Debugging

All Versions

Examples

The Python Debugger: Step-through Debugging with _pdb_

The Python Standard Library includes an interactive debugging library called *pdb* . *pdb* has extensive capabilities, the most commonly used being the ability to 'step-through' a program.

To immediately enter into step-through debugging use:

```
python -m pdb <my_file.py>
```

This will start the debugger at the first line of the program.

Usually you will want to target a specific section of the code for debugging. To do this we import the pdb library and use *set_trace()* to interrupt the flow of this troubled example code.

```
import pdb

def divide(a, b):
    pdb.set_trace()
    return a/b
    # What's wrong with this? Hint: 2 != 3

print divide(1, 2)
```

Running this program will launch the interactive debugger.

```
python foo.py
> ~/scratch/foo.py(5)divide()
-> return a/b
(Pdb)
```

Often this command is used on one line so it can be commented out with a single # character

```
 import pdf; pdb.set_trace()
```

At the *(Pdb)* prompt commands can be entered. These commands can be debugger commands or python. To print variables we can use *p* from the debugger, or python's *print* .

```
(Pdb) p a
1
(Pdb) print a
1
```

To see list of all local variables use

```
locals
```

build-in function

These are good debugger commands to know:

```
b <n> | <f>: set breakpoint at line *n* or function named *f*.
# b 3
# b divide
b: show all breakpoints.
c: continue until the next breakpoint.
s: step through this line (will enter a function).
n: step over this line (jumps over a function).
r: continue until the current function returns.
l: list a window of code around this line.
p <var>: print variable named *var*.
# p x
q: quit debugger.
bt: print the traceback of the current execution call stack
up: move your scope up the function call stack to the caller of the current function
down: Move your scope back down the function call stack one level
step: Run the program until the next line of execution in the program, then return control back
next: run the program until the next line of execution in the current function, then return cont
return: run the program until the current function returns, then return control back to the debu
continue: continue running the program until the next breakpoint (or set_trace si called again)
```

The debugger can also evaluate python interactively:

```
-> return a/b
(Pdb) p a+b
```

```
(Pdb) p a+b
3
(Pdb) [ str(m) for m in [a,b]]
['1', '2']
(Pdb) [ d for d in xrange(5)]
[0, 1, 2, 3, 4]
```

Note:

If any of your variable names coincide with the debugger commands, use an exclamation mark ' **!** ' before the var to explicitly refer to the variable and not the debugger command. For example, often it might so happen that you use the variable name ' **c** ' for a counter, and you might want to print it while in the debugger. a simple ' **c** ' command would continue execution till the next breakpoint. Instead use ' **!c** ' to print the value of the variable as follows:

```
(Pdb) !c
4
```

Via IPython and ipdb

If IPython (or Jupyter) are installed, the debugger can be invoked using:

```
import ipdb
ipdb.set_trace()
```

When reached, the code will exit and print:

```
 /home/usr/ook.py(3)<module>()
      1 import ipdb
      2 ipdb.set_trace()
----> 3 print("Hello world!")

ipdb>
```

Clearly, this means that one has to edit the code. There is a simpler way:

```
from IPython.core import ultratb
sys.excepthook = ultratb.FormattedTB(mode='Verbose',
                                     color_scheme='Linux',
                                     call_pdb=1)
```

This will cause the debugger to be called if there is an uncaught exception raised.

Remote debugger

Some times you need to debug python code which is executed by another process and and in this cases rpdb comes in handy.

> rpdb is a wrapper around pdb that re-routes stdin and stdout to a socket handler. By default it opens the debugger on port 4444

Usage:

```
# In the Python file you want to debug.
import rpdb
rpdb.set_trace()
```

And then you need run this in terminal to connect to this process.

```
# Call in a terminal to see the output
$ nc 127.0.0.1 4444
```

And you will get pdb promt

```
> /home/usr/ook.py(3)<module>()
-> print("Hello world!")
(Pdb)
```

Syntax

Parameters

Remarks