Python programming — Debugging

Finn Årup Nielsen

DTU Compute
Technical University of Denmark

October 30, 2013

Overview

print, pprint

logging

pdb: The Python Debugger

Spyder

PuDB

Introspection with Paul Butler's debugging decorator

Before major debugging

Write **unit tests**: Consider test-driven development where you first write the test code and then implement the functionality.

Run **pylint**: This will check you code for style and perhaps discover "real" bugs

Print

`print`: While ok in development `print` statements should usually not occure in the finished code whether executed or not (comment out).

For nested structures, such as dictionaries within lists within dictionaries consider `pprint` (note the extra "p")

```
import pprint, requests

response = requests.get("https://ofirehose.com/feed.json").json()
pprint.pprint(response["items"][0])
```

This gives you a better indentation of the nested structure.

logging module

Set different level of logging messages: DEBUG, INFO, WARNING, ERROR

Useful for, e.g., for a web application that never should error, but always return something useful.

Consistent formatting with timing information

Setting of the logging level

Redirection of the logging output: standard error, log files.

Simple logging example

```
import logging, requests

logging.debug("Requesting feeds from ofirehose")
try:
    response = requests.get("httpps://ofirehose.com/feed.json").json()
    feeds = response["items"]
except Exception as e:
    feeds = []
    logging.warn("Could not download feeds from ofirehose: " + str(e))

for feed in feeds: print(feed['content'])
```

This will lead to a logging message from the "warn" call

```
WARNING:root:Could not download feeds from ofirehose: No connection ...
```

More elaborate logging example . . .

Setting up logfile, format and logging level:

```
import logging, os.path

logger = logging.getLogger("openfeed")              # Name of logger
filename = os.path.expanduser("~/openfeed.log")     # log file
hdlr = logging.FileHandler(filename)
formatter = logging.Formatter("%(asctime)s %(levelname)s %(message)s")
hdlr.setFormatter(formatter)                # Format for each log line
logger.addHandler(hdlr)
logger.propagate = False                    # No stderr output
logger.setLevel(logging.DEBUG)              # Changing log level
logger.info("Logging setup")                # Logging that the log is setup
```

Now the `logger` object is setup that we can use for logging:

. . . More elaborate logging example

```
logger.debug("Requesting feeds from ofirehose")
try:
    response = requests.get("httpps://ofirehose.com/feed.json").json()
    feeds = response['items']
except Exception as e:
    feeds = []
    logger.warn("Could not download feeds from ofirehose: " + str(e))

for feed in feeds: print(feed['content'])
```

The logfile `openfeed.log` now contains:

```
2013-10-02 16:20:39,604 INFO Logging setup
2013-10-02 16:21:01,034 DEBUG Requesting feeds from ofirehose
2013-10-02 16:21:01,054 WARNING Could not download feeds from ofireho ...
```

Logging in modules . . .

In the module `submodule.py`:

```
import logging
from logging import NullHandler

log = logging.getLogger(__name__)      # The log gets the name of the module
log.addHandler(NullHandler())          # Avoids "No handlers" message if no logger

def some_function():
    log.debug("In some_function()")    # A log message to the module log
    return "Hello, World"
```

In importing module `usermodule.py` for example:

```
import submodule

log = logging.getLogger()              # This includes the submodule logger too
log.setLevel(logging.DEBUG)
handler = logging.StreamHandler()
handler.setFormatter(logging.Formatter('%(asctime)s %(levelname)s %(name)s: %(message)s'))
log.addHandler(handler)

submodule.some_function()
```

. . . Logging in modules

How to make it shut up:

With no logger:

```
import submodule

submodule.some_function()
```

Or by adjusting the logging level:

```
import submodule

log = logging.getLogger()                 # This includes the submodule logger too
log.setLevel(logging.WARNING)
handler = logging.StreamHandler()
handler.setFormatter(logging.Formatter('%(asctime)s %(levelname)s %(name)s: %(message)s'))
log.addHandler(handler)

submodule.some_function()
```

Python debugger: pdb

The Python Debugger is a module, `pdb`, for interactive code debugging

The perhaps most simple usages is to insert a breakpoint:

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

When reached, the debugger is started with the prompt “`(Pdb)`”

Pdb command examples

help/h: Displays the list of commands

step/s: Single step the program, step into functions.

next/n: Single step in the current function

pp/p: Pretty printing/printing a variable

cont/c: Continue execution

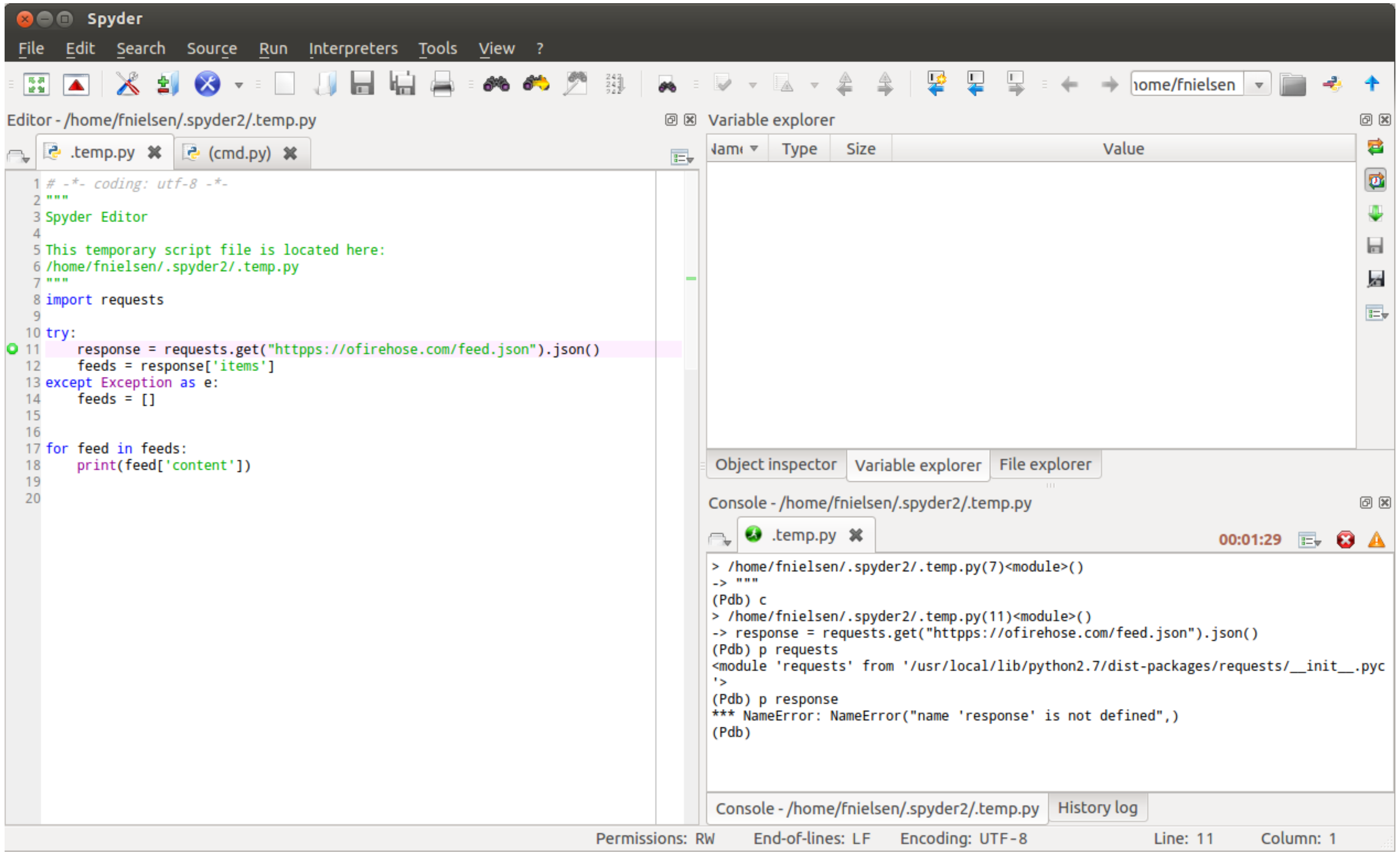
quit/q: Quit the debugger

. . .

pdb example

```
import requests
import pdb

try:
    pdb.set_trace()
    response = requests.get("httpps://ofirehose.com/feed.json").json()
    feeds = response['items']
except Exception as e:
    feeds = []

(Pdb) n
InvalidSchema: InvalidS....json'",)
> <stdin>(3)<module>()
(Pdb) pp requests.get("httpps://ofirehose.com/feed.json")
*** InvalidSchema: InvalidSchema("No connection adapters were fou ...
```

pdb in Spyder



```python
# -*- coding: utf-8 -*-
"""
Spyder Editor

This temporary script file is located here:
/home/fnielsen/.spyder2/.temp.py
"""
import requests

try:
    response = requests.get("https://ofirehose.com/feed.json").json()
    feeds = response['items']
except Exception as e:
    feeds = []


for feed in feeds:
    print(feed['content'])

```

```
> /home/fnielsen/.spyder2/.temp.py(7)<module>()
-> """
(Pdb) c
> /home/fnielsen/.spyder2/.temp.py(11)<module>()
-> response = requests.get("https://ofirehose.com/feed.json").json()
(Pdb) p requests
<module 'requests' from '/usr/local/lib/python2.7/dist-packages/requests/__init__.pyc
'>
(Pdb) p response
*** NameError: NameError("name 'response' is not defined",)
(Pdb)
```

pdb is available in Spyder. Breakpoints may be added with the mouse or keyboard (F12).

PuDB

PuDB, a console-based Python debugger.

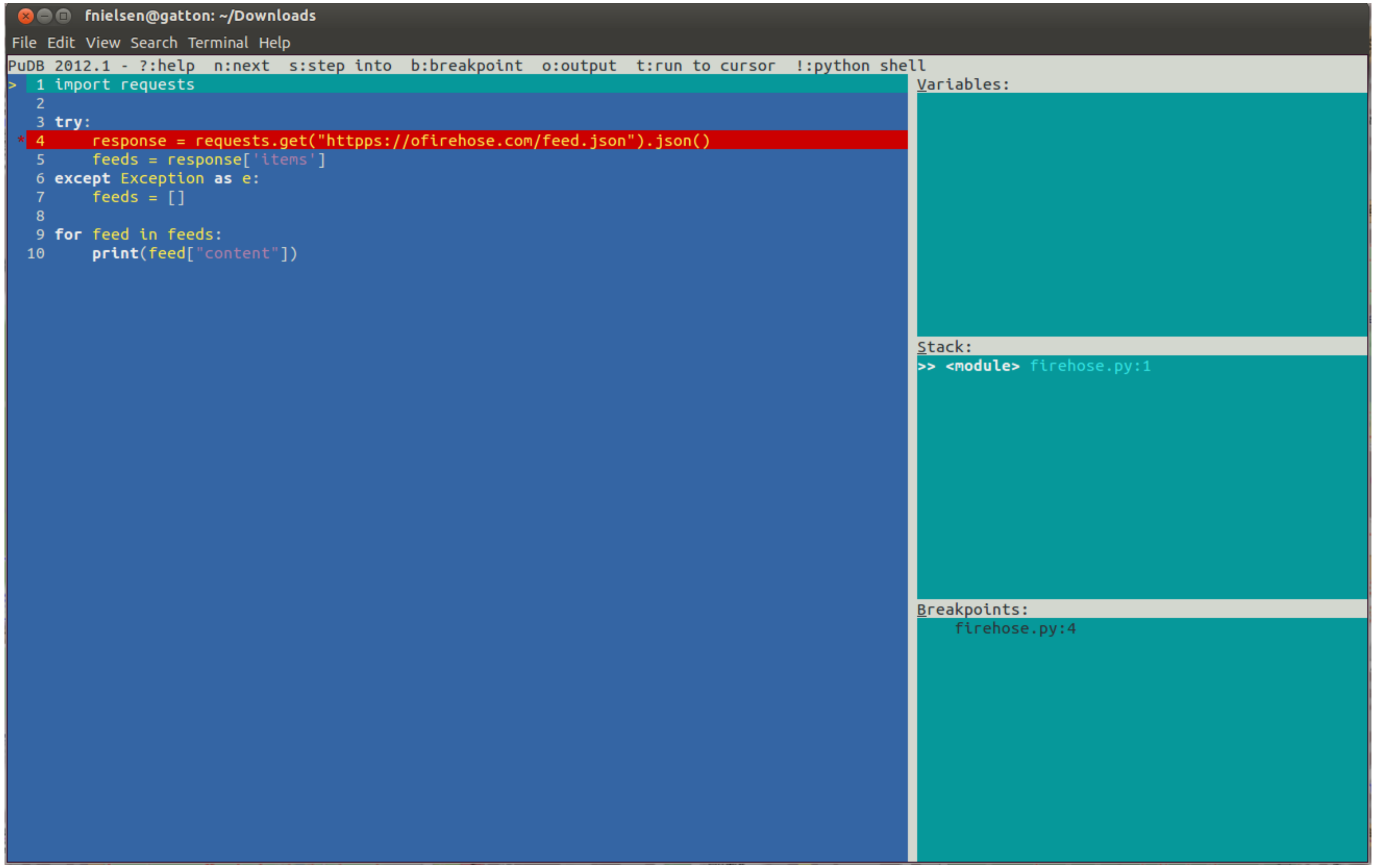
Consider the file `firehose.py`

```
import requests

try:
    response = requests.get("httpps://ofirehose.com/feed.json").json()
    feeds = response['items']
except Exception as e:
    feeds = []

for feed in feeds:
    print(feed["content"])
```

Run pudb on the file:

```
$ pudb firehose.py
```

PuDB


fnielsen@gatton: ~/Downloads
File Edit View Search Terminal Help

```
PuDB 2012.1 - ?:help  n:next  s:step into  b:breakpoint  o:output  t:run to cursor  !:python shell
>  import requests

   try:
*      response = requests.get("httpps://ofirehose.com/feed.json").json()
       feeds = response['items']
   except Exception as e:
       feeds = []

   for feed in feeds:
       print(feed["content"])
```

```
Variables:

Stack:
>> <module> firehose.py:1

Breakpoints:
    firehose.py:4
```

Regular expression debugging

```
import re
re.compile("...[here goes a complicated regular expression]", re.DEBUG)

re.compile(r"(?:-)*(?:\d{1,3}(?:,\d{3})*(?:\.\d*)?|\d+(?:\.\d*)?)",
           flags=re.DEBUG)
```

Will perhaps(?) give a better overview of the regular expression.

Python debugging

```
max_repeat 0 65535
  subpattern None
    literal 45
subpattern None
  branch
    max_repeat 1 3
      in
        category category_digit
    max_repeat 0 65535
      subpattern None
        literal 44
        max_repeat 3 3
          in
            category category_digit
    max_repeat 0 1
      subpattern None
        literal 46
        max_repeat 0 65535
          in
            category category_digit
  or
    max_repeat 1 65535
      in
        category category_digit
    max_repeat 0 1
      subpattern None
        literal 46
        max_repeat 0 65535
          in
            category category_digit
```

Traceback and logging . . .

Catching uncaught exceptions in the log with traceback.

First, troublesome example code (where is the bug(s)), — a simple daemon for monitoring file lengths in a directory:

```
import os

def monitor_lengths(dirname="."):
    lengths = {}
    while True:
        for filename in os.listdir(dirname):
            filename = os.path.join(dirname, filename)
            if os.path.isfile(filename):
                length = len(open(filename).read())
                if filename in lengths:
                    if lengths[filename] != length:
                        print(filename)
                else:
                    lengths[filename] = length
```

. . . Traceback and logging

Exception traceback modified from the cookbook (Martelli et al., 2005, section 8.4) with output to a log.

```
import logging
import cStringIO
import traceback

try:
    monitor_lengths()
except Exception as e:
    f = cStringIO.StringIO()
    traceback.print_exc(file=f)
    msg = f.getvalue().replace("\n", "\\\\")   # On one line
    logging.critical(msg)
```

Paul Butler’s debugging decorator

Paul Butler debugging decorator to decorate a function that misbehaves

```
def report(function):
    def wrap(*args, **kwargs):
        wrap.call_count += 1
        indent = '   ' * report._indent
        fc = "{}({})".format(function.__name__,
                             ", ".join(map(str, args) +
                             map(lambda (k, v): "{}={}".format(k, v), kwargs.items())))
        print("{}{} called #{}".format(indent, fc, wrap.call_count) )
        report._indent += 1
        return_value = function(*args, **kwargs)
        report._indent -= 1
        print("{}{} returned with value {}".format(indent, fc, str(return_value)) )
        return return_value
    wrap.call_count = 0
    return wrap

report._indent = 0
```

Now the `@report` decorator can be applied on functions:

. . . Paul Butler-like debugging decorator

Decorating the troublesome function with Paul Butler's debugging decorator:

```
@report
def not_really_fibonacci(n, dummy=0):
    if n in [0, 1, 2]:
        return n
    else:
        return not_really_fibonacci(n-1, dummy) + not_really_fibonacci(n-3, dummy)
```

Run the program:

```
>>> not_really_fibonacci(4, dummy=84)
not_really_fibonacci(4, dummy=84) called #1
  not_really_fibonacci(3, 84) called #2
    not_really_fibonacci(2, 84) called #3
    not_really_fibonacci(2, 84) returned with value 2
    not_really_fibonacci(0, 84) called #4
    not_really_fibonacci(0, 84) returned with value 0
  not_really_fibonacci(3, 84) returned with value 2
  not_really_fibonacci(1, 84) called #5
  not_really_fibonacci(1, 84) returned with value 1
not_really_fibonacci(4, dummy=84) returned with value 3
3
```

More information

Andrew Dalke, Tracing python code: The use of `sys.settrace` and `linecache` for printing executed lines.

Summary

`print` should rarely appear in a finished program

Better to use logging module

Pdb is the Python debugger with a simple command-line interface.

Pdb functionality is available in Spyder and in Pudb (and likely other IDE)

Python is a programming language with introspection: You can trace the program and, e.g., query the function name.

References

Martelli, A., Ravenscroft, A. M., and Ascher, D., editors (2005). *Python Cookbook*. O'Reilly, Sebastopol, California, 2nd edition.