|  |  |
| --- | --- |
| **Testing** | **Evidence** |
| In version one I set up the program with a pre-made recipe so I could focus on getting the scale factor to work. The user inputs how many people the recipe should serve, then the program divides this by how many people it originally served to find the scale factor. This is what all the values in the amounts list will be multiplied by.  I used two basic scale factors to test that this was working – one third of the serves and double the serves. These both created the expected scale factor. |  |
| In version two I focused on the next component of the code – altering the amounts of each ingredient in the recipe using the scale factor. I looped through the amounts variable and multiplied each value in the list by the scale factor. This new value is appended to a list of all the new amounts. Then the new recipe is outputted, with each ingredient and its relevant unit and amount on one line. At this point, n/a still shows up.  To test this I used the one third and double scale factor again, because it is easy to see that the numbers are correct. |  |
| In version three I started to replace the hard-coded recipe with user input. For this version I was focusing on the ingredients and their units. I trialled using a menu-based input function, however I came across some issues with this:   * It is easy for the user to get mixed up when inputting, and put the units and ingredients out of order so they don’t correspond to each other. * My output method relies on ingredients, units, and amounts being the same length for everything to get outputted correctly. If ingredients is shorter, not everything will get printed. If it’s longer, an error message will show.   I decided that this was not the best way to get input from the user. |  |
| In version four, I replaced the menu-based input with a different method. If the user does not want to input an ingredient they press quit, otherwise it carries on to the units for them. This ensures that the lists are the same lengths, and the units and ingredients pair up well. The program still wasn’t perfect, though, because there are other aspects I need to take out of hard coding. |  |
| In version five, I added the amounts and the original servings to the recipe\_input function. I decided to do this in a different version because I had to make sure that I was still able to perform calculations on these amounts (they had to be float and int). Now that I had gotten rid of the hard-coded recipe, I could comment out the print statement crediting the original recipe. Also, now that the entire recipe was user-inputted, there were no more errors with the list length in output. |  |
| In version six I added error handling for the boundaries. This was only relevant for the scale factor. So, if the new amount of servings that the user entered creates a scale factor that is greater than three or smaller than one quarter, the program stops them and prompts them to enter a different amount. I put the new servings input down with the scale factor calculation, and put both in a while True loop. This loops infinitely until the user enters a new serving amount that leads to a scale factor that is between 0.25 and 3. Only then will the infinite loop break and the program will carry on to alter the amounts and output the new recipe. |  |
| In version seven I added exceptional error control for the float and int inputs. I created two functions – force\_int and force\_num. Force\_int will loop infinitely until the user enters a whole number, while force\_num will do the same except the user can enter a whole number or a decimal. Either way, it’ll get converted to a float when it is stored. I replaced the input statements with these functions for the input of the ingredient amounts and the serving inputs (both old and new).  There are two testing screenshots shown here. One is with entirely expected results to make sure the base program was still working. Then, the second is me typing in random letters, floats when there’s meant to be ints, leaving inputs blank, etc. Anything to try and break the program. |  |
| Version eight is what I plan to be the final version of this program. I’ve gone through and deleted any unnecessary comments and print statements to make the code easier to read. Usability wise, I added a welcome/instruction message so the user knows what to do. I tried to keep it simple to not overwhelm them, though, as I’ve tried to make the input prompts as descriptive as possible. Also, for the final recipe output, I made an adjustment so that when you write n/a for the unit it will format the output differently and not include any unit in that line. This allows for things like “2 eggs” and “3 cups of flour” to exist in the same recipe. |  |
| For my final testing, I ran through my testing plan (bottom of the planning document). First, I started with expected numbers for the ingredient amount input. Then I inputted strings and left it blank, and these both reacted as expected. Then I inputted expected amounts for the serving input. Then I tested an expected boundary for the scale factor (I didn’t mean to do it out of order, I was just testing the serving input). Then I tested invalid serve inputs (floats, strings, empty input fields). Then I tested expected scale factors (1/3, 2). Then I tested the remaining expected boundary (1/4). After that I was meant to test unexpected boundaries, but I realised that 0.24 would be hard to create. So, instead, I did 1/5 and 3.1. With the boundary unexpected values done, it was time for the exceptional unexpected testing of the scale factor. I tested a scale factor of 7, then 0.1. | [video is below the table or can be found in the screenshots folder] |

