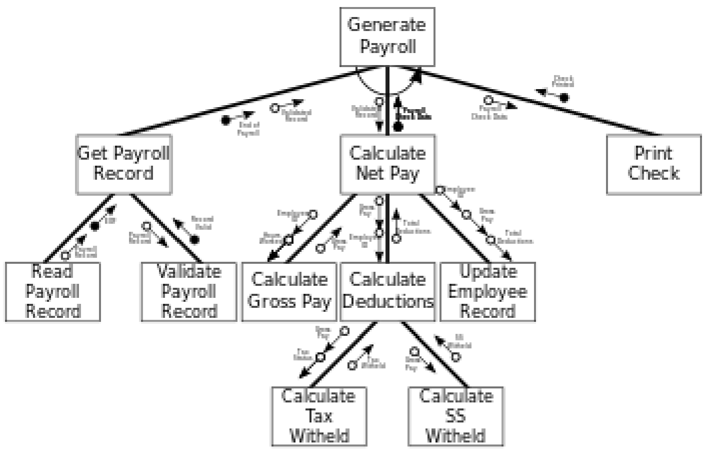
Decomposition and Abstraction

Decomposition

Decomposition is about breaking a problem down into smaller parts which are more manageable. It's something we do all the time - taking a large project and turning it into a to-do list of small, manageable items. However, when programming this becomes even more important - we cannot write a computer program containing several million lines of code from one end to the other. We have to break it down into different sections, different modules. This means that different people can work on different parts of the program, but also even when we are working on our own we can work on one module at a time, then test it, before integrating it with the whole program.

This is an example of a program structure diagram that might be used to break down a programming task into component parts:



If a problem is decomposed properly it will mean that each component can be tackled independently of the others - you can work on them on their own, without needing to know what is happening within another module. Ideally, each part should also be of a similar level of complexity or detail.

The advantages of decomposing a problem are:

- The different modules can be worked on by different people, or teams, at the same time, which will increase productivity
- A small module is easier to think about and solve than a large problem
- It may be possible to process modules in parallel to each other, allowing for faster processing of the problem as a whole.

However, there are risks of decomposition:

- You have to understand the problem very well to successfully decompose it
- If parts of a problem are worked on in parallel, it may be difficult, or impossible, to recombine them into a single solution at the end.

A large computer program will often be broken up into modules called Subprograms, or Subroutines. In Python, and some other programming languages, these are often called Functions. A Function is a self contained piece of code that can be ‘called’ and run from within other pieces of code. We will cover a section on them later.

We use Subprograms because:

- They make code easier to read
- They make testing easier
- They make a program more structured
- They make a program shorter - you don’t have to write the same code over and over again, just create one subprogram and ‘call’ it whenever you need it

Abstraction

Abstraction is about making a program that represents the real world, but is not identical to the real world. When creating a program like a game the programmer does not attempt to include every single detail of the real world, but rather includes those which are relevant, important and interesting. Take something as simple as rolling a dice - a real dice has weight, may have rounded corners, makes a particular noise when it roles, obeys complex laws of physics when it bounces around when we drop it… none of this is *necessary* to include in a computer program. If we were creating a board game of monopoly, we needn't simulate all of these details - we could just randomly generate a number between 1 and 6 and put it on the screen. We may choose to include other details of a dice roll - like the image of the side of a die - if we think it will make the program better, but we might also leave them out to keep things simple.

Abstraction is all about choosing which are the important details to include, and which we can safely leave out. Including too many details makes a program too complex and may make the end user experience overwhelming. Leaving out important details will make a final program ineffective or inaccurate.

Question 1 - If you were creating a computer version of chess, what details of playing chess might you leave out?

Question 2 - What details of playing chess would it be essential that you included?

Investigating Requirements

Before creating a new computer program it is essential that we take the time to really think about and understand what we are trying to achieve. This is particularly important if we are creating a program for someone else - i.e. we are employed by a company to write software for them. We may be creating a program that we, ourselves, are never going to use, about a topic we don't understand at all. Therefore, it's important we properly investigate exactly what the end users want, and what the program should achieve.

We can break a program down into its basic functions -

- Inputs - what data needs to be entered into the system, and how will it be entered (typed in? Via mouse click? Via game controller?)
- Outputs - what data should be displayed, and how (as an image, a graph, a table, an animation?)
- Processing - what rules govern how the input gets turned into the output? What are the calculations and manipulations that are required?
- Initialisation - what should variables be set to when the program first starts?

Each of these categories should be considered, and designed carefully.

Methods for investigating the needs of a system include -

- Interviewing potential users
- Setting a questionnaire for users
- Watching a user use the old system (shadowing)