# Python debugging

Reuven M. Lerner, PhD
reuven@lerner.co.il

## Debugging

```
def hello():
  lalala


>>> hello()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in hello
NameError: global name 'lalala' is not defined
```

2

## Stack traces

- Extremely useful
- Takes some time to read them, but it’s worth it
- In a complex program, the stack trace will be much deeper — start with the newest, and only work back as far as necessary

## pdb

- pdb — Python debugger
- Comes with Python
- Easy to use

4

## Using pdb

```
import wc # from exercise

import pdb

pdb.run('wc.count_words()')
```

5

| | |
|---|---|
| p | prints expression value |
| pp | prints expression with pretty-print (pprint) |
| n | go to the next line |
| p | go to the previous line |
| r | run the program to the end |
| u *num* | run until line *num* |
| j *num* | jump to line *num* |
| b | show all breakpoints |
| l | show current line of code, with context |
| h | show help |

6

## Setting breakpoints

```
b filename:linnum

b filename:linenum, 'string_to_eval'

b function

b function, 'string_to_eval'
```

## A little hint

- Inside of pdb, a leading “b” is seen as a command — so you can’t assign to b!

```
b = 123     # won’t work
```

- If you must assign to b, use a leading ! sign:

```
!b = 123    # solves the problem
```

8

## Invoke pdb from code

- Import the set_trace function from pdb:

```
from pdb import set_trace
```

- Now, whenever you invoke pdb.set_trace() in your code, you will be put into pdb

## ipdb

- Part of the IPython project
- Replaces the standard Python interpreter with IPython for pdb

## Auto-debugging

- Want to have IPython start the debugger whenever you encounter an error?  Just type

```
%pdb
```

- And ipdb will be started whenever you encounter an error!

## Manual debugging

- Meanwhile, if you encounter an error while in IPython, you can use the magic %debug command to enter the debugger where the exception happened:

```
%debug
```

## code

- Another way to start interactive Python during your code execution:

```
import code

code.interact(local=locals())
```

- Control-D (Unix) or Control-Z (Windows) then continues execution
- exit() exits from the program + debugging

## Debug on error

- sys.excepthook is the function that Python executes when there is an error
- By replacing this with a function of our choosing, we can have a program enter the debugger when an error occurs
- The function takes three arguments: The error type, the value, and the traceback

## Simple example

```
#!/usr/bin/env python
import sys, traceback, pdb
def expanded_error(type, value, tb):
    print "Custom error!"
    traceback.print_exception(type, value, tb)
    pdb.pm()
sys.excepthook = expanded_error

x = 'abc'
print x[10]
```

15

## pudb

```
pip install pudb
```

- A more graphical debugger!

```
from pudb import set_trace

set_trace()
```

- or

```
python -m pudb my-script.py
```

16

## bugjar

- Another option: bugjar!
- It uses Tk, but isn’t terribly ugly

```
pip install bugjar

bugjar test.py
```

## Frames

- Invoking sys._getframe() will give you the current execution frame
- Passing a numeric argument goes that many frames up / back in the stack
- _getframe() returns an object of type frame
- That gives you the line number, source code, globals, locals, and more

## Fun with frames

```
sys._getframe().f_code.co_name

sys._getframe().f.code.co_filename



def callersname():

  return sys._getframe(1).f_code.co_name
```

19

## __debug__

- This constant is always defined except if you invoke Python with the -O (optimize) flag
- This allows you to put debugging output in your program without risking production performance
- You cannot assign to it!

## Inspecting things

- The “inspect” module lets you examine four things:
  - Type checking
  - Getting source-related info
  - Inspecting classes and functions
  - Examining the interpreter stack

## Type checking

- inspect.getmembers() shows all members of an object

```
[('__add__', <method-wrapper '__add__' of str object at
0x1002aab70>), ('__class__', <type 'str'>), ('__contains__', <method-
wrapper '__contains__' of str object at 0x1002aab70>), ('__delattr__',
<method-wrapper '__delattr__' of str object at 0x1002aab70>),
('__doc__', 'str(object) -> string\n\nReturn a nice string representation
of the object.\nIf the argument is a string, the return value is the same
object.'), ('__eq__', <method-wrapper '__eq__' of str object at
0x1002aab70>), ('__format__', <built-in method __format__ of str
object at 0x1002aab70>), ('__ge__', <method-wrapper '__ge__' of str
object at 0x1002aab70>), ('__getattribute__',  ... )]
```

## Type checking

- Instead of looking there, you can use a number of convenience functions:
    - ismodule
    - isclass
    - isroutine   # any kind of function or method
    - isfunction  # user-defined function

## Logging

- Instead of print, you can log debugging information to STDOUT
- The “logging” module gives you amazing flexibility on this front

## Simple logging

```
import logging

logging.basicConfig(level=logging.DEBUG)

logging.debug('hello, log')
```

## Logging options

- logging.basicConfig() takes many options:
- filename: Write to this file
- filemode: Mode with which to open file
- format: Python format string for writing
- dateformat: Date/time format
- level: Minimum level to log

26

## Logging functions

```
logging.critical()

logging.error()

logging.warning()

logging.info()

logging.debug()
```

## The source code

- You can use inspect on any object to learn more about it
- All are get* functions, all listed online

```
getcallargs(f)

getclasstree(f)
```

28

## Inspecting existing objects

- inspect lets you look at existing objects
- Use the right name that people use

29

## memory_profiler

```
pip install memory_profiler
```

- Use the @profile decorator on any function whose memory usage you wish to profile
- Now invoke the program with -m memory_profiler:

```
python -m memory_profiler program.py
```

- When your function is invoked, you'll see its memory footprint!

## Sample file

```
#!/usr/bin/env python

@profile

def get_etc_filename_lengths():

    import os

    files = os.listdir('/etc/')


    for filename in files:

        print len(filename),


get_etc_filename_lengths()
```

31

## Profiling the memory use

```
Line #    Mem usage    Increment   Line Contents
================================================
     3   11.023 MiB    0.000 MiB   @profile
     4                             def get_etc_filename_lengths():
     5   11.027 MiB    0.004 MiB       import os
     6   11.031 MiB    0.004 MiB       files = os.listdir('/etc/')
     7
     8   11.031 MiB    0.000 MiB       for filename in files:
     9   11.031 MiB    0.000 MiB           print len(filename),
```

## Useful tools

- pep8
- pyflakes, pylint
- tabnanny
- trace
- timeit
- profile

## Optimizing

- http://wiki.python.org/moin/PythonSpeed/PerformanceTips

## Python style guide

- Indentation: Use 4 spaces!
  - Never mix tabs and spaces
- Limit line length to 79 characters
- Imports should be on separate lines
  - So don’t “import os, sys”
- PEP8