Introduction to programming

Handling errors and testing

Whilst you have been writing your Python programs in the lab exercises it is likely that you have come across problems when trying to run your code. You will have seen the red error messages if you are using IDLE with information about what has gone wrong with the program. These error messages are vital in understanding what has gone wrong with your program and interpreting them correctly will allow you to fix issues faster.

Types of errors

There are two types of errors that can be introduced in to a program. A **syntax** error is where the programmer has misspelt a keyword or not followed the necessary rules for writing a program. For example, in Python if you were to try and print out a string without quotation marks:

```
>>> print Good morning!
SyntaxError: invalid syntax
```

In IDLE, the red **SyntaxError** message is generated and this will be similar for other text editors or IDEs. Most other programming languages will also generate these sorts of error messages but may handle what happens next differently. In Python, if your program has a syntax error then the program will not run. In fact, if you are using the editor in IDLE, it will not even attempt to run the program. In this way, Python is very similar to compiled languages such as Java or C where the compiler catches any syntax errors before the program is compiled ready to run. This is in contrast to some languages such as Javascript where the code will start to run, printing out any statements or executing any functions until an error is encountered.

Python doesn't check variable names at run time to see if they exist before use.

```
msg = "Good Morning!"
print "Hello,"
print message
```

For example, in the above code the program will run and the first print statement will be displayed to the user. However, as **message** does not exist, a **run-time exception** will be generated.

The other type of error is harder to identify and prevent. **Logic** errors are where the programmer believes the correct implementation of a statement or function has been written but it behaves differently. This can also lead to **exceptions** being generated by the programming language.

```
x = 3
y = 4
average = x + y / 2
print(average)
```

The above code won't produce any errors but it is clear that the programmer is attempting to find the average of 3 and 4. The answer should be 3.5 however if you were to run the above code you would get the answer of 5 displayed (as brackets are required to make sure the mathematical operators work in the sequence that is intended). This is an unexpected result and constitutes an error in the program as the value stored in **average** could now filter to other parts of the program and cause further unexpected results.

Exceptions

There are certain times when a program is running when it will encounter a problem and can't correctly handle it. This will result in the program raising an **exception** and exiting. This was mentioned in an example previously where a programmer has misspelt a variable name or tried to use one that hasn't been assigned a value. In this instance, a **NameError** exception will be generated and the program will stop:

```
Traceback (most recent call last):
  File "C:\Users\james.bubb\Desktop\err.py", line 3, in <module>
    print message
NameError: name 'message' is not defined
```

There are many types of exceptions covering all sorts of situations such as files not being found, division by zero and incorrect indices being used with lists. For example, try typing in **1 / 0** (one divided by zero) in to your Python shell and you will generate the **ZeroDivisonError** exception. Here are some of the more common exceptions in Python:

Exception	Occurs when…
EOFError (End of File)	Using **input** or **raw_input** reaches the end of a file and there is no more data to read.
IOError (Input / Output)	A file or URL does not exist when trying to open with the **open** function.
IndexError	An invalid index is attempted to be used on a list.
NameError	A variable name is used which doesn't exist.
SyntaxError	A program breaks the rules of Python's syntax.
TypeError	A program tries to use a function with a mismatched data type

A full list of exceptions in Python can be found at
https://docs.python.org/2/library/exceptions.html

Handling exceptions
One particular exception you may have encountered is when trying to obtain a numeric input from the user. You could wrap the **raw_input** function in the **int** function to convert the number they type in to an integer (instead of the default string they enter).

```
number = int(raw_input("Enter a number:"))
```

This will be fine so long as the user enters a valid integer number. If the user enters a string by mistake, an exception will be raised.

```
Traceback (most recent call last):
  File "<pyshell#150>", line 1, in <module>
    number = int(raw_input("Enter a number:"))
ValueError: invalid literal for int() with base 10: 'James'
```

You cannot trust that the user will enter a digit and not generate the error but Python will allow us to **handle** the exception where you provide instructions to complete if the data entered is invalid. This is done by using a **try..catch** block of code.

```
while True:
    try:
        number = int(raw_input("Please enter a number: "))
        break
    except ValueError:
        print "That isn't a valid number.
```

Using a **try..catch** block, the code indented after the **try:** statement is run as normal but if it raises an exception, the program jumps to the **except** statement and performs the action after the indentation there. If you try the above program, you will see that if you enter a valid integer number, there is no problem and the program finishes. However if you enter a string, the **ValueError** exception is raised and this is **caught** by the **except** block, which prints out the error message. You will notice that you have to specify the type of exception to catch, in this case it is **ValueError** but the principle works for other types of exceptions.

Sometimes there is a potential for there to be multiple exceptions that need to be caught. In Python this can be done simply by putting the list of exceptions in parentheses (), separated by commas.

```
try:
    with open('file.txt', 'r') as f:
            print int(f.readline())
except (ValueError, IOError):
    print "Something went wrong!"
```

The above code will produce the same error message regardless of whether the exception is related to a problem with the file or with the changing of the input from the file into an integer. Exceptions can therefore be separated on to different lines so you can output different messages accordingly:

```
try:
    with open('file.txt', 'r') as f:
            print int(f.readline())
except IOError:
    print "File could not be opened"
except ValueError:
    print "File contains non-numbers"
```

Performing a final action

There may be times that you wish to perform a final action, regardless of whether an action has completed successfully or an exception was raised. This can be done with the **final** statement in Python.

```
try:
    with open('file.txt', 'r') as f:
            print int(f.readline())
except IOError:
    print "File could not be opened"
except ValueError:
    print "File contains non-numbers"
finally:
    print "File I/O complete."
```

In the above extended example, the print statement under the **finally** block will always be displayed to the user. This is regardless of whether or not the file operation completed. This can be useful to display a message to the user to let them know that the program is continuing and it has managed to deal with the exception if one occurred.

Testing programs

In software development it is crucially important to test programs before they are made available for release to users. There are many different approaches to testing programs, each with applications to which they are suited to. Later in your apprenticeship, software testing will be looked at in depth so this section will just be an overview of doing some basic testing on the programs you are currently writing.

There may be different people involved in the software testing stage with some companies employing their own dedicated software testing staff. It usually isn't a good idea for a programmer to test their own code as they may make assumptions or take in to account the way the code has been written thereby not really giving a true reflection of the program. Depending on the stage a program is at, the testers may be the **end user** who will ultimately be using the program e.g. a customer.

Introducing software testing at a high-level, you can think of there being two types: **black box testing** and **white box testing**. Black box testing is where the program's functionality is checked and verified to be working without looking at or considering the actual code. In this way, other programmers, testers or ultimately the end user can use the program and see if it performs as they expect it to. White box testing is done by other programmers or testers who have examined the actual code of the program. With this extra knowledge they will be aware of how the program is supposed to react in certain situations. Therefore the tests will be more in-depth, ensuring the logic of the overall program is sound.

As mentioned, the approach to testing software is a broad subject with every company having their own methods.

Test plans

Part of a testing strategy is to design, implement and review a test plan. In its simplest form a test plan can be a table with a list of tests that are to be performed, the data to be used, the expected result and the actual result achieved when the program is tested. A test plan would usually be created by a specifically employed tester or by another programmer. The plan can then be handed to another tested for them to perform the same tests.

For example, consider the snippet of code below:

```
def get_number(msg="Please enter a number:"):
    while True:
        try:
            number = int(raw_input(msg))
            break
        except TypeError:
            print "That isn't a valid number"
    return number


number1 = get_number()
number2 = get_number("And another one:")

print number1, "divided by", number2, "=", float(number1)/float(number2)
```

The program defines a function to retrieve a number safely from the user and then asks for two numbers which are then divided. The test plan for such a piece of code could look something along the lines of this:

Test id	Description	Data	Expected Result	Actual Result	Notes
1	Program divides two numbers correctly	Num1 = 10 Num2 = 2	**5**		
2	Program divides correctly when second number is larger	Num1 = 2 Num2 = 10	**0.2**		
3	Program does not divide by zero	Num1 = 10 Num2 = 0	**0**		
4	Program doesn't accept invalid numbers	Num1 = "test"	Text displayed: **"That isn't a valid number"**		
5	get_number() displays the default message	n/a	Text displayed: **"Please enter a number:"**		
6	get_number() displays the provided argument message	get_number("And another one:")	Text displayed: **"And another one:"**		

Tests 1-4 could be considered black box as the tests are designed to check if the functionality of the program is OK without digging into how the program is achieving the end result. Tests 5 and 6 fall under the white box testing category. In order to investigate these tests the tester would need to know that there is a function called **get_number()** which can take an optional message argument.

Take a moment to copy the above code and work through the test plan yourself. Make a note of what the actual result is when you run the test and any notes you think are relevant.

Once the test plan has been implemented the rest of the table can be completed which may look like this.

Test id	Description	Data	Expected Result	Actual Result	Notes
1	Program divides two numbers correctly	Num1 = 10 Num2 = 2	**5**	**5**	**Passed**
2	Program divides correctly when second number is larger	Num1 = 2 Num2 = 10	**0.2**	**0.2**	**Passed**
3	Program does not divide by zero	Num1 = 10 Num2 = 0	**0**	**ZeroDivison Error**	**Failed** **Program crashed, uncaught exception raised.**
4	Program doesn't accept invalid numbers	Num1 = "test"	Text displayed: **"That isn't a valid number"**	**ValueError**	**Failed** **Program crashed, uncaught exception raised.**
5	get_number() displays the default message	n/a	Text displayed: **"Please enter a number:"**	**"Please enter a number:"**	**Passed**
6	get_number() displays the provided argument message	get_number("And another one:")	Text displayed: **"And another one:"**	**"And another one:"**	**Passed**

Running through the program, most of the tests passed. However test 3 failed as the program does not consider if the user enters 0 for the second number which causes the **ZeroDivisionError** problem. Also, test number 4 which tries to enter a string instead of a number, failed. The actual result gave us a **ValueError** exception whereas the code is checking for a **TypeError** exception.

After implementing a test plan as above, the next step is to review the failed tests and make suggestions for them to be remedied. Test 3 could be remedied by trying the division calculation and catching the exception.

```
try:
    result = float(number1)/float(number2)
    print number1, "divided by", number2, "=", result
except ZeroDivisionError:
    print number1, "divided by", number2, "=", 0
```

Test 4 can simply be fixed by updating the code in the get_number function to catch a **ValueError** instead of a **TypeError**.

```
def get_number(msg="Please enter a number:"):
    while True:
        try:
            number = int(raw_input(msg))
            break
        except ValueError:
            print "That isn't a valid number"
    return number
```

Then the tests can be re-tried to see if the fixes to the program have resolved the failures that were occurring previously.

Test id	Description	Data	Expected Result	Actual Result	Notes
3	Program does not divide by zero	Num1 = 10 Num2 = 0	**0**	**0**	**Passed**
4	Program doesn't accept invalid numbers	Num1 = "test"	Text displayed: **"That isn't a valid number"**	**"That isn't a valid number"**	**Passed**

It's worth noting that generally after changes have been made to the code of a program, all of the tests in a test plan would be re-run. Making changes to code can fix one problem but introduce another. This is where **automated testing** would make re-running the existing tests which previously passed less painful. In the example program used, it is too simple for the changes to have had any effect.