Abstraction

Representational abstraction
Abstraction allows us to remove unnecessary detail from a problem leaving only the essential features thereby making it easier to solve. Maps are examples of representational abstraction.

Abstract generalisation
With abstract generalisation we identify common (general) characteristics thereby enabling us to group similar constructs together into a hierarchy

Abstract generalisation is also the ability to see patterns so that we can recognise problems or parts of problems that we may have solved before. Lots of programming is concerned with reusing pieces of code that were originally developed for other solutions. Even within a piece of code we are writing we may notice that quite a lot of our code is repeated. It is our ability to notice those repetitions that help us write more succinct and generalisable code using functions perhaps.

Procedural abstraction
Abstract away the actual values used in a computational method. In that sense algebra and formulas are abstractions. The following expressions have the same form:

```
(1+2) x 3
(7+9) x 2
(5.5 + 12.3) x 18.1
```

We can abstract them away algebraically as:

```
(a+b) x c
```

Functional abstraction
- A functional abstraction maps an input to an output. The function returns a value given a certain input.
- Functional abstraction is an extension of procedural abstraction. A procedural abstraction might form part of a functional abstraction.

```
def calc(a,b,c):
    return (a+b)*c
print(calc(1,2,3))
```

- In programming we abstract details using functions. We do not need to know how functions work to use them. The functions themselves can be a black box to us. The details of how the function works have been abstracted away.

Data abstraction
The details of how the data are represented are hidden. We do not need to worry how ASCII characters, real numbers and integers are represented. Real numbers can be represented using exponent and mantissa in binary, but we do not need to concern ourselves about this when we a writing programs. We can have more complex abstract data types. These include queues, stacks graphs, trees, hash tables, dictionaries and vectors.

Problem abstraction
Remove details of a problem until you are left with a problem that you already know how to solve. This allows us to use solutions that have been applied to analogous problems. For instance Euler solved the Konigsberg bridge problem (is it possible to cross all bridges only once) by reducing it down to a graph problem, that he already knew how to solve.

Information hiding
In OOP, this is where data that do not contribute to the essential characteristics of an object are hidden. These attributes and methods are private and not accessible from outside an object. Essential characteristics of the object can be accessed via an interface. Also we do not need to concern ourselves with local variables in functions.

Decomposition
Decomposition is the breaking down of a complex problem into smaller more manageable problems that are easier to solve. Each component of the program completes a specific task. This allows algorithms to be more modular and therefore more intuitive.

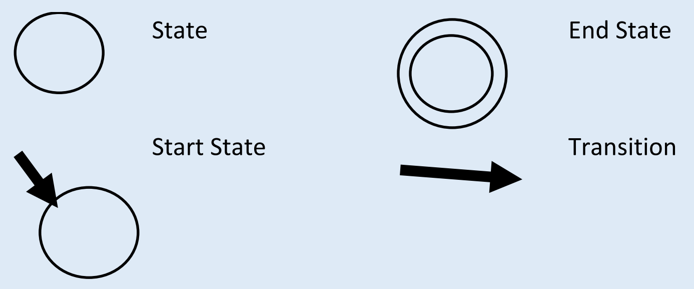
Composition
Composition is combining the procedures together to form compound procedures in order to solve a greater part of the problem that each of the procedures can solve separately. Specifying the interface between the components is important otherwise they would not fit together.

Automation
- Putting models into action using algorithms
- Putting the abstractions into algorithms and putting the algorithms into code.
- Developing computer models that concentrate on the essence of a problem. The models are a simplified representation of reality where assumptions are made.
- For instance, weather forecast models use mathematical models and physics to model the atmosphere as a fluid, which is a good way to run simulations and predict the weather up to a few days ahead.
- It is not possible to model the billions of variables, so simplifications are made to help solve the problem. As computers get more powerful and algorithms improve and we have more data we get better at predicting the weather.

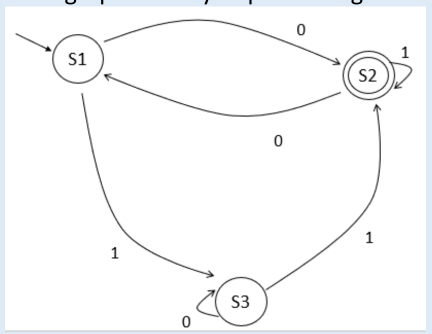
Finite State Machines (FSMs)

FSMs are a model of computation that allow us to understand how computers work. FSMs consist of a set number of states that allows the transition between states and are determined by a fixed set of inputs and have a set of outputs.

Notation for FSM



Finite state diagrams are a graphical way of presenting finite state machines.



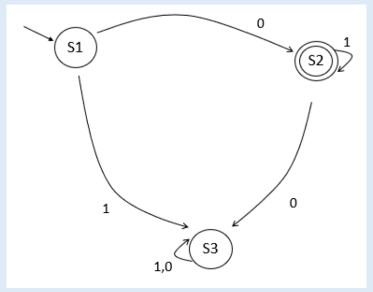
- S1, S2 and S3 are the states
- Each transition edge has an input value
- S1 is the Start State
- S2 is the Accept State)
- For the input sequence to be valid the sequence must end on the accept state (S2)

Example sequences
- 0 0 0 1 1 - Valid
- 1 0 0 1 - Valid
- 1 0 1 0 - Invalid
- 1 0 1 - Valid

State transition tables are another way of representing FSM.

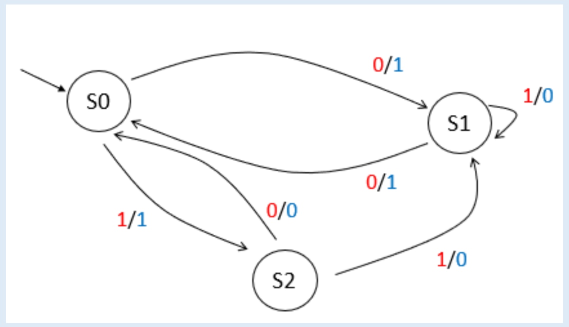
Start state	Input	New State
S1	0	S2
S1	1	S3
S2	0	S1
S2	1	S2
S3	0	S3
S3	1	S2

Trapping invalid input – In the following example S3 captures invalid input there is no what to transition to another state once S3 has been achieved.



Mealy Machines
- The FSM we have looked at so far have a valid and invalid state. The valid state is the accept state
- Mealy machines are a type of FSM that have outputs on each transition and have no end state

Example Mealy Machine



The red number is the input and the blue number is the output.

Input	Output
0 1 1 1 0 1 1 0 1	1 0 0 0 1 1 0 1 1
1 1 1 0 0 1 0 0 0	1 0 0 1 1 1 1 1 1
1 0 1 0 0 1 1 1 0	1 0 0 1 1 0 0 0 1

Corresponding state transition diagram

Start state	Input	New State	Output
S0	0	S1	1
S0	1	S2	1
S1	0	S0	1
S1	1	S1	0
S2	1	S1	0
S2	0	S0	0