Tips for Debugging Your Python Code

Frank Burkholder

Objectives

- Define debugging
  - Talk will focus on interactive debugging using a debugger
- Step 0: Write code that is eas(ier) to debug
  - Motivating example - Bubblesort
- Step 1: Demonstrate **pdb** - the Python Standard Library debugger
- List common **pdb** commands

Debugging code

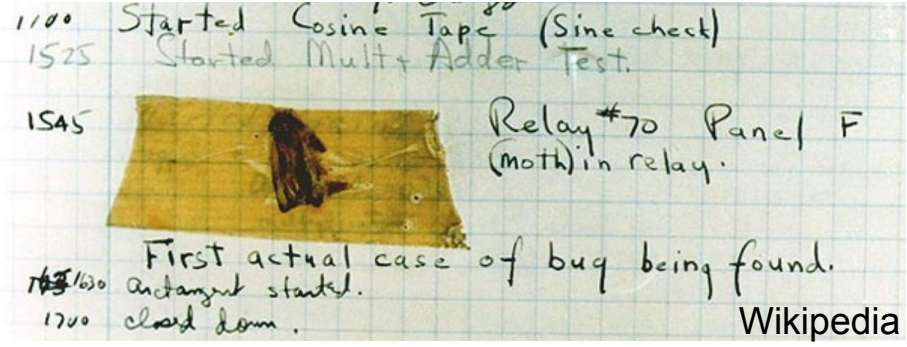
Debugging is the process of finding problems within a computer program that prevent correct operation.

The terms "bug" and "debugging" are popularly attributed to Admiral Grace Hopper in the 1940s. While working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system.

M. Hopper says she did not coin the term; “debug” had been used in aeronautics previously.

-- paraphrased from Wikipedia

Rear Admiral (then Commodore) Grace M. Hopper, 1984



Wikipedia

Common ways to debug code

Debugging tactics can involve:

- **print debugging** (a.k.a trace debugging). TRON (Trace On) - use print statements to show items under investigation
- **interactive debugging** - use a debugger during program execution to inspect values
- **unit testing** - writing tests that check for proper outputs of classes and functions
- **profiling** - quantifies space and time complexity of a program

-- paraphrased from Wikipedia

This talk: interactive debugging

Debugging tactics can involve:

- **print debugging** (a.k.a trace debugging). TRON (Trace On) - use print statements to show items under investigation
- **interactive debugging** - use a debugger during program execution to inspect values (**pdb** - the Python Standard Library debugger)
- **unit testing** - writing tests that check for proper outputs of classes and functions
- **profiling** - quantifies space and time complexity of a program

-- paraphrased from Wikipedia

Step 0: Making your code easier to debug

Coding/debugging project: Bubblesort algorithm

Goal: sort a list from low to high

Strategy: on each pass the through the list, “bubble” the biggest number to the “top” (right) as you iterate through each pair of list items

Initial: [3, 2, 4, 1]

Pass 1: [**3, 2**, 4, 1] -> [**2, 3**, 4, 1]

[2, **3, 4**, 1] -> [2, **3, 4,** 1]

[2, 3**, 4, 1**] -> [2, 3, **1, 4**]

Pass 2: [**2, 3**, 1, 4] -> [**2, 3**, 1, 4]

[2, **3, 1**, 4] -> [2, **1, 3,** 4]

Pass 3: [**2, 1**, 3, 4] -> [**1, 2**, 3, 4]

Bubblesort pseudocode

```
def bubblesort(list):
    num_el = len(list)
    num_passes = num_el - 1
    for pass in range(num_passes):
        for i in range(num_el - 1 - pass):
            if list[i] > list[i+1]:
                swap(list[i], list[i+1])
```

Easier to debug?

Option 1:

```
def bubblesort(lst):
    [lst.append(lst.pop(0) if i == len(lst) - 1 or lst[0] < lst[1] else
     lst.pop(1)) for j in range(0, len(lst)) for i in range(0, len(lst))]
    return lst
```

source: https://stackoverflow.com/questions/57350378/bubble-sorting-with-one-list-comprehension

Easier to debug?

Option 2:

```
def swap(lst, i):
    tmp = lst[i]
    lst[i] = lst[i+1]
    lst[i+1] = tmp

def bubblesort(lst):
    '''Bubblesort, more sparse and easier to debug'''
    num_el = len(lst)
    num_passes = num_el - 1
    for p in range(num_passes):
        for i in range(num_el - 1 - p):
            if lst[i] > lst[i+1]:
                swap(lst, i)
    return lst
```

Remember Zen of Python

Option 2 was easier to debug. If you write Python with the Zen of Python in mind, your code will be easier to debug, too.

From the Zen of Python:

…

```
Sparse is better than dense.
Readability counts.
```

…

Step 1: interactive debugging with **pdb**

pdb, the Python Standard Library debugger

- **pdb** - **P**ython **D**e**b**ugger
- Part of the Python Standard Library
  - It's always available
- Can be executed via the command line, or by insertion of `breakpoint()` into your script.
  - Does not depend on an IDE, always available to you
- Allows interactive debugging
  - As the program executes, it will stop at a `breakpoint()` and allow you to see the value of variables, see where you are in the stack (the sequence of function calls), and allow you to continue execution line-by-line, to another `breakpoint()`, or continue execution outside of the debugger.

How to start & use pdb, the interactive debugger

Step 0: Find a place in your code where you'd like to start tracing execution (usually just before you have a problem), and type `breakpoint()`.

Before:

```
def bubblesort(lst):
    '''Bubblesort, more sparse and easie
    num_el = len(lst)
    num_passes = num_el - 1
    for p in range(num_passes):
        for i in range(num_el - 1 - p):
            if lst[i] > lst[i+1]:
                swap(lst, i)
    return lst
```

Insert `breakpoint()`:

```
def bubblesort(lst):
    '''Bubblesort, more sparse and easier to debug'''
    breakpoint()
    num_el = len(lst)
    num_passes = num_el - 1
    for p in range(num_passes):
        for i in range(num_el - 1 - p):
            if lst[i] > lst[i+1]:
                swap(lst, i)
    return lst
```

Step 1: Execute the script, e.g. `$ python bubblesort.py`

Step 2: Use pdb commands to query/navigate

Try it for yourself

0: Add a `breakpoint()` to bubblesort.py
1: Execute the script `$ python bubblesort.py`

Common pdb commands

command	description
p	Print the value of an expression.
pp	Pretty-print the value of an expression.
n	Continue execution until the next line in the current function is reached or it returns.
s	Step into a function/class
c	Continue execution and only stop when a breakpoint is encountered.
unt	Continue execution until the line with a number greater than the current one is reached. With a line number argument, continue execution until a line with a number greater or equal to that is reached.
l	List source code for the current file. Without arguments, list 11 lines around the current line or continue the previous listing.
ll	List the whole source code for the current function or frame.

command	description
b	With no arguments, list all breaks. With a line number argument, set a breakpoint at this line in the current file
w	Print a stack trace, with the most recent frame at the bottom. An arrow indicates the current frame.
u	Move the current frame count (default one) levels up in the stack trace (to an older frame).
d	Move the current frame count (default one) levels down in the stack trace (to a newer frame).
h	See a list of available commands.
h pdb	Show the full pdb documentation.
q	Quit the debugger and exit.
!	Will override a command to see a variable value. For example, to see the value of variable n instead of n for next: !n

Reference

- Real Python: Python Debugging with pdb
- Documentation, Python Debugger