Debugging

When writing code mistakes are a certainty. We need to rapidly recover from mistakes to avoid frustration.

Most of tips here work in either Spyder or repl.it.

The main tip: **read the error message**. It will tell you (roughly) where in the code the problem is (the line number) and some general reason for the error. This may not be understandable at first, but once the problem is found, try linking the error message to the problem in your mind. That way the next time you see the same error message the easier it will be to find the problem.

Asking for help

If the tips below do not work then you may want to ask other students, demonstrators, or module leads. Please go ahead. However, please note two points about asking by email:

1. It's really hard to answer a problem without sufficient detail. Writing out the detail is tedious. It's always best to *email the code*. Copy-pasting code into email is error prone (particularly Python that relies on spacing). Including screen shots is very rarely helpful, as the person receiving the message wants to run the code, not type it in again (and make their own mistakes, or accidentally correct yours without noticing).
2. The university email server blocks attached files that end `.py`. To email a Python script as an attachment either compress it in an archive (`.zip` by preference) or change the file extension (for example to `.txt`).

Code does not run

Python will not run scripts that have syntax errors that it can spot.

Syntax errors often result from

- mis-spelling a variable name (`witdh` instead of `width`);
- an error in assignment (`v + 3` instead of `v = 3`, when `v` has not yet been assigned);
- not closing a bracket.

Strategies

Use editor features

In most cases the error message will give you the line with the problem. The IDE editor will also highlight problems (look for red lines in the margins, or red underlines of code).

Check earlier lines

Sometimes Python isn't smart enough. This happens often when a bracket isn't closed. If you can't spot the error on the indicated line, always check the lines *before* that indicated for problems with brackets.

Comment lines out

Sometimes code can be very confusing and it's hard to spot where the syntax error is. Try commenting out lines that don't appear relevant to the error. If they do change or remove the error that can indicate the (or a) problem was in the line just commented out.

Code runs but gives error

Standard reasons for this are

- the code tries to do the right thing with the wrong input;
- the code uses features in the wrong way.

Examples

An example of the first would be

```
x = 0
y = 1 / x
```

It's fine to divide by a number that's not zero: it's just the particular value of `x` that's the problem.

An example of the second would be

```
x = [1, 2, 3]
v = [10, 11, 12]
z = v[x]
```

The final line tries to index into a list using another list. The code may have wanted to get certain entries of the list, in which case slicing or some `numpy` features would be better. Alternatively, the code may have intended to loop over the indexes encoded in `x`.

A final example would be

```
def my_add(a, b):
  result = a+b

x = my_add(1, 2)
y = 1 / x
```

The issue here is that the function has no `return` statement. This means nothing is returned, so the variable `x` is set to `None`. It makes no sense to use this in Python mathematical expressions, so an error results.

Strategies

`print` stuff out

The examples above occurred because, when writing a line of code (say `y = 1 / x`) the coder had assumed "sensible" input. When the code actually runs the value of `x` turned out not to match the assumptions (so `x` was zero, or not a number).

Once you know what values cause problems you can track back through your code to find where the assumptions failed.

To find the values, add `print` statements before the line causing the problem. When you know which term is causing the problem, work back and add more `print` statements to discover how it was set. For example, in the final example, change the code to

```
x = my_add(1, 2)
print("x is", x)
y = 1 / x
```

We then see

```
x is None
```

output to the screen. This tells us something is up with the `my_add` function; we could add `print` statements to the function internals to check what it's doing, and (hopefully) then see the missing `return`.

Variable explorer

This works in Spyder only. There is a tab in the top right (with Help, and Plots) called the Variable Explorer that shows the current values of all variables. You can run the whole script, or some lines of the script (highlight them, right-click, "Run selection"), and then see the values collected in one window.

The advantage of this approach is that you don't have to change your code, and you can see all the values you might need. The disadvantage is that it's tedious to select lines to run if dealing with a problem in, say, the middle of a long loop.

Debuggers

Both repl.it and Spyder include debuggers. This combine the previous strategies and allow to get information about the state of your code, particularly the values of your variables, even in complex situations. There is documentation for Spyder and briefer documentation for repl.it.

The idea is to add a *breakpoint* to the code. This is done by clicking in the margin of the editor next to the line number. In Spyder this will add a red dot; in repl.it a blue dot. When the code is then run *using the debugger*, it will pause every time it reaches the breakpoint. The value of any and all variables can be checked at this point.

The code pauses *before* executing the line that the breakpoint is on. We can also step the code forward one line at a time to check the effect of each.

Breakpoints are particularly useful when there is a problem with loops. Add a breakpoint inside the loop and check the values at each stage.

Code runs, gives no errors, but the wrong result

This usually results from no implementing an algorithm exactly right.

Many of the strategies needed match those above: check the values of the variables at each step. The difference is that the computer will not tell you where the problem is. You need to know what the result should be.

Example

Compute the sum of 2^{-n} for $n \in 0, 1, \dots, \infty$. The answer should be 2. We can't go to infinity, so go to a large number.

```python
total = 0
summand = 1
for n in range(1000):
  summand = summand / 2
  total = total + summand
print(total)
```

Each time through the loop we compute the next term within the sum, the `summand`, by dividing the previous one by 2. We accumulate that.

If you run this code you will find the total being 1. Changing the range of the sum makes no difference if it is large enough. The problem with the code is that we should add the `summand` to the `total` *before* updating it.

Strategies

Test in the simplest possible cases

We want to find the step that is going wrong. This is hard when there are many complex steps. So first check in the easiest cases.

What happens in the example if the range for n is restricted to $n = 0$ only? Analytically we immediately see that the sum is $2^{-0} = 1$. Changing the `range` statement in the code to `range(1)`, which runs from zero up to (but not including) one, the code outputs `0.5`. We see the loop fails at the first step. This tells us that `summand` must have the wrong value, or `total` is incorrectly initialised. These can be checked with `print` statements or in a debugger.

Test classes of problems

If the simple tests work, think about what *qualitatively* changes in the "real" problem you are trying to solve. Then find a simple test for your qualitative change.

As a toy example, look at

```python
def get_first(v):
  return v[0]
```

This function returns the first element of a list. It works fine when a list has one object or five objects. But if an *empty* list is passed in it will give an error.

This is linked to the "zero, one, infinity" rule. Unless there is a very special case, your function should work (or should check) if an object exists (is not zero), if it's unique (just one of them), or if it could contain arbitrarily many things. That gives three qualitative cases the code should work on (or exclude as possible inputs).