Compound interest calculator:

First need to define some variables…

```
initial_input = float(input("What is your starting value? ")) #allows user to input starting cash value
interest_annual = float(input("What is your annual intertest rate as a percentage? ")) #asks user to input annual interest rate
term_length = "" #sets term_length variable to empty so it can later be defined
term_type = "" #same again
term_number = 0 #same again
```

```
while term_length == "": #This start a loop where term_length is empty and so it will keep running through the programme until defined.
    term_length = str(input("Is it daily/monthly/yearly compounding? ")).strip().lower() #Asks the Q and lets user answer. strip removes any spaces or special characters
    if term_length == "monthly": #If user inputs monthly, term_length is set to month and term number is equal to 12 as this will be used to calculate monthly interest r
        term_type = "month"
        term_number = 12

    elif term_length == "yearly": #same as previous if command, when user returns something other than monthly, the prev. step will skip and come to this
        term_type = "year"
        term_number = 1

    elif term_length == "daily": #same again
        term_type = "day"
        term_number = 365

    else: #if user enters anything but monthly, daily or yearly it will come to this step
        print("Invalid input please respond with daily, monthly or yearly") #lets user know input was invalid
        term_length = "" #sets term_length back to empty so loop will run again
```

Use a while command to open a loop, so while term_length is empty (which it is as we pre-defined it) this loop will run and ask user to input whether it is daily, monthly or yearly compounding. The code then defines the term type and term number based on the input term length. Code will check each if and elif statement in order, if neither are correct, it will run the ‘else’ code.

```
term_length = "" #sets term_length back to empty so loop will run again
```

Previously this line was not there, which meant the term_length was not reset and therefore the program would skip to the next part which would result in…

```
Is it daily/monthly/yearly compounding? g
Invalid input please respond with daily, monthly or yearly
How many 's would you like to calculate for? 12
```

There is a missing word in the last line as our next input line requires term_type to be defined else it will stay empty…

```
term_amount = int(input("How many " + term_type + "'s would you like to calculate for? "))
```

This also meant term_number was not defined and as such when turned to float(term_number) it would be 0.0 and would cause errors in the later calculations (since it would be used to divide the annual interest rate by the term number and divisions by 0 are not able to be done).

```
acting_interest = (interest_annual/term_number)
acting_interest = ((acting_interest/100) + 1)
```

Sets a new variable to be used in the calculation. First line divides* the annual interest by the term number to give the equivalent monthly or daily interest (yearly will jsut be divided by 1 and so stay the same). Second line divides by 100 and adds 1 to give the multiplier value. E.g. 12% annual interest divided by 12 is 1% monthly interest, as a decimal this is 0.01 but the multiplier of the original value is 101% so we multiply by 1.01 in the calcualtion.

*Same reason we had problems when term_length = 0

```
contribution_check = "" # makes new variable and sets it to empty
contribution_check = input("Any monthly contributions? Yes/No ").strip().lower() #
while contribution_check != "": # checks if new variable is empty, if it is the fo
    if contribution_check == "Yes".strip().lower(): #if yes the following line wil
        monthly_contribution = float(input("Monthly contribution amount: "))
    elif contribution_check == "No".strip().lower(): #if no the following line wil
        monthly_contribution = float(0.0)
    else:                                                  # if neiher yes or no, the
        print("Invalid input please respond with yes or no.")
        contribution_check  = "" #sets the variable to empty again so the program
        monthly_contribution = "bad input"

    print(monthly_contribution)
```

This part is similar to the loop before, program is intended to check whether user has replied yes or no or repeat the question, if user replies yes they are then prompted to provide the monthly contribution amount.

However becuase the line

```
contribution_check = input("Any monthly contributions? Yes/No
").strip().lower()
```

Came BEFORE the while loop, and the while loop says while contribution check is not empty to run th eloop, the loop would run continuously.

So to fix it this line had to be placed just under the while line and the while loop changed to only run if it IS empty e.g. the variable is empty so it asks to give a yes or no answer and then runs depending on that answer. Giving the following corrections...

```
while contribution_check == "": # checks if new variable is empty, if it is the following program will run
    contribution_check = input("Any monthly contributions? Yes/No ").strip().lower() # asks user if there are monthly contributions
    if contribution_check == "Yes": #if yes the following line will run
        monthly_contribution = float(input("Monthly contribution amount: "))
    elif contribution_check == "No": #if no the following line will run
        monthly_contribution = float(0.0)
    else:                                          # if neiher yes or no, the following line will run
        print("Invalid input please respond with yes or no.")
        contribution_check  = "" #sets the variable to empty again so the program will start again from 'while'
        monthly_contribution = "bad input"

    print(monthly_contribution)
```

Encountered one more error as I realised the strip().lower() was useless for the if and elif commands when it is already applied to the input, however I was then meant to change from “Yes” to “yes” and “No” to “no” else the input always puts Yes to yes and No to no meaning the condition can never be satisfied. Easy fix.

Note – the print at the end is just to check the value is correct and is only temporary.

Now we have the monthly contribution we can calculate the totals.

**at this point I realised I had only allowed for monthly contributions and not daily or yearly contributions but decided to carry on for now and think about altering the program at a later date to include these as it would become increasingly complex. BUT if there is an option for monthly contributions it needs ot be able to be applied to daily, monthyl or yearly compound calculations to give correct totals.

Therefore if we call our monthly contribution = x, then in the case that compounding is yearly, but contributions are monthly, we can have annual_contribution as 12x (since the montlhy contribution is not compounded till the end of the year we can just write it as the annual contribution.

In the scenario that we are daily compounding but adding a monthly contribution it is slightly more complex.

The idea I had in mind was to make a new variable that be a vector of days in the month with each day being ‘0’ and the 30th day being ‘1’. Then run a for loop for each day when there is a 1 you can add to the total. You can set the number of days in a range to run the for loop for. Below is some testing of codes for this concept:

```python
test_vert += [1]
print(test_vert)
checker_thing = ""
for numb in test_vert:
    if numb == 0 and checker_thing == "":
        print("this is zero")
    else:
        print("this is 1")

print(2*test_vert)

test_new = 45 % 10
print(test_new)
max = 3
cheese_v = range(0,max)

for numbers in cheese_v:
    print("shush")
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

shush
shush
PS C:\Users\surya> & C:/Users/surya/AppData/Local/Microsoft/WindowsApps/python3.11.exe c:/Users/surya/#test.py
[0]
[0, 1]
this is zero
this is 1
[0, 1, 0, 1]
5
shush
shush
shush
```

Testing the range and modulus functions^

```python
while contribution_check == "": # checks if new variable is empty, if it is the following program will run
    contribution_check = input("Any additional contributions? Yes/No ").strip().lower() # asks user if there are monthly contributions
    if contribution_check == "yes": #if yes the following line will run
        monthly_contribution = float(input("Monthly contribution amount: "))
    elif contribution_check == "no": #if no the following line will run
        monthly_contribution = float(0.0)
    else:                                           # if neiher yes or no, the following line will run
        print("Invalid input please respond with yes or no.")
        contribution_check  = "" #sets the variable to empty again so the program will start again from 'while'
        monthly_contribution = "bad input"
```

Here we are going to add a new input to ask whether the contribution is daily/monthly or yearly, we will need a new variable t store the contribution type as well and we will call this contribution_type...

```
contribution_type = input("Is the contribution daily, weekly, monthly or yearly? ").strip().lower()
if contribution_type not in( "daily", "weekly", "monthly", "yearly"): #checks if the users input is
    contribution_type = str("Invalid input. Please respond with daily, weekly, monthly or yearly") #
    print(contribution_type) #prints in terminal to let user know input is invalid
elif contribution_check == "yes" and contribution_type == "monthly": #if yes the following line will
    additional_contribution = float(input("Monthly contribution amount: "))
elif contribution_check == "yes" and contribution_type == "daily":
    additional_contribution = float(input("Daily contribution amount: "))
elif contribution_check == "yes" and contribution_type == "yearly":
    additional_contribution = float(input("Yearly contribution amount: "))
elif contribution_check == "yes" and contribution_type == "weekly":
    additional_contribution = float(input("Weekly contribution amound"))
```

Added contribution_type to be defined by user and changed from montlhy_contribution to additional_contribution to avoid having to make more variables.

**Encountered a problem as if you respond ‘no’ to whether there is a contribution, the program would still carry on asking for contribution amount.

```
if contribution_check not in ("yes", "no"):
    print("Invalid response please respond yes or no")
elif contribution_check == "yes":
    contribution_type = input("Is the contribution daily, weekly, monthly or yearly? ").strip().lower() #asks user to define the frequency of contributions
```

Fixed this by adding a previous check whether there is a response to the contribution question and then to only run the program if contribution check is responded with ‘yes’.

This also meant the ‘and’ and ‘or’ sections became redundant because we’ve already ran the checks for contribution check. Also could remove the else statement and the last Elif line could be brought into the main program again:

```
while contribution_check == "": #starts a loop for while contribution check is empty to run the following program
    contribution_check = input("Any additional contributions? Yes/No ").strip().lower() #asks user to input yes or no, not that
    if contribution_check == "yes": #check if user responded 'yes'
        contribution_type = input("Is the contribution daily, weekly, monthly or yearly? ").strip().lower() #if user responded y
        while contribution_type not in( "daily", "weekly", "monthly", "yearly"): #starts another loop if the input is invalid
            print("Invalid input, please respond daily, weekly, monthly or yearly") #prints message to say input is invalid
            contribution_type = input("Is the contribution daily, weekly, monthly or yearly? ").strip().lower() #repormpts the u
        if contribution_type == "monthly": #checks if it is equal to 'monthly'
            additional_contribution = float(input("Monthly contribution amount: ")) #if so it prompts user for the monthly contr
        elif contribution_type == "daily":
            additional_contribution = float(input("Daily contribution amount: "))
        elif contribution_type == "yearly":
            additional_contribution = float(input("Yearly contribution amount: "))
        elif contribution_type == "weekly":
            additional_contribution = float(input("Weekly contribution amount: "))
    elif contribution_check == "no": #if user responds no, this program will run
            additional_contribution = float(0.0) #contribution is set to zero
            contribution_type = "No additional contribution"
    else:
        print("Invalid input. Please respond yes or no") #if user responds anything other than yes or no the next line will run
        contribution_check = "" #this sets contribution check back to empty meaning our while loop at the top will run again
```

Fixed code, new while loop added so that if user enters an invalid response to the contribution type, they will be asked again. The program runs as follows:

1. User is asked if there are any additional contributions
2. If user replies ‘no’, the additional contribution is equalled to 0 and contribution type is no additional contributions
3. If user enters ‘yes’, the user is then asked for the contribution frequency (daily, weekly, monthly or yearly)

4. If user doesnt respond with one of these 4 inputs, it will print ‘Invalid input’ and ask for the contribution frequency again.
5. When user responds with one of the 4 contribution frequencies, they will be asked to provide the amount of the contribution and this will be stored in the variable ‘additional_contributions’

```
if contribution_type == "monthly": #checks if it is equal to 'monthly'
    additional_contribution = float(input("Monthly contribution amount: "))
elif contribution_type == "daily":
    additional_contribution = float(input("Daily contribution amount: "))
elif contribution_type == "yearly":
    additional_contribution = float(input("Yearly contribution amount: "))
elif contribution_type == "weekly":
    additional_contribution = float(input("Weekly contribution amount: "))
```

This can actually be reduced to one line and have the float ask the question dependant on the users input still…

```
if contribution_type in ("daily", "weekly" "monthly", "yearly"): #checks if it is equal
    additional_contribution = float(input(contribution_type + " contribution amount: "))
```

Uses the contribution type to ask the question in the input command instead of having a different line for each different input.

```
term_type = "days"
term_amount = int(input("how many days would you like to calculate for? "))
term_schedule = [0] #creates a new variable and sets it as a lsit

for day in range(1, term_amount): #starts a loop for each 'day' in term amount
    if day % 30 == 0: #if the remainder of the day when divided by 30 is 0, the next line will run
        term_schedule.append(1) # this will add a 1 into the list
    else: # if the remainder is more than 0 the next line will run
        term_schedule.append(0) #this will add a zero

print(term_schedule) #the result is that every 30th day will have a 1 and not a 0, therefore later
#case of monthly contributions but daily compounding
```

In a test IDE running this new program whereby the days are put into a list with a 0 or a 1 depending on the reaminder when divided by 30, this would later be used to add a monthly contribution for each time there is a 1. To allow more felxibility, instead of using ‘30’ we can add a variable that can be based on the contribution type e.g. weekly would be multiples of 7.

We have all the current tools to calculate the final output, and now just struggling with how to print the calculations on a term-by-term basis, for now we will calculate the total output.

For this we need a separate variable for the interest rate on the monthly contributions, as if we a re compounding monthly but contributing yearly, the interest rate for the

contribution should be equal to the yearly interest rate, whilst the monthly interest rate should be equal to the yearly interst rate/12

```
if contribution_type == "daily":
    contribution_term_number = 365
elif contribution_type == "weekly":
    contribution_term_number = 52
elif contribution_type == "monthly":
    contribution_term_number = 12
elif contribution_type == "yearly":
    contribution_term_number = 1
else:
    contribution_term_number = term_number

contribution_interest = (interest_annual/contribution_term_number)
contribution_interest = ((contribution_interest/100) + 1)
```

This will assign a number to a new variable for the contribution term number, which will be used to calculate the contribution interest rate from the annual interest, using the same formula as before but replacing the variable to equal contribution_interest instead of our preivous acting_interest to avoid overwriting the previous variable.

Next we need to convert the contribution frequency to the same units, which we already have all the variables for so we can use the following equation:

```
contribution_frequency = round(term_amount*(contribution_term_number/term_number))
```

1. Where term_amount is the user's previous input for how long to calculate for e.g. user is prompted 'how many months do you want to calculate for?' and they type 36 , this is equal to 36
2. Contribution_term_number is the frequency that the contribution will take place in a one year time period e.g. if this is daily, then it's 365, monthly is 12 etc...
3. Term_number is the frequency of compounding in a one year period, so even though user input 36 months, in a one year period this would still be equal to 12.

For an example, we will do 36 months compounding and calculate the amount of contributions if we are contributing once per year

36 months to calculate for * (1 year in every year /12 months in every year) = 3 years

Now that is everything needed for the calculation for the total new balance, we can just write the equation out...

```
contribution_compound = (additional_contribution * ((contribution_interest**contribution_frequency) - 1))/(contribution_interest - 1)
compound_total = (initial_input*(acting_interest**term_amount)) + contribution_compound #calculates the initial input compounded and
```

Where the first line calculates the sum of the monthly contributions compounded and the second calculates the sum of the initial input compounded plus the monthly contributions total to give the new total balance.

```
print("Your new total is: " + str(round(compound_total, 2)))
```

This line prints the new total for the user.

Then added more prints to give the user more information on including, EAR (Effective Annual Rate), percentage increase and total earned...

```
deposit_total = (initial_input + (contribution_frequency * additional_contribution)) #calculates total
print("Your total deposits are: " + str(deposit_total)) #prints total deposits
print("Your new total is: " + str(round(compound_total, 2))) #prints a message saying your new total is
print("You have earned " + (str(round(compound_total - deposit_total, 2))) + " interest.") #calculates
increase_percentage = (100 * (compound_total/deposit_total)) - 100 #calculates the percentage increase
print("This is an increase of " + str(round(increase_percentage, 2)) + "%") #prints the percentage incr
effective_annual_rate = ((1 + ((interest_annual/100)/term_number))**term_number) - 1 #calcualtes effect
print("Your effective annual interest rate is: " + str(round(100 * effective_annual_rate, 2)) + "%") #p
```

For an example where someone opens a bank account with a 3.45% annual interest rate, monthly compounding and deposits an initial £500, with additional regular monthly contributions of £150 and we want to work out the total after 2 years (24 months)...

```
Your total deposits are: 4100.0
Your new total is: 4257.24
You have earned 157.24 interest.
This is an increase of 3.84%
Your effective annual interest rate is: 3.51%
```

Since we already previously made contribution_term_number to deal with different formats of time given e.g. days, months years etc...

We will create an emtpy list and for ranges 0 up to the amount of time we are calculating for (in our case the term_amount) we will run a for program...

This program will check if the term is divisble by the contribution_term_number (e.g. 30 for months, 365 for days 7 for weeks etc...)

If it is, then it will put a 1 in the list, and if not it will put a 0 in the list...

```
term_list = []
for term in range(1, (term_amount + 1)):
    if term % (contribution_term_number) == 0:
        term_list.append(1)
    else:
        term_list.append(0)
```

(We do term_amount + 1 because otherwise it will not calculate the final term number e.g. if term_amount is 24, it will run from 1 up until 23, when it is 24 it no longer satisfies the range and therefore stops the loop.

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0
, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0
, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
```

This is the wrong output as it is printing ‘1’ every 12 days and not every 30 days

```
term_number = 365
contribution_term_number = 30
term_amount = 365
contribution_frequency =round(term_amount*(contribution_term_number/term_number))
print(contribution_frequency)


term_list = []
for term in range(1, (term_amount + 1)):
    if term % (contribution_frequency) == 0:
        term_list.append(1)
    else:
        term_list.append(0)

print(term_list)
print(len(term_list))
```

The resulting code would fix it for certain circumstance but ran into issues when the term number was less than the contribution term number – it would generate a list of the term number evne if it was smaller (for example it would make a list with 12 integers) and therefore if we contribute daily, there is 365 days in the year and we cant fit that information into a list of 12 length.

```
range_finder = max(term_number, contribution_term_number)
term_list = []
for term in range(1, (range_finder + 1)):
```

Therefore I included a new variable range_finder which would be equal to whichever is higher out of the two term numbers and use that in out for loop, so the highest number will always be what the list is created to be the size of.

```
if term % (contribution_frequency) == 0:
    term_list.append(1)
```

Still getting problems as this line is only printing a 1 if the remainder is equal to 0, since in the test case contribution_frequency is 730, it only prints a 1 on the 730th number in the lsit, where it should print every day.

Okay there was a lot of trial and error so I will try to simplify the now working code ins teps…

1. Need to find the largest set of data that the time is stored as e.g. 730 days compounding, 24 months contributing, in this case the largest set is 730. We used the range_finder to do this between the term_amount and the contribution_frequency. This way it does it between the amount of time passed for compounding and the number of contributions added. Therefore if the contributions added are daily but compounding is only monthly or yearly, we still use the amount of days to represent the data.
2. Run a 'for' loop for each number in the range from 1 to our range finder + 1. The reason we use teh plus one is because the range command will calculate only up to the number defined and not including the defined number.
3. Inside the loop we have an 'if' command where it will run a line of code depending if the condition is met: <if the number in the range is divisible by the interval of each contribution with 0 remainder>. It will add a 1 to our term list. First we need to calcualte the time between each contribution (contribution interval) and can do this by dividing the amount of time calculated for formatted into the smallest units we are using (range_finder) by the amount of contrbutions (contribution_frequency) and rounding down to the nearest integer.
4. Inside the same 'if' line we divide the number from the range by the contribution interval and check if the remainder is equal to zero, if it is we can add a 1 to the list.
5. We then have an else command, in the case the number is not divisible by the contribution interval with no remainder, we add a 0 to the list.
6. This entire thing is also within an 'if' command to ensure it only runs when there is an additional contribution, if there is not, we have an else command at the end to make sure the list fills with 0s.

7. New working code below...

```
term_list = [] #creates a new variable and sets it as an empty list
range_finder = max(contribution_frequency, term_amount) #this finds the highest
                                                        #It will help define ho
                                                        #For example, if we onl
                                                        #We would be unable to
if contribution_check == "yes".strip().lower(): #check if there is additional c
    for term in range(1, (range_finder + 1)): #runs a loop for each number from
                                              #e.g. if the range is 12 months, th
        if term % (range_finder//contribution_frequency) == 0: #range_finder//c
                                                               #as an example,
                                                               #so then dividi
                                                               #in this exampl
                                                               #in this case t
                                                               #then it checks

            term_list.append(1) #if there is no remainder, we add a 1 into the
        else:
            term_list.append(0) #if there is a remainder, then the if term is r
else: #if there is no additional contribution...
    for term in range(1, (range_finder + 1)): #for each term in range
        term_list.append(0) #add a 0 to the list
```

Next we want to be able to rpint these results on a term by term basis, so in our new list for every ‘1’ we want to times the value by the acting interest rate and add the contribution. If it is ‘0’ we dont add the contribution. We can run the code like this...

```
running_total = initial_input
running_deposit = initial_input

for integer in term_list:
    if integer == 1:
        running_total = (running_total*(acting_interest)) + additional_contribution
        running_deposit += additional_contribution
        print(str(round(running_total, 2)) + "      " + str(running_deposit))
    else:
        running_total = (running_total*(acting_interest))
        print(str(round(running_total, 2)) + "      " + str(running_deposit))
```

Running total and running deposit are created to keep track of value over time. Since they both start at the value which the user input at the start we cna equal it to initial_input. Since we are calculaitng term by term, the term length for each calulcation is 1, meaning the interest rate is to the power of 1. As this has no effect on the value we can leave this out.

One problem we ran into with this code is that the acting_interest is the interest per term for the compounding, and as such may be different to the interest of the contribution. E.g. If we have yearly compounding at 3.5% annual interest rate, the program will set acting_interst to 3.5%/1 which is still 3.5%, however if we have monthly contributions and therefore the terms we are calcualting for are monthly, the acting_interest will be too high. To fix this, we need to make a new variable that converts the interest rate to the suitable term.

```
contribution_interest = interest_annual/contribution_term_number
```

Since we already had the user define the annual interest and the number of contributions per term, we can calculate the interest rate for the contribution term.

We then need to tell the program which interest to use so we can use a min code to determine which is the smaller value, as this will be the one with more terms and as such be our acting interst in the term-by-term basis.

```
running_interst = min(acting_interest, contribution_interest)
```

And then changin our for loop before, replacing acting_interest with running_interest...

```
for integer in term_list:
    if integer == 1:
        running_total = (running_total*(running_interest)) + additional_contribution
        running_deposit += additional_contribution
        print(str(round(running_total, 2)) + "      " + str(running_deposit))
    else:
        running_total = (running_total*(running_interest))
        print(str(round(running_total, 2)) + "      " + str(running_deposit))
```

```
contribution_interest = ((interest_annual/contribution_term_number)/100) + 1
```

^Fixed format of contribution interest to be in same format as acting_interest (percentage/100)

Now I want to be able to print the information in an orderly table with headings and clearly defined areas for each heading.

```
header = ("| " + "Term Number" + " | " + "Deposited" + " | " + "Interest Earned" + " | " + "New Total" + " |"
print(header)
underline = ""
for char in header:
    if char == "|":
        underline += "|"
    else:
        underline += "-"
print(underline)
```

In a test, we made a variable called header and ran a for loop for each character in header, so that if there is an “|” the underline will also print this, else it will print a “-” the result is this…

```
| Term Number | Deposited | Interest Earned | New Total |
|-------------|-----------|-----------------|-----------|
```

Since we also want to print ther term number and the interest earned, we will need to create new variables which calculate for these.

```
else:
    running_total = (r
term_counter = 1 term_counter += 1
    term_counter += 1
```

To count the terms we simply made a new variable called term_counter starting at 1, for each iteration of the for loop this will go up by 1.

```
interest_earned = 0
```

```
interest_earned = running_total - running_deposit
```

```
else:
    running_total = (running_total*(running_interest))
    term_counter += 1
    interest_earned = running_total - running_deposit
```

Interest earned is equal to the new total minus the deposit total for each term and so this is also put into the for loop to update with each iteration.

```
print("|", term_counter + " " * (11 - len(term_counter)), "|",
    str(Deposited) + " " * (9 - len(str(Deposited))), "|",
    str(Interest_earned) + " " * (15 - len(str(Interest_earned))), "|",
    str(New_total) + " " * (9 - len(str(New_total))), "|")
```

Again, in a test, testing the print function whereby each item is printed and then followed by and amount fo spaces equal to the amount of characters in the header it belongs to minus the amonut of charcaters in the item in string form. E.g In the ‘Deposited’ Header, this is 9 letters long and 500 is 3 characters long, so we print the 500 and add 6 spaces to make sure the text is aligned within the box that contains the header. Giving the following for this example…

```
| Term Number | Deposited | Interest Earned | New Total |
|-------------|-----------|-----------------|-----------|
| month 3     | 500       | 50              | 550       |
```

I wanted to make the term number ‘month 3’ be capitalised for the first letter so ran a short for loop...

```
term_type = "month"
term_table_type = ""
first_letter = "yes"
for char in term_type:
    if first_letter == "yes":
        char = char.upper()
        first_letter = "no"
        term_table_type += char
    else:
        char = char.lower()
        term_table_type += char
```

Where we have avaribale called first_letter set to “yes”, the for loop says if first_letter is set to “yes”, capitalise this character, and then changed first_letter to no so it wont run again. INstead it will run the else command which says to have this charcter in lower case. This is stored in the new variable term_table_type which is then used to replace term_type in the print command.

Input these into normal program and changed to respective variables...

```
for integer in term_list: #checks for each integer in the list for a 1 or a 0
    if integer == 1: #if the integer is 1
        running_total = (running_total*(running_interest)) + additional_contribut
        running_total = round(running_total, 2) #rounds running total to nearest
        running_deposit += additional_contribution #running deposit is the pervio
        term_counter += 1 #adds one to previousvalue of term counter, ensuring it
        interest_earned = round((running_total - running_deposit), 2) #calcualtes
        term_printer = term_table_type + " " + str(term_counter) #updates term_pr

        print("|", term_printer + " " * (11 - len(term_printer)), "|", #prints ea
            str(running_deposit) + " " * (9 - len(str(running_deposit))), "|", #
            str(interest_earned) + " " * (15 - len(str(interest_earned))), "|",#
            str(running_total) + " " * (9 - len(str(running_total))), "|")
    else:                                                                  #w
        running_total = (running_total*(running_interest)) # running total is equ
        running_total = round(running_total, 2)
        term_counter += 1
        interest_earned = round((running_total - running_deposit), 2)
        term_printer = term_table_type + " " + str(term_counter)
        print("|", term_printer + " " * (11 - len(term_printer)), "|",
            str(running_deposit) + " " * (9 - len(str(running_deposit))), "|",
            str(interest_earned) + " " * (15 - len(str(interest_earned))), "|",
            str(running_total) + " " * (9 - len(str(running_total))), "|")
```

```
term_table_type = "" #new variable used
first_letter = "yes" #assigns a new vari
for char in term_type: #for every charac
    if first_letter == "yes": #If first_
        char = char.upper() #charcter ch
        first_letter = "no" #first lette
        term_table_type += char #charact
    else: #if first_letter = no (or anyt
        char = char.lower()
        term_table_type += char
```

```
term_printer = term_table_type + " " + str(term_counter)
```

Also made new variable to mroe easily keep track of the term being printed, this is alos in the for loop so it keeps updated.

One problem I ran into was that this is based on the term_type, which is always equal to the compound term type and not the contribution term type. Therefore if the contribution term is more common than the compound term, it will end up priting the wrong term type. E.g. We compound yearly for 2 years, but contribute each month, we print a breakdown for each month but the term type prints 'Years@...

```
| Term Number |
|-------------|
| Year 1      |
| Year 2      |
| Year 3      |
| Year 4      |
| Year 5      |
| Year 6      |
| Year 7      |
| Year 8      |
```

To try an fix this, I first went back further up into our code when the contribution term number was assigned and added a new varaibel called contributio_term_type, whereby daily = day, weekly = week and so on...

```
if contribution_type == "daily": #each
    contribution_term_number = 365
    contribution_term_type = "day"
elif contribution_type == "weekly":
    contribution_term_number = 52
    contribution_term_type = "week"
elif contribution_type == "monthly":
    contribution_term_number = 12
    contribution_term_number = "month"
elif contribution_type == "yearly":
    contribution_term_number = 1
    contribution_term_type = "year"
else:···
```

```
acting_term_type = "" #new variable called acting_term_type, will be used to assign the term type based on which is larger
if contribution_term_number > term_number: #if the contribtuion term nuymber is larger than the term number...
    acting_term_type = contribution_term_type #the acting term type should be equal to the contribution term type
else:
    acting_term_type = term_type #else it should stay as the term type
```

Then added an if command where if the contribution term number is larger than the term number, the acting term number would be updated to equal the contribution term type, else it will equal the term type.

```
for char in acting_term_type:
    if first_letter == "yes":
        char = char.upper() #
```

Then updated this in the for loop

Term Number	Deposited	Interest Earned	New Total
Month 1	650.0	1.44	651.44
Month 2	800.0	3.31	803.31
Month 3	950.0	5.62	955.62
Month 4	1100.0	8.37	1108.37
Month 5	1250.0	11.56	1261.56
Month 6	1400.0	15.19	1415.19
Month 7	1550.0	19.26	1569.26

And finally gave the correct format.

Next we added a £ symbol for each quantity representing a price. But this then shifted the table by one across...

Term Number	Deposited	Interest Earned	New Total
Day 1	£1000.0	£0.14	£1000.14
Day 2	£1000.0	£0.28	£1000.28
Day 3	£1000.0	£0.42	£1000.42

This can be fixed by adding a 1 into the equation when working out how many spaces we need after the item has been put inder the header…

```
"£" + str(running_deposit) + " " * (9 - (len(str(running_deposit)) + 1)),
```

So now the number of spaces is equal to (the number of characters in header) - (length of item under the header + 1).

Now the calculator is fully working, there is one not that the total compound interest is not always equal to the compound interest added up over a term-by-term basis as it is based on ideal conditions vs the term-by-term being based on real conditions. I plan to give first the estimated value and then provide an insight into the accuracy, before asking user it they would like the table and a graph to be printed.

Before that however, I wanted to print the graphs. As the information will need to be stored before printing the graphs, I made 4 empty lists, one representing each header…

```
cumulative_total = []
cumulative_deposit = []
stored_interest = []
term_iterations = []
```

Inside the for loop where each iteration is calculated, I'll add the value to these lists using (for example) cumulative_total.append(running_total) for each respective list…

```
cumulative_total.append(running_total)
cumulative_deposit.append(running_deposit
stored_interest.append(interest_earned)
term_iterations.append(term_printer)
```

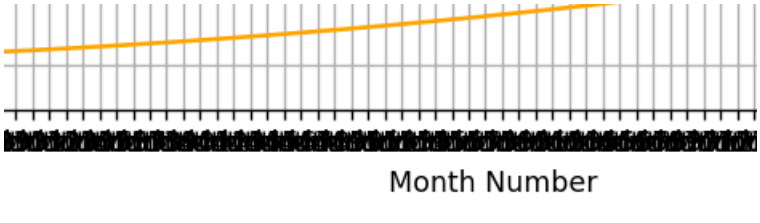
Next we need to go back to the very top and import the python graph/plotting library

```
import matplotlib.pyplot as plt
```

And use the following plot commands for the graph…

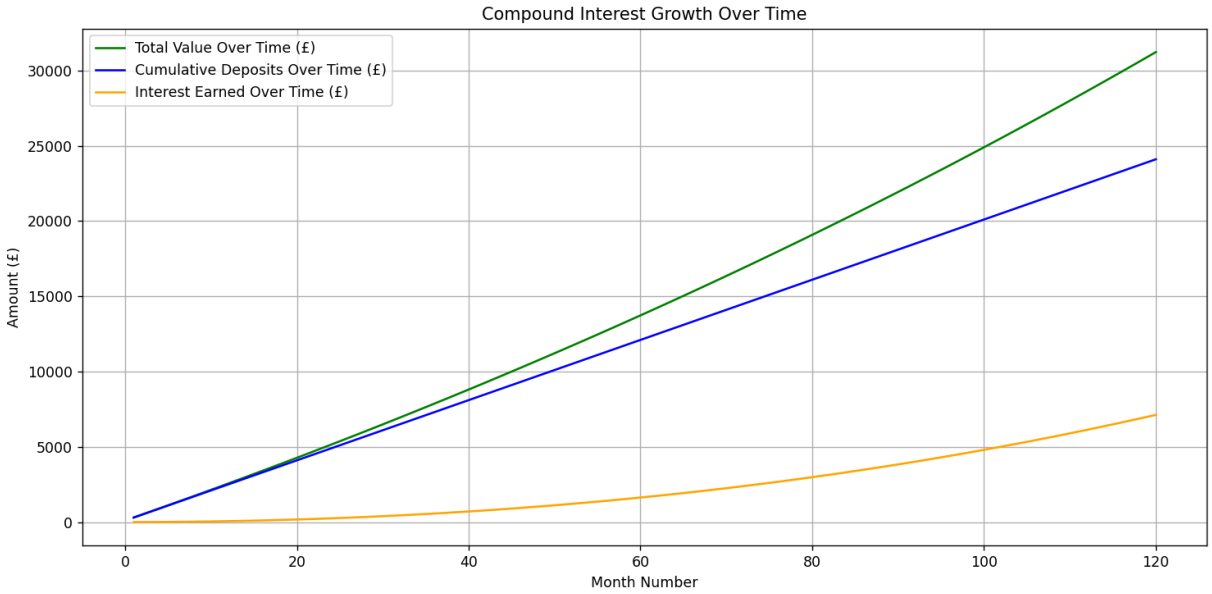
```
plt.figure(figsize=(12, 6))

plt.plot(term_iterations, cumulative_total, label = 'Total Value Over Time (£)', color = 'green')
plt.plot(term_iterations, cumulative_deposit, label = 'Cumulative Deposits Over Time (£)', color = 'blue')
plt.plot(term_iterations, stored_interest, label = 'Interest Earned Over Time (£)', color = 'orange')

plt.title("Compound Interest Growth Over Time")
plt.xlabel(term_table_type + " Number")
plt.ylabel("Amount (£)")
plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```

The next problem I ran into was that when the amount of itereations was very large, the bottom of the graph would get squised together and not be displayed properly...




```
term_iterations.append(term_counter)
```

So i changed the term_iterations list to be input with only the number of the term as opposed to the term name and the number



Which made the graph much easier to read.

I want to make this graph optional, so I made a new input...

```
graph_checker = input("Would you like this displayed in graph form? (yes/no) ")
```

And then placed the plot commands into an if statement, where it will only print the graph when user types ‘yes’ as a response to the prompt.

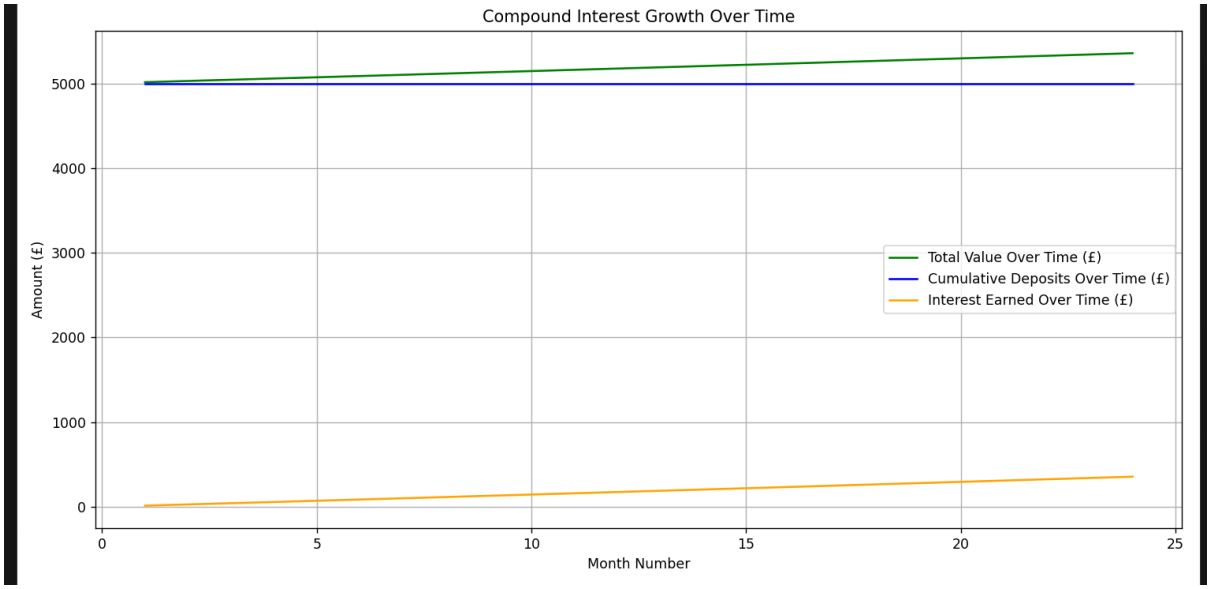
```python
if graph_checker == "yes":
    plt.figure(figsize=(12, 6))

    plt.plot(term_iterations, cumulative_t
    plt.plot(term_iterations, cumulative_d
    plt.plot(term_iterations, stored_inter

    plt.title("Compound Interest Growth Ov
    plt.xlabel(term_table_type + " Number"
    plt.ylabel("Amount (£)")
    plt.legend()
    plt.grid(True)
    plt.tight_layout()
    plt.show()
```

Added the same for the table in terminal – it is now optional to be printed

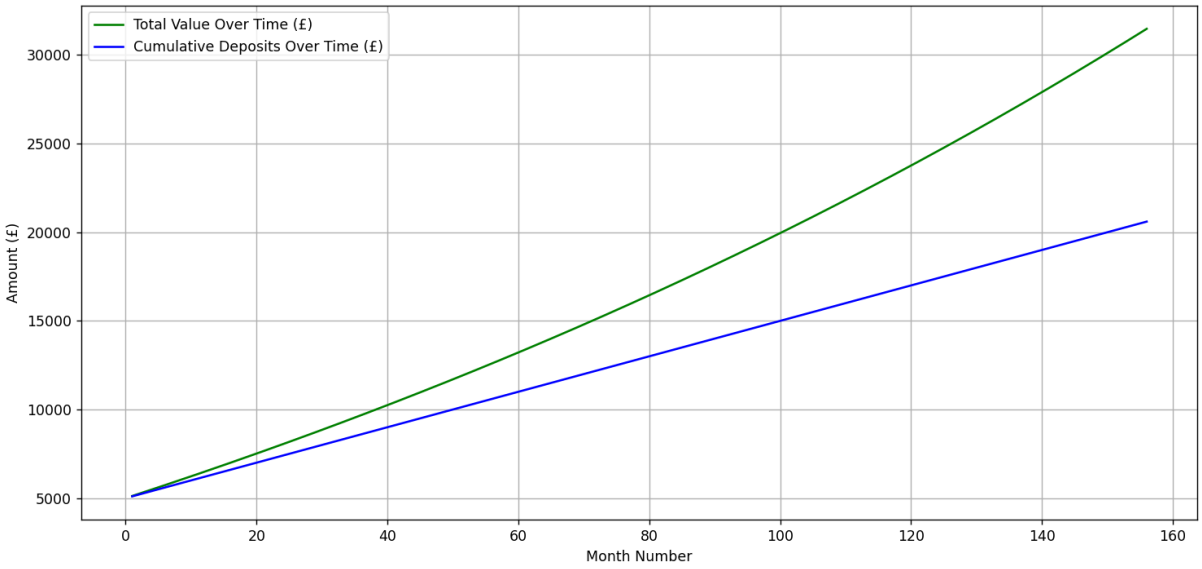
The terminal results so far:

```
PS C:\Users\surya> & C:/Users/surya/AppData/Local/Microsoft/WindowsApps/python3.11.exe "c:/Users/surya/Compound interest calc.py"
What is your starting value? 5000
What is your annual interest rate as a percentage? 3.45
Is it Daily, monthly or yearly compounding? monthly
How many months would you like to calculate for? 24
Any regular additional contributions? Yes/No no

Your total deposits are: £5000.0
Your new total is: £5356.65
You have earned £356.65 interest.
This is an increase of 7.13%
Your effective annual interest rate is: 3.50508%

Please note that the above is an estimate based on idealistic conditions and as such has a deviation of up to 0.5 % either side.
If you want to view the a more acurate estimate based on term-by-term growth, please select 'yes' when asked if you would like to view the table.

Would you like to see term-by-term growth in table format? (yes/no) yes
```

```
Would you like to see term-by-term growth in table format? (yes/no) yes

| Term Number | Deposited | Interest Earned | New Total |
|-------------|-----------|-----------------|-----------|
| Month 1     | £5000.0   | £14.38          | £5014.38  |
| Month 2     | £5000.0   | £28.8           | £5028.8   |
| Month 3     | £5000.0   | £43.26          | £5043.26  |
| Month 4     | £5000.0   | £57.76          | £5057.76  |
| Month 5     | £5000.0   | £72.3           | £5072.3   |
| Month 6     | £5000.0   | £86.88          | £5086.88  |
| Month 7     | £5000.0   | £101.5          | £5101.5   |
| Month 8     | £5000.0   | £116.17         | £5116.17  |
| Month 9     | £5000.0   | £130.88         | £5130.88  |
| Month 10    | £5000.0   | £145.63         | £5145.63  |
| Month 11    | £5000.0   | £160.42         | £5160.42  |
| Month 12    | £5000.0   | £175.26         | £5175.26  |
| Month 13    | £5000.0   | £190.14         | £5190.14  |
| Month 14    | £5000.0   | £205.06         | £5205.06  |
| Month 15    | £5000.0   | £220.02         | £5220.02  |
| Month 16    | £5000.0   | £235.03         | £5235.03  |
| Month 17    | £5000.0   | £250.08         | £5250.08  |
| Month 18    | £5000.0   | £265.17         | £5265.17  |
| Month 19    | £5000.0   | £280.31         | £5280.31  |
| Month 20    | £5000.0   | £295.49         | £5295.49  |
| Month 21    | £5000.0   | £310.71         | £5310.71  |
| Month 22    | £5000.0   | £325.98         | £5325.98  |
| Month 23    | £5000.0   | £341.29         | £5341.29  |
| Month 24    | £5000.0   | £356.65         | £5356.65  |
```



Changed to not show the interst line on the graph



So now it displays more cleany.

Next steps:

1. Add a column for when deposit is added (depsoit added per term) DONE
2. Add an option to not include contribuion on first term

Coming back to this I think I should first tidy up some of the code before adding more features. I have made a backup file incase anything goes wrong. I will try to define some of my functions and simplify some parts of the code fro example...

```
term_table_type = "" #new variable used in
first_letter = "yes" #assigns a new variabl
for char in acting_term_type: #for every ch
    if first_letter == "yes": #If first_let
        char = char.upper() #charcter chnag
        first_letter = "no" #first letter c
        term_table_type += char #character
    else: #if first_letter = no (or anythir
        char = char.lower()
        term_table_type += char
```

The for loop I made to ensure the first letter is capitalised is redundant as python has an inbuilt feature to allow for this already, as such I can condesnse tis to one line using .capitalise()...

```
term_table_type = acting_term_type.capitalize() #capitalizes the first letter of the term type
```

```
def compound_info():
    while term_length == "": #This start a
        term_length = str(input("Is it Daily
        if term_length == "monthly": #If use
            return "month", 12

        elif term_length == "yearly": #same
            return "year", 1

        elif term_length == "daily": #same
            return "day", 365

        else: #if user enters anything but
            print("Invalid input please resp
            term_length = "" #sets term_leng
term_type, term_number = compound_info
```

Next, we have taken out while loop for term_type and term_length and placed it within a new function called compound_info().

I just had to replace the lines under each condtion where I would set term_length and term_type manually and instead used ‘return’ to save 2 values. Note that these values are not stored in a variable until the function is ran using the line at the bottom...

Term_type, term_number = compound_info...

This tells the program to define term type and legnth based on the program compound info where the first returned value is stored in term_type and te second is stored in term_length.

The while command ‘while term_length == “”’ can just be changed to while true as this will just keep running until a return is given.

```
def contribution_info():
    while True: #starts a loop for while contribution check is empt
        contribution_check = input("Any regular additional contribu
        if contribution_check == "yes": #check if user responded '
            contribution_type = input("Is the contribution daily, w
            while contribution_type not in( "daily", "weekly", "mon
                print("Invalid input, please respond daily, weekly,
                contribution_type = input("Is the contribution dail
            if contribution_type in ("daily", "weekly", "monthly",
                additional_contribution = float(input(contribution_
                return additional_contribution, contribution_type
        elif contribution_check == "no": #if user responds no, this
                additional_contribution = float(0.0) #contribution
                contribution_type = "No additional contribution"
                return additional_contribution, contribution_type
        else:
            print("Invalid input. Please respond yes or no") #if us
```

Also done for the function to gather information regarding contribution type if any. I will continue to do this for each function, as not only will this tidy up the code but may help in the goal of being able to add multiple banks later down the line as we can simply recall on the function to run again as opposed to the whole code itslef.

```
def compound_info():···
def scenario_details(term_type):···
def contribution_info():···
def contribution_figs(contribution_type):···
def pre_calculations(interest_annual, term_number, contribution_t
def compound_interest_calculations(additional_contribution, contr
def find_num_terms(contribution_check, contribution_frequency):···
def print_table(acting_interest, contribution_interest, initial_t
def print_graph(term_iterations, cumulative_total, term_table_typ
```

Code is now sorted into 9 different functions

1. Compound_info() will ask for the compound frwuency e.g. daily, monthly, yearly and stores this in variable term_type
2. Scenario_details asks for more details, namely, staring value, annual interest rate and the amount of term_types user wants to calculate for.
3. Contribution_Info will ask for user to input whether they want to add regular contributions into the calculation, if they do, they will then be asked for the

amount and the frequency of the contribution. If there is no contribution then it will be set at 0.0.

4. Contribution_figs calls upon contribution_type which is defined and returned by contribution_info and then returns the term type and term number. E.g. contribution is monthly, so term type is month, and term_number is 12.
5. Pre_calcualtions takes the annual interest, term number for compound and contribution and the term amount, from these values it calculates and then returns the acting interest, contribution interest rate and the contribution frequency.
6. Compound_calculations calls upon the following variables previously defined by other functions

```
compound_interest_calculations(additional_contribution, contribution_interest, initial_input, contribution_frequency, acting_interest, term_amount):
```

And calculates the compoud interest with contributions added, the total deposts, the effective annual interest rate, the percentage increase and prints this information to the user. The funciton doesnt return any of these values as they have all either already been defined or are only relevant to this function.

7. Find_num_terms is a function used to find the maximum number of terms between the compound term number and the monthly term number, e.g. if there is 2 years compound with yearly compound this is onyl 2 terms, but contribution is monthly so this is equal to 12 * 2 = 24 terms. Next the function will create a list with 0s and 1s, where each 1 represents a term that will have the contribution added on. This is returned in the list term_list.
8. The print_table function takes the following inputs...

```
def print_table(acting_interest, contribution_interest, initial_input, contribution_term_number, term_number, term_type, term_list):
```

And then calculates the values for total deposits and new value on a term b term basis, it does this by going through the term_list and for every 0 it will apply the the compound interest formula and for each 1 it will apply the compound interest formula + the regular contribution. This function also prints a header for the table and prints the information in a table format.

9. Finally, print_graph takes the term_iterations, term_table_type, cumulative total and cumulative_deposit and prints this in a graph if the user wants to. The graph is based on the table outputs and as such is only available after the table is printed.

```
' + "Deposited" + "
```

Added a new column in table that only prints the deposited amount, if there is no deposit the space will be left blank...

```
| Month 11  |          | £5000.0  | £234.0  | £5234.0  |
| Month 12  | £500.0   | £5500.0  | £255.81 | £5755.81 |
| Month 13  |          | £5500.0  | £279.79 | £5779.79 |
```

```
"£" + str(deposit_added) + " " * (9 - (len(str(deposit_added)) + 1)), "|",
" " * 9, "|",
```

So if there is deposit added it prints the deposit value with the £ symbol and then adds on the amount of spaces necessary so that it matches the header spacing.
If there is no deposit added it simply adds 9 blank spaces, still matching the header size.

```
first_term = ""
while first_term == "":
    first_term = str(input("Do you want to appply contribution to the first " + term_type)).strip().lower()
    if first_term == "yes":
            contribution_compound = (additional_contribution * ((contribution_interest**contribution_frequency) - 1))/(contribution_interest -
            compound_total = (initial_input*(acting_interest**term_amount)) + contribution_compound #calculates the initial input compounded a
    elif first_term == "no":
            contribution_compound = (additional_contribution * ((contribution_interest**contribution_frequency) - 1))/(contribution_interest -
            compound_total = ((initial_input*(acting_interest**term_amount)) + contribution_compound) - (additional_contribution * (contributio
    else:
        print("Invalid input, Please respond yes or no.")
        first_term = ""
deposit_total = (initial_input + (contribution_frequency * additional_contribution)) #calculates total deposited by user
```

New loop within the compound interest calculations that asks for user input on whether they want to add the regular contribution to the first term. If the asnwer is yes, we will calculate the answer the same method we prviously used.
If no then we will use the same method again but then take away one terms contribution times by the interest rate to the power of the amount of terms. This is because this is added on the first term, it gets compounded over the same amount of terms as the initial deposit, so to remove it from the calcualtion we also need to compound it over time.

```
- (additional_contribution * (contribution_interest**contribution_frequency))
```

First version above

```
- (additional_contribution * (acting_interest**term_amount))
```

Changed to the second version as the contribution is compounded based on the compounding term and interest as opposed to the contribution interest and terms.

```
deposit_total = ((initial_input + (contribution_frequency * additional_contribution)) - additional_contribution)
```

Also adjusted the deposit total in the case that user does not want to include first term for contribution, taking away one iteration of the additional deposit.

Now I need to also adjust this when using the table for calculations, therefore we adjusted this current function to return first_term so that we can reuse it in a later function.

```
if first_term != "yes":
    range_start = 2
    term_list.append(0)
else:
    range_start = 1
```

I did this by starting an if command whereby if first_term is anything other than “yes” the new variable range_start will be equal to 2 and the term_list will be started with a 0. If it is yes thr range_start will be 1…

```
    for term in range(range_start, (round(range_finder) + 1)):
```

We then replaced our range from 1 to range_finder to be from range_start to range_finder so we avoid printing a 1 on the first iteration of term_list if we dont want to add contribution for the first term.