# Debugging in Python

Kevin Rue-Albrecht
Oxford Biomedical Data Science Training Programme

2020-05-04 (updated: 2020-05-04)

## Debugging

### What is it?

Finding and reducing the number of defects in a computer program.

### Errors

| Error | Effect | Challenge |
|---|---|---|
| Syntax | Code is formally correct, but does not mean what you intended. | Easy |
| Semantic | Code is meaningless. | + |
| Algorithmic | Code is formally and semantically correct, but is the wrong solution for the problem. | ++ |
| Complex | Code is correct and provides correct solution, but is impacted by system effects (e.g., memory allocation, concurrency, dependencies). | Hard |

## Why is it hard?

### Example

```python
piggybank = 12

piggybank += 10

piggybank = 'empty' # Cause of the error

piggybank *= 2      # Error without consequence

piggybank += 10     # Error finally causes crash
```

### Error

```
Traceback (most recent call last):

  File "<ipython-input-1-c21e790c53a4>", line 9, in <module>
    piggybank += 10

TypeError: can only concatenate str (not "int") to str
```

## Coding techniques to avoid bugs

- Write legible code
  - Make it easy to read: quicker to understand and quicker to find a bug
  - Variable and function naming
- Write tests
  - Check if what you have written is correct
- Write incrementally
  - Write a block of code, test, then proceed
- Reuse code
  - Use functions; code needs only be debugged once
  - Even for trivial things

## Coding techniques to avoid bugs

- Write documentation in the code (comments)
  - Make it easy to spot if intention is different from outcome
- Avoid syntax errors
  - There are tools for this: lint, pyflakes, ...
- Use assertions to check assumptions
  - Catch problems early
- Use extensive logging
  - Tell what the script is about to do, what it just did
  - Check intermediate results, report unusual properties

## Debugging is easier with logging

- Logging helps to understand
  - When/where a failure occurs
  - What the state of the program was when the failure occurred
- Always use the logging facility
  - Do *not* use print statements
  - Logging formats messages consistently
  - Logging can be redirected (file, server, web)
  - Verbosity of logging can be controlled by the user

## Loggign is easy

```python
import logging as L

L.basicConfig(level=L.DEBUG)

L.info("reading data")

data = [list(range(10)),
        list(range(20)),
        list(range(8))]

L.info(f'processing {len(data)} data sets')

min_mean = 4

for idx, d in enumerate(data):
    L.debug(f'iteration {idx+1}')
    mean = sum(d) / len(d)
    L.info(f'mean for data set {idx+1}: {mean}')
    if mean < min_mean:
        L.warn(
        f'mean less than expected: {mean} < {min_mean}')

L.info(f'processed {len(data)} data sets')
```

```
INFO:root:reading data
INFO:root:processing 3 data sets
DEBUG:root:iteration 1
INFO:root:mean for data set 1: 4.5
DEBUG:root:iteration 2
INFO:root:mean for data set 2: 9.5
DEBUG:root:iteration 3
INFO:root:mean for data set 3: 3.5
logging2.py:21: DeprecationWarning: The 'warn' function is deprecated, use 'w
  f'mean less than expected: {mean} < {min_mean}')
WARNING:root:mean less than expected: 3.5 < 4
INFO:root:processed 3 data sets
```

### Python logging module

- Part of the standard library
- Very useful
- Very flexible
- Very easy to use

### Good logging practice

- Output what you are going to do
- Output what you have done
- Output anything that is unexpected
- Output anything that is of interest

# Program failures

```
################################################## User code ##################################################
python CGAT/scripts/bed2graph.py  --help
Traceback (most recent call last):
  File "CGAT/scripts/bed2graph.py", line 91, in <module>
    sys.exit(main(sys.argv))
  File "CGAT/scripts/bed2graph.py", line 54, in main
    (options, args) = E.Start(parser, argv=argv)
  File "/Users/andreas/devel/cgat/CGAT/Experiment.py", line 868, in Start
    (global_options, global_args) = parser.parse_args(argv[1:])
```

```
################################################ 3rd party code ################################################
############################################## Python stack trace ##############################################
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 1388, in parse_args
    stop = self._process_args(largs, rargs, values)
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 1428, in _process_args
    self._process_long_opt(rargs, values)
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 1502, in _process_long_opt
    option.process(opt, value, values, self)
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 786, in process
    self.action, self.dest, opt, value, values, parser)
  File "/Users/andreas/devel/cgat/CGAT/Experiment.py", line 441, in take_action
    self, action, dest, opt, value, values, parser)
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 808, in take_action
    parser.print_help()
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 1648, in print_help
    file.write(self.format_help())
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 1636, in format_help
    result.append(self.format_option_help(formatter))
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 1611, in format_option_help
    formatter.store_option_strings(self)
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 333, in store_option_strings
    self.indent()
  File "/Users/andreas/devel/cgat-install/conda-install/envs/cgat-s/lib/python3.6/optparse.py", line 247, in indent
    raise ValueError('debug error')
ValueError: debug error
```

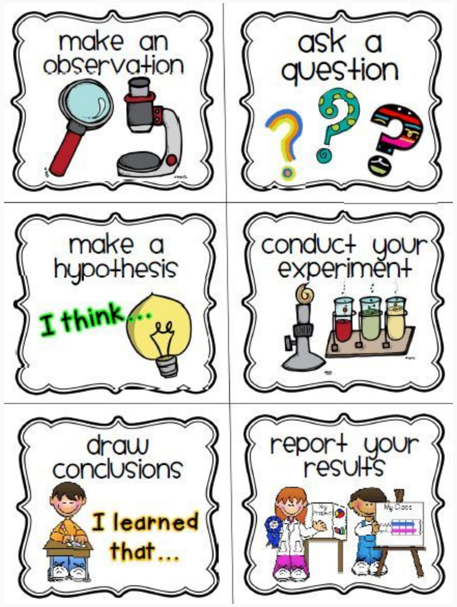
## But very often

```
# output generated by CGAT/scripts/bed2graph.py tests/bed2graph.py/srf.hg19.bed.gz tests/bed2graph.py/srf.hg19.bed.gz
# job started at Mon Sep 18 09:53:06 2017 on MRCs-MacBook-Pro.local -- ff97abb0-2dae-4591-8389-490159770956
# pid: 12117, system: Darwin 16.7.0 Darwin Kernel Version 16.7.0: Thu Jun 15 17:36:27 PDT 2017; root:xnu-3789.70.16~2/RELEASE_
# loglevel                                  : 1
# output                                    : full
# random_seed                               : None
# short_help                                : None
# stderr                                    : <_io.TextIOWrapper name='<stderr>' mode='w' encoding='US-ASCII'>
# stdin                                     : <_io.TextIOWrapper name='<stdin>' mode='r' encoding='US-ASCII'>
# stdlog                                    : <_io.TextIOWrapper name='<stdout>' mode='w' encoding='US-ASCII'>
# stdout                                    : <_io.TextIOWrapper name='<stdout>' mode='w' encoding='US-ASCII'>
# timeit_file                               : None
# timeit_header                             : None
# timeit_name                               : all
<NO OUTPUT HERE>
# job finished in 0 seconds at Mon Sep 18 09:53:06 2017 --  0.20  0.06  0.00  0.00 -- ff97abb0-2dae-4591-8389-490159770956
```

# Debugging techniques

## The Zen Approach

## The Scientific Approach



Debugging costs time – you want to fix bugs quickly.

## The Zen of Debugging

**Introspection**

- Look at the code location where a bug occurs
- Think about possible causes (experience helps)
- Catches semantic bugs

**Understanding**

- Read the documentation
- Read the code
- Understand intention and organization of the code

**Observation**

- Observe the code in action
- Print statements, assertions
- Logging messages
- Use a debugger

A Bug's life can last for:

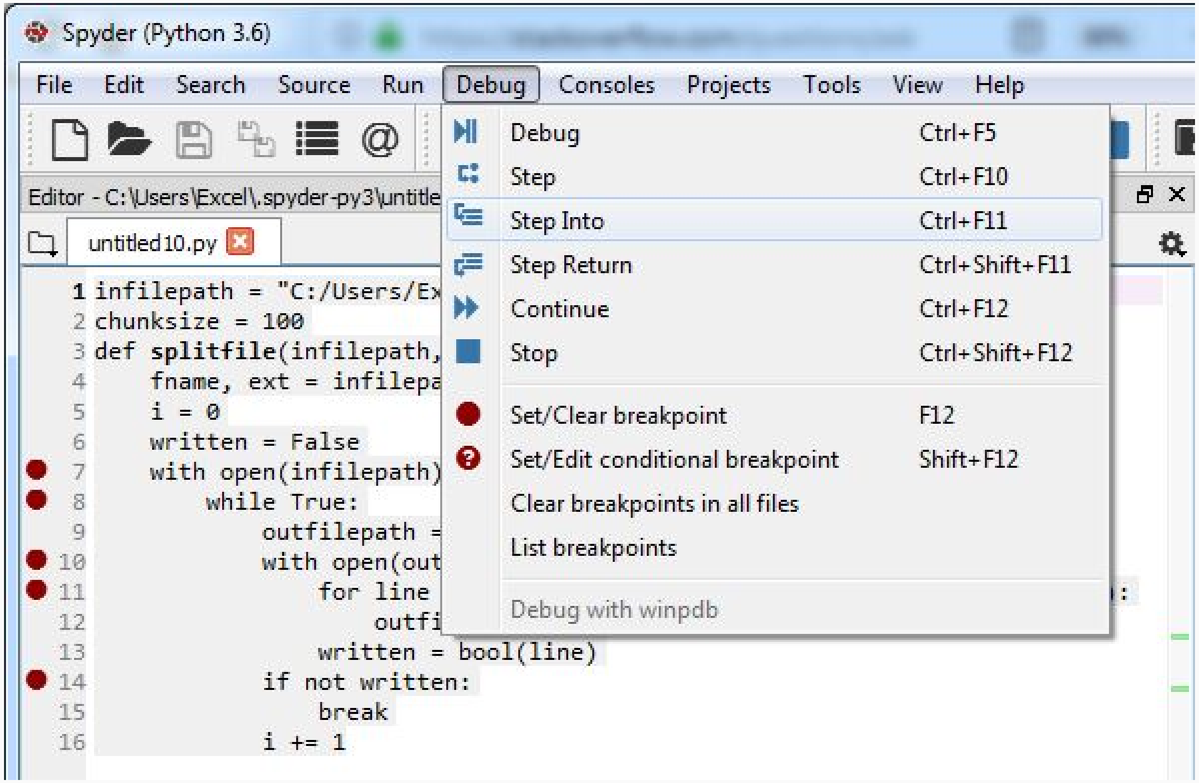
- Minutes
- Hours
- Days
- Weeks
- ... forever

Working around is an option.

## Using a debugger

- Can step through the code line by line
- Can chose whether to step through functions
- Can run the code up to a specific location - breakpoint
    - Breakpoints can be activated/deactivated
    - Breakpoints can be conditional
- Can view values of all variables at each step
- When debugging, programs run slower

# Debugging in Spyder



Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\Users\Excel\.spyder-py3\untitle
untitled10.py

| | |
|---|---|
| Debug | Ctrl+F5 |
| Step | Ctrl+F10 |
| Step Into | Ctrl+F11 |
| Step Return | Ctrl+Shift+F11 |
| Continue | Ctrl+F12 |
| Stop | Ctrl+Shift+F12 |
| Set/Clear breakpoint | F12 |
| Set/Edit conditional breakpoint | Shift+F12 |
| Clear breakpoints in all files | |
| List breakpoints | |
| Debug with winpdb | |

```
infilepath = "C:/Users/Ex
chunksize = 100
def splitfile(infilepath,
    fname, ext = infilepa
    i = 0
    written = False
    with open(infilepath)
        while True:
            outfilepath =
            with open(out
                for line
                    outfi
                written = bool(line)
            if not written:
                break
            i += 1
```

## The Scientific Method

Useful if bug only appears under certain conditions.

- with some input data, but not all
- when running on a certain machine
- when memory is low
- when running in a multi-threaded version

Aim: Use an experimental approach to make a bug reproducible

## Making a bug reproducible

### Bug depends on input data

- Split data set into smaller and smaller chunks

### Bug depends on conditions

- Run program under controlled conditions
- Dedicated machine
- Clean operating system
- Minimum system/external libraries

## Reporting bugs in 3rd party software

- Raise an issue on GitHub
- Document the bug to enable reproduction
    - Input data
    - Command line options
    - Expected output
    - Actual output
    - Environment (how installed, versions, ...)
    - Ideally: a small test-data set
- For extra karma: offer a fix (pull request)

## Profiling

The code works, but is slow or uses too many resources.

- Core dump
- Performance measurement
    - Time
    - Memory
    - I/O
- The profile package is part of the Python standard library

## Job stats from the shell

```
$ /usr/bin/time -v python script.py
      Command being timed: "python script.py"
      User time (seconds): 0.01                           # How long your code took
      System time (seconds): 0.00                         # How long system code took
      Percent of CPU this job got: 73%
      Elapsed (wall clock) time (h:mm:ss or m:ss): 0:00.02 # How long it took in real time
      Average shared text size (kbytes): 0
      Average unshared data size (kbytes): 0
      Average stack size (kbytes): 0
      Average total size (kbytes): 0
      Maximum resident set size (kbytes): 5544            # Maximum memory consumed
      Average resident set size (kbytes): 0
      Major (requiring I/O) page faults: 0
      Minor (reclaiming a frame) page faults: 1473
      Voluntary context switches: 27
      Involuntary context switches: 2
      Swaps: 0
      File system inputs: 0
      File system outputs: 0
      Socket messages sent: 0
      Socket messages received: 0
      Signals delivered: 0
      Page size (bytes): 4096
      Exit status: 0
```

**If %CPU is low:** Are you reading/writing a lot of data? Are you waiting to download data from a service?

## Debugging Summary

- Bugs will happen
  - Use good coding practices to minimise bugs and make them easier to spot
  - Logging makes debugging easier
- Debugging is essential
  - Easy to do using the Spyder debugger and variable explorer
  - Can take a long time
  - May need to work around or even give up
- Profiling can help you to optimise code
  - If your code works but is running slowly
  - If it uses too many resources

## Exercise

Run the Spyder debugger on the following code.

```python
items = [1, 2, 3, 'test', 4]

for i in range(len(items)):
    item = items[i]
    value = item // 2
```