1811/2807/7001ICT
Programming Principles

School of Information and Communication Technology
Griffith University

Trimester 1, 2024

18 Debugging

When programs don’t do what was intended, the process of fixing them is called *debugging*.

This section has some general advice about how to debug, and how not to have to do it.

18.1 Not bugs – defects

The term “bug” is cute, but implies that they are things that are going to creep into our programs, no matter what we do.

They should be called *defects*, and they get there because *we put them there*.

The best way to make programs without defects is to not make the mistakes in the first place.

We do that by thinking carefully and designing before we code, and by testing a lot as we code.

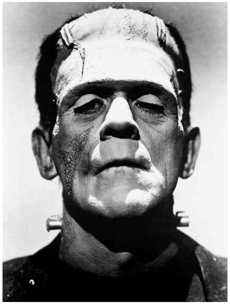
Even so, the best programmers still introduce some defects, albeit less than novice programmers.

18.2 Understanding your program

A defect is an instance where what the program really does is not what was intended.

This implies that we did not really understand what we created.

To remove the defect, we need to understand the program better.


misunderstood creation

If you change something without really understanding the problem, even if it seems to fix the problem, chances are you have introduced another defect that will become apparent later.

While you are learning to program, every bug is an opportunity to understand programs better.

Programmers never stop learning.

18.3 Reductionism is not a dirty word

Most so-called alternative health practitioners claim scientific medicine is bad because it is *reductionist*, and not *holistic*.

We are more than the sum of our parts, but the scientific understanding of our parts sure helps understand the whole.

Lucky for us, programs don't care if you chop them into little pieces.

person debugger

If you try to think about all of a big program at the same time, you will never be able to isolate the problem.

18.4 Test as you code

Never type in all of a big program and only then test.

Type in a few statements, compile, test, understand, debug, and repeat until done.

If you do this, any defects you find will already be isolated to the last few lines you typed.

18.5 Peek inside

If a doctor wants to understand what's wrong with a patient, she will perform tests and scans to find out what's really going on under the skin.

The programmer's problem is that we can't typically watch our programs run.

Sometimes we can, for example with the debugger built into PyCharm, but what about when there is no debugger available?

Add extra printing statements to:

- confirm that a location in the code is actually reached; and
- see what values are actually in the variables, and how they change.

18.6 Isolate with comments

If you have already completed a program, and can’t isolate a problem, particularly if the program is crashing, use comments to remove large chunks of your program until the problem goes away.

Then slowly uncomment, line-by-line, until the problem occurs again.

18.7 Don’t be too clever

Write clear code, as if you are expecting to have to debug it. *You always will.*

> “Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.”
> – Brian W. Kernighan

> “Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”
> – Martin Golding

18.8 Debugging in PyCharm

The key concept you need to know to use a source-level debugger is the *break-point*.

This is a point in your program where you want the debugger to halt your program temporarily, so you may inspect the program's variables.

Inspecting the variables helps you check that variables *really* have the types and values that you think they have.

Following is a program that attempts to solve this problem:

> *Problem:* An Australian tourist in the USA, has an old road map that shows the road distances between towns in only miles. She would like to know the total length of her trip in kilometres. Write a program that lets her enter distances in miles and print the total trip length in kilometres.

```
# file: TripLength.py
# Calculate a total trip length in km,
# given distances in miles.

MILES_PER_KM = 0.621371

def milesToKm(miles):
   """convert miles to km"""
   km =  miles // MILES_PER_KM
   return km
```

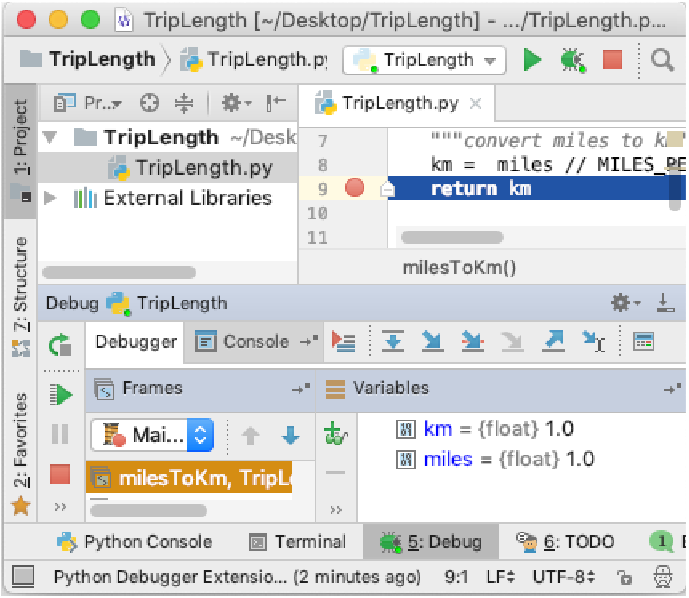
```
print("Total trip calculator. " +
   "End with a -ve leg length.")
miles = float(input("Enter the length " +
   "of a trip leg (miles): "))
totalKm = milesToKm(miles)
while miles >= 0:
   totalKm += milesToKm(miles)
   miles = float(input("Enter the length " +
      "of a trip leg (miles): "))
print("Total trip = {:.1f} km".format(totalKm))
```

A quick test of this program doesn't look quite right.

```
$ python3 TripLength.py
Total trip calculator. End with a -ve leg length.
Enter the length of a trip leg (miles): 1
Enter the length of a trip leg (miles): 1
Enter the length of a trip leg (miles): 1
Enter the length of a trip leg (miles): -1
Total trip = 4.0 km
$
```

By clicking next to the line numbers in the `PyCharm` editor, we can set a break-point (the red dot).

This marks the line *before which* the execution will pause for us to inspect variables.



```
    """convert miles to k
    km =  miles // MILES_PE
    return km
```

milesToKm()

```
km = {float} 1.0
miles = {float} 1.0
```

Start debugging with the Run ▶ Debug... command instead of Run ▶ Run...

When it pauses at a break-point the lower pane shows the types and values of the variables currently in scope.

To continue choose:

- to single-step the program; or
- to resume execution to the next break-point or the end of the program.

Find and rectify the defects in this program with the aid of the debugger.

Section summary

This section covered:

- bugs as defects and misunderstandings;
- strategies for isolating bugs;
- strategies for peeking inside the operation of a running program; and
- how to use the debugger in PyCharm.