National University of Singapore
School of Computing
CS1010X: Programming Methodology
Semester II, 2019/2020

Debugging Exercises I

When you program, you will certainly make mistakes, both simple typing mistakes and more significant conceptual bugs. Even expert programmers might produce code which contain errors. It's important to know how to find and fix errors in your code. In computer science, this process is called debugging.

Interpreting error messages

To start, perform the following steps:

1 Download and save `debug1.py` on your computer.

2 Open `debug1.py` using IDLE.

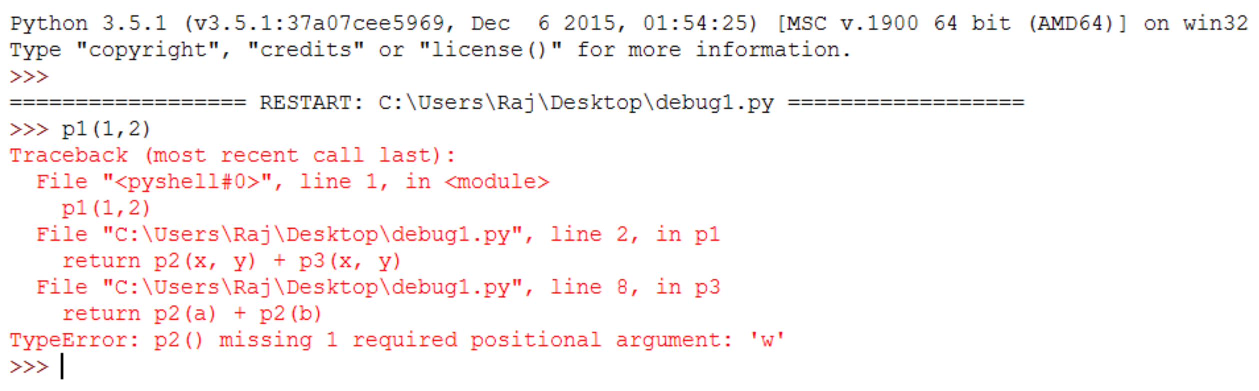
3 Select Run > Run Module or press F5 to run `debug1.py`.

4 Note that the Python shell pops up with no output. The three functions in `debug1.py` have been successfully defined.

Now evaluate the following expression by typing it in the shell, then press enter:

```
p1(1, 2)
```

You should encounter a red error message:



```
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:54:25) [MSC v.1900 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> 
================== RESTART: C:\Users\Raj\Desktop\debug1.py ==================
>>> p1(1,2)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    p1(1,2)
  File "C:\Users\Raj\Desktop\debug1.py", line 2, in p1
    return p2(x, y) + p3(x, y)
  File "C:\Users\Raj\Desktop\debug1.py", line 8, in p3
    return p2(a) + p2(b)
TypeError: p2() missing 1 required positional argument: 'w'
>>> |
```

Don't panic. There is helpful information about the error that we can use to fix it. The first part of the error message is the Traceback. It shows a sequence of executions (the call stack) that leads to the error. By tracing through the call stack, we can easily find the source of the error.

Tracing the call stack

In this example, the call stack has three layers.

I) *Layer One:*

```
File "<pyshell#0>", line 1, in <module>
    p1(1, 2)
```

This is where the trouble begins! The first layer indicates the very first execution that leads to function/statement executions in subsequent layers.

There are typically 4 information contained in a layer's message:

- `File "<pyshell#0>"`: the file or the shell that contains the executed statement. In this case, the statement is entered from our shell.
- `line 1`: the line at which the executed statement is. In this case, the statement is at the 1st line of the shell i.e. it's the first statement entered into the shell.
- `in <module>`: the module or the function that contains the executed statement. In this case, the statement is in global scope.
- `p1(1, 2)`: the executed code that resulted in the error.

So the first layer simply says that there was an error encountered in the first line of what you entered in the Python Shell `p1(1, 2)`.

II) *Layer Two:*
The second layer tells us that the error was encountered in the execution of function `p1`, on line 2 of the file `debug1.py`.

```
File "C:\Users\Raj\Desktop\debug1.py", line 2, in p1
    return p2(x, y) + p3(x, y)
```

If you read the source, you will realize that the execution of `p1(1,2)` in layer 1 leads to the execution of

```
    return p2(x, y) + p3(x, y)
```

At this point, you still do not know the actual problem so continue reading the error message.

III) *Layer Three:*

```
File "C:\Users\Raj\Desktop\debug1.py", line 8, in p3
    return p2(a) + p2(b)
```

The third layer tells us that executing the previous line

```
    return p2(x, y) + p3(x, y)
```

leads to the execution of

```
    return p2(a) + p2(b)
```

Making sense of the error encountered

Immediately after this, you see "TypeError". This is the second part of the error message which gives you the details of the error encountered.

```
TypeError: p2() missing 1 required positional argument: 'w'
```

This tells you that at this point, the interpreter ran into a type error. The description that follows explains that `p2` is missing 1 argument. When looking at the definition of `p2`, you will realize that it requires 2 arguments, but only one argument is passed. This results in the error message which says that an argument is missing. This is the source of the error.

Inserting print statements

If you are lucky, the error can be easily deduced from the error message. Unfortunately, sometimes error messages can be quite vague and you might not know what the actual cause is. In this section we are going to learn another common debugging technique: adding `print` statments.

Let us examine the following code, which aims to find the solutions of a quadratic equation $ax^2 + bx + c = 0$. The values of the coefficients a, b, and c are given at the beginning of the program.

```
import math

a = 1
b = 2
c = -3

delta = b * b - 4 * a * c

s1 = (-b - math.sqrt(delta)) / (2 * a)
s2 = (-b + math.sqrt(delta)) / (2 * a)

print("Solution 1: x = ", s1)
print("Solution 2: x = ", s2)
```

The above code can be found in the file `debug2.py`. Download and execute it in IDLE.

If you execute this code, you get two possible solutions: $x = -3$ or $x = 1$. This looks great. Let us test it further by trying the values $a = 548$, $b = 753$, $c = 162784$. Try executing the code in IDLE. What do you get?

If you have followed the instructions above, you will see an error message:

```
Traceback (most recent call last):
  File "/Users/user/Downloads/debug2.py", line 9, in <module>
    s1 = (-b - math.sqrt(delta)) / (2 * a)
ValueError: math domain error
```

Applying what you have learnt in the previous section, we see that the error occurs at line 9, and the message is about a `math domain error`. This message may seem confusing. If you are brave enough, open your notebook and work the solution of the equation out on the paper to see what's wrong...

As the number seems too scary to calculate on paper, and you may be too lazy to do the math, let's use the `print` function to examine values of some variables in our code to find out the bug. $\wedge \smile \wedge$

Let us read our code again. In line 7, we compute `delta`. We know that if `delta` is negative, the square root is undefined. Is the value of `delta` negative in our code? Let's check this by printing out the value `delta`. We add

```
print("Delta = ", delta)
```

right below the assignment

```
delta = b * b - 4 * a * c
```

Note that the Python interpreter executes your program line by line, and stops immediately when an error is encountered. Therefore if you add the `print` to the end of the program, you won't be able to see the value of `delta` because when the interpreter encounters the error, it has not reached the print statement yet.

Now examine the value of `delta`. Is it negative?

You should obtain a negative value for `delta`. This clarifies the error message which you have gotten. The `math.sqrt()` function does not know how to handle a negative value, hence the math domain error message. At this point, we are now quite sure that the error is caused by the negative value of `delta`. (Remember when you input $a = 1$, $b = 2$, $c = -3$, the program works. `delta` is positive in that case.)

Fixing the bug

We can easily fix this program with an `if`-statement. For example, if `delta` is negative, print a line which says that there is no solution. If it is positive or zero, print the two (or one) solutions. We would leave this to you as an exercise. Happy bug fixing!

Print is simple

The above example demonstrates the simplicity and power of using `print` to check the values of some variables in your program. This way of debugging is simple, and the print function is available in almost every programming language although it might have different names such as `printf` in C or `System.out.println` in Java. Even so, the general process is the same: inject print statements into places where you suspect that something might have gone wrong and check if the values printed out match your expectations.

Congratulations, you are now ready to do some real programming, equipped with one of the simplest but most powerful debugging methods!