# Lesson 3, Week 1: Error messages and debugging

---

## AIM

— to learn to read error messages

— to learn line-by-line method of debugging

After this lesson, you will be able to

* find useful information in Julia's error messages

* find errors in a single line of Julia code and fix them

* step through Julia code line by line, fixing errors as you find them

* occasionally go back to fix errors you missed

## Debugging some almost-Lesson-1 code

Suppose by mistake we tried to run Lesson 1 using the code below[1].

```
mystiring1 = "Hello, world
prinln(mystring1)
```

You find it doesn't run (and, depending how you are trying to run it, you may see an error message).

By examining the code line by line you can find out why. Here, when we try to run the first line from the REPL, it hangs, without any error message. Your job now is to find out why, and you start with a careful look at the code. Is it valid code? No! There is no closing double quote character to indicate the end of the string value. As Julia sees it, you are still entering characters in a string[2].

Fix that line by adding the missing double quote character at the end.

[1] NB: this works best if you start with a fresh REPL. It contains typos, you should be careful to type the example below exactly as it is, typographical errors and all

[2] If you put these two lines in a .jl file and tried to use `include()` run it, you would get the error message " LoadError: syntax: incomplete: invalid string syntax" which is more helpful, but not very much more. Fact is, there are many ways that syntax errors can occur, and Julia does not try very hard to tell you exactly what the error is.

Here's a good tip: also add a semi-colon after the closing double quotes. In the REPL, this suppresses the output of that line, which means you have less to read and see fewer distractions. Of course, in some cases you want to see the output of evaluating a line of code, this isn't always the right thing to do.

Ok, so now the code becomes

```
mystiring1 = "Hello, world";
prinln(mystring1)
```

When we run the first line in the REPL, as expected see no output. So we run the second line. Oh dear. Suddenly lots of lines on the screen and some of them in red!

The line starting with ERROR is the important one[3]: this is the error message. In it, Julia gives you some indication of what the problem might be. Here, the message is `UndefVarError: prinln not defined`. But of course that is easy to fix also, so now the code becomes

```
mystiring1 = "Hello, world";
println(mystring1)
```

And now we get an error: again the message `UndefVarError: mystring1 not defined` . How unexpected! And it seems wrong: the line is valid Jullia code! When we examine the `println()` line there seems to be nothing wrong with the variable name, how can it be undefined?? This often happens in debugging: some errors mask others. You fix the first ones you find, and eventually they reveal errors earlier in your code. Here of course there is a spelling error in the line above it. If we don't spot that right away, the fact that the error message says `mystring1 not defined` should help. It means this is the first time that Julia sees the name `mystring1`. Whatever we believed about creating it earlier is false: either there is no line at all for creating it, or the line is wrong. Find the line where it should have been created, and fix the error. Finally, all correct when we change for the last time to

```
mystring1 = "Hello, world";
println(mystring1)
```

## Some points about debugging

Line by line examination in this way is by no means the only way to find bugs, but it is a good place to start learning how to debug code. You should practice it a lot and become skilled at interpreting error messages.

In fact, debugging is an absolutely vital skill for programmers. Just as for writing code, here are many, many styles of debugging[4]. But at the heart of them all is the ability to read through code with a very critical, searching eye. That ability is what you should cultivate.

[3] So important that in most terminals it come up bright red.

[4] As we'll see in the next lesson, writing code is fundamentally an act of creative expression. Debugging too allows you to go about it your own way.

If you go on to program a lot, you will probably want to learn about software to help you with the task. There is quite a lot out there[5], but in this course we do not go into that.

## Review and summary

* Debugging line by line is a very useful skill to learn.
* Error messages in Julia contain useful hints, but also a lot of other detail.
* Error messages in Julia[6] cannot guarantee that they point at the actual error.
* Line by line output is not quite the same in the REPL as it is for running a .jl file with the same code.
* For line by line output, it can be helpful to suppress output with a semi-colon at the end of a line.

[5] For Julia, the main supports are the `Debugger.jl` package, which is very useful but takes a while to learn, and the `Juno` development interface, which is not specially for debugging but has a lot of features that can help with it—and these can be extended by using `Debugger.jl`.

[6] Or any other language, for that matter.