

Final

by Ryan Easton

Submission date: 24-May-2026 10:37PM (UTC+0100)

Submission ID: 284525280

File name: 66125_Ryan_Easton_Final_727104_708726957.docx (270.93K)

Word count: 3016

Character count: 14057

ACCA4020 Introduction to Computer Programming

Ryan Easton

Introduction

Requirement for the program

Implementing the code provided by the flowchart

Testing

Correcting the mistakes

Reflection

Introduction

The student has been tasked with creating a code designed to track how many spaces are available in a car park and to keep track of how many spaces are used and how many are free. The student has been given a flowchart to implement the code. The student will then use troubleshooting methods to find and remove various problems in the code to make sure the code works as intended. Once the code has been corrected, the student will then test the code against a few test cases provided to see how the code functions under controlled circumstances. The student will then improve the logic of the code to ensure the code is working efficiently and without mistakes, both in syntax and in logic. The student will then reflect on the journey of implementing this code and the challenges of this code and how the challenges have been overcome.

Requirements for the program.

The program has been given to the student by another team, who has made a flowchart detailing what the team wanted for the code. The requirements are as follows:

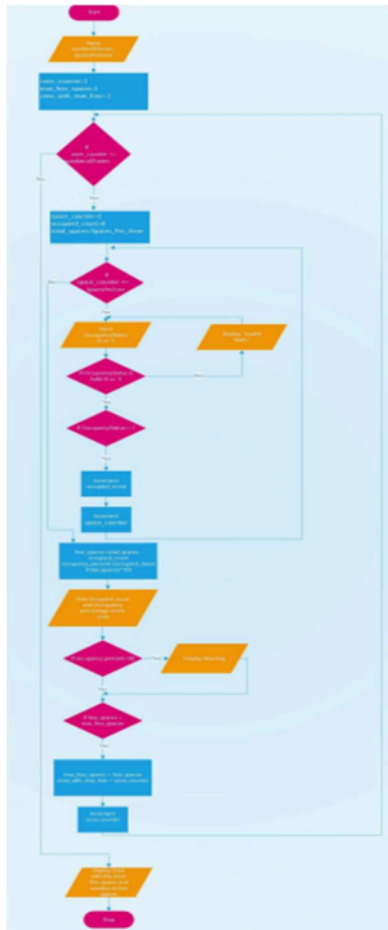
“Ask the user to input: Number of zones Number of parking spaces per zone for each space,

prompt whether it is: Occupied (1) Free (0)

For each zone: Count the number of occupied spaces. Calculate the percentage of occupied spaces. Display a warning if occupancy exceeds 80%.

For all zones: Display the zone with the most free spaces.

Only accept valid inputs (0 or 1); re-prompt if invalid.”



The student was given a flowchart to translate into python code, and into a working program. This structure is easy to follow and implement. The flowchart has given a clear start-to-finish and is clear on what part of the code is supposed to go where. However, the flowchart also has a few spelling errors. These spelling errors can break a code completely. For instance, in the flowchart, there is a “spacesPerZone” variable. This variable is implemented to keep an integer that was entered into the code without it being locked into the code. This code will then keep it stored until the code is finished. There is another instance of this variable; it is however named “spaces_Per_Zone”. The student understands that this variable is supposed to be the same as the “spacesPerZone” variable, but Python does not recognize that as the same variable. This is one of the problems found in the flowchart.

Implementing the Code

The flowchart given to the student was translated into python. The student had copied the flowchart word for word but could not get the code to function. This was because the flowchart given had a few inconsistencies regarding the naming of Variables, alongside some spelling errors. These spelling errors were making different variables despite looking like it should be part of the same variable. For example, the Variable "spacesPerZone" would dictate a variable for this specific name. There is also a Variable in the flowchart names "spaces_Per_Zone". This variable dictates a different variable, even though it is clearly supposed to be the same variable as the "spacesPerZone" variable. With a Human's logic, this is clearly supposed to be one-in-the-same, a variable that dictates how many zones are necessary to work with. But a machine works very logically, without much in the way of being "close enough" to one of the variables to group them together. The machine will look at these 2 variables as completely different variables instead of these being close enough to assume that these are supposed to be the same. Another problem that was found by the student was near the end of the flowchart, there is a variable named "free_spaces" that needed to update its integer to find which zone had more spaces free. The problem lies in the simple fact that there is no variable to update. This is simple to fix, simply add a variable named "free_spaces" to make this variable usable and allow it to update and allow the program to function as intended. There are also misspellings in the "display Invalid status" line. This does not matter for the machine as it will just print out the letters, but for a Human that is noticeable. This does not impact the code's functionality, and this is an extremely easy fix. The flowchart also does not do anything with the occupancy status being set to 0. This does not impact the code but for the purposes of tracking which space is occupied, and which will help the code to accurately print out correct numbers.

The student has translated the code into Python. The student will now turn the flowchart into functional code by adding the relevant variables. These variables will allow the code to dynamically change while the program is running. This saves the programmer from having to change every single number in the program manually and allows the program to remember certain characters and numbers. This allows different characters to be added to the program itself instead of editing static numbers, saving time and effort for both the user and the programmer. For example, if the numbers were fixed, then the program would only work with those numbers. If something were to change, like the par park getting bigger or smaller, the program will not be able to adapt to the change as the numbers are already fixed to the previous par park, not the new car park.

The student then uses loops to ensure that the program can determine how many zones are necessary, how many spaces are in the zone, and determine which zone has more space available. The loops are essential to ensure that the program can determine if the

inputs are valid, determine if the inputs are valid before proceeding to record the inputs to look back on for future calculations. These calculations will let the program determine if a zone has more free space. This is one of the requirements given to the student.

Next is the mathematical aspect of coding. The student will determine what needs to be added, subtracted, divided, and multiplied for the program to function as intended. The student will implement this for the loops and for the zone with the highest free spaces.

Testing

To ensure that the program works efficiently, the student will test the program using a series of tests. These tests are designed to test the validation of inputs and to make sure that the program works as intended. These tests will reveal if there are any syntax problems or logical problems. The student will then be able to fix these problems and make the program more efficient. To test the program, the student will use various inputs, some logical, some absurd, in order to check the program's validation and see if the program will allow a non-number input into a 0 or 1 only input. The tables below are the test cases for the program.

Test case	zones	spaces	Input per zone
a	2	4	Zone 1: 1,1,0,0, Zone 2: 0,0,1,1
b	2	3	Zone 1: 1,1,1 Zone 2: 1,1,1
c	1	4	Zone 1: 1,2,0, %
d	2	3	Zone 1: 1,0,1 Zone 2: 0,0,1

These test cases are designed to be controlled environments. The point of these tests is to show how the program functions under certain circumstances.

Test Case a

```

input the number of zones:2
input the number of spaces per zone:4
space 1 zone 1
is space 1 occupied 1=yes 0=no: 1
space 1 is occupied
space 2 zone 1
is space 2 occupied 1=yes 0=no: 1
space 2 is occupied
space 3 zone 1
is space 3 occupied 1=yes 0=no: 0
space 3 is not occupied
space 4 zone 1
is space 4 occupied 1=yes 0=no: 0
space 4 is not occupied
space 5 zone 2
is space 5 occupied 1=yes 0=no: 0
space 5 is not occupied
space 6 zone 2
is space 6 occupied 1=yes 0=no: 0
space 6 is not occupied
space 7 zone 2
is space 7 occupied 1=yes 0=no: 1
space 7 is occupied
space 8 zone 2
is space 8 occupied 1=yes 0=no: 1
space 8 is occupied
Summary
zone 1 has the most free spaces
free spaces = 4
total spaces = 8
4 occupied
50.0% space used
Process finished with exit code 0
|

```

“Test case a” functions mostly as intended. The spaces count as they should, and the zones count as the zones should increment. The zone should be increased when the space limit is found. The space limit is inputted by the user. In this case, the zones should be increased after space 4. the space, however, should reset back to 1 as it is a new zone with a new loop. The zones count after 4 loops, which is correct as the spaces per zone has 4 inputted. There were 2 zones inputted and the program looped 2 times. Zone 1 does not have the highest free space zones as both zones have an equal number of zones free, being 2 each and 2 spaces occupied. The student can fix this by adding a line of code that checks if both zones are equal and prints a line of text recognizing this. The free spaces were calculated correctly as well as the total number of zones present. The percentage of space used is also correct as 4/8 is 50% of the total.

Test Case B

```

input the number of zones:2
input the number of spaces per zone:3
space 1 zone 1
is space 1 occupied 1=yes 0=no: 1
space 1 is occupied
space 2 zone 1
is space 2 occupied 1=yes 0=no: 1
space 2 is occupied
space 3 zone 1
is space 3 occupied 1=yes 0=no: 1
space 3 is occupied
space 4 zone 2
is space 4 occupied 1=yes 0=no: 1
space 4 is occupied
space 5 zone 2
is space 5 occupied 1=yes 0=no: 1
space 5 is occupied
space 6 zone 2
is space 6 occupied 1=yes 0=no: 1
space 6 is occupied
Summary
zone 1 has the most free spaces
free spaces = 0
total spaces = 6
6 occupied
100.0% space used
warning! high amount of spaces occupied

Process finished with exit code 0
|

```

“Test Case b” as is the case with “Test Case a” has similar problems. The numbers count as they should, the spaces increment as intended, the zones count as intended, the same problem as “Test Case a” because the zone with the highest free spaces doesn’t correctly determine that both are equal. As mentioned in Test Case a, this is easily fixed with a line of text determining if the zones are of an equal amount. Test b shows that the warning prompt works for the program because the program should look at the percentage of space, and it exceeds 80 percent. It should display a warning that the zone has a lot of space filled.

Test Case c

```

input the number of zones:1
input the number of spaces per zone:4
space 1 zone 1
is space 1 occupied 1=yes 0=no: 1
space 1 is occupied
space 2 zone 1
is space 2 occupied 1=yes 0=no: 2
Invalid status, try again
space 2 zone 1
is space 2 occupied 1=yes 0=no: 0
space 2 is not occupied
space 3 zone 1
is space 3 occupied 1=yes 0=no: %
Traceback (most recent call last):
  File "F:\Python Report\Report.py", line 17, in <module>
    occupancyStatus = int(input(f'is space {space_counter+1} occupied 1=yes
ValueError: invalid literal for int() with base 10: '%'

Process finished with exit code 1
|

```

Test case c showcases a different problem, and this problem is from a different one from both tests "a" and "b". This problem is a validation problem. The problem lies in adding a non-number to the input that asks for either 0 or 1. as it is not a number, the program does not know what to do with it, since the lines of relevant code need an integer to function correctly. Otherwise, the program will instead crash, which is not ideal for both the program and the programmer. This is not as easy a fix as the previous problem as this will need a full validation code. A Validation code is a few lines of code that ensure that someone cannot enter something the student doesn't want them to enter. To do this, the code will check what is entered and allow the input to be saved if it is valid. If it is not, the lines of code will block the entry and ask for a valid entry. If the next input is not valid again, the cycle will repeat until a valid input is entered. The second problem is in the loop itself. The loop is in a fixed number after it is entered by the user. If there is an invalid number entered, the loop continues even though it is supposed to ask to input a valid number. The test asked for a 2 to be added. The 2 was not valid, but the loop continued regardless. The loop asked for 4 entries and 4 entries were given. This is not a problem with the loop's logic but Moreso a problem with correcting the loop's number to incorporate a faulty entry. This can be fixed by ensuring that the loop doesn't continue until the number inputted is valid. The reason the program gave a crash report was because the input was set as an integer. So, when an input was added that was not an integer was inputted, the program couldn't process the input as the program was looking for an integer, not a string, which is what forced the program to stop. This test uses a different symbol for an input. this showcased that the validation was not present and needed to be fixed.

Test Case d

```
input the number of zones:2
input the number of spaces per zone:3
space 1 zone 1
is space 1 occupied 1=yes 0=no: 1
space 1 is occupied
space 2 zone 1
is space 2 occupied 1=yes 0=no: 0
space 2 is not occupied
space 3 zone 1
is space 3 occupied 1=yes 0=no: 1
space 3 is occupied
space 4 zone 2
is space 4 occupied 1=yes 0=no: 0
space 4 is not occupied
space 5 zone 2
is space 5 occupied 1=yes 0=no: 0
space 5 is not occupied
space 6 zone 2
is space 6 occupied 1=yes 0=no: 1
space 6 is occupied
Summary
zone 1 has the most free spaces
free spaces = 3
total spaces = 6
3 occupied
50.0% space used
Process finished with exit code 0
```

Test Case d works a lot better than the other test cases as the program works completely as intended. The spaces increment as intended. The zones count on the correct loop, after 3 valid inputs the zone increments by 1. The free spaces are correctly added, and the total spaces are correctly added. The total space occupied is correct, and the percentage of space used is also correct. 3 free spaces out of 6 total spaces are 3/6. This is 50% of the total space used. This test was not as strenuous as the other tests as this test simply asks for 2 zones with 3 spaces. The other tests had asked for a number that should not be submitted or a symbol that should not be allowed to be entered.

```

input the number of zones:2
input the number of spaces per zone:2
space 1 zone 1
is this space occupied (1=yes, 0=no):1
space 1 is occupied

space 2 zone 1
is this space occupied (1=yes, 0=no):2
Invalid input, please enter 1 or 0.
space 2 zone 1
is this space occupied (1=yes, 0=no):0
space 2 is not occupied

space 1 zone 2
is this space occupied (1=yes, 0=no):%
invalid input, try again
space 1 zone 2
is this space occupied (1=yes, 0=no):1
space 1 is occupied

space 2 zone 2
is this space occupied (1=yes, 0=no):1
space 2 is occupied

Summary
zone 1 has the most free spaces
free spaces = 1
total spaces = 4
3 occupied
75.00% space used

Process finished with exit code 0
|

```

The student will now look back on the code and implement the necessary fixes and correct the mistakes the test cases have shown. The student has fixed the verification issue within the code by using While loops instead of for loops. This allowed the program to dynamically loop on itself without incrementing the loop amount. This means that, if there is a problem, like if the user decides to put in a letter or a symbol that is not a number instead of a number, the program can now repeat the process until a valid input is inputted. The program will then continue the process without fault. Another change made is the use of continue keyword. The continue command will allow the process to skip certain processes, if necessary, then move on unless another keyword stops it. Another fix that the student has implemented is the wrong zone being displayed as having the freest space. The student initially had the zone_counter variable the same as the max_with_zone_free variable. This sets the zone_with_max_free variable to whatever the zone_counter was at the time. This is a logical error as the program itself worked as intended but not what the programmer intended. Instead, the logic has been updated so that the zone_with_Max_free variable now checks how many free spaces there are in a loop. It will then keep a number of how many free spaces there were and allow the loop to go again. It will then check its saved number against the new number in the 2nd loop and save the bigger number. This will give an accurate number instead of just giving the same number as the zone_counter. Using the .2f for the percentage formats gives 2 decimal points. This doesn't change much in the code, but makes the percentage read better.

Reflection

The student had some difficulties with the code functioning and the flowchart. The first problem the student faced was reading and following the flowchart's direction to translate into python code. The problem lies in changing logical progression by human standards and translating it into machine language. Following a flowchart itself is not difficult, but the decision of what lines of code to use to emulate the same progression was difficult. Questions like "should this be used as an if statement or a while statement?" and "how many variables are in this flowchart and how and where do these variables need to be implemented" were initially difficult to place. This problem was circumvented by using "trial-and-error" testing, which can be time consuming. The student has learned a lot from these problems, such as problem solving and troubleshooting. The problem-solving aspect is the trial-and-error style of testing and allows the student to find problems efficiently. An example of this is found in test cases "a" and "b". The problem found was the lack of attention to where a variable and its increment was in the code. Through trial and error, the student realized that the variable was not correctly displaying the correct number. Also, a problem found in Test Case "c" shows that there was no validation in regard to inputting symbols and letters. Instead of retrying the loop and allowing the user to try a new input, the script crashed. This was because the input line was an integer input line, and there was no "try / except" structure. The try/except structure is designed so that the code that is nested in the "try" structure will attempt to run as much as it can, provided the input is correct. If the input is not correct, the "except" structure will display a message, saying "invalid input, try again" and looping back to the input section of the code. This allows the code to function if there is an input that is not 0 or 1. Another thing the student had focused on was cleaning up the code. Initially, the code was messy, with a lot of redundant statements where while would've been more useful. Cleaning up the code allows the code to run more efficiently without needing to check and re-check lines of code that were not doing much anyway.

The student had noticed that the flowchart had a few errors in it. The student had found that interpreting the errors while adding what the flowchart needed was stressful and the student felt like the problems found in the code could have been from either the flowchart or from the student's programming. The student had still dealt with the problems given by the flowchart and their own logical errors. The flowchart also had some spelling errors, such as `print(invalid statis)` and naming variables "spacesPerZone" and "spaces_Per_Zone". The student used camelCase to ensure that they do not make the same mistake, in addition to renaming the variables given to the same naming conventions.

The test cases have proven to be very effective in learning what the problems were with the program. Test case a and b showed that there was a logical error in the code. This would've

been found eventually, but the student had found the error a lot sooner than using the Trial-and-error style. The tests were also useful because it allows the programmer to see how efficiently the code functions under a controlled environment. The student had found a problem with the spaces not resetting back to 1 after a loop has been fully completed. This was fixed by changing the space counter to outside of the if statements and adding it to the end of the while statements. This is because initially, the space counter was inside the if statement, and as long as the condition was met, it would increment. Now it will increase as intended and reset as intended.

Final

GRADEMARK REPORT

FINAL GRADE

GENERAL COMMENTS

75/100

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

GRADING FORM: ACCA4020 RUBRIC 2026

RYAN EASTON

75

TASK 1A [20]

Correct implementation of flowchart (even with errors) [8] Code runs (even if incorrect output) [5] All test cases attempted [4] Screenshots/evidence provided [3]



The flowchart is mostly implemented as given, and test cases are executed. Evidence of some errors (e.g., invalid input or incorrect outputs) is shown, although screenshots or outputs could be clearer.

15

TASK 1B [15]

Correct outputs recorded [5] Identification of error types [5] Clear explanation of errors [5]



Some errors were identified, but important issues such as validation or equal-zone logic are explained. However these could have been improved by adding the proper screenshots.

12

TASK 2A [20]

Errors correctly fixed [10] Improved logic (loops, conditions) [6] Input validation added [4]



Most errors are fixed, including validation and general logic improvements. However the reporting of the validated outputs could have been improved by adding the screenshots and the corresponding explanation.

15

TASK 2B [20]

All test cases re-run [6] Outputs clearly recorded [6] Comparison with old results [4] Explanation of fixes [4]



Most test cases are re-tested with generally correct outputs. Improvements are evident, though comparisons or explanations could be more detailed with outputs.

15

TASK 3 [10]

Understanding of difficulty [5] Impact on design/efficiency [5]



Good explanation of difficulties and some reference to improved approaches using data structures. Shows reasonable understanding of efficiency

7

TASK 4 [15]

Identifies challenges [5] Examples from assignment [5] Reflection on testing/learning [5]



11