PAL Consolidation 7 *University of Hertfordshire – Week 9 – December 2015 –*

Aim & Objectives

- Review last session
- More exercises of Test Driven Development
- How to test

Introduction

In this session we'll be looking at more examples of testing and defending our code against bad operation.

Exercise - Testing Bad Code

Open the resource '*Resources/Week9/worstcode*' on the website where you should see a program that is particularly susceptible to being broken by the user. We'll start by analysing potential flaws in general and use this to work out how to generate good test cases.

Getting user input

When getting user input the first thing that needs to come to your mind is "*what if the user doesn't enter what I want them to enter?*". If you want a number a user will give you a letter. If you want a word they'll give you an empty string. If you want a number in a range they'll give you a number outside that range. If you want a positive value they'll give you a negative.

The point is, the user will always keep you on your toes. The weakest part of a computer is the bit that connects the screen to the keyboard. To get around these issues, before we do something that may cause our program to crash, we should check it's at least got a fighting chance!

Numbers

There isn't a particularly good method for checking how valid a number is in Python. Checking how valid each character is introduces it's own overhead [1]. It's far better in this case to catch the case where a cast from a string to a number fails, for example:

```
# isNumber()
#
# Check string is valid number.
#
# @param s The string to be checked.
# @return True iff valid number.
def isNumber(s) :
    try :
        float(s)
        return True
    except ValueError :
        return False
```

The code shown above is exactly how it looks, the '`try`' is the program literally "*trying*" to run the code you've asked it. If an error occurs, it stops running in the '`try`' block and it makes an effort to "*catch*" the error it's looking for with '`except`'. It then runs some other code instead defined inside the '`except`' block. This structure resembles '`if`' and '`else`' relationship.

The line in that code that will cause the program to "*throw*" an "*exception*" is the '`float(s)`' line, where a string like '`"hello"`' cannot be made into a number. "*throw*" is how we talk about the computer finding there was an error during execution and "*exception*" is the way we refer to that error that was given out by the computer. We say that these cases are exceptional.

Tip: Keep the code inside a try-catch scenario as minimal as possible as it usually adds significant overhead to the speed of your program. Usually you only want to use them for inputs you don't control, otherwise you simply need to design your algorithms better if they throw exceptions.

[1] '*http://stackoverflow.com/questions/354038/how-do-i-check-if-a-string-is-a-number-float-in-python*'

Other Values

For other values you'll have to be more creative. You could look at characters by inspecting them individually, the length of the string, whether the string refers to something that already exists - the list is endless. It all depends on what valid data you expect from the user which always depends on what you want to do with it, numbers are just the most common case.

Checking your Math

The next place where you are likely to get issue is in the mathematics in your program. There are literally infinite ways most programs can go wrong and it's your job to go from this large number to just one issue you can solve. Below are some ways programs can go wrong.

Divide by Zero

This one is infamous. You need to make sure that there is no way your program will allow the computer to divide by zero as this will cause a crash. Most people have punched '`1 / 0`' into an old Casio calculator and got the predictable "*Error!*" on the display. The reason this happens is because mathematicians cannot decide what the answer is to the problem, therefore the output is undefined. Rather than output something incorrectly, computer scientists chose not to output anything at all and to throw an exception in the case it occurs. Make sure you watch out for this in your programs and test for it where possible!

General Algorithms Issues

One of the other aspects that usually goes wrong is the algorithm simply doesn't work. The computer not crashing, the program compiling and the code programmed as planned are not measures of how well it *actually* works to solve the task. Your method to solve a problem just may not work and it's a general sense you need to learn that you may just be trying hard at the wrong solution. Some general advice to prevent this:

- Plan your algorithm on paper before hand and make sure you understand how it works.
- Write tests for your algorithm as specific key stages to make sure sections work how you think they should.
- As a friend what they think of what you have planned or written and try to get some constructive feedback. Sometimes it takes many minds to tackle a problem!

Resources & Further Reading

'*http://homepages.herts.ac.uk/~db12aba/*' – All content from these sessions updated weekly.

'*http://code.org/*' – A good resource testing your programming skills.

‘*http://stackoverflow.com/*’ – Highly recommended online help for programmers (NOTE: Employers are interested to know whether you’re an active member of this site!).

‘*http://draw.io*’ – A very good, free online drawing tool that exports to many formats, including ‘XML’ and ‘JPG’.