Teaching guide: Data validation and authentication

This resource will help with understanding robust and secure programming through the data validation and authentication elements. It supports elements of Section 3.2.11 of our GCSE Computer Science specification (8525). The guide is designed to address the following learning aims:

- Recognise the inherent unreliability of data entered by a user.
- Investigate ways to validate data using code.

A good rule of thumb when writing programs that involve user input is to always assume that the user will make mistakes and write your programs accordingly. By way of an example let's look at a guessing game. An instance of this game could look like this:

```
> Enter the number to be guessed 36
> Enter a guess between 1 and 100 fifty
$$$ Program crashed $$$
```

The user hasn't necessarily done anything wrong – they've entered the string `'fifty'` as their guess, but because our computer program was expecting a string that could be converted to an integer, and because the string `'fifty'` cannot be directly converted into an integer it caused the entire program to crash.

Different types of validation

In using a program, some users will make mistakes, and crashing the program every time they do will be frustrating. Code can be written that validates the user's input before carrying on. Validation does not mean that the user's input is always entered correctly (for example they may enter an incorrect combination of username and password), but it means that the format of their input is correct. A complete list of the necessary validation for our user's guess would be:

- minimum length (they must enter something: this is called a presence check)
- type (the input should be able to be converted to an integer)
- range (the integer representation of their input should be between 1 and 100).

The actions that result from the input not passing one or more of these validation attempts should be to prompt the user to enter another guess and, if we were being helpful, to give a message that explains why their input wasn't valid.

One possible high-level plan for this is:

1. user enters a guess
2. if the length of this guess is zero then
    a. output an error message
    b. go to step 1
3. convert the input to an integer
4. if this causes an error
    a. 'catch' the error
    b. output an error message
    c. go to step 1
5. if the integer is not between 1 and 100
    a. output an error message
    b. go to step 1

In step 4a the error that would be caused by converting to an integer is 'caught'. This means that the program does not crash and when an error is found it is caught by a safety net around that part of code. We can use the following syntax in our pseudo-code to represent this safety net:

```
TRY
   # do some code that might cause an error
CATCH
   # if an error is found then do this code
ENDTRY
```

An example of **TRY–CATCH** when attempting to convert a string to an integer is:

```
string_input ← USERINPUT
TRY
   int_input ← STRING_TO_INT(string_input)
CATCH
   OUTPUT 'input could not be converted to integer'
ENDTRY
```

Coding validation routines

The entire validation procedure above should be written to continue to loop until the data is fully valid and, as we don't know how long this will be, apart from the fact it must happen at least once, then a **REPEAT-UNTIL** or **WHILE-ENDWHILE** loop would seem appropriate.

```
valid ← False
OUTPUT 'Enter a guess between 1 and 100'
REPEAT
    guess_as_string ← USERINPUT
    # presence check
    IF LEN(guess_as_string) = 0 THEN
        OUTPUT 'You have not entered anything'
    ELSE
        TRY
            # type check (will go straight to CATCH if it
            # fails)
            guess ← STRING_TO_INT(guess_as_string)
            # range check
            IF guess < 1 OR guess > 100 THEN
                OUTPUT 'Must be between 1 and 100'
            ELSE
                # if all checks passed then input is valid
                valid ← True
            ENDIF
        CATCH
            OUTPUT 'Must enter an integer (e.g. 42)'
        ENDTRY
    ENDIF
UNTIL valid
```

Taking this a step further

The code below could be used to get user input between a lower and upper bound:

```
SUBROUTINE get_input(lower, upper)
    OUTPUT 'Enter a number between', lower, 'and', upper
    number ← USERINPUT
    RETURN number
ENDSUBROUTINE
```

We can combine our knowledge of validation and subroutines to amend this code so the subroutine will not return a value unless it is an integer within the correct range (this example uses a **WHILE** loop instead of **REPEAT-UNTIL** for variance):

```
SUBROUTINE get_input(lower, upper)
    OUTPUT 'Enter a number between', lower, 'and', upper
    number_as_string ← USERINPUT
    # continue to loop until number is returned
    WHILE True
        TRY
            number ← STRING_TO_INT(number_as_string)
            IF number < lower OR number > upper THEN
                OUTPUT 'Number not within bounds'
            ELSE
                RETURN number
            ENDIF
        CATCH
            OUTPUT 'Not an integer'
        ENDTRY
    ENDWHILE
ENDSUBROUTINE
```

Data validation code is often very repetitive, so becoming adept at putting this code inside subroutines that can be reused will speed up the development of the code as well as making the code more structured, shorter and less likely to contain errors.

Authentication routines

The use of usernames and their associated passwords is a very common method of authentication. Using the techniques so far we could ask a user to enter their username followed by their password and compare what is entered against the list of stored usernames and passwords.

As an overview:

1. User enters a username
2. User enters a password
3. Loop over every stored username-password combination
    a. if the username entered matches a stored username then check if the corresponding passwords match
    b. if the corresponding passwords also match then allow the user to progress

There are many finer details that would be involved in implementing this such as:

- validating the user input to make sure they have entered a valid username and a valid password, eg a presence check to make sure they have entered something for both
- ensuring that the loop terminates either when no further usernames are available or when a successful username-password match has been found.