Lab 3: Debugging

Ahmed Eldeeb

2023-08-21

Table of contents

Debugging

Today we're going to explore the concept of debugging and some of the tooling that allows for debugging python code.

It's widely recognized and accepted that any sizeable code base will have a non-trivial number of bugs (where does the term come from?). The main goal of testing is to make sure the main expected cases are behaving correctly.

Sometime you code doesn't do what you expect or want it to do, and it's not clear just by reading through it, what the problem is.

In interpreted languages (esp. those with interactive shells like python) one can sometimes run the code piece by piece and inspect the state of the variables.

If the code involves a loop or a function, a common practice is to judiciously place a few print statements that dump the state of some variables to the terminal so that you can spot the problem by tracing through it.

When the code involves multiple functions and complex state, this strategy starts to break down. This is where you start rolling up your sleeves and invoking a debugger!

Debugging can be a slow process, so you typically start a debugging session by deciding which line in your code you would like to start tracing the behavior from, and you place a breakpoint. Then you can have the debugger run the program up to that point and stop at it, allowing you to:

1- inspect the current state of variables 2- step through the code line by line 3- step over or into functions as they are called 4- resume program execution

advanced debugging

Some bugs are really tricky to catch. Those are typically the bugs that happen very rarely, and in unclear circumstances. In statistical computing and data analysis settings these might be conditions that happen in iterative algorithms sometimes, but not very often, and can be hard to reproduce.

One way to deal with these bugs is to start the debugging session with placing one or more **conditional** breakpoints. These are similar to regular breakpoints but are not going to cause the debugger to stop the execution of the program and hand you the controls unless a specific condition (that you specify) evaluates to true as the program is executing that particular line of code. You may use some conditions that are similar to what you would place in an assert statement (conditions that shouldn't happen) or any other conditions that you think may be associated with the occurrence of the anomalous behavior your are debugging.

Integrated GUI debugger (with VS Code)

Today we will experiment with the visual debugging tools integrated with IDEs. We will do that in VS Code (unless you have another IDE with debugger integration). We will load a piece of code, go through it to understand what it does, then try to discover the problem with it and fix it.

Here's a piece of code that implements the binary search algorithm to locate the first occurence of a number in a list of numbers:

```
import math
def binary_search(lst, T):
    L = 0
    R = len(lst) - 1
    while L < R:
        m = math.floor((L + R) / 2)
        if lst[m] <= T:
            L = m + 1
        else:
            R = m - 1
    if lst[L] == T:
        return L
    return -1
```

There are a couple of things not quite right with this implementation, even though it will run and produce correct results for some cases.

Here's another piece of code implementing merge sort (also with some bugs in it):

```
def merge_sort(lst):
    n = len(lst)
    if n == 1:
        return lst
    return merge(merge_sort(lst[:n//2]), merge_sort(lst[n//2:]))
```

```
def merge(lst1, lst2):
    merged = []
    i, j = 0,0
    while i < len(lst1) and j < len(lst2):
        if i < len(lst1) and j < len(lst2) and lst1[i] < lst2[j]:
            merged.append(lst1[i])
            i += 1
        else:
            merged.append(lst1[j])
            j += 1
    while i < len(lst1):
        merged.append(lst1[i])
        i += 1
    while j < len(lst2):
            merged.append(lst2[j])
            j += 1
    return merged

merge_sort([3,1,5,1,6,3,9,12,8])
```

You can use this to practice stepping inside functions, and thinking about recursion.

Incidentally, if you first sort, then find, you can get the quantile of a particular value within a collection (you'll need to adjust the binary search a little to achieve this).

Alternatively you could start by implementing (without using any existing functions) a function that inverst the order of the words in a string, and debug it until it works.

Here's a version of this function where I injected a couple of bugs, if you prefer to start from there:

```
def reverse_words(input):
    working = list(input)
    invert(working)
    start = 0
    for i, c in enumerate(working):
        if c == ' ' and i != start:
            invert(working, start, i)
            start = i+1
    return ''.join(working)

def invert(lst, start=None, end=None):
    if None == start:
        start = 0
    if None == end:
        end = len(lst)-1
```

```
    while start < end:
        tmp = lst[start]
        lst[start] = lst[start]
        lst[end] = tmp
        start += 1
        end -= 1

reverse_words("These are my words.   I have spoken!")
```

Next time we will touch briefly on how to do debugging without an IDE with debugger integration.