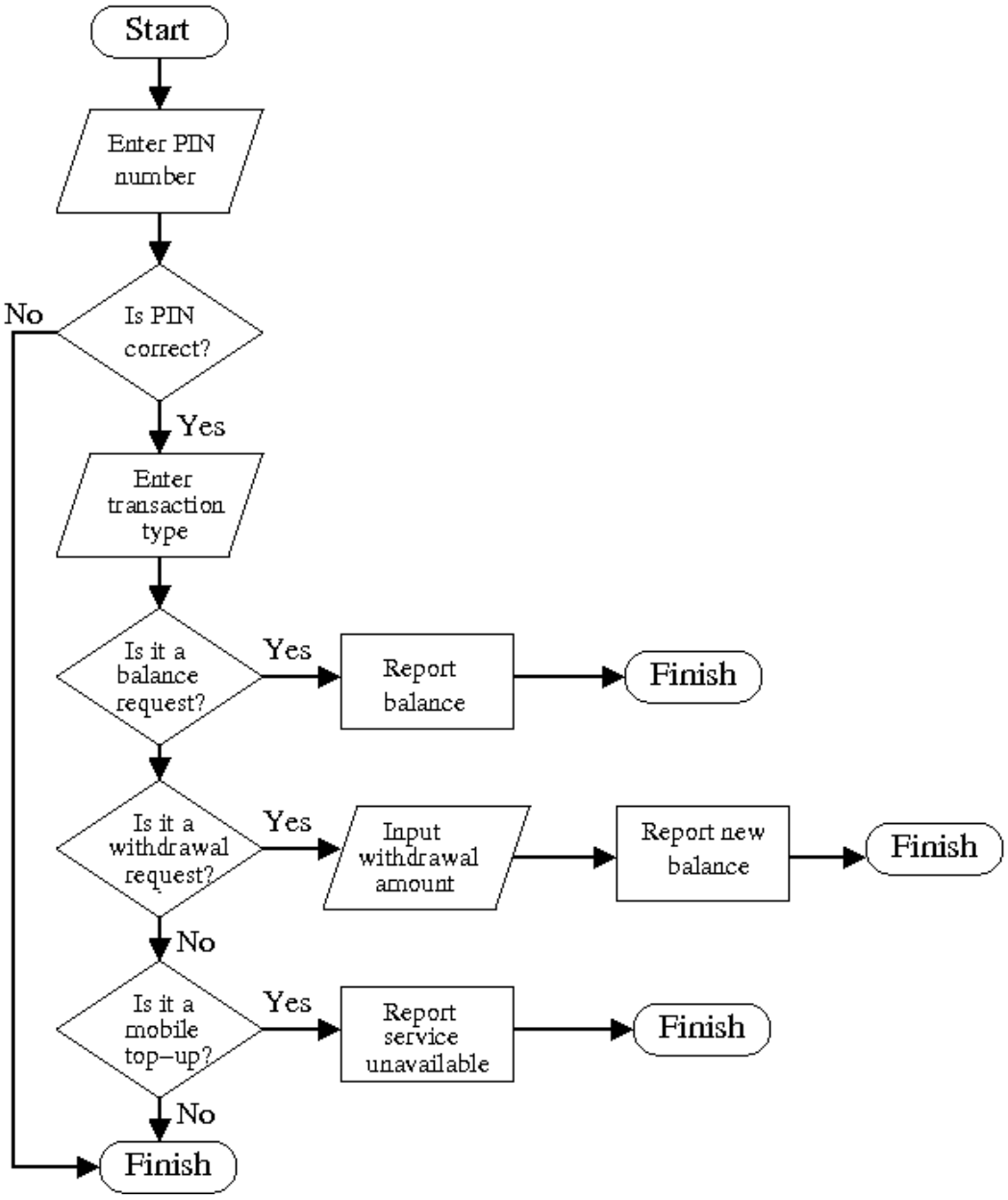
Python Practice 2: Conditionals and Functions

The aim of this lab class is to give you experience of using conditional statements in Python, and in structuring code through the use of functions. You will write a program which simulates some of the behaviour of a *cashpoint machine* or *ATM*. The following *flowchart* shows the underlying logic of the system you will implement:



1 Some useful tips on programming and debugging

This lab requires you to write some reasonable amount of code, that you must produce for yourself from scratch. Here are some tips on how to proceed, both with *programming* (i.e. writing code) and with *debugging*, which is the task of finding errors in the code you have written, or figuring out why it doesn't behave as you expect.

1. When writing a larger program, it is good to proceed *incrementally*, i.e. to save and test (i.e. run) your code each time you make a significant change. Doing so makes it easier to identify and resolve errors at each stage. This is much easier than trying to write your code in one go, and then discovering that you have a large number of errors to fix.
2. When your code is only partly written, you might find it useful to put print statements in place of code blocks that are not yet written. This can allow you to run your incomplete code, so as to observe whether execution proceeds as you expect, even though some of the code is not yet in place. For example, if you're writing a conditional `if-else` structure, you might start by putting a print statement for the `else` code block, allowing you to get on with writing and testing the `if` code block.
3. If Python prints an error message when you test your code, study the error message: it may help you discover the problem in your code, e.g. pointing out a syntactic error.
4. Print statements can be used in various ways to help you understand why your code is not working as you intended:
   - For example, you can print out a value computed as part of some larger task as a way to check that it has been computed correctly.
   - You can add print statements that signal whether the IF or ELSE case of a conditional has been followed as you expect. If the wrong case is followed, you may have specified the condition incorrectly, or incorrectly calculated one of the values being tested.
   - Print statements can also help you find the source of an error, when you are having trouble locating this, e.g. if your code exits in a way you don't expect and you can't see why. In this case, it may be useful to add print statements at various points, printing strings such as `"*POINT-1*"`, `"*POINT-2*`", etc. If there are, say, four such statements, but only the first two messages get printed, this suggests the error is located in the part of your code between the second and third print statements.
5. Finally, don't forget the critical importance of correct indentation in Python programming. Spyder will help you achieve correct indentation. Pressing TAB or DEL in the 'indentation zone' will cause the cursor to move in or out one level of indentation.

2 The programming task

The envisaged scenario is that a bank user approaches the ATM and inserts their card. We imagine that the ATM then reads the card details and uses them to access key information from the bank's central computer, namely: (i) the card owner's *true* PIN, and (ii) their current balance. The ATM then calls the code that you will write, which checks that the user knows the *correct* PIN, and if so, then provides ATM services to the user.

Download the code files `SimpleCashpoint.py` and `test_cashpoint.py` from the module homepage, and open them in Spyder. The first file (`SimpleCashpoint.py`) contains a 'dummy' (i.e. empty) definition of the `cashpoint` function, which consists of a single print statement (which prints a message that the function has not yet been defined). It is your task to complete this function definition, so as to implement the system described by the flowchart.

The second file (`test_cashpoint.py`) contains some test cases. If you run this file (by pressing F5), it first imports the `cashpoint` function from the first file, and then calls it with different parameters, i.e. specifying different PIN numbers and different current balances. The first call to the function in that file is given as:

```
result = cashpoint('1234',3415.55)
```

Note how the result 'returned' by the function call is here *assigned* to a variable (`result`), so that it can subsequently be printed out (in the next line of the test file). This doesn't make much sense at this stage, as the function is not yet written to return a result, but it will be useful later on. A function that does not specifically return a result instead returns `None` (which is a special *null value* in Python), and it is this value that is printed when the test file is run. Run the test file now, to check that it correctly finds the first code file, and runs as you expect.

3 Thinking through the problem logic

Look again at the flowchart on Page 1. Using this information, you can write down the logical steps that the your code must follow, when the `cashpoint` function that you are defining is called.

1. Ask the user to input their PIN.
2. Check whether the PIN value entered matches the true PIN (i.e. the value given in the function call, such as `'1234'` above). If they match, then continue as below, otherwise print a suitable message and finish.
3. Ask the user to choose their transaction, by printing a numbered list of options, and reading the value entered by the user.
4. If the input is 1 (for balance request), print the balance information, and exit.
5. If the input is 2 (for a withdrawal), ask the user for the withdrawal amount, report the adjusted balance amount, and exit.
6. If the input is 3 (for mobile phone top-up), then (for now) just print a message that the service is unvailable, and exit.
7. If the input is something else, then print a suitable message and exit.

4 A first attempt in Python

Next, have a go a writing a first definition of the `cashpoint` function, completing the dummy definition given in the file `SimpleCashpoint.py`. First, save a copy of the file with a modified name (say `SimpleCashpoint_v1.py`), and develop your code in this new file. You will need to modify the testing file also, so that it imports from the new file. Some hints:

1. Where the user is required to provide input (e.g. their PIN, transaction choice, or withdrawal amount), you can use the `input` function (as in the first lab).

   NOTE that the test file call to the `cashpoint` function specifies the correct PIN as a *STRING* (of digits), not as a number (e.g. integer). Recall also that the `input` function returns what has been entered as a string, so this can be directly compared to the correct PIN (although care must be taken about stray *space* characters being added to the input).
2. Where there is a choice of how to proceed, e.g. what to do depending on whether the PIN values match, you can use an `if-else` conditional statement. Where there are more than two options (w.r.t. transaction type), you might use an `if-elif-else` statement.

Test your code thoroughly, to ensure that it behaves as you expect. Show it to a demonstrator for feedback.

5 A more realistic definition: using `return`

Before you proceed, you should again save your code with a different name, and work in the new file (e.g. `SimpleCashpoint_v2.py`).

If you have written it well, your first definition of the `cashpoint` function may display a reasonable version of the *visible* behaviour we might expect, i.e. of the pattern of interaction of a user with the ATM. However, it would not be much use in a real ATM, as for this purpose, the function would need to *return* information to the ATM system that called it, so that the ATM would know how to proceed. For example, if the user requested a withdrawal, the ATM would need to know how much money to issue before returning the user's card. If the PIN was entered incorrectly, then the ATM should know to give back (or perhaps withhold) the user's card, etc.

To add this functionality to our code, we can use *return* statements. As we saw in class, a return statement in a function has the form "`return <value>`". When executed, it causes the function's execution to terminate at this point, with the specified value being returned. For example, the following function tests if a number is positive (greater than or equal to 0), and **returns** (or *gives back*) the value `True` if it is, or otherwise gives back the value `False`.

```
def is_positive(n):
    if n >= 0:
        return True
    else:
        return False
```

The value returned might then be assigned to a variable, as in the following:

```
>>> is_positive(3)
True
>>> is_positive(-2)
False
>>> result = is_positive(5)
>>> result
True
```

Extend your code by adding return statements, so that it returns, to the system that has called it, a value that provides the system with the information it needs to proceed. For some cases, this returned value can be a single string, such as (e.g.) `"PIN-error"` or `"balance-request"`. For the case of a *withdrawal*, however, the information must specify both that a withdrawal is requested, but also the *amount* to be withdrawn. This can be handled by returning the two pieces of information as a *pair*, i.e. with a return statement such as:

```
    return ("withdrawal-request",amount)
```

Recall how the code in the test file (`test_cashpoint.py`) tries to collect the value returned by the function call, so that it can be printed. Hence, you should now see the results that are returned by your code being printed when the test file is run.

6 Breaking the task down, using functions as subroutines

Again, before you go any further, save your code with a different name (`SimpleCashpoint_v3.py`) and work in the new file. The code you have written so far is hopefully fairly readable, but if the

functionality is extended much further, it could easily become long-winded and hard to read. For example, a real ATM might allow three attempts at entering the PIN before refusing to continue. It might allow you to conduct several transactions in one visit to the ATM. The amount you are allowed to withdraw is typically restricted, based on various factors (e.g. your current balance, and a maximum daily withdrawal amount). If such behaviour was achieved by further adding and embedding conditionals, then our function definition could soon be very long indeed.

Many programming languages address this problem by allowing users to specify named chunks of code, known as *subroutines*. A subroutine has the advantage of being *reusable* in different parts of the program (whilst being specified only once itself), and also — by fulfilling a *conceptually coherent subtask* — making the higher level code that calls it much easier to read. In Python, this idea of subroutines is realised by defining *functions*.

Develop your program by defining sub-functions to package up some of the required functionality. The aim is to give the overall program more complex behaviour, without making the top-level `cashpoint` function itself much more complicated. Some suggestions for how to proceed follow.

6.1 PIN testing

Define a function `check_PIN` to cover the PIN checking part of the task. The function will ask the user to input their PIN, compare this to the true PIN, printing an error message if it is wrong, etc. The function might return a *boolean* value, i.e. returning `True` if the check succeeds, and `False` otherwise. In that case, a call to this function can appear as the *condition* of the relevant `if-else` statement in the `cashpoint` function, as in the following (where the `"..."`s represent sections of code that I have not filled out):

```
def cashpoint(truepin,balance):
    if check_PIN(truepin):
        ...
        ... # Case where PIN check succeeds
    else:
        ...
        ... # Case where PIN check fails


def check_PIN(truepin):
    ...
    ... # Code asking user to input their pin
    ... # Returns True or False, depending on success of check
```

Observe how this approach simplifies the definition of the top-level `cashpoint` function, by delegating some of the work to the `check_PIN` function, which performs a conceptually coherent subtask. Next, extend the `check_PIN` function to allow the user *three* attempts at entering their PIN before final rejection. This can be done by modifying only the definition of the `check_PIN` function, i.e. without complicating the top-level `cashpoint` function.

6.2 Withdrawal function

Define a function to cover the withdrawal sub-task. So far, users have been allowed to withdraw any amount of money, which is unrealistic. Allow only withdrawals that do not put the account into the red. Also, assume there is a limit to the amount that can be withdrawn with any visit

to the ATM, e.g. 100 pounds. If the user requests an amount of money that is not allowed, they should be told so, with a zero withdrawal amount being signalled (i.e. returned) to the higher level. As a further embellishment, withdrawal amounts might be restricted to multiples of 10.

6.3 Mobile phone top-up function

Define a function to provide reasonable mobile top-up functionality. This should require the user to enter the mobile number twice in succession, and check that the same value was entered both times. The money amount allowed for the top-up might be restricted to a multiple of 10, and to not exceed the current balance.