Robust Programs – Gist

When designing programs we need to ensure that they are robust.
Failure to do this may result in two critical outcomes:

1. The program wont function correctly / will not do the job its supposed to do.
2. The program may be unsecure and data that it holds may become compromised.

Defensive design strategies minimise these issues:

- Input Sanitisation
- Input Validation - Anticipating Misuse
- Authentication
- Maintainability of Code
- Testing

Input Sanitisation - Data entered by user is cleaned of any unwanted characters that the user may enter. E.g removing spaces to avoid SQL injection attack.

SQL Injections Structured Query Language is used to lookup data in a database.
When you log in to an account, you will add your username and password into a couple of input boxes.
When you press 'enter', in the background these two pieces of information are added to an SQL statement:

E.g: SELECT account WHERE username = "bjones" AND password = "pa$$w0rd"

With SQL injections, you can 'bolt on' some SQL to the end of your password. This will then alter the SQL statement and allow you to access the accounts of other users.
Using input sanitisation we can remove certain characters which could enable the user to misuse the system.

Input Validation

Input validation is when a system will check that the input meets certain criteria, so to ensure that the data is in an acceptable form.
For example, if a user is to input their email address to enable them to sign up to a user account, input validation can check to see if the entered email address in the expected form (contains an @ symbol and ends with a domain type (.co.uk)). If the entered address doesn't contain these items then it can be rejected in order to ensure that only valid data is entered by the user.

Whitelists and Blacklists

Software developers will often use whitelists or blacklists to help them plan against unwanted inputs.
A **whitelist** is a lists of data that the program being created, should accept. All other data should be rejected by the program.
A **blacklist** is a list of data that the program being created, should reject. All other data should be accepted by the program.

Authentication

having passwords to only allow certain users…
…and potentially limiting the access of the various parts of the system (access rights).
File access rights – Read Only, Read/Write, Full access

Maintainability

Code is written in such a way that when other programmers are asked to develop the code, they can make full sense of it, therefore reducing the chance of introducing coding mistakes / bugs.
This is achieved by:

- Comments (e.g program name, description, purpose of functions and variables, etc)
- Indentation
- Formatting (code grouped in logical blocks and split up using blank lines)

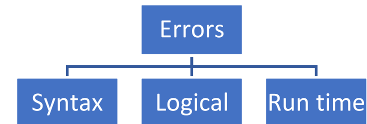
Testing - Gist

In order to check that **the program works as expected** we test it. Testing also **detect errors** in a piece of code. These errors cause the program to run incorrectly or run not as expected.

Errors:

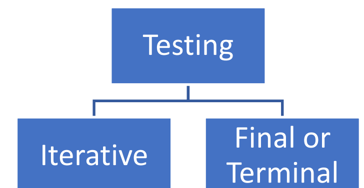
Syntax errors are when the code does not follow the programming language rules. (Grammar of the language)
Translators can only execute a program if it is syntactically (grammar of coding) correct.
Example: Print (“Hello) → print(“Hello”)

Logic errors: causes a program to produce an incorrect output. It does not affect the running of the program.
Example:
To produce a sum of two numbers:
Sum = Number1 * Number 2 *Logical error*
Sum = Number1 + Number2 *Correct code*



Runtime errors may stop the program from running or cause it to crash.
Examples could be: Running out of memory
names = [“John”,”Harry”, “Sam”]
print names[4] *Out of range because the array has only three elements.*

Testing:



Iterative testing is testing the code as you write it.
This could be testing the code line by line or a section at a time.
Once tested and feedback is received you then alter your code as required.

Final or Terminal testing is carried out at the end of the program when all modules are complete and the program is tested as a whole to ensure that it functions as it should.

Test Plan

A test plan should be made before the development of program begins. A test plan will outline exactly what you are going to test. It will anticipate all possible issues and will include test data to test these issues.

Test Data:

It is used to check what happens when a range of predefined test data is entered in the program.
When testing the program it is important to use a range of test data

Normal/Valid – Data that is correct e.g for age for teenager numbers between 13 and 19 inclusive
Boundary – Data of correct type which is on the very edge of being valid e.g. for teenagers 13 - 19
Invalid – Correct data type which should be rejected. E.g age greater than 19 for a teenager
Erroneous – Incorrect data type e.g age is “seven”.

Example of a Test plan. This data is planned to validate an “Age” field for teenager applicants

Test Num	Test Data	Type of Data	Expected output	Actual output	Comment
1.	15	Normal	15	15	Output as expected. No action required.
2.	13	Boundary	13	13	Output as expected. No action required.
3.	19	Boundary	19	19	Output as expected. No action required.
4.	12	Invalid	Error message	Error message	Output as expected. No action required.
5.	20	Invalid	Error Message	Error Message	Output as expected. No action required.
7.	Fifteen	Erroneous	Error Message	Error Message	Output as expected. No action required.