02-BreakPoints

February 10, 2015

Contents

0.1 Debugging

- Debugging is the **process of finding errors** in a program
- very important part of programming
- can take considerible time
- mastering debugging techniques can save you a lot of time

0.1.1 Literature

- **Python docs on errors and exceptions**
  - http://docs.python.org/2/tutorial/errors.html

0.1.2 Overview

1. Introduction/ Motivation
2. Debugging in with IPython `%debug`
3. Exercise
4. **Debugging with break-points**
5. **Exercise**
6. Debugging with the logger module
7. Exercise
8. Nose tests
9. Exercise

0.1.3 Debugging with break-points

Break points are a traditional method of inspecting how a program works. **Break points are points in a program where the debugger stops each time the program flow passes these points.** If the program is not run with a debugger, break-points are ignored. It can be used to understand how data is changed as the program runs. It is an essential tool in compiled languages, because you cannot manipulate data live in these languages, only view them.

In many programming languages break-points are handled by the IDE and a single click next to the line number in the editor switches them on or off.

`ipdb` module

In the following examples I am going to use the Ipython debugging module, which wraps the orignal `pdb` commands (i.e. uses the same commands as the standard `pdb`). To install `ipdb` just type

```
pip install ipdb
```

in your terminal. If that fails for some reason, you can change the `import` line below to

```
import pdb
```

And you should be able to get the same things done. However, the `ipdb` provides nice syntax highlighting and generally a nicer view of the stack trace.

`set_trace()`

In python, break-points are defined with the `set_trace()` function:
Unfortunately the ipdb is broken in the IPython notebook at the moment. Therefore I am showing you this in the terminal. The following code is saved out as `scripts/01.py`. Open a terminal and run it as
`python scripts/01.py`.

```
# scripts/01.py
import ipdb as pdb

def functionWithBreakpoint():
    number = 4
    number += 3
    pdb.set_trace()
    number += 2
    return number

print functionWithBreakpoint()
```

We can play a little bit around with it and try out the commands we know already from the `%debug` magic we introduced before.

Commands

(courtesy of http://georgejhunt.com/olpc/pydebug/pydebug/ipdb.html)

- s(tep) – Execute the current line, stop at the first possible occasion (either in a function that is called or in the current function).
- n(ext) – Continue execution until the next line in the current function is reached or it returns.
- unt(il) – Continue execution until the line with a number greater than the current one is reached or until the current frame returns.
- r(eturn) – Continue execution until the current function returns.
- c(ont(inue)) – Continue execution, only stop when a breakpoint is encountered.

Stepping though several break-points

```
# scripts/02.py
import ipdb as pdb

def functionWithBreakpoint():
    number = 0
    for i in range(10):
        number += 2
        pdb.set_trace()

        numnber -= 1
        pdb.set_trace()

    return number

print functionWithBreakpoint()
```

```
In [ ]:
```