Error handling

Syntax mistakes and debugging

Sometimes code will not run because of a coding mistake. Such errors should simply be fixed. Try to run the following code and see what happens. Does the error message help you pin down what went wrong? Now fix the mistake and run it properly.

```
warning = "Winter is coming"
print(warn)
```

Hopefully when you get an error, the error message is useful in determining what has gone wrong. Sometimes the error message highlights where the error takes place. Sometimes, the program doesn't recognise there is a problem until later in the code, so it may highlight a later line rather than the one with the error, but it should give a hint about where to start looking for the mistake. Specifically: if the program stopped on a particular line, the error should be on that line or earlier in the code.

Sometimes a program doesn't display an error message but just doesn't do what it should either. If you are struggling to get a program to do what it should and you aren't sure which parts of the program are causing problems, one technique is to put some dummy output into the program.

For example, say you have an `if` statement but you don't know if it is running as you expected.

```
i=1
while i<100:
    if i % 2 == 0:
        i = i + 1

    i = i + 2
```

If you want to know whether the code inside the `if` statement is ever running, one way is to print a debugging message. This is a message that you write just for testing and that you don't intend to include in the finished program.

```
i=1
while i<100:
    if i % 2 == 0:
        i = i + 1
        print("This bit ran")

    i = i + 2
```

Running this, you can see whether the program ever ran the block of code inside the `if` statement by looking whether the text `"This bit ran"` is ever displayed.

If you want to know how many times a loop ran, you can add and print out a counter. For example:

```
i=1
counter=0
while i<100:
    counter = counter + 1
    if i % 2 == 0:
        i = i + 1
        print("This bit ran")

    i = i + 2
print(counter)
```

Once you are finished testing, you can delete or comment out these test outputs.

It can also help to write programs in small chunks and then run them, because if you know it ran fine until you added a line/loop/function/whatever, you can be pretty sure the error is in the bit you just added.

The rest of this page is not about coding errors, which should be fixed before finishing a piece of code (especially before handing it in to be marked!). The rest of this page is about errors that occur as the program is running, called exceptions.

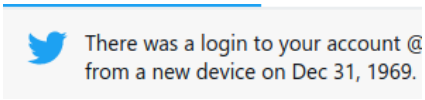
Failing gracefully

Often, the way errors are displayed is not particularly user-friendly. It is better to handle the error in a way that causes your program to *fail gracefully*, that is, to do something nice to communicate to the user what has happened.

For example, look at the following picture that I took at Nottingham railway station. It seems that some diagnostic code is output to the screen to help diagnose the error (the bits like "`Addr = 164 (A4h)`" and "`Script = P1148M1:NT01-V2.0.0`" look like debugging output to me; I'm not sure what the repeated "`.  .`" are about). This sort of output is fine if you are trying to fix an error, but these screens were like this for weeks. It is not particularly graceful to show diagnostic or debugging readouts to your customers - far nicer to have displayed a friendly message apologising that the screen is not currently able to show the information it should.

Here are a few examples I've collected of non-graceful failures. A well-written program should notice something has gone wrong and fail gracefully, rather than outputting gibberish to the user.

I got this warning in the Twitter app - extremely strange that someone was able to log in decades before Twitter existed, but why did it take them so long to tell me about it?

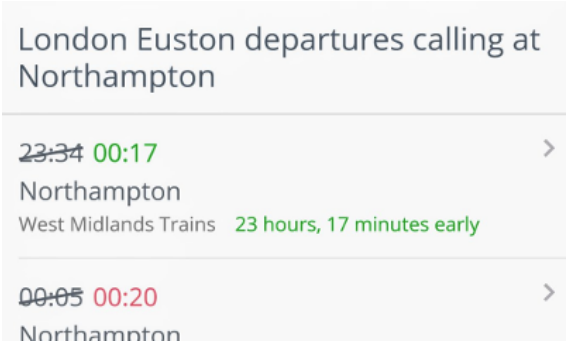


(It may help to explain this error if you know that some computer systems measure time as the number of seconds that have passed since 1 Jan 1970.)

I got an email that contained this text. What do you think was supposed to happen here?

Please do not reply to this email. If you have any questions or need help, please contact us at %{helpdesk_email} or visit %{contact_us}

I was looking up times for a delayed train and saw this. Can you tell what has gone wrong?



You can't really do anything about a computer crash, but it always amuses me to see BSoD in the wild.

Exceptions

An *exception* halts the code and displays an error message. The rest of the code does not run.

Try running the following code.

```
x = 2
print(5/x)
print("Now do something else")
```

Now edit the code so that the value assigned to `x` is zero, and run it again. This should raise an exception called `ZeroDivisionError`. Python displays an error message and halts the script. Note that the string `"Now do something else"` isn't printed because the script is halted before it gets to this point.

Try and except

One way to handle code that might throw an exception is to use a `try`/`except` block. This stops an exception from automatically halting the program and displaying an error to the user. Python to *tries* to run the code in the `try` block and then runs the code in the `except` block if and only if an exception is raised. This means a friendlier message can be displayed to the user, and that the rest of the program can still run.

Run the code below. Note that the exception is handled by the `except` code, and so the script does *not* halt and the string `"Now do something else"` *is* printed.

```
x = 0
try:
    print(5/x)
except ZeroDivisionError:
    print("Oh dear, you tried to divide by zero.")
print("Now do something else")
```

You can catch different exceptions and do different things with them. Here, try changing the value of `x` to `x=0` and running the code again.

```
x = "chickens"
try:
    print(5/x)
except ZeroDivisionError:
    print("Oh dear, you tried to divide by zero.")
except TypeError:
    print("Oh dear, you tried to divide by something that isn't a number.")
print("Now do something else")
```

You should catch the exceptions you expect might happen and deal with them appropriately, leaving unexpected exceptions to halt the program in the usual way so you don't hide that an unusual error occurred. Partly, if an unexpected error occurs that makes Python want to halt the script, overriding this when you don't know what happened might be dangerous and cause problems later in the program.