state):
  if state.n == 1:
    return state.result
  state.result *= state.n
  state.n -= 1
  raise RestartFrame(inspect.currentframe())
```

Listing 7: Factorial with tail recursion.

5.4 Tail Call Optimization

Another advantage of our approach is that it enables tail call optimization [1]. Instead of allocating frames for a tail-recursive function call, we raise a *RestartFrame* exception introduced in Section 4 and save the function *state* in a separate variable, see Listing 7.

Our thus written *factorial* function does not produce StackOverflow exceptions as it would in CPython. Accordingly, we can compute `factorial(50000)` in our prototype, but not in CPython. This transformation of moving the state into a parameter and replacing the tail call with raising an exception could be done automatically by the interpreter.

5.5 Limitations

Our implementation has four noteworthy limitations:

(1) When a function is restarted in our prototype, side-effects of the previous iteration are not reset. This means that any modification to global variables, mutable parameters (only references are reset), and IO will keep any modification the first iteration of the function made, possibly leading to different behavior during the next function call. This limitation

is shared by all other E'N'C implementations that we are aware of.

(2) When we modify a function in our prototype, only the call site in the current activation of the function is modified. This means that other references to the function are not updated and still reference the unmodified version. We discuss a possible solution for this limitation in Section 7.

(3) As our prototype works by exchanging code objects, we do not support restarting C-functions that are provided by C-extensions or the interpreter. We are only able to modify and restart the extension wrapper functions.

(4) Unlike in Squeak/Smalltalk, Python does not automatically open a debugger when it encounters an uncaught exception, making our *recover* use case only possible with some prior anticipation by the programmer, as demonstrated in Section 5.3. This is a limitation imposed by the termination model exception handling that Python uses, as explained in Section 2.

6 RELATED WORK

In this section, we discuss work related to our approach and implementation.

Edit-and-continue debugging in other languages. Other runtime environments implement E'N'C as well, namely Squeak/Smalltalk, the JVM, and JavaScript in browsers supporting the Chrome debugging protocol. We do not claim contributions to E'N'C itself, our goal was to bring this feature to CPython.

Live Multi-language Development and Runtime Environments. The multi-language runtime of Squimera [8] contains a modified version of PyPy [2], an alternative interpreter for Python. This modified interpreter supports E'N'C debugging similarly to our implementation with similar limitations. The purpose of this paper was to show that E'N'C debugging can also be implemented in CPython without requiring major changes to the underlying architecture.

reloadr. Reloadr[7] is a Python project for hot code reloading. It works via annotations on functions or classes prior to starting the program. Once the program is running, reloadr can exchange the function implementation at run-time, meaning that the next call to the function will call the modified behavior.

While reloadr supports hot-code reloading implemented without modification to the interpreter, it does not support modifying the current execution of a function and thus cannot support E'N'C debugging. Additionally, reloadr relies on RedBaron[8] for the source code modification, which provides its functionality by generating full syntax trees via its own parser. Because of this separate implementation, RedBaron and subsequently reloadr do not support the full Python 3.7 syntax at the time of writing.

Our approach sidesteps such obstacles by being a small modification to the interpreter, with no need for the considerably more complex features of RedBaron.

[7]https://github.com/hoh/reloadr
[8]https://github.com/PyCQA/redbaron

live-py-plugin. The live-py-plugin[9] works by continuously restarting program execution, thus achieving a live experience for the programmer. In contrast to our solution, the live-py-plugin offers a trace of the program execution over time, while we offer the ability to edit the state in a snapshot of the program during execution. This discussion on different perspectives on live- and exploratory programming is extensive [11] [10] and outside the scope of this paper. However, by the nature of this solution, it does not work well with large code bases, as the short feedback loop is impossible to achieve with time-consuming setup of the relevant runtime state.

The approach of the live-py-plugin is also not suited for our *recover* use case, as a potentially valuable state is discarded after each modification of source code.

Similarly to the reloadr project, the solution here is also more complex than our approach in both code complexity, setup and resource consumption.

Interactive Python Development. There are other solutions that allow for the interactive development of Python programs, which is in a limited way also useful for interactive debugging. Jupyter notebooks [6] on top of IPython achieves this more interactive development of Python, by allowing the program to be split into multiple blocks that can be edited and restarted on their own, persisting the global state of all previous executions. Similarly to the live-py-plugin, this approach is limited by the restarting approach and thus is not suited for our *recover* use case.

Tail-call optimization. The tco project[10] aims at optimizing tail-calls in Python. While the approach is similar to ours at run-time, their solution is implemented in pure Python and works via annotations. There are no benchmarks available for the tco project, thus we do not know how our approaches (raising exceptions versus added dispatch for annotations) compare performance-wise. Our prototype however aims to achieve E'N'C debugging and enables tail-call optimization as a side benefit. Tail-call optimization is not the core contribution, but a side-effect of this work.

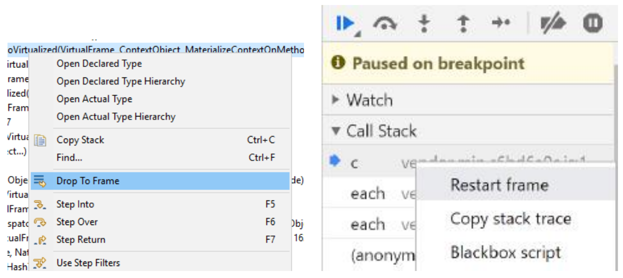
7 FUTURE WORK

Engaging the debugger. In Section 5.1 we mentioned a performance penalty of up to 23%. There are multiple possible ways to confine this penalty to the context of debugging, where performance is not as important as during run-time. The easiest way is to have a separate binary for Python, that is used on development systems for debugging while production systems run the regular CPython without E'N'C support. Another way is to activate the E'N'C functionality via a *-debug* flag. In this case the performance impact would be minimal when not debugging, as only the flag would have to be checked.

Another possibility would be to only add E'N'C functionality when a debugger gets imported via `import pdb`. With this variant, we would only start copying frames at the point in the program where PDB is imported, thereby only introducing the performance penalty from that point forward.

We aim to test the usefulness and performance impact of these ideas in future work.

[9]https://github.com/donkirkby/live-py-plugin
[10]https://github.com/baruchel/tco



(a) Eclipse (b) Chrome

Figure 1: E'ɴ'C debugging tooling for restarting in Eclipse and Chrome.

Tool support. While our prototype integrates in PDB, modifying existing functions via the CLI is cumbersome compared with the existing solutions in Eclipse, Chrome, or Squeak/Smalltalk, see Figure 1. Integration of our prototype with a graphical Python IDE should be possible by exposing our additions to PDB graphically. This could improve the usability of E'ɴ'C in our prototype and the *exploratory programming* scenario immensely.

Persistent patching. With our current solution, the current call site of the function is modified, i.e. the pointer is modified. Any other references to the same function are not updated and still point to the unmodified version. This is due to our implementation choice of implementing the hot-code patching inside the `PyEval_EvalFrameEx` function as opposed to further down in the stack where we would still have access to the function object, as explained in Section 4.

One way to extend this functionality would be to keep a record of all code-objects we have patched (i.e. save the mapping of the old pointer to the new pointer in a Python `dict`) and redirecting any call to this code-object (old pointer) to our modified version (new pointer). This would introduce additional overhead to any function call, which should be minimal as long as no functions are modified and the mapping-structure is empty. First tests of this idea look promising and we hope to expand onto this idea in future work.

Modifying function arguments. In our current prototype, we allow the programmer to modify the argument-count and -name of the function it is modifying. We are able to support this because we have a copy of the entire frame. We chose to copy the frame because it was the simplest way to reset a frame in order to restart a function. With more time and better knowledge of the interpreter, it would probably be possible to implement E'ɴ'C without this copy. This would remove the copy operations thereby reducing performance as well as memory overhead. How big the performance impact would be is another interesting question for future work.

8 CONCLUSION

In this paper, we have shown what it takes to support E'ɴ'C debugging in CPython. Our proposed solution can be implemented in just under 300 lines of code and introduces a performance penalty of at most 23%. We have demonstrated the added utility of E'ɴ'C debugging in Python and suggested ways to circumvent the performance penalty in performance critical usage scenarios.

In conclusion, E'ɴ'C debugging is easy to implement in CPython and can offer significant usability benefits to programmers.

REFERENCES

[1] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and interpretation of computer programs* (2 ed.). MIT Press.

[2] Carl Friedrich Bolz, Antonio Cuni, Maciej Fijalkowski, and Armin Rigo. 2009. Tracing the Meta-level: PyPy's Tracing JIT Compiler. In *Proceedings of the 4th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09).* ACM, New York, NY, USA, 18–25. https://doi.org/10.1145/1565824.1565827

[3] Mikhail Dmitriev. 2004. Profiling Java Applications Using Code Hotswapping and Dynamic Call Graph Revelation. In *Proceedings of the 4th International Workshop on Software and Performance (WOSP '04).* ACM, New York, NY, USA, 139–150. https://doi.org/10.1145/974044.974067

[4] Marc Eaddy and Steven Feiner. 2005. *Multi-language edit-and-continue for the masses.* Technical Report CUCS-01505. Columbia University Department of Computer Science.

[5] Kirk Glerum, Kinshuman Kinshumann, Steve Greenberg, Gabriel Aul, Vince Orgovan, Greg Nichols, David Grant, Gretchen Loihle, and Galen Hunt. 2009. Debugging in the (Very) Large: Ten Years of Implementation and Experience. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09).* ACM, New York, NY, USA, 103–116. https://doi.org/10.1145/1629575.1629586

[6] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian E Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica B Hamrick, Jason Grout, Sylvain Corlay, et al. 2016. Jupyter Notebooks-a Publishing Format for Reproducible Computational Workflows.. In *ELPUB*. 87–90.

[7] Bil LEWIS. 2003. Debugging Backwards in Time. In *proceedings of the Fifth International Workshop on Automated Debugging (AADEBUG 2003), September.*

[8] Fabio Niephaus, Tim Felgentreff, Tobias Pape, Robert Hirschfeld, and Marcel Taeumel. 2018. Live Multi-language Development and Runtime Environments. 2 (2018). Issue 3.

[9] David Pacheco. 2011. Postmortem Debugging in Dynamic Environments. *Queue* 9, 10, Article 12 (10 2011), 12 pages. https://doi.org/10.1145/2039359.2039361

[10] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. 2016. How Live Are Live Programming Systems? Benchmarking the Response Times of Live Programming Environments. In *Proceedings of the Programming Experience 2016 (PX/16) Workshop (PX/16).* ACM, New York, NY, USA, 1–8. https://doi.org/10.1145/2984380.2984381

[11] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding: A Literature Study ComparingPerspectives on Liveness. In *The Art, Science, and Engineering of Programming 3.1 (July 23, 2018).* https://doi.org/10.22152/programming-journal.org/2019/3/1

[12] David Sandberg. 1988. Smalltalk and Exploratory Programming. *ACM Sigplan Notices* 23, 10 (1988), 85–92. https://doi.org/10.1145/51607.51614

[13] Jason Trenouth. 1991. A Survey of Exploratory Software Development. *Comput. J.* 34, 2 (1991), 153–163. https://doi.org/10.1093/comjnl/34.2.153

[14] Guido van Rossum. 1995. *Python Reference Manual.* Technical Report CS-R9525. Centrum voor Wiskunde en Informatica (CWI).

[15] Guido van Rossum. 1998. What is Python? Executive Summary. http://www.python.org/doc/essays/blurb.html. Accessed: 2018-02-01.

[16] Andreas Zeller. 2005. *Why Programs Fail: A Guide to Systematic Debugging.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.