NoPdb

Ondřej Cífka

Nov 25, 2021

CONTENTS

CHAPTER

ONE

INTRODUCTION

NoPdb is a **programmatic** (non-interactive) **debugger** for Python. This means it gives you access to **debugger-like superpowers** directly from your code. With NoPdb, you can:

- **capture function calls**, including arguments, local variables, return values and stack traces
- **set "breakpoints"** that trigger user-defined actions when hit, namely:
  - **evaluate expressions** to retrieve their values later
  - **execute arbitrary code**, including modifying local variables
  - **enter an interactive debugger** like *pdb*

Note: NoPdb should run at least under CPython and PyPy. Most features should work under any implementation as long as it has `sys.settrace()`.

CHAPTER

TWO

CONTENTS

2.1 Getting Started

2.1.1 Capturing function calls

The functions *capture_call()* and *capture_calls()* allow capturing useful information about calls to a given function. They are typically used as context managers, e.g.:

```
with nopdb.capture_call(fn) as call:
    some_code_that_calls_fn()

    print(call)  # see details about how fn() was called
```

Note: Only calls to pure-Python functions can be captured. Built-in functions and C extensions are not supported.

To have a concrete example, let's first define some simple functions to work with:

```
>>> def f(x, y):
...     z = x + y
...     return 2 * z
>>> def g(x):
...     return f(x, x)
```

Now let's try calling `g()` and capturing the call to `f()` that will be made from there:

```
>>> with nopdb.capture_call(f) as call:
...     g(1)
4
>>> call
CallCapture(name='f', args=OrderedDict(x=1, y=1), return_value=4)
>>> call.args['x']
1
>>> call.return_value
4
>>> call.locals
{'x': 1, 'y': 1, 'z': 2}
>>> call.print_stack()
  File "<stdin>", line 2, in <module>
  File "<stdin>", line 2, in g
  File "<stdin>", line 1, in f
```

The object returned by *capture_calls()* will always contain information about the *most recent* call within the context manager block. To capture *all* the calls, we can use *capture_calls()* (in the plural):

```
>>> with nopdb.capture_calls(f) as calls:
...     g(1)
...     g(42)
4
168
>>> calls
[CallInfo(name='f', args=OrderedDict(x=1, y=1), return_value=4),
 CallInfo(name='f', args=OrderedDict(x=42, y=42), return_value=168)]
```

Both *capture_call()* and *capture_calls()* support different ways of specifying which function(s) should be considered:

- We may pass a function or its name, i.e. `capture_calls(f)` or `capture_calls('f')`.
- Passing a method bound to an instance, as in `capture_calls(obj.f)`, will work as expected: only calls invoked on that particular instance (and not other instances of the same class) will be captured.
- A module, a filename or a full file path can be passed, e.g. `capture_calls('f', module=mymodule)` or `capture_calls('f', file='mymodule.py')`.
- If no arguments are supplied, calls to *all* Python functions will be captured.

2.1.2 Setting breakpoints

Like conventional debuggers, NoPdb can set breakpoints. However, because NoPdb is a *non-interactive* debugger, its breakpoints do not actually stop the execution of the program. Instead, they allow executing actions scheduled in advance, such as evaluating expressions.

To set a breakpoint, call the *breakpoint()* function. A breakpoint object is returned, allowing to schedule actions using its *eval()*, *exec()* and *debug()* methods.

Using the example from the previous section, let's try to use a breakpoint to capture the value of a variable:

```
>>> with nopdb.breakpoint(f, line=3) as bp:
...     z_values = bp.eval('z')  # Get the value of z whenever the breakpoint is hit
...
...     g(1)
...     g(42)
4
168
>>> z_values
[2, 84]
```

Note: There are multiple ways to specify the breakpoint location (see the reference for *breakpoint()* for a detailed description of all the parameters). Like in a classical debugger, we can pass a `filename` and a `line` number. Like above, we can also pass a `function` (or its name). Note that lines are always counted from the beginning of the file or notebook cell, and the breakpoint will be triggered *just before* executing the given line.

A more convenient option is to provide the *source code* of the desired line (ignoring surrounding whitespace), for example:

```
with nopdb.breakpoint(f, line='return 2 * z') as bp:
    ...
```

`line` can also be omitted, in which case the breakpoint will be triggered every time the given function is called.

A conditional breakpoint can be set using the `cond` parameter.

Not only can we capture values, we can also modify them!

```
>>> with nopdb.breakpoint(f, line=3) as bp:
...     # Get the value of z, then increment it, then get the new value
...     z_before = bp.eval('z')
...     bp.exec('z += 1')
...     z_after = bp.eval('z')
...
...     g(1)  # This would normally return 4
6
>>> z_before
[2]
>>> z_after
[3]
```

Warning: Assigning to local variables is somewhat experimental and only supported under CPython (the most common Python implementation) and PyPy.

2.1.3 The `NoPdb` class

Another way to use NoPdb is by creating a *`NoPdb`* object. The object can either be used as a context manager, or started and stopped explicitly using the *`start()`* and *`stop()`* methods. This can be useful if we want to set multiple breakpoints or call captures in a single context:

```
with nopdb.NoPdb():
    f_call = nopdb.capture_call(f)
    g_call = nopdb.capture_call(g)
    z_val = nopdb.breakpoint(f, line=3).eval('z')

    g(1)
```

Or alternatively:

```
dbg = nopdb.NoPdb()
f_call = dbg.capture_call(f)
g_call = dbg.capture_call(g)
z_val = dbg.breakpoint(f, line=3).eval('z')

dbg.start()
g(1)
dbg.stop()
```

Note: While it is possible to create multiple *`NoPdb`* objects, they cannot be active simultaneously. Starting a new instance will pause the currently active instance.

2.2 API Reference

nopdb.**capture_call**(*function: Union[Callable, str, None] = None*, ***, *module: Optional[module] = None*, *file: Union[str, os.PathLike, None] = None*, *unwrap: bool = True*) → nopdb.call_capture.CallCapture

Capture a function call.

The returned object can be used as a context manager, which will cause the capturing to stop at the end of the block.

If multiple calls occur, the returned object will be updated as each call returns. At the end, the returned object will contain information about the call that was the last to return.

Parameters

- **function** (*Callable or str, optional*) – A Python callable or the name of a Python function. If an instance method is passed, only calls invoked on that particular instance will be captured.
- **module** (*ModuleType, optional*) – A Python module. If given, only calls to functions defined in this module will be captured.
- **file** (*str or PathLike, optional*) – A path to a Python source file. If given, only calls to functions defined in this file will be captured. If a string is passed, it will be used as a glob-style pattern for `pathlib.PurePath.match()`. If a path-like object is passed, it will be resolved to a canonical path and checked for an exact match.
- **unwrap** (*bool, optional*) – Whether or not to unwrap *function* when it is a wrapper (e.g. produced by a decorator). Only works when *function* is given as a callable. Defaults to *True*.

Returns An instance of *CallInfo* which also works as a context manager.

Return type *CallCapture*

nopdb.**capture_calls**(*function: Union[Callable, str, None] = None*, ***, *module: Optional[module] = None*, *file: Union[str, os.PathLike, None] = None*, *unwrap: bool = True*) → nopdb.call_capture.CallListCapture

Capture function calls.

The return value is an initially empty list, which is updated with a new item as each call returns. At the end, the list will contain a *CallInfo* object for each call, following the order in which the calls returned.

The return value can also be used as a context manager, which will cause the capturing to stop at the end of the block.

Parameters

- **function** (*Callable or str, optional*) – A Python callable or the name of a Python function. If an instance method is passed, only calls invoked on that particular instance will be captured.
- **module** (*ModuleType, optional*) – A Python module. If given, only calls to functions defined in this module will be captured.
- **file** (*str or PathLike, optional*) – A path to a Python source file. If given, only calls to functions defined in this file will be captured. If a string is passed, it will be used as a glob-style pattern for `pathlib.PurePath.match()`. If a path-like object is passed, it will be resolved to a canonical path and checked for an exact match.

- **unwrap** (*bool, optional*) – Whether or not to unwrap *function* when it is a wrapper (e.g. produced by a decorator). Only works when *function* is given as a callable. Defaults to *True*.

Returns A list of *CallInfo* objects which also works as a context manager.

Return type *CallListCapture*

nopdb.**breakpoint**(*function: Union[Callable, str, None] = None*, ***, *module: Optional[module] = None*, *file: Union[str, os.PathLike, None] = None*, *line: Union[int, str, None] = None*, *cond: Union[str, bytes, code, None] = None*, *unwrap: bool = True*) → nopdb.breakpoint.Breakpoint

Set a breakpoint.

The returned *Breakpoint* object works as a context manager that removes the breakpoint at the end of the block.

The breakpoint itself does not stop execution when hit, but can trigger user-defined actions; see *Breakpoint.eval()*, *Breakpoint.exec()*, *Breakpoint.debug()*.

At least a function, a module or a file must be specified. If no function is given, a line is also required.

Example:

```
# Stop at the line in f that says "return y"
with nopdb.breakpoint(function=f, line="return y") as bp:
    x = bp.eval("x")              # Schedule an expression
    type_y = bp.eval("type(y)")   # Another one
    bp.exec("print(y)")           # Schedule a print statement

    # Now run some code that calls f
    # ...

print(x, type_y)  # Retrieve the recorded values
```

Parameters

- **function** (*Callable or str, optional*) – A Python callable or the name of a Python function. If an instance method is passed, only calls invoked on that particular instance will trigger the breakpoint.
- **module** (*ModuleType, optional*) – A Python module.
- **file** (*str or PathLike, optional*) – A path to a Python source file. If a string is passed, it will be used as a glob-style pattern for `pathlib.PurePath.match()`. If a path-like object is passed, it will be resolved to a canonical path and checked for an exact match.
- **line** (*int or str, optional*) – The line at which to break. Either of the following:
  - The line number, counted from the beginning of the file.
  - The source code of the line. The code needs to match exactly, except for leading and trailing whitespace.
  - *None*; in this case, a *function* must be passed, and the breakpoint will be triggered every time the function is called.

  Note that unlike in *pdb*, the breakpoint will only get triggered by the exact given line. This means that some lines will not work as breakpoints, e.g. if they are part of a multiline statement or do not contain any code to execute.

- **cond** (*str, bytes or CodeType, optional*) – A condition to evaluate. If given, the breakpoint will only be triggered when the condition evaluates to true.
- **unwrap** (*bool, optional*) – Whether or not to unwrap *function* when it is a wrapper (e.g. produced by a decorator). Only works when *function* is given as a callable. Defaults to *True*.

Returns The breakpoint object, which also works as a context manager.

Return type *Breakpoint*

nopdb.**get_nopdb**() → nopdb.nopdb.NoPdb

Return an instance of *NoPdb*.

If a *NoPdb* instance is currently active in the current thread, that instance is returned. Otherwise, the default instance for the current thread is returned.

class nopdb.**NoPdb**

The main NoPdb class.

Multiple instances can be created, but only one can be active in a given thread at a given time. It can be used as a context manager.

add_callback(*scope: nopdb.scope.Scope, callback: Callable[[frame, str, Any], Any], events: Iterable[str]*) → nopdb.common.Handle

Register a low-level callback for the given type(s) of events.

Parameters

- **scope** (Scope) – The scope in which the callback should be active.
- **callback** (*TraceFunc*) – The callback function. It should have the same signature as the function passed to `sys.settrace()`, but its return value will be ignored.
- **events** (*Iterable[str]*) – A list of event names (`'call'`, `'line'`, `'return'` or `'exception'`); see `sys.settrace()`.

Returns A handle that can be passed to *remove_callback()*.

Return type Handle

breakpoint(*function: Union[Callable, str, None] = None, *, module: Optional[module] = None, file: Union[str, os.PathLike, None] = None, line: Union[int, str, None] = None, cond: Union[str, bytes, code, None] = None, unwrap: bool = True*) → nopdb.breakpoint.Breakpoint

Set a breakpoint.

The returned *Breakpoint* object works as a context manager that removes the breakpoint at the end of the block.

The breakpoint itself does not stop execution when hit, but can trigger user-defined actions; see *Breakpoint.eval()*, *Breakpoint.exec()*, *Breakpoint.debug()*.

At least a function, a module or a file must be specified. If no function is given, a line is also required.

Example:

```
# Stop at the line in f that says "return y"
with nopdb.breakpoint(function=f, line="return y") as bp:
    x = bp.eval("x")              # Schedule an expression
    type_y = bp.eval("type(y)")   # Another one
    bp.exec("print(y)")           # Schedule a print statement
```

(continues on next page)

(continued from previous page)

```
    # Now run some code that calls f
    # ...

print(x, type_y)  # Retrieve the recorded values
```

Parameters

- **function** (*Callable or str, optional*) – A Python callable or the name of a Python function. If an instance method is passed, only calls invoked on that particular instance will trigger the breakpoint.
- **module** (*ModuleType, optional*) – A Python module.
- **file** (*str or PathLike, optional*) – A path to a Python source file. If a string is passed, it will be used as a glob-style pattern for `pathlib.PurePath.match()`. If a path-like object is passed, it will be resolved to a canonical path and checked for an exact match.
- **line** (*int or str, optional*) – The line at which to break. Either of the following:
  - The line number, counted from the beginning of the file.
  - The source code of the line. The code needs to match exactly, except for leading and trailing whitespace.
  - *None*; in this case, a *function* must be passed, and the breakpoint will be triggered every time the function is called.

  Note that unlike in *pdb*, the breakpoint will only get triggered by the exact given line. This means that some lines will not work as breakpoints, e.g. if they are part of a multiline statement or do not contain any code to execute.
- **cond** (*str, bytes or CodeType, optional*) – A condition to evaluate. If given, the breakpoint will only be triggered when the condition evaluates to true.
- **unwrap** (*bool, optional*) – Whether or not to unwrap *function* when it is a wrapper (e.g. produced by a decorator). Only works when *function* is given as a callable. Defaults to *True*.

Returns The breakpoint object, which also works as a context manager.

Return type *Breakpoint*

capture_call(*function: Union[Callable, str, None] = None*, ***, *module: Optional[module] = None*, *file: Union[str, os.PathLike, None] = None*, *unwrap: bool = True*) → nopdb.call_capture.CallCapture

Capture a function call.

The returned object can be used as a context manager, which will cause the capturing to stop at the end of the block.

If multiple calls occur, the returned object will be updated as each call returns. At the end, the returned object will contain information about the call that was the last to return.

Parameters

- **function** (*Callable or str, optional*) – A Python callable or the name of a Python function. If an instance method is passed, only calls invoked on that particular instance will be captured.

- **module** (*ModuleType, optional*) – A Python module. If given, only calls to functions defined in this module will be captured.
- **file** (*str or PathLike, optional*) – A path to a Python source file. If given, only calls to functions defined in this file will be captured. If a string is passed, it will be used as a glob-style pattern for `pathlib.PurePath.match()`. If a path-like object is passed, it will be resolved to a canonical path and checked for an exact match.
- **unwrap** (*bool, optional*) – Whether or not to unwrap *function* when it is a wrapper (e.g. produced by a decorator). Only works when *function* is given as a callable. Defaults to *True*.

Returns An instance of *CallInfo* which also works as a context manager.

Return type *CallCapture*

capture_calls (*function: Union[Callable, str, None] = None, *, module: Optional[module] = None, file: Union[str, os.PathLike, None] = None, unwrap: bool = True*) → nopdb.call_capture.CallListCapture

Capture function calls.

The return value is an initially empty list, which is updated with a new item as each call returns. At the end, the list will contain a *CallInfo* object for each call, following the order in which the calls returned.

The return value can also be used as a context manager, which will cause the capturing to stop at the end of the block.

Parameters

- **function** (*Callable or str, optional*) – A Python callable or the name of a Python function. If an instance method is passed, only calls invoked on that particular instance will be captured.
- **module** (*ModuleType, optional*) – A Python module. If given, only calls to functions defined in this module will be captured.
- **file** (*str or PathLike, optional*) – A path to a Python source file. If given, only calls to functions defined in this file will be captured. If a string is passed, it will be used as a glob-style pattern for `pathlib.PurePath.match()`. If a path-like object is passed, it will be resolved to a canonical path and checked for an exact match.
- **unwrap** (*bool, optional*) – Whether or not to unwrap *function* when it is a wrapper (e.g. produced by a decorator). Only works when *function* is given as a callable. Defaults to *True*.

Returns A list of *CallInfo* objects which also works as a context manager.

Return type *CallListCapture*

remove_callback (*handle: nopdb.common.Handle*) → None

Remove a callback added using *add_callback()*.

Parameters **handle** (*Handle*) – A handle returned by *add_callback()*.

start () → None

Start this instance.

Called automatically when the object is used as a context manager.

property started

stop () → None

Stop this instance.

Called automatically when the object is used as a context manager.

class nopdb.**Breakpoint**(*nopdb: NoPdb*, *scope: nopdb.scope.Scope*, *line: Union[int, str, None] = None*, *cond: Union[str, bytes, code, None] = None*)

Bases: nopdb.common.NoPdbContextManager

A breakpoint that executes scheduled actions when hit.

Breakpoints are typically created with *nopdb.breakpoint()*. The breakpoint object works as a context manager that removes the breakpoint on exit.

enable() → None
Enable the breakpoint again after calling *disable()*.

disable() → None
Disable (remove) the breakpoint.

debug(*debugger_cls: Type[bdb.Bdb] = <class 'pdb.Pdb'>*, ***kwargs*) → None
Schedule an interactive debugger to be entered at the breakpoint.

Parameters

- **debugger_cls** (*Type[bdb.Bdb], optional*) – The debuger class. Defaults to pdb.Pdb.
- ****kwargs** – Keyword arguments to pass to the debugger.

eval(*expression: Union[str, bytes, code], variables: Optional[Dict[str, Any]] = None*) → list
Schedule an expression to be evaluated at the breakpoint.

Parameters

- **expression** (*str, bytes or CodeType*) – A Python expression to be evaluated in the breakpoint's scope.
- **variables** (*Dict[str, Any], optional*) – External variables for the expression.

Returns An empty list that will later be filled with values of the expression.

Return type list

exec(*code: Union[str, bytes, code], variables: Optional[Dict[str, Any]] = None*) → None
Schedule some code to be executed at the breakpoint.

The code will be executed in the breakpoint's scope. Any changes to local variables (including newly defined variables) will be preserved in the local scope; note that this feature is somewhat experimental and may not work with Python implementations other than CPython and PyPy.

Parameters

- **code** (*str, bytes or CodeType*) – Python source code to be executed in the breakpoint's scope.
- **variables** (*Dict[str, Any], optional*) – External variables for the code. These may not conflict with local variables and will *not* be preserved in the local scope.

class nopdb.**Scope**(*function: Union[Callable, str, None] = None*, *module: Optional[module] = None*, *file: Union[str, os.PathLike, None] = None*, *unwrap: bool = True*)

class nopdb.**CallInfo**
Information about a function call.

name
The name of the function's code object.

Type str

file
The path to the file where the function was defined.

Type str

stack
The call stack.

Type traceback.StackSummary

args
The function's arguments.

Type dict

locals
Local variables on return.

Type dict

globals
Global variables on return.

Type dict

return_value
The return value.

print_stack(*file=None*) → None
Print the call stack.

class nopdb.**CallCapture**(*nopdb: NoPdb*, *scope: Scope*)
Bases: nopdb.call_capture.BaseCallCapture, nopdb.common.NoPdbContextManager, nopdb.call_capture.CallInfo

enable() → None
Start capturing again after calling *disable()*.

disable() → None
Stop capturing.

class nopdb.**CallListCapture**(*nopdb: NoPdb*, *scope: Scope*)
Bases: nopdb.call_capture.BaseCallCapture, nopdb.common.NoPdbContextManager, list, typing.Generic

enable() → None
Start capturing again after calling *disable()*.

disable() → None
Stop capturing.

INDEX