J277 - 2.3 Producing robust programs

2.3 – Producing robust programs

Sub topic	Guidance
2.3.1 Defensive design	
☐ Defensive design considerations: ○ Anticipating misuse ○ Authentication ☐ Input validation ☐ Maintainability: ○ Use of sub programs ○ Naming conventions ○ Indentation ○ Commenting	**Required** ✓ Understanding of the issues a programmer should consider to ensure that a program caters for all likely input values ✓ Understanding of how to deal with invalid data in a program ✓ Authentication to confirm the identity of a user ✓ Practical experience of designing input validation and simple authentication (e.g. username and password) ✓ Understand why commenting is useful and apply this appropriately
2.3.2 Testing	
☐ The purpose of testing ☐ Types of testing: ○ Iterative ○ Final/terminal ☐ Identify syntax and logic errors ☐ Selecting and using suitable test data: ○ Normal ○ Boundary ○ Invalid ○ Erroneous ☐ Refining algorithms	**Required** ✓ The difference between testing modules of a program during development and testing the program at the end of production ✓ Syntax errors as errors which break the grammatical rules of the programming language and stop it from being run/translated ✓ Logic errors as errors which produce unexpected output ✓ Normal test data as data which should be accepted by a program without causing errors ✓ Boundary test data as data of the correct type which is on the very edge of being valid ✓ Invalid test data as data of the correct type but outside accepted validation limit ✓ Erroneous test data as data of the incorrect type which should be rejected by a computer system ✓ Ability to identify suitable test data for a given scenario ✓ Ability to create/complete a test plan

J277 - 2.3 Producing robust programs

Robust programming

Programs that function correctly shouldn't break or produce errors. Avoid these problems by using defensive design:

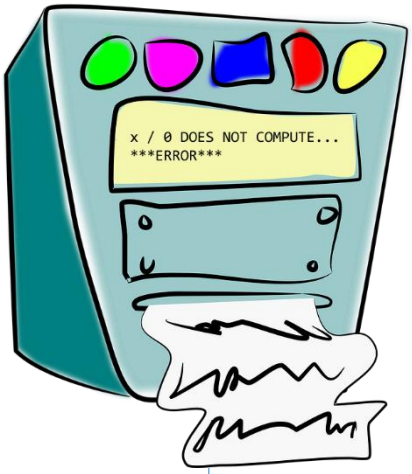
Anticipate and prevent misuse by users.

Keep code well-maintained.

Reduce errors by testing.

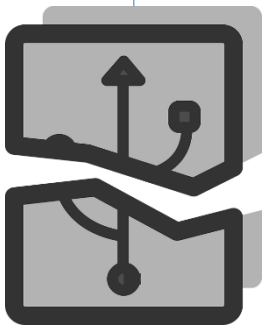
Defensive design: Anticipating misuse

Even with valid inputs there are a number of reasons why a program could crash. These should be trapped by the programmer with exception handling code.



Division by zero.

Communication error.
E.g. lost connection to host.



Printer out of paper.
Printer out of ink.
Paper jam.

Out of disk space.
File not found.
End of file.
Invalid data in file.

A user might also misinterpret the on-screen prompts, or enter data into the wrong input box. A programmer should plan for all possible eventualities.

J277 - 2.3 Producing robust programs

Defensive design: Authentication

Confirming the identity of a user before allowing access. Passwords or biometrics are usually associated with a username.

The most common way of doing this is to ask for a **username** and **password**. The entered details are then checked against a database containing valid accounts.

reCaptcha is a method used to protect online forms against bots. A bot can automatically submit data in online forms creating spam. A robust program needs to identify the user as a human and not another program. Often the user has to type the words they see on-screen which are presented as pictures in a format that would be difficult for a program to decipher, instead of text.

For high security sites such as financial services **'two-factor' authentication** is becoming popular. This means that after the user enters a valid user name and password, the system sends an SMS text 'authentication code' to the designated mobile phone. They then have to enter this as well.

Biometrics is method that checks some physical feature of the authorised person such as their fingerprint. The user puts their thumb on a fingerprint entry device, the data is sent off to a database and checked against their valid data.

J277 - 2.3 Producing robust programs

Defensive design: Input validation

Input validation: Checking if data meets certain criteria before passing it into a program.

Check digit	the last one or two digits in a code are used to check the other digits are correct	bar code readers in supermarkets use check digits
Format check	checks the data is in the right format	a National Insurance number is in the form LL 99 99 99 L where L is any letter and 9 is any number
Length check	checks the data isn't too short or too long	a password which needs to be six letters long
Lookup table	looks up acceptable values in a table	there are only seven possible days of the week
Presence check	checks that data has been entered into a field	in most databases a key field cannot be left blank
Range check	checks that a value falls within the specified range	number of hours worked must be less than 50 and more than 0
Spell check	looks up words in a dictionary	MS Word uses red lines to underline misspelt words

J277 - 2.3 Producing robust programs

Maintainability

Well maintained code is easier for other programmers to understand. They can change parts of the code without causing problems elsewhere.

1. #Write **comments** to explain what is happening at each stage. #ensures the input is restricted to a y or an n

```
do
    for count = 1 to 10
        print("Hey!")
    next count
    again = input("Enter y to go again")
    again.lower
until again !="y"
```

2. Use **indentation** to make the program flow easier to see and show selection and iteration code branches.

3. Use of **whitespace** to easily see where functions begin and end.

`H` - poor choice. What does it mean?

`height` - better choice. The value to be inputted will be a number.

4. Descriptive **variable** and **function** names.

5. Use of **sub-programs** to separate parts of the program.

```
def greet():
    name = input("Enter your name: ")
    print("Hello,", name, "!")
    # Additional functionality can be added here

# Main program
print("Welcome!")
greet()
print("Goodbye!")
```

J277 - 2.3 Producing robust programs

2.3 – Producing robust programs

Sub topic	Guidance
2.3.1 Defensive design	
☐ Defensive design considerations: ○ Anticipating misuse ○ Authentication ☐ Input validation ☐ Maintainability: ○ Use of sub programs ○ Naming conventions ○ Indentation ○ Commenting	**Required** ✓ Understanding of the issues a programmer should consider to ensure that a program caters for all likely input values ✓ Understanding of how to deal with invalid data in a program ✓ Authentication to confirm the identity of a user ✓ Practical experience of designing input validation and simple authentication (e.g. username and password) ✓ Understand why commenting is useful and apply this appropriately
2.3.2 Testing	
☐ The purpose of testing ☐ Types of testing: ○ Iterative ○ Final/terminal ☐ Identify syntax and logic errors ☐ Selecting and using suitable test data: ○ Normal ○ Boundary ○ Invalid ○ Erroneous ☐ Refining algorithms	**Required** ✓ The difference between testing modules of a program during development and testing the program at the end of production ✓ Syntax errors as errors which break the grammatical rules of the programming language and stop it from being run/translated ✓ Logic errors as errors which produce unexpected output ✓ Normal test data as data which should be accepted by a program without causing errors ✓ Boundary test data as data of the correct type which is on the very edge of being valid ✓ Invalid test data as data of the correct type but outside accepted validation limit ✓ Erroneous test data as data of the incorrect type which should be rejected by a computer system ✓ Ability to identify suitable test data for a given scenario ✓ Ability to create/complete a test plan

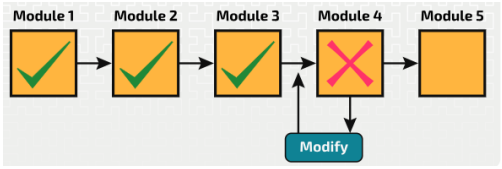
J277 - 2.3 Producing robust programs

The purpose and types of testing

Four main reasons why a program should be thoroughly tested before being given to a user:

To check the program meets the requirements.	To ensure there are no logic errors.	To check the program has an acceptable performance and usability.	Ensure unauthorised access is prevented.

Iterative Testing

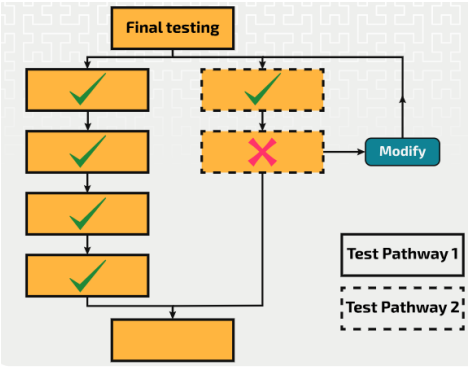


Performed whilst the software is being developed.

The programmer writes an individual section of code or part of the project and then tests it to ensure that it functions correctly. It is likely that errors will be detected, and these are resolved before the next section of the program is developed and tested.

Final/Terminal Testing

Performed at the end of development



This stage happens after the individual sections or modules of the system have been tested, to ensure that the system works as a whole and that it meets the requirements of the project.

J277 - 2.3 Producing robust programs

2.3 – Producing robust programs

Sub topic	Guidance
2.3.1 Defensive design	
☐ Defensive design considerations: ○ Anticipating misuse ○ Authentication ☐ Input validation ☐ Maintainability: ○ Use of sub programs ○ Naming conventions ○ Indentation ○ Commenting	**Required** ✓ Understanding of the issues a programmer should consider to ensure that a program caters for all likely input values ✓ Understanding of how to deal with invalid data in a program ✓ Authentication to confirm the identity of a user ✓ Practical experience of designing input validation and simple authentication (e.g. username and password) ✓ Understand why commenting is useful and apply this appropriately
2.3.2 Testing	
☐ The purpose of testing ☐ Types of testing: ○ Iterative ○ Final/terminal ☐ Identify syntax and logic errors ☐ Selecting and using suitable test data: ○ Normal ○ Boundary ○ Invalid ○ Erroneous ☐ Refining algorithms	**Required** ✓ The difference between testing modules of a program during development and testing the program at the end of production ✓ Syntax errors as errors which break the grammatical rules of the programming language and stop it from being run/translated ✓ Logic errors as errors which produce unexpected output ✓ Normal test data as data which should be accepted by a program without causing errors ✓ Boundary test data as data of the correct type which is on the very edge of being valid ✓ Invalid test data as data of the correct type but outside accepted validation limit ✓ Erroneous test data as data of the incorrect type which should be rejected by a computer system ✓ Ability to identify suitable test data for a given scenario ✓ Ability to create/complete a test plan

J277 - 2.3 Producing robust programs

How to identify syntax errors

A syntax error occurs when code written does not follow the rules of the programming language. Examples include:

- Misspelling a statement, eg writing `pint` instead of `print`
- Using a **variable** before it has been declared
- Missing **brackets**, eg opening a bracket but not closing it

```
# Example: Syntax Error
print("Hello, World!"
```

```
# Correction
print("Hello, World!")
```

How to identify logic errors

A logic error is an error in the way a program works. The program simply does not do what it is expected to do.

- Logic errors can have many causes, such as:
- Incorrectly using logical operators, eg expecting a program to stop when the value of a variable reaches 5, but using <5 instead of <=5
- Incorrectly using **Boolean operators**
- Unintentionally creating a situation where an **infinite loop** may occur
- Incorrectly using brackets in calculations
- Unintentionally using the same variable name at different points in the program for different purposes
- Referring to an element in an **array** that falls outside of the scope of the array

```
# Example: Logic Error
num1 = 5
num2 = 3
sum = num1 - num2  # Incorrect operation

print("The sum of", num1, "and", num2, "is:", sum)
```

```
# Correction
sum = num1 + num2
```

J277 - 2.3 Producing robust programs

2.3 – Producing robust programs

Sub topic	Guidance
2.3.1 Defensive design	
☐ Defensive design considerations: ○ Anticipating misuse ○ Authentication ☐ Input validation ☐ Maintainability: ○ Use of sub programs ○ Naming conventions ○ Indentation ○ Commenting	**Required** ✓ Understanding of the issues a programmer should consider to ensure that a program caters for all likely input values ✓ Understanding of how to deal with invalid data in a program ✓ Authentication to confirm the identity of a user ✓ Practical experience of designing input validation and simple authentication (e.g. username and password) ✓ Understand why commenting is useful and apply this appropriately
2.3.2 Testing	
☐ The purpose of testing ☐ Types of testing: ○ Iterative ○ Final/terminal ☐ Identify syntax and logic errors ☐ Selecting and using suitable test data: ○ Normal ○ Boundary ○ Invalid ○ Erroneous ☐ Refining algorithms	**Required** ✓ The difference between testing modules of a program during development and testing the program at the end of production ✓ Syntax errors as errors which break the grammatical rules of the programming language and stop it from being run/translated ✓ Logic errors as errors which produce unexpected output ✓ Normal test data as data which should be accepted by a program without causing errors ✓ Boundary test data as data of the correct type which is on the very edge of being valid ✓ Invalid test data as data of the correct type which is not on the validation limit ✓ Erroneous test data as data of the incorrect type which should be rejected by a computer system ✓ Ability to identify suitable test data for a given scenario ✓ Ability to create/complete a test plan

J277 - 2.3 Producing robust programs

Selecting and using suitable test data

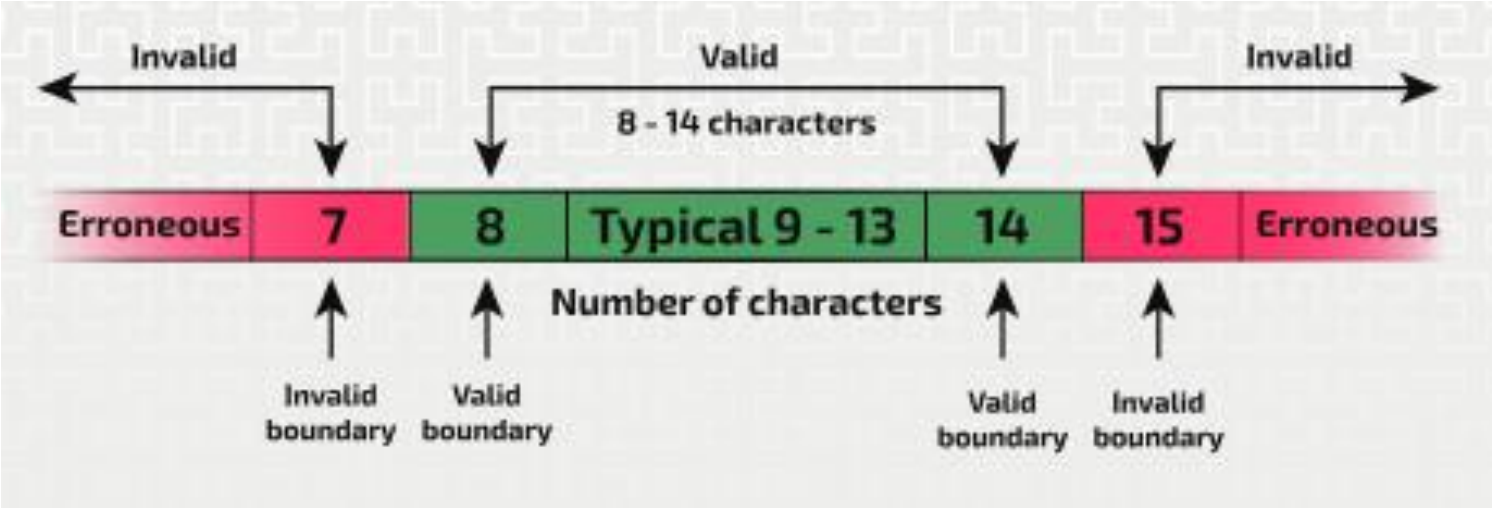
Test data is data that is used to test whether or not a program is functioning correctly. Ideally, test data should cover a range of possible and impossible inputs, each designed to prove a program works or to highlight any flaws. Three types of test data are:

Normal data - typical, sensible data that the program should accept and be able to process.

- **Boundary data** - valid data that falls at the boundary of any possible ranges, sometimes known as extreme data.

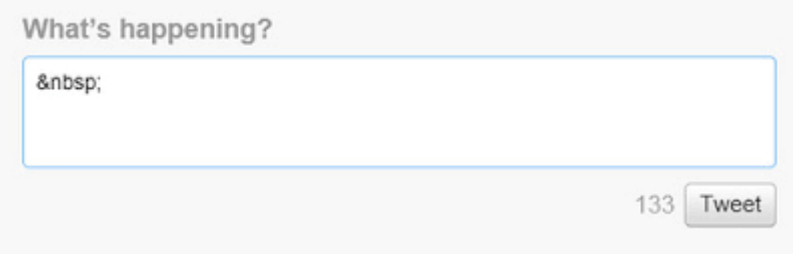
- **Erroneous data** - data that the program cannot process and should not accept.

Types of test data for a password checking system:



Selecting and using suitable test data

In September 2017, Twitter announced it was testing doubling the number of characters in a tweet from 140 to 280 characters. Twitter's character limit is a holdover from the app's early days when tweets were sent as texts, which were limited to 160 characters. It has since become one of the product's defining characteristics. A typical test table that could be used:



Test No.	No. characters input	Type of test	Reason for the test
1	67	Normal data	Check the new functionality doesn't break the original working code. Check the character counter updates correctly.
2	280	Boundary data	Check the maximum number of characters. Check the character counter updates correctly to 0.
3	281	Invalid data	Check only tweets up to 280 characters are accepted. Check the character counter updates correctly to negative number.
4	0	Invalid data	A blank tweet should not be stored.
5	1	Boundary data	Check the minimum number of characters.