

## Programming Standards Levels 1 & 2

*A discussion of, and example plans & programs for Standards 1.45, 1.46, 2.45, 2.46*

**By**

Anthony Robins & Sandy Garner  
Computer Science  
The University of Otago  
anthony@cs.otago.ac.nz

Version: 1.0 (16 December 2011)

This document: <http://www.cs.otago.ac.nz/staffpriv/anthony/programming-standards-notes.pdf>

Example code: <http://www.cs.otago.ac.nz/staffpriv/anthony/programming-standards-code.zip>

### Introduction

This is a discussion of the standards listed above, including some example plans and programs, and a discussion of the issues that they raise. The document contains the following sections:

- Introduction 1
- The design of the Standards 2
- Step-ups for Standards 1.45 and 1.46 4
- Step-ups for Standards 2.45 and 2.46 5
- A note on the examples 6
- Examples for 1.46 – Checkerboard 6
  - Achieved
  - Merit
  - Excellence
- Examples for 2.45 – Hangman 11
  - Flowchart
  - Excellence
- Examples for 2.46 – Hangman 16
  - Excellence
  - Merit
  - Achieved

**Disclaimer:** *We are not school teachers, NCEA experts, or NZQA / Ministry officials. Teaching resources released via official sources are the definitive guide. All of the following is simply our advice / opinion about how we see the Standards.*

## The design of the Standards

These “programming standards” cover creating a plan for a program (1.45 and 2.45), and using a plan to code / implement a program (1.46 and 2.46).

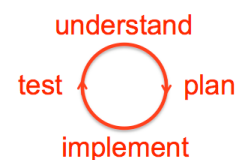
### Cover the standards together

At both levels the standards are **designed to be covered together** (1.45 with 1.46, and 2.45 with 2.46). While it is possible to do one without the other, we suggest that there are many good reasons to cover them together as intended:

- (1) Writing a computer program is a complex interactive process, a continuing cycle of planning, implementation, testing and refinement (see figure).

Covering the standards together allows this process to occur naturally. Covering one without the other creates an artificial situation (cutting a diagonal line across the cycle in the figure that separates understanding and planning from implementation and testing). We think this is pedagogically risky.

#### Programming process



- (2) This material will probably be challenging for students. They need to understand programming language constructs (such as for loops or Boolean conditions) to do either standard, but doing both together gives them twice as much allocated classroom time to come to terms with the concepts.
- (3) Assessing a planning standard (1.45 or 2.45) on its own will be very difficult, because it’s hard to assess the correctness of a plan. Even more problematic, it would be hard for students to assess the correctness / quality of their own plans. Unlike a program, there is no concrete feedback from a “plan”.
- (4) Assessing an implementation standard (1.46 or 2.46) on its own is very difficult, because students must be given an “abstract plan” as a starting point (see the Standards’ explanatory notes). What level of plan (Achieved Merit or Excellence – these will be very different!) and why? There seem to be pedagogical risks to all options (e.g. giving an Excellence level plan to a student who can’t cope could be setting them up to fail).

### Common themes

There are strong similarities and themes running through the Standards.

Level 1 covers certain programming concepts, Level 2 adds modular structure (procedures / methods / functions) and indexed data structures (arrays / lists). But apart from the new material at Level 2, the structure and themes of the Standards at both levels are the same.

Using font and colour as cues, we’ll try and identify the common themes **across** Standards and **over** the step-ups within Standards (compare next three pages).

## Level 1

### Use of programming constructs / procedural structure:

The main theme running across the standards and over the step-ups is how well the required programming constructs (variables, data types, actions, conditions, control structures like loops and selection) have been used.

**Achieved:** The required constructs have been used and everything works.

**Merit:** The constructs are / procedural structure is “well chosen”, i.e. has no obvious faults (and the student worked independently).

**Excellence:** The constructs are / procedural structure is a “well structured, logical solution”, i.e. the plan / program is efficient, hard to improve on. (See Explanatory Note 9 in each Standard.) Also...

The program is flexible and robust, i.e. uses constants, variables, and derived values appropriately. (See Explanatory Note 8 in each Standard, and Level 2 discussion below.)

The distinction between Merit “well chosen” and Excellence: “well structured, logical solution” may be a tricky one, it is the focus of the section “Examples for 1.46 - Checkerboard” (pg 6).

### Testing:

The plan / program needs to be tested. Achieved: On expected inputs. Merit: On expected and boundary inputs. Excellence: On expected, boundary and illegal inputs. For example if expected inputs are 1 to 10 (inclusive) then: expected inputs are e.g. 1, 3, 5, 9; boundary inputs are 0, 1, 10, 11; illegal inputs are e.g. “f”, “hello”, or <return> (no input at all).

### Code layout and comments:

This applies only to program code (so only to 1.46). The code must be set out clearly. The step-ups are based on the quality of the comments.

## Level 2

### Use of programming constructs / procedural and modular structure:

As for Level 1 the main theme is how well the required constructs (which now include at least one indexed data structure (array / list) and modules (procedures / functions / methods)) are used. The language focuses on issues relating to modular structure, but the same issues relating to procedural structure (as for Level 1 above) are also relevant!

There is still a tricky distinction between Merit “well chosen” and Excellence: “well structured, logical solution”. (See Explanatory Note 10 in each Standard.)

At excellence level this theme includes the requirement of specifying / using “variables, constants, and derived values effectively so as to increase the flexibility and robustness of the program”. Basically we don’t want changes in one part / value of the program to cause other parts to break. (See Explanatory Note 9 in each Standard.)

Hopefully the section “Examples for 2.46 - Hangman” (page 11) will also help in both cases.

### Testing: Same as Level 1

**Code layout and comments:** 2.46 only, same as Level 1.

## Step-ups for Standards 1.45 and 1.46

### 1.45 Construct a plan...

#### Achieved

*Construct a plan for a basic computer program for a specified task involves:*

- specifying variables and their data types
- specifying a procedural structure that combines actions, conditions and control structures
- specifying a set of test cases with expected inputs for testing the program.

#### Merit

*Skilfully construct a plan for a basic computer program for a specified task involves:*

- independently constructing the plan
- specifying a procedural structure with well-chosen actions, conditions and control structures
- specifying a set of test cases with expected and boundary inputs for testing the program.

#### Excellence

*Efficiently construct a plan for a basic computer program for a specified task involves:*

- constructing a flexible and robust plan
- specifying an effective procedural structure that constitutes a well-structured logical solution to the task
- specifying a comprehensive set of test cases with expected, boundary and invalid input for testing the program.

### 1.46 Construct a program...

#### Achieved

*Construct a basic computer program for a specified task involves:*

- implementing a plan for a basic program in a suitable programming language
- setting out the program code clearly and documenting the program with comments
- testing and debugging the program to ensure that it works on a sample of expected inputs.

#### Merit

*Skilfully construct a basic computer program for a specified task involves:*

- independently implementing the plan for a basic program in a suitable programming language that uses a procedural structure with well-chosen actions, conditions and control structures
- documenting the program with variable names and comments that accurately describe code function and behaviour
- testing and debugging the program in an organised way to ensure that it works on expected and boundary inputs.

#### Excellence

*Efficiently construct a basic computer program for a specified task involves:*

- constructing a basic program which uses actions, conditions and control structures effectively to increase the flexibility and robustness of the program
- using an effective procedural structure that results in a well-structured, logical solution to the task
- setting out the program code concisely and documenting the program with succinct comments that explain and justify decisions
- comprehensively testing and debugging the program in an organised and time-effective way to ensure the program is correct on expected, boundary and invalid inputs.

## Step-ups for Standards 2.45 and 2.46

### 2.45 Construct a plan...

#### Achieved

*Construct a plan for an advanced computer program for a specified task...:*

- specifying variables, their scopes and data types
- specifying an indexed data structure
- specifying a modular structure for the program, including details of the procedural structures of the modules
- specifying a set of expected input cases for testing the program.

#### Merit

*Skilfully construct a plan for an advanced computer program for a specified task involves:*

- independently constructing the plan
- specifying well-chosen scopes for the variables
- specifying well-chosen parameters for the modules
- specifying a set of expected and boundary input cases for testing the program.

#### Excellence

*Efficiently construct a plan for an advanced computer program for a specified task involves:*

- specifying modules (including their procedural structures) that constitute a well-structured logical decomposition of the task
- specifying variables, constants, and derived values effectively so as to maximise the flexibility and robustness of the plan
- specifying a comprehensive set of expected, boundary and exceptional input cases for testing the program.

### 2.46 Construct a program...

#### Achieved

*Construct an advanced computer program for a specified task involves:*

- implementing a plan for an advanced program in a suitable programming language
- setting out the program code clearly and documenting the program with comments
- testing and debugging the program to ensure it works on a sample of expected input cases.

#### Merit

*Skilfully construct an advanced computer program for a specified task involves:*

- independently implementing a plan for an advanced program in a suitable programming language that uses well-chosen scopes for variables, and well-chosen parameters for modules
- documenting the program with variable and module names and comments that accurately describe code function and behaviour
- testing and debugging the program in an organised way to ensure it works on inputs that include both expected and boundary cases.

#### Excellence

*Efficiently construct an advanced computer program for a specified task involves:*

- constructing an advanced program where the modules (including their procedural structures) constitute a well-structured logical decomposition of the task
- using variables, constants, and derived values effectively so as to increase the flexibility and robustness of the program
- setting out the program code concisely and documenting the program with comments that explain and justify decisions
- comprehensively testing and debugging the program in an organised and time effective way to ensure the program is correct on expected, boundary and invalid inputs.

## A note on the examples

The examples in this document are **not intended to be fully worked** model teaching resources (see Ministry / NZQA resources for that). Nor are they intended to be complete across all Standards and all levels. Instead they are chosen to **focus on certain key issues**, to help interpret and elaborate the Standards. (Examples at each Level will hopefully be useful at both Levels.)

Example code is all in Python, as this seems to be the most popular (text based) language, but code should be clear enough for those familiar with other languages to interpret. We use Python 3, to convert to Python 2 all `input( )` should be `raw_input( )` and all `print(“this”)` should be `print “this”` instead.

## Examples for 1.46 – Checkerboard

This example presents programs relating to Standard 1.46. The point of this example is to explore the interpretation of the main theme of the Level 1 Standards, the **use of programming constructs / procedural structure** as discussed on page 3. In particular the distinction between:

Achieved: just using the constructs,

Merit: “a procedural structure with well-chosen actions, conditions and control structures”), and

Excellence: “an effective procedural structure that constitutes a well-structured logical solution to the task”.

### Task

The example task is to print a square checkerboard pattern of size n. In a graphical environment (e.g. with Scratch) these might be black and white squares, the background to a game. But here using Python and text, we'll just make the pattern out of X and O. So for n = 6, the desired output is:

```
XOXOXO
OXOXOX
XOXOXO
OXOXOX
XOXOXO
OXOXOX
```

In a real program we assume n has been read in from the user and checked, but in the examples below we just set n = 6 for simplicity. Other useful background notes for the examples are below.

## Programs

For the examples on the next 3 pages:

The # comments for each program explain its properties and why it is classified as Achieved, Merit or Excellence.

Each of the examples runs and produces the output shown on page 6.

### Achieved

Note this example would have met the old (1.46 Version 1) Standard's definition of Excellence! It uses nested loops, and complex Boolean conditions. But there is a lot wrong with this code, and the current version of the Standard tries to capture and assess this.

### Merit

This example has none of the obvious problems of the Achieved code. The control structures and conditions are “well chosen”.

### Excellence

This example depends on an insight involving the **sum** of **row** and **column** positions (indexes) as shown for example values to the right. Notice that the summed indices form a checkerboard of odd and even values. Using a test for even numbers (see Note 1 below) we can now write a very compact program, the Excellence example.

	row	0	1	2	3
col	0	0	1	2	3
1	1	2	3	4	5
2	2	3	4	5	6
3	3	4	5	6	7

The insight required to see this solution can be taken as evidence of a well structured logical solution (evidence that the student has thought deeply about the task). The program itself is compact and efficient, it cannot be significantly improved.

### Notes

- (1) We can check if a number is even by seeing it divides by 2 with a remainder == 0. So to test if x is even: `x % 2 == 0` is true for all even values of x.
- (2) If we want (in Python) to print something, then have the next thing printed on the same line, we use `end=""`. So:

```
print('X', end="")
print('0')
```

will create the output:

X0

When using this technique, at the end of a row of printing, we need to:

```
print()
```

to move the point of printing down to start the next row. These techniques are used in the example programs to print the checker-board row by row.

```

### Achieved example #####
#
# The overall approach is to set and use Boolean variables that
# represent the states of the rows and columns, and use these to
# determine what should be printed. This code works but it's abominable!
#
# It uses while loops instead of for loops. It tries to test every
# combination to determine if numbers are even or odd. It uses a sequence
# of redundant “if” statements instead of elif (else if). It uses an
# unnecessary number of Boolean variables, and unnecessary complicated
# Boolean tests. So...
#
# A plan (1.45) would be clumsy and complicated. The code (1.46)
# sets everything out fully and mechanically, clearly missing a real
# understanding of the language constructs involved. It works but
# there's plenty wrong with it, so Achieved.

n = 6
row = 0
while row < n:
    if row == 0 or row == 2 or row == 4 or row == 6 or row == 8:
        even_row = True
        odd_row = False
    if row == 1 or row == 3 or row == 5 or row == 7 or row == 9:
        odd_row = True
        even_row = False
    col = 0
    while col < n:
        if col == 0 or col == 2 or col == 4 or col == 6 or col == 8:
            even_col = True
            odd_col = False
        if col == 1 or col == 3 or col == 5 or col == 7 or col == 9:
            odd_col = True
            even_col = False
        if even_row == True and even_col == True: print('X', end="")
        if even_row == True and odd_col == True: print('O', end="")
        if odd_row == True and even_col == True: print('O', end="")
        if odd_row == True and odd_col == True: print('X', end="")
        col += 1
    print()
    row += 1

```

```
### Merit example #####  
#  
# The overall approach is the same as the Achieved example, trying to  
# set and use Boolean variables that represent the states of the rows  
# and columns.  
#  
# But this example fixes all of the problems with the Achieved. The loops  
# are well chosen (for loops instead of while). The conditions are well  
# chosen - instead of setting out every possibility or combination  
# explicitly we use sensible tests and combinations of Boolean variables.  
#  
# The code (1.46) - and assume also a plan (1.45) - use well chosen  
# conditions and control structures. It's a Merit solution. But the plan /  
# program still does unnecessary work, it's clear how to improve it, so it  
# isn't quite an Excellence solution.
```

```
n = 6  
for row in range(0, n):  
    even_row = False  
    if row % 2 == 0: even_row = True  
    for col in range(0, n):  
        even_col = False  
        if col % 2 == 0: even_col = True  
        if even_row and even_col: print('X', end="")  
        elif even_row and not even_col: print('0', end="")  
        elif not even_row and even_col: print('0', end="")  
        elif not even_row and not even_col: print('X', end="")  
    print()
```

```
### Excellence example #####  
#  
# Depends on the insight that when row + col is even we print "X"  
# and when it is odd we print "0". That insight leads us to a solution  
# that doesn't need multiple Boolean variables and tests.  
#  
# A plan (1.45) would be, and the code (1.46) here is a very compact and  
# efficient solution to the task - a "well structured logical solution".  
  
n = 6  
for row in range(0, n):  
    for col in range(0, n):  
        if (row + col) % 2 == 0:  
            print("X", end="")  
        else:  
            print("0", end="")  
    print()
```

## Examples for 2.45 – Hangman

### Task

Plan (2.45) and implement (2.46) a program that allows the user to play games of Hangman. The player chooses how many games to play (up to some maximum number e.g. 3).

Each game the program chooses a word at random from a hard coded list. The player enters single letter guesses one at a time. After each guess the program provides feedback about whether the guess was correct, and if so where that letter occurs in the word.

The program must cope with words with repeated letters (e.g. “mississippi”). Assume that all letters will be lower case (e.g. the word will never be “Mississippi”).

A game ends when the player has guessed all the letters in the word, or when the player has exceeded the maximum number of guesses (e.g. 12), in which case the player should be told what the word was.

### Note

This task has been left intentionally vague in some respects so as to let the examples illustrate some of the issues that might arise in task interpretation (what is an “expected” input? – what quality of “feedback” is required?).

### Plans

The first example plan (page 12) is in the form of a flowchart. With additional supporting material it could form the main element of a plan, or it could represent a first / early draft of what developed into the second example.

The second example plan (starting page 13) is a fully developed plan for the example Excellence program (see the next section). It consists of explanatory text and pseudocode. A first draft of this plan should be developed before any coding begins. But implementing the code and testing the program will almost always reveal new issues. The plan and the program usually evolve as part of a continuous cycle of understanding, planning, implementation and testing. This text represents a “finished” plan that would have been developed along with the resulting program.

### Programs

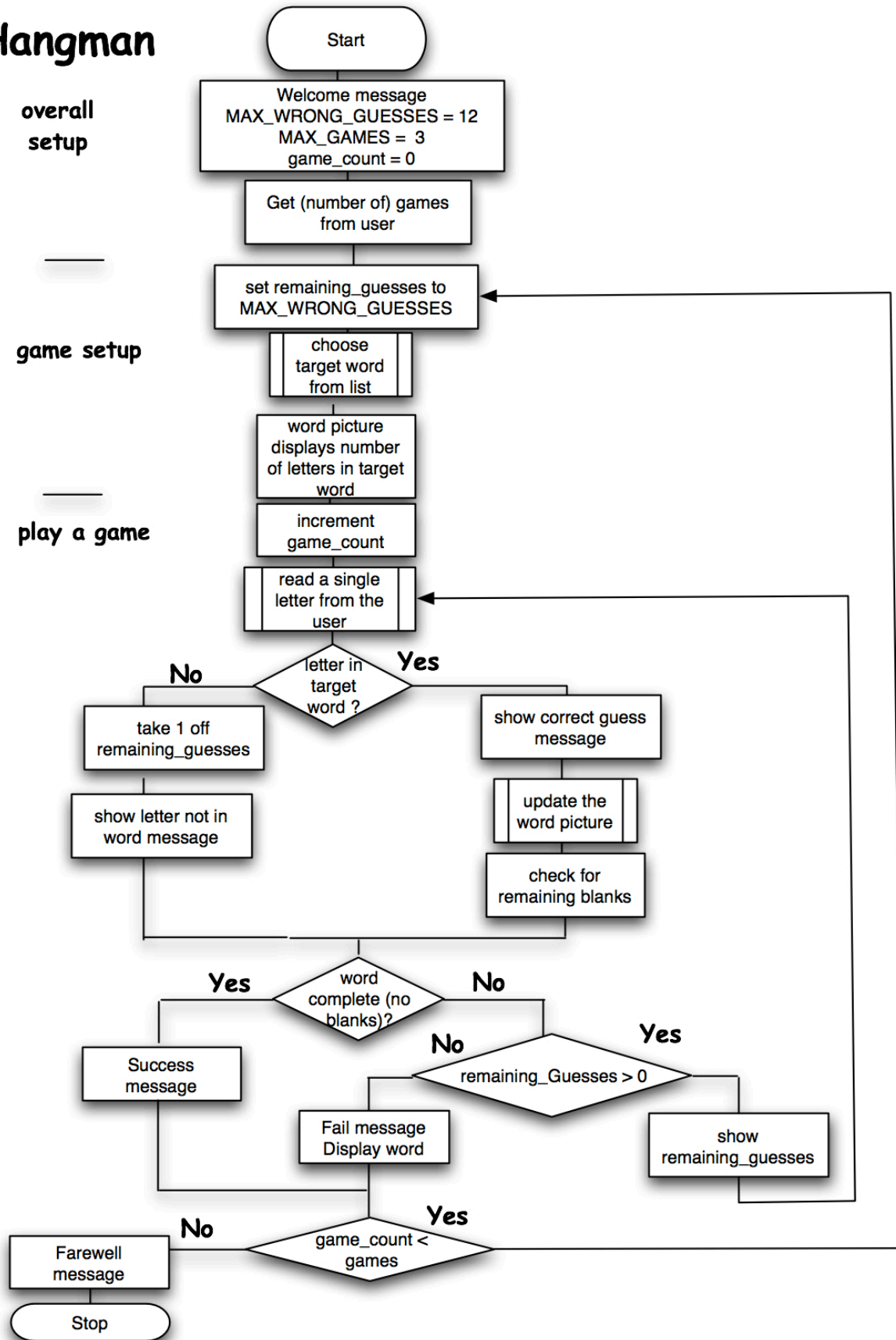
Example programs (2.46) follow in the next section (page 16).

# Hangman

overall  
setup

game setup

play a game



### Excellence plan for Hangman

This is the second example plan described on page 11. For the sake of comparing the plan with the corresponding Excellence program (next section) we will present it in the same order as the program - modules, globals, main routine.

#### Modules / routines

What routines are required? What aspects of the task does it make sense to isolate as discrete modules? (Such modules are typically reused, i.e. called from several places in the program.)

(1) Read in an integer representing the number of games to be played. This is only used once, but the type and range checking required make it a bit cumbersome, making it worth abstracting this task into a module.

Inputs: none.

Outputs: single integer value in range 0 to MAX\_GAMES

Pseudocode:

```
get_games()
    while games is not an integer value in range
        try set games = input integer
        except print an error message "not an integer"
    return games
```

Testing: if MAX\_GAMES is 3 then expected values are 0 - 3, boundary values are -1, 0, 3, 4, and illegal inputs are e.g. x, fred, or <return> (no input).

(2) Choose a random word from the word list. This is used only once, and is currently simple, but it is a discreet task worth abstracting into a module. There could be more complex strategies for choosing a word (e.g. starting with simple words and getting harder). The use of a module allows for this future possibility to be developed without cluttering the main routine.

Inputs: the word list.

Outputs: a single word.

Pseudocode:

```
choose(wordlist)
    rand = random number in range 0 to length of wordlist
    return word in wordList at position rand
```

Note: The use of random number generation requires program to import the random module.

(3) Read in a single letter (representing the Player's current guess). This is called many times per game from the main game loop.

Inputs: a list of letters guessed already so that duplicates can be rejected.

Outputs: a single character (should be a letter).

Pseudocode:

```
get_letter(all_letters_guessed)
    while letter is not a string of length 1 i.e. not a single character
        letter = input()
        if letter in list all_letters_guessed set letter to illegal value so loop repeats
    return letter
```

Testing: Test with expected inputs (single letters) and illegal inputs (too long e.g. 'abc' or too short <return>). Boundary cases doesn't really apply, but test with single characters that are not letters (expected behaviour: an input like '2' will not cause any problems but will just be a wasted guess because it cannot match any letters in the target word).

(4) Given a letter and a word, get the position(s) of the letter in the word (if any). For example given 'a' and 'abba' return [0, 3]. This is called for every letter entered by the Player following (3).

Inputs: a letter and a word.

Outputs: a list of the position(s) of the letter in the word (if any).

Pseudocode:

```
get_positions(letter, word)
  for i from 0 to length of the word
    if word at position i == letter then add position i to positions
  return positions
```

(5) Update a word picture to add a correctly guessed letter at its correct positions. This is called after (3) and (4) if the letter guessed by the player occurs at some positions in the word.

Inputs: the current word picture list e.g. ['a', '\_', '\_'], the list of positions where letter occurs (should never be empty [ ]) e.g. [1], the letter e.g. 'x'.

Outputs: an updated word picture list with the letter added at the specified position(s) e.g. ['a', 'x', '\_'].

Pseudocode:

```
update_picture(word_picture, positions, letter)
  for i from 0 to length of word_picture
    if i is in the list positions change word_picture at i to letter
  return word_picture
```

Note: (4) and (5) could be combined in to a single module which is more efficient (there is no need to preserve a separate list of positions). *[I've kept it this way to aid comparison with the Merit example.]*

## Global variables

These are variables that are accessible to the main routine and to all other modules / routines. They describe the global state of the program (i.e. they do not change from game to game, but apply to every game).

MAX\_GAMES - The maximum number of games a player can play. Should be constant. Suggested testing value 3.

MAX\_WRONG\_GUESSES - The maximum number of guesses allowed in a game. Should be constant. Suggested testing value 12.

wordList - List of words that can be chosen for the game. To keep things simple for testing there are currently only two words in the list ["axe", "mississippi"], but more can be added (no other changes to program code are required).

games - gets set to the number of games the player wants to play 0 to MAX\_GAMES

## Main Routine

The main routine draws on the modules and global variables described above to implement the required number of games. It prints a welcome message and asks the player how many games to play.

Each game starts with a welcome message and a message with a word picture showing the number of letters in the target word. The main game loop begins. Each pass through the loop a single input letter is read from the user. If the letter is in the target word an updated word picture is printed, otherwise the number of guesses remaining is decreased by one (and the user sees a message telling them how many guesses they have left). The main game loop ends when the player has guessed all the letters, or has no guesses remaining (in which case the player is told what the word was).

Note: With respect to the program requirements the main indexed data structures are wordList (each game one element is selected from wordList as the target word), and word\_picture (each correct letter results in setting and changing individual positions / letters). The program also treats individual words like indexed data structures by examining them one position / character at a time.

### Pseudocode:

```
print welcome messages and information about game play
games = get_games() # to read the number of games to play
for the number of games chosen
    # set up the variables for this game
    word = choose(wordlist) # to choose the target word for the game
    remaining_guesses = MAX_WRONG_GUESSES
    word_picture = ['_', '_', ...] for the length of word
    all_letters_guessed is empty at this point
    print a message and starting word picture

    # main game loop reading and processing input letters from player
    while the word isn't guessed and remaining_guesses > 0
        letter = get_letter(all_letters_guessed)
        add letter to all_letters_guessed
        positions = get_positions(letter, word)

        if positions is empty the letter wasn't in the word
            remaining_guesses decreased by 1
        else the letter was in the word
            word_picture = update_picture(word_picture, positions, letter)
            print message and word_picture

    if no more '_' in word_picture then all letters are guessed!
        we're done so set Boolean flag to end the main game loop
        print message of congratulations
    else there are still letters to guess
        print remaining_guesses
        if remaining_guesses == 0 main game loop will end so print message
after the games loop is finished print farewell message
```

## Examples for 2.46 – Hangman

### Programs

These programs implement the Hangman task (page 11). There is an example at each level, Excellence (matching the plan in the previous section), Merit and Achieved. They are similar in structure so as to help relate the three versions, and should be read in order from Excellence to Achieved.

In all versions there are only two words in the list (see task), just to keep it simple. More words can easily be added. The Excellence program is robust, the Merit and Achieved programs would need further changes to accommodate the longer list because of their inflexible design.

**These programs should be run and tested to explore their behaviour.**

The following notes describe their strengths & weaknesses, and discuss the issues that arise.

### Excellence program

The example program below meets all the criteria. (There are two different copies of the Excellence program, one with - possibly excessive - Excellence level comments, and one with few comments to illustrate just the code.)

The use of global constants and derived values (like the length of the word list) means that the program is flexible and robust (add new words to the word list and everything works fine).

When inputting the number of games to play it handles expected, boundary and illegal values. When inputting letter guesses it handles all cases in the sense that it accepts any single character and the program executes as planned. But it doesn't check that the input is actually a lower case letter in range 'a' .. 'z', so an input like '2' will just be treated as an incorrect guess. The program could be improved by rejecting inputs that are not letters.

*Nice features:*

The task requirements are vague when it comes to behaviour and output, and this program exceeds a minimum interpretation. It has features which are not specified, but which are obviously sensible. For example it: keeps a record of letters guessed so far so as to avoid duplicates; and it presents feedback to the player relatively nicely in the form of a simple “word picture”, e.g. [ 'm', '\_', 's', 's', '\_', 's', 's', '\_', '\_', '\_', '\_' ].

Excellence level students will often *exceed a minimum interpretation* of the task like this and include “nice features”.

The standard doesn't explicitly account for such nice features, but they could be recognised within the first Excellence level bullet point which covers “a well-structured logical decomposition of the task”. Nice features can be taken as evidence of a well structured logical decomposition (evidence that the student has thought deeply about the task).

## Merit program

The example program meets all the criteria (if the student worked independently). The output about the positions of correctly guessed letters is not user friendly, e.g.:

```
Yes that is in the word at the following positions [2, 3, 5, 6] .
```

(The player would probably need to keep track with a pen and paper.) But overall the program meets the specification of “providing feedback”.

But note the hard coded values (especially the length of the word list in the choose routine). That’s not a flexible and robust design - if new words are added to the list they will never be used unless the code is changed! Comments and testing are also not up to Excellence level, so this is definitely Merit.

When inputting the number of games to play it handles expected and boundary values (an illegal value like ‘x’ crashes the program). When inputting letter guesses it could be improved in the same way as the Excellence example above.

### *Remaining issue:*

You can fool the program. It keeps a count of the number of letters guessed correctly, and when that equals or exceeds the length of the word it assumes that all letters are guessed correctly. But nothing prevents a player from entering the same correct letter multiple times, for example entering ‘a’ three times when the word is ‘axe’ the program counts three correct letters and tells the player they have guessed correctly.

There was nothing in the specification about this, and the program meets the criteria and works correctly for expected inputs (if we expect a player not to enter letters more than once), so it’s a Merit program. But spotting and fixing this kind of problem / unexpected input is evidence of Excellence (see Excellence *Nice features*).

## Achieved program

As per the comments in the code, the example program was created by taking the Merit example and breaking it in multiple ways to illustrate different kinds of problem (while preserving the overall structure of the program that should be familiar by now). Real student examples will probably have a lot of variation!

Despite the multiple code problems the example (with one required fix) works, so it meets the criteria. Currently the program chooses the same word every game. But that is easy to fix (move one line of code), so you’d probably point it out to the student and get them to fix it, then the program could be considered working.

In terms of testing, the program only works for expected inputs for number of games (and there is no upper limit!). Lower boundary and illegal inputs cause it to crash.

*Remaining bug:*

The program still contains a significant bug. It's a more serious version of the problem noted in the Merit example, where the player can enter just one correct letter multiple times to “win” a game. In fact, if the word is mississippi and the player enters only “s” repeatedly, they are told that they have entered 4 of 11 letters correctly, then 8 of 11, then 12 of 11, then 16 of 11, and so on – the program never stops!

The Merit program used a “>=” test on the length of the word, so it would stop (in this example) at 12 of 11 correct.

This Achieved program uses an “==” test, which is often unsafe for exactly this kind of reason!

We would still consider the program “working” with this bug, but spotting and fixing such issues is evidence of thinking beyond Achieved level.

```
#####
## Written by Anthony Robins, 7 December 2011, using Python 3.2.1.

## This program implements a game of Hangman. The Player chooses how many games to play.

## For each game the program selects a target word from a hardcoded list. The player
## enters a sequence of letters to try and guess the word. After each guess the player
## gets feedback in the form of a "word picture", e.g. having guessed 'a' which is in the
## word (at the first position of a three letter word):
## ['a', '_', '_']
## where '_' represents a letter not yet guessed. This form of word picture is used because
## it is informative and easy to implement. It could be displayed in a less cluttered
## format with a minor extension to the program.

## A game ends when all letters are guessed correctly, or after MAX_WRONG_GUESSES
## incorrect guesses.

## Note that positions in lists are indexed from 0, so in ['a', 'b', 'c']
## the element 'a' is at position 0.

## Note: All letters in the game are lower case e.g. 'mississippi',
## or input checking would need to handle both cases e.g. 'm' and 'M'.

## The program could be further improved by adding an ASCII graphic output representing
## the incorrect guesses, by presenting the "word picture" in a more readable format,
## by merging get_positions and update_picture into one routine (there is no need to
## preserve a separate list of positions), or by doing better checking of user input - see
## routine get_letter.

## [Even for excellence level the comments in this version are pretty over the top!]

import random

#####
# Routines:

# Returns an integer in the range 0 to MAX_GAMES representing the number of games to be played.
def get_games():
    # Start with an illegal value then loop until a legal value is entered.
    games = -1
    while not games in range(0, MAX_GAMES + 1): # Range goes up to not including, so +1 to include.
        print("Please enter the number of games you want to play from 0 to", MAX_GAMES,)
        # Try to set a new value for games, but check that the input is an integer.
        try:
            games = int(input())
        except ValueError:
            print("That wasn't an integer, please try again!")
            # Games remains unchanged at -1 so the loop will repeat.
    return games

# Returns a single word chosen at random from input list.
def choose(wordList):
    rand = random.randrange(0, len(wordList))
    return wordList[rand]
```

```
# Returns a single character. Nothing prevents the player from entering a character that
# isn't a lower case letter, e.g. '2'. In this case the program continues fine, but the player
# has wasted a guess. The program could be further improved by checking to see that the input
# is a legal lower case letter 'a' .. 'z'.
```

```
# Takes as input a list of all letters guessed so far. This is used to reject duplicate inputs.
```

```
def get_letter(all_letters_guessed):
    letter = ""
    # Loop until a single character is read that hasn't been used already.
    while len(letter) != 1:
        letter = input("Please enter a single letter: ")
        # If the input is a duplicate the loop will repeat
        if letter in all_letters_guessed:
            print("You already guessed that letter, please try again")
            letter = ""
    return letter
```

```
# Returns a list of the positions that letter is found in word.
# For example letter = 'a' and word is 'abac' then returns [0, 2]
```

```
def get_positions(letter, word):
    # Initially the list of found positions is empty.
    positions = []
    # Check each position in the word.
    for i in range(len(word)):
        # If the character at this position == the input letter then add position to list.
        if word[i] == letter:
            positions += [i]
    return positions
```

```
# Returns a list representing a word picture (of the state of correct guesses so far).
# Only called when word picture needs updating with a new correct letter.
# Takes as input the current word picture, the list of positions where letter occurs,
# and the letter to be added at these positions.
```

```
def update_picture(word_picture, positions, letter):
    # Consider each position in the word picture.
    for i in range(len(word_picture)):
        # If the current position is in the list of positions...
        if i in positions:
            # .. change the picture from "_" to letter at this position.
            word_picture[i] = letter
    return word_picture
```

```
#####
```

```
# Global variables:
```

```
# These apply to all games.
```

```
# Note: Python has no simple way of declaring a constant, so here we just follow
```

```
# the common naming convention of all caps (indicating that MAX_WRONG_GUESSES
# should never change though in fact nothing prevents it).
```

```
MAX_GAMES = 3 # The maximum number of games a player can play. Should be constant.
```

```
MAX_WRONG_GUESSES = 12 # Max number of guesses allowed in a game. Should be constant.
```

```
wordList = ["axe", "mississippi"] # List of words that can be chosen for the game.
```

```
games = 0 # Gets set to the number of games the player wants to play (0 to MAX_GAMES)
```

```
#####
# Main routine:

print("Welcome to Simple Hangman!\n")
print("You will be asked how many games you want to play.")
print("In each game I choose a word and you try to guess one letter at a time.")
print("In each game you only get 12 incorrect guesses!\n")
games = get_games() # Asks player how many games to play.

# Play the number of games requested (range goes up to not including, so +1 to include)
# If games == 0 then no games will be played, proceed to the statement after for loop.
for game_count in range(1, games + 1):
    # Initialise the variables that apply to this game.
    word = choose(wordList) # Target word for this game is chosen randomly from wordList
    guessed = False # Set to True if the player has guessed all the letters in the word.
    remaining_guesses = MAX_WRONG_GUESSES
    word_picture = ["_" for i in range(len(word))] # Make initial word picture all "_".
    all_letters_guessed = [] # Each letter guessed is added to list so duplicates can be avoided.

    print("\nWelcome to game ", game_count)
    print("I have chosen a word for you to guess:\n", word_picture)

    # Main loop for this game, repeats until guessed is True or there are no guesses left.
    while( (not guessed) and (remaining_guesses > 0) ):
        # Ask the player for a letter, and add it to the list of guesses so far.
        letter = get_letter(all_letters_guessed)
        all_letters_guessed += [letter]
        # Get a list of positions where the letter occurs in the target word.
        positions = get_positions(letter, word)

        if positions == []:
            # The letter occurred in no positions (list is empty), so that's a guess gone.
            remaining_guesses -= 1
            print("Sorry that letter is not in the word")
        else:
            # The letter occurred in one or more positions, update and display word picture.
            word_picture = update_picture(word_picture, positions, letter)
            print("Yes that is in the word - so far you have guessed\n", word_picture)
            # Now check to see if all letters in the target word are guessed.
            # This is true when the word picture contains no more "_" blank positions.

            if not "_" in word_picture:
                # The word is guessed, set the flag guessed = True to end the game loop.
                guessed = True
                print("Congratulations - you have guessed the word correctly!")
            else:
                # The word is not guessed, print remaining guesses (if 0 the game loop ends).
                print("You have", remaining_guesses, "guesses left")
                if remaining_guesses == 0:
                    print("Sorry, game over. The word was:", word)

# When all games have been played we end up here.
print("\nThanks for playing - goodbye!")
```

```
#####  
# Excellence program, second version, with few comments, to focus on the code.  
  
import random  
  
def get_games():  
    # Start with an illegal value then loop until a legal value is entered.  
    games = -1  
    while not games in range(0, MAX_GAMES + 1):  
        print("Please enter the number of games you want to play from 0 to", MAX_GAMES,)  
        try:  
            games = int(input())  
        except ValueError:  
            print("That wasn't an integer, please try again!")  
            # Games remains unchanged at -1 so the loop will repeat.  
    return games  
  
def choose(wordList):  
    rand = random.randrange(0, len(wordList))  
    return wordList[rand]  
  
def get_letter(all_letters_guessed):  
    letter = "  
    # Loop until a single character is read that hasn't been used already.  
    while len(letter) != 1:  
        letter = input("Please enter a single letter: ")  
        if letter in all_letters_guessed:  
            print("You already guessed that letter, please try again")  
            letter = "  
    return letter  
  
def get_positions(letter, word):  
    positions = [  
    for i in range(len(word)):  
        if word[i] == letter:  
            positions += [i]  
    return positions  
  
def update_picture(word_picture, positions, letter):  
    for i in range(len(word_picture)):  
        # If the current position is in the list of positions...  
        if i in positions:  
            # .. change the picture from "_" to letter at this position.  
            word_picture[i] = letter  
    return word_picture  
  
MAX_GAMES = 3  
MAX_WRONG_GUESSES = 12  
wordList = ["axe", "mississippi"]  
games = 0
```

```
print("Welcome to Simple Hangman!\n")
print("You will be asked how many games you want to play.")
print("In each game I choose a word and you try to guess one letter at a time.")
print("In each game you only get 12 incorrect guesses!\n")
games = get_games()

for game_count in range(1, games + 1):
    word = choose(wordList)
    guessed = False
    remaining_guesses = MAX_WRONG_GUESSES
    word_picture = ["_" for i in range(len(word))] # Make initial word picture all "_"
    all_letters_guessed = []

    print("\nWelcome to game ", game_count)
    print("I have chosen a word for you to guess:\n", word_picture)

    # Main loop for this game, repeats until guessed is True or there are no guesses left.
    while( (not guessed) and (remaining_guesses > 0) ):
        letter = get_letter(all_letters_guessed)
        all_letters_guessed += [letter]
        positions = get_positions(letter, word)

        if positions == []:
            remaining_guesses -= 1
            print("Sorry that letter is not in the word")
        else:
            word_picture = update_picture(word_picture, positions, letter)
            print("Yes that is in the word - so far you have guessed\n", word_picture)

        if not "_" in word_picture:
            guessed = True
            print("Congratulations - you have guessed the word correctly!")
        else:
            print("You have", remaining_guesses, "guesses left")
            if remaining_guesses == 0:
                print("Sorry, game over. The word was:", word)

print("\nThanks for playing - goodbye!")
```

```
#####
## This program implements a game of Hangman. The Player chooses how many games to play.

## For each game the program selects a target word from a hardcoded list. The player
## enters a sequence of letters to try and guess the word. If a letter is correct the
## player is told the positions of the letter in the word. If a letter is incorrect
## the player is told this.

import random

# Returns a single word chosen at random from input list
def choose(wordList):
    rand = random.randrange(0, 2) # There are only 2 words in the list at the moment.
    return wordList[rand]

# Returns a single letter entered by the user (may not be an actual letter!).
def get_letter():
    letter = ""
    while(len(letter) != 1):
        letter = input("Please enter a single letter: ")
    return letter

# Returns a list of positions in the word where we found the letter.
def get_positions(letter, word):
    positions = []
    for i in range(len(word)):
        if word[i] == letter:
            positions += [i]
    return positions

# Main routine:

wordList = ["axe", "mississippi"] # List of words that can be chosen for the game.

print("Welcome to Simple Hangman!")
print("\nPlease enter the number of games you want to play from 0 to 3")
print("In each game I choose a word and you try to guess one letter at a time.")
print("In each game you only get 12 incorrect guesses!\n")

# Get the number of games to play.
games = -1
while not games in range(4): # Games must be 0 1 2 or 3
    print("Please enter number of games")
    games = int(input())

# Play 1 2 or 3 games
for game_count in range(1, games + 1):
    word = choose(wordList)
    print("\nWelcome to game ", game_count)
    print("I have chosen a word for you to guess. It is", len(word), "letters long.")

    guessed = False # Set to True if the player has guessed all the letters in the word.
    remaining_guesses = 12
    n_correct_letters = 0
```

```
while( (not guessed) and (remaining_guesses > 0) ):

    # Ask the player for a letter and get the positions of the letter.
    letter = get_letter()
    positions = get_positions(letter, word)

    # If the letter occurs in no positions one guess is gone, if it does occur print positions.
    if positions == []:
        remaining_guesses -= 1
        print("Sorry that letter is not in the word")
    else:
        print("Yes that is in the word at the following positions", positions)
        n_correct_letters += len(positions) # number of correct letters increases by number of positions
        print("You have guessed", n_correct_letters, "out of", len(word), "letters correctly so far")

    # If the number of correct letters is equal to the letters in the word then the word is guessed!
    # [Actually that's not necessarily true - see notes - Anthony]
    if n_correct_letters >= len(word):
        guessed = True
        print("Congratulations - you have guessed all the letters correctly!")
    else:
        print("You have", remaining_guesses, "guesses left")
        if remaining_guesses == 0:
            print("Sorry, game over. The word was:", word)

print("\nThanks for playing - goodbye!")
```

```
#####
## The comments in this program are NOT the kind of comments I would expect from a student,
## they are my comments pointing out what is wrong.

## This example should be read AFTER the Excellence and Merit examples.

## I wanted this program to be recognisably similar to the Excellence and Merit examples, so I have
## taken the merit example and broken it a bit to produce a simpler, clumsier program with bugs.

## So this program still uses two modules OK, and still uses some sophisticated thinking in the routine
## get_positions.

## There could be huge variation in Achieved level programs and an actual example might be
## considerably clumsier than this!

import random

def get_letter(games): # This routine is passed an input parameter that it doesn't need/use!
    letter = ""
    while(len(letter) != 1):
        letter = input("Please enter a single letter: ")
    return letter

def get_positions(letter, word): # Same as Excellence and Merit examples just for convenience.
    positions = []
    for i in range(len(word)):
        if word[i] == letter:
            positions += [i]
    return positions

wordList = ["axe", "mississippi"]
r = random.randrange(0, 2) # r is used to choose a word at random from the list but this statement
    # is in the wrong place! Wrong scope. It should be inside the game loop.
    # As it is set here the program chooses the same word every game.
    # This might not be spotted if the list of words was longer.
    # This would be easy enough to point out to the student and fix, then
    # this bug would be fixed. (PS - r is also a lousy name)

g = 12 # This statement is redundant, see declaration inside game loop.
n = 0 # This statement is redundant, see declaration inside game loop.
# These statements also make the variables global, they don't need to be, so it's poor scope.

print("Welcome to Simple Hangman!")
print("\nPlease enter the number of games you want to play")
print("In each game I choose a word and you try to guess one letter at a time.")
print("In each game you only get 12 incorrect guesses!\n")

print("Please enter number of games")
games = int(input())
game_count = 1
```

```
while game_count <= games: # While loop is a clumsy choice here.
    # If "<" instead of "<=" another bug would exist.
    word = wordList[r] # see comment about r above.
    print("New game, the word has this many letters", len(word))

    guessed = False # guessed is never used anywhere - this line should be deleted
    # guessed was obviously left over from a discarded idea!
    g = 12 # For remaining guesses, g is a lousy name.
    n = 0 # For number of correct letters, n is a lousy name.

    while( g > 0 ): # Number of guesses will be used as the only exit condition - see below.
        letter = get_letter(games) # This routine is passed an input parameter that it doesn't need/use!
        positions = get_positions(letter, word)

        if positions == []:
            g = g - 1
            print("Sorry that letter is not in the word")
        else:
            print("Yes that is in the word at the following positions", positions)
            n = n + len(positions) # number of correct letters increases by number of positions
            print("You have guessed", n, "out of", len(word), "letters correctly so far")

        if n == len(word): # Should be ">=" see discussion of bugs in the notes.
            g = 0 # signals that the word is correctly guessed by setting the number of guesses left to 0
            # This is bad design - we can't tell if the game ended with them running out of guesses
            # or guessing correctly!
            print("Congratulations - you have guessed all the letters correctly!")
        if n != len(word): # Much simpler if this line is an "else:" rather than redundant if.
            print("You have", g, "guesses left")

    game_count = game_count + 1

print("Sorry, game over. The word was:", word) # Player sees this even if they guessed correctly!
print("Thanks for playing - goodbye!")
```