Computational Thinking in Action: Preventing Fraud

Let's try and apply what we've learned!

Imagine that we're working in the security department of a major national bank. A customer - let's call him Tom - calls us on a Saturday morning: he's upset that his wallet was taken at a bar and money was withdrawn from his account at 1am last night. Police investigation reveals the thief saw Tom inputting his pin when buying a drink, stole the wallet when Tom was a few mojitos deep, went to a cashpoint, input the PIN incorrectly once, got the PIN right the second time, and withdrew a substantial amount of money from Tom's account.

This is a serious problem! We have a very unhappy customer. Luckily, we have technology at our disposal: our ATMs have cameras we can use to film users, and we have photographs of our customers' faces to compare them with users. Let's apply computational thinking and object-oriented programming to come up with a general solution.

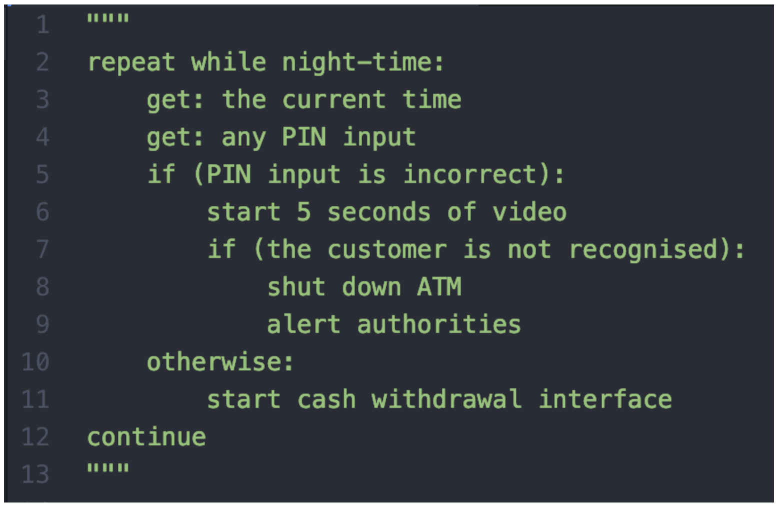
This application might be tricky, but we want to show you the power of computational thinking and programming in a real-world example. Don't worry if not all of what follows makes sense.

A. Abstraction

The type of problem here is checking whether night-time attempts by customers to withdraw money are fraudulent or not. We want our solution to apply to all our future customers, not just Tom.

D. Decomposition

Here we reflect. What do we need to get, set, and check, in order to solve our problem? Let's write some instructions that bear some resemblance to syntactically correct Python code, but don't — strictly speaking — count as Python code. I.e., let's write some pseudo-code:



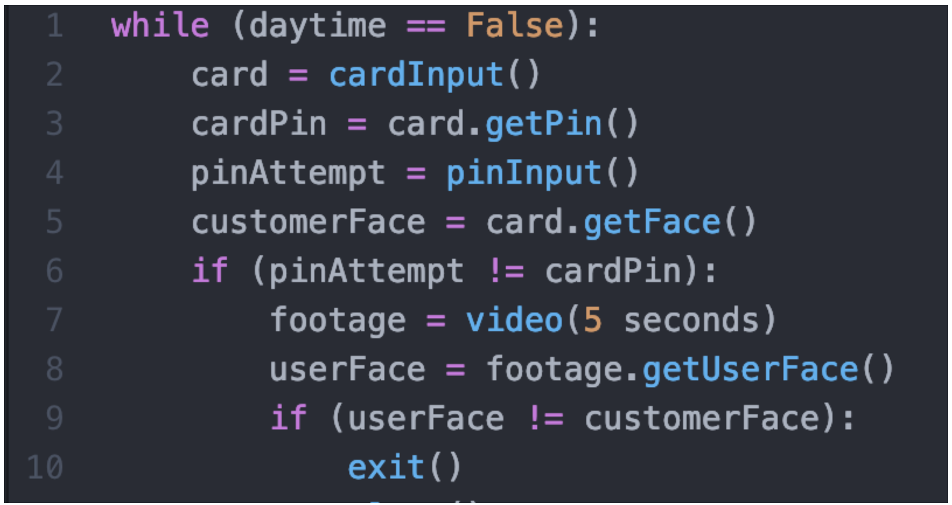
```
"""
repeat while night-time:
    get: the current time
    get: any PIN input
    if (PIN input is incorrect):
        start 5 seconds of video
        if (the customer is not recognised):
            shut down ATM
            alert authorities
    otherwise:
        start cash withdrawal interface
continue
"""
```

Side note: don't worry about the pair of triple quotes (""""); this just makes everything in between technically a ***comment***, rather than code. This was written in the text editor Atom, where you can write Python. The pair of triple quotes prevents the computer from trying to interpret the pseudo-code as Python.

Notice the use of blocks, defined by their indentation. The Repeat while block is the largest, and has a 'scope' that encompasses every other instruction. Everything within that block is repeated over and over until some condition (that it's night-time) becomes False. As soon as it's not night-time (and not before) we Continue. The first If-statement opens up a new block, in which further instructions are contained: we'll shut down the ATM and Alert the authorities *if and only if*, first, it's night-time, second, the PIN input was incorrect, and third, the customer is not recognized; in that order.

T. Translation:

We did well! But we need to be language-precise. We need to translate our pseudo-code into Python code. Take a look at this code. We'll explain it shortly.



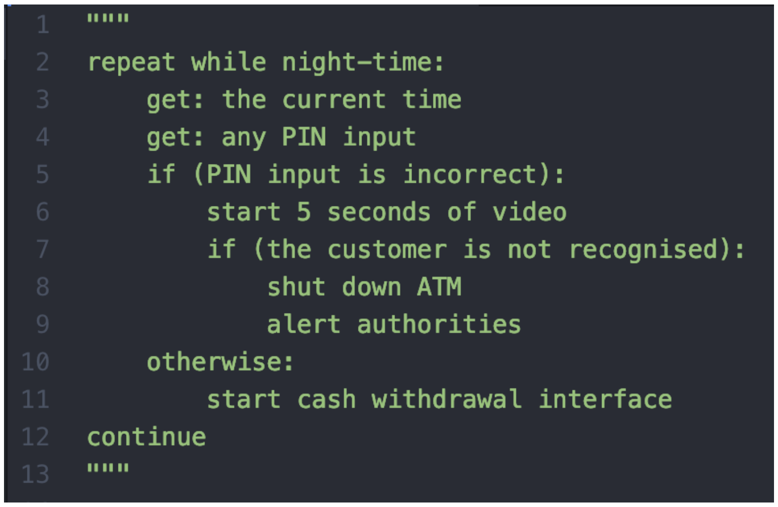
```
while (daytime == False):
    card = cardInput()
    cardPin = card.getPin()
    pinAttempt = pinInput()
    customerFace = card.getFace()
    if (pinAttempt != cardPin):
        footage = video(5 seconds)
        userFace = footage.getUserFace()
        if (userFace != customerFace):
            exit()
```

This is much more colorful. Nice!

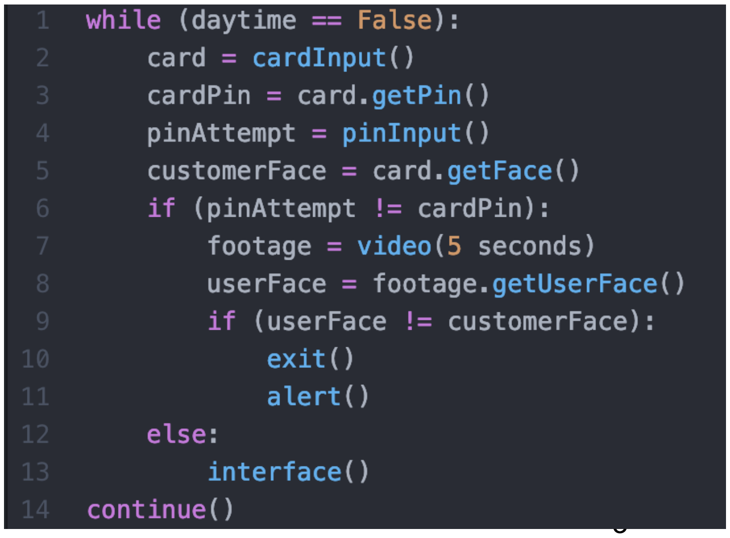
We never reinvent the wheel when programming, which means that when we're coding, we often make use of classes and functions defined elsewhere either by ourselves or by others. The above is no exception. What we're assuming here is that:

- The variable 'daytime' has been defined above this code and is set to False/True appropriately by some other code that tracks whether it's daytime or nighttime.
- The class *Card* has been defined elsewhere, and this class contains intrinsic variables storing the PIN and an image of its owner's face, as well as functions like getPin() and getFace(), which allow us to get a particular card's values of those variables, respectively.
- The function cardInput() takes the real input of a real credit or debit card and outputs an instance of the class *Card*.
- The pinInput() function takes a real-life PIN input and outputs a list of integers.
- The video() function starts filming the user of the ATM, can be given durations of time as inputs, and outputs an instance of the class *Footage*, which has been defined elsewhere. We've built this class so that it has a function for facial recognition: getUserFace().

Let's look at the pseudo-code in the Decomposition phase, and the Python code in the Translation phase, side-by-side:



```
"""
repeat while night-time:
    get: the current time
    get: any PIN input
    if (PIN input is incorrect):
        start 5 seconds of video
        if (the customer is not recognised):
            shut down ATM
            alert authorities
    otherwise:
        start cash withdrawal interface
continue
"""
```

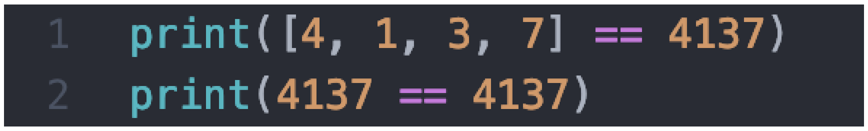


```
while (daytime == False):
    card = cardInput()
    cardPin = card.getPin()
    pinAttempt = pinInput()
    customerFace = card.getFace()
    if (pinAttempt != cardPin):
        footage = video(5 seconds)
        userFace = footage.getUserFace()
        if (userFace != customerFace):
            exit()
            alert()
    else:
        interface()
continue()
```

The code on the left is the pseudo code, or the output of the Decomposition stage, and the code on the right is the actual Python code, or the output of the Translation.

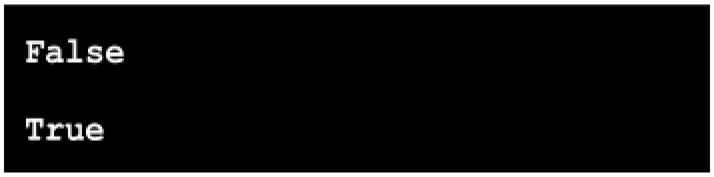
Notice how we've replaced our pseudo control-flow structure 'repeat while' with an actual Python control-flow structure, a while loop. With the Python, we're doing a heavy amount of object-oriented programming, relying on classes and functions that we must define elsewhere in order for our code to work. These classes - *Card* and *Footage* - and functions - cardInput(), pinInput(), card.getFace() etc - must all be defined somewhere and do the appropriate things to the appropriate types of thing.

Check out the first if-statement too. This compares the user's attempted pin and the card's actual pin. Remember: the '!=' operator is the opposite of the '==' operator; it checks for inequality between two things. Accordingly, the value of the pinAttempt variable, and the value of the cardPin variable, need to be of the same type: they need to be either both lists, like [4, 1, 3, 7], or both strings, like '4137', or both integers, like 4137. If they are of different types, then we could have a situation where the program thinks that the pinAttempt and the cardPin are different, when in fact they're really the same, just because the pinAttempt is being stored as the list [4, 1, 3, 7] and the cardPin is being stored as a string '4137'. To reiterate this, the code:



```
print([4, 1, 3, 7] == 4137)
print(4137 == 4137)
```

Outputs:



```
False

True
```

O. Optimisation:

Phew! We've done very well so far. It's in this stage that we now ask: could we have reduced the number of instructions in our code? Could we have written instructions that execute faster?

There are almost always many, many ways to accomplish the same goal, and as we've seen, how we cut up the tasks depends on the tools at our disposal. Indeed we could build a bespoke function whose entire *raison d'être* is to do what the Python code above does, in which case, if we execute that function in another piece of code, we've reduced 14 lines to one line! When we’re dealing with Python, we always have a fantastic range of tools at our fingertips: we can find out about them with a simple Google search. But as we’ve seen, by defining classes and functions, we can also make our own tools *with* Python’s tools.