Concepts of Computer Science

Number Systems

Chapter Goals

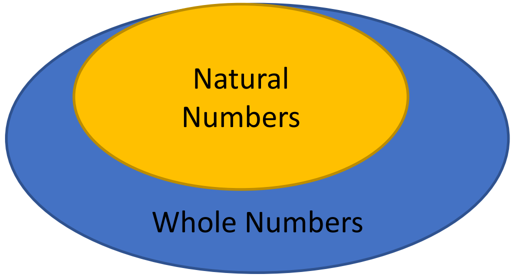
- Lecture 1:
  - Categories of numbers and positional notation
  - Converting numbers between bases
  - Relation between bases 2, 8, and 16

- Lecture 2:
  - Arithmetic in binary
  - Importance of binary in computing

- Number Systems, Lecture 1:
  - Categories of numbers
  - Converting numbers between bases
  - Relation between bases 2, 8, and 16

Numbers

Numbers



- Different categories of numbers

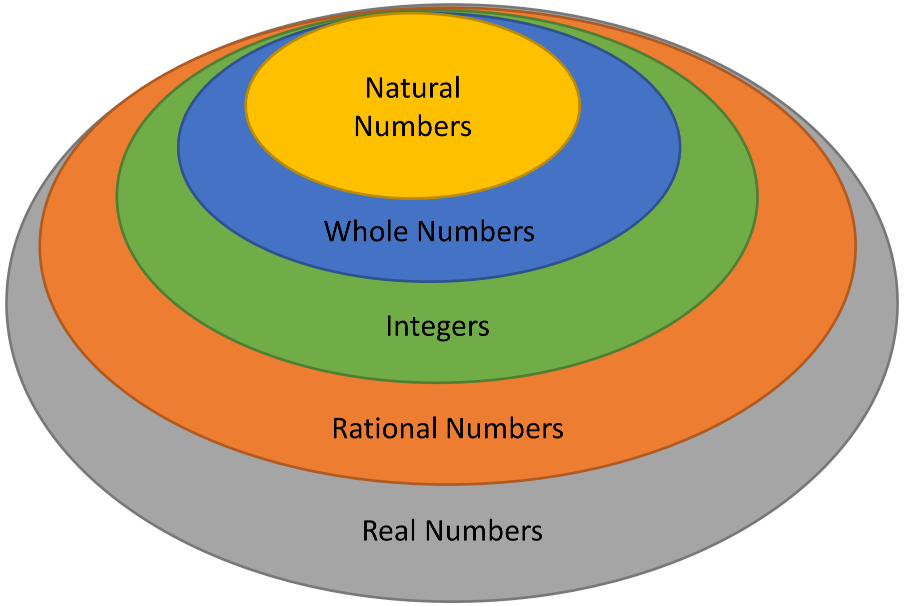
- Natural Numbers $\mathbb{N}$:
  - The counting numbers achieved by adding 1
  - {1, 2, 3, 4, …}

- Whole Numbers $\mathbb{W}$:
  - The natural numbers AND zero
  - Also called non-negative integers
  - {0, 1, 2, 3, 4, …}

This classification can cause arguments amongst mathematicians

Numbers

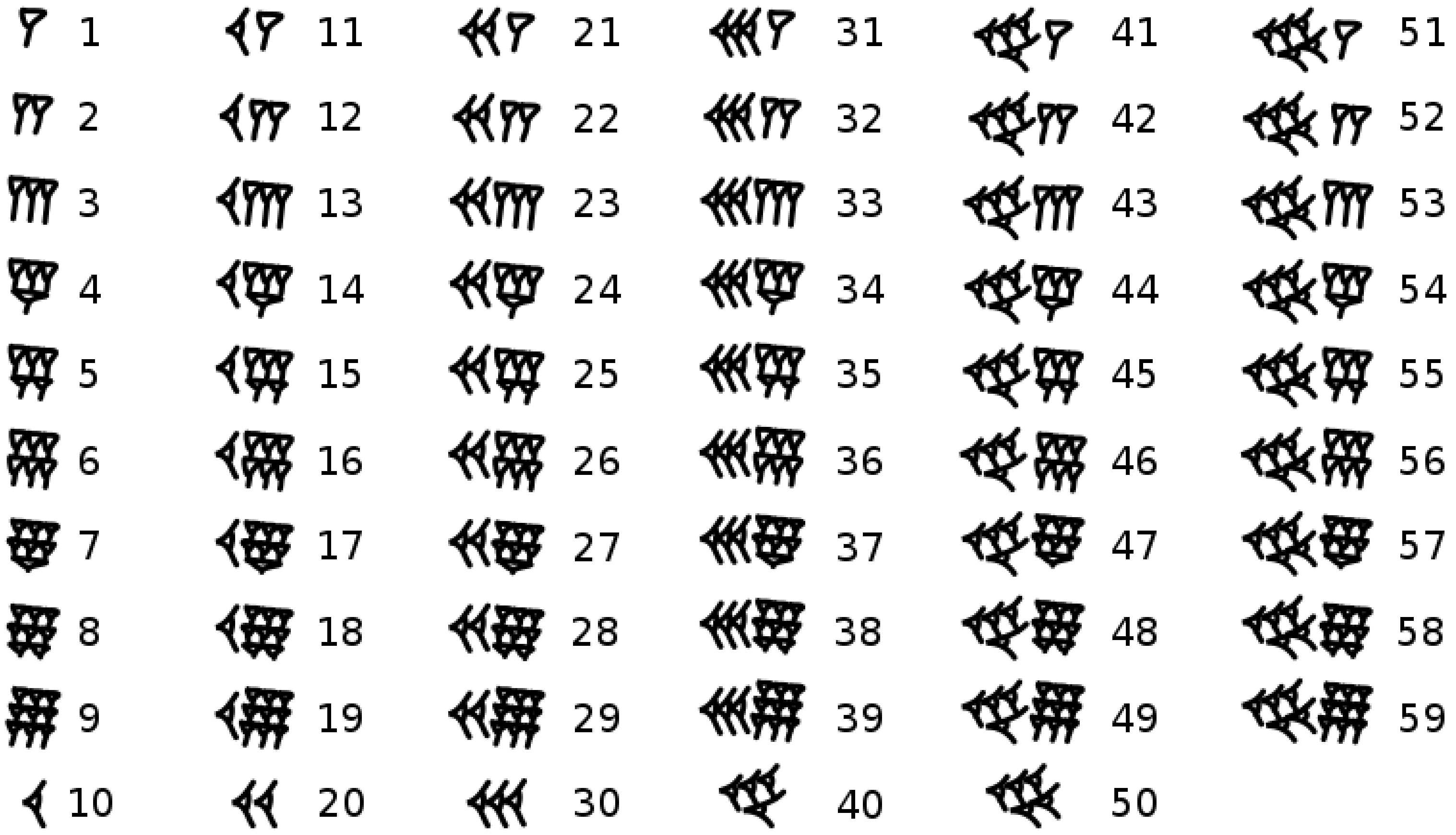
- Integers $\mathbb{Z}$:
  - The Whole Numbers and negative Natural Numbers
  - {…, -4, -3, -2, -1, 0, 1, 2, 3, 4, …}
- Rational Numbers $\mathbb{Q}$:
  - Integer, or a quotient of an integer and a non-zero integer.
  - {Integers, 3/8, -5.28, …}
- Real Numbers $\mathbb{R}$:
  - All “non-imaginary” numbers



Writing numbers

- When writing a number we use **digits**: {0, 1, 2, 3, …, 9}
- The number of digits available to us defines the **base** of the number we are representing:
  - Base 2: {0, 1} (binary)
  - Base 3: {0, 1, 2}
  - …
  - Base 8: {0, 1, 2, 3, 4, 5, 6, 7}
  - Base 9: {0, 1, 2, 3, 4, 5, 6, 7, 8}
  - Base 10: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9} (decimal)
  - …
  - Base 16: {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F} (hexadecimal)
  - Base 2311: {0, 1, …, 9, A, B, …, Z, …, 👽, 💩, ☃} (I just made this up)

Sexagesimal (base 60)



Why?

Image: Josell7 - File:Babylonian_numerals.jpg, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=9862983

Writing numbers

- To avoid **ambiguity**, we can state the base a number is in.

Is “357” in base 8? base 10? base 16?

- We write number in brackets (sometimes) and include the base in **subscript**:

$$(101100101)_2 = (357)_8 = (239)_{10} = (165)_{16}$$

Common bases and their digits

Binary (base 2):

- Digits: 0,1

Octal (base 8):

- Digits: 0,1,2,3,4,5,6,7

Decimal (base 10):

- Digits: 0,1,2,3,4,5,6,7,8,9

Hexadecimal (base 16):

- Digits: 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Why might these be common?

We'll come back to some of them later

Writing bigger numbers

- If we have only 10 digits, how do we write larger decimal numbers?
- In decimal, for any number **greater** than 9, we need to use multiple digits…
- We use **positions** to modify what that digit **represents**:

357

This is really 3 hundreds, 5 tens, and 7 ones…

Positional Notation

Positional Notation
(more in pre-reading material)

- Represent a number by **digits** in **positions**, indexed from the **rightmost** position.

- Increase in position results in increase in magnitude:

Decimal example:

Units	10^0	=	1
Tens	10^1	=	10
Hundreds	10^2	=	100
Thousands	10^3	=	1000

…

The superscript here denotes the position

Positional Notation

(more in pre-reading material)

- To get our value, we multiply the digit at a position by the base raised to the power of it's position:

- E.g. in **decimal** (base 10):

$$357 = 3 \cdot 10^2 + 5 \cdot 10^1 + 7 \cdot 10^0$$

- General form, in base 10, an *n* digit number is represented as:

$$d_{n-1} \ldots d_2 d_1 d_0 = d_{n-1} \cdot 10^{n-1} + \ldots + d_2 \cdot 10^2 + d_1 \cdot 10^1 + d_0 \cdot 10^0$$

Generalisation to other bases
(more in pre-reading material)

- A base is a representation scheme that describes the available number of unique symbols available to represent numbers.

- In **decimal** (base 10) we have 10 unique symbols: {0,1,2,…,9}
- In **octal** (base 8) we have 8 unique symbols: {0,1,2,…,7}

- The general form of positional notation for base *b*:

$$d_{n-1}\ldots d_2 d_1 d_0 = d_{n-1} \cdot b^{n-1} + \ldots + d_2 \cdot b^2 + d_1 \cdot b^1 + d_0 \cdot b^0$$

Converting between bases

Convert from base *b* to decimal

- Use general formula as we saw a moment ago…
- Convert d_i to decimal and multiply it by b^i, then sum the results:

$$\begin{aligned}(357)_8 &= 3 \cdot 8^2 + 5 \cdot 8^1 + 7 \cdot 8^0 \\ &= 3 \cdot 64 + 5 \cdot 8 + 7 \cdot 1 \\ &= 192 + 40 + 7 \\ &= (239)_{10}\end{aligned}$$

Convert from base *b* to decimal

- Use general formula as we did for positional notation:
- Convert d_i to decimal and multiply it by b^i, then sum the results:

$$
\begin{aligned}
(AE4)_{16} &= A \cdot 16^2 + E \cdot 16^1 + 4 \cdot 16^0 \\
&= A \cdot 256 + E \cdot 16 + 4 \cdot 1 \\
&= 10 \cdot 256 + 14 \cdot 16 + 4 \cdot 1 \\
&= 2560 + 224 + 4 \\
&= (2788)_{10}
\end{aligned}
$$

Note here that the hexadecimal “A” is equivalent to “10” in decimal

Convert from decimal to base *b*

- Algorithm (convert decimal integer to base b):
    - While number is not zero
        - Divide number by new base b
        - Note down remainder
        - Replace number by quotient
    - The remainders listed in reverse order is the value in base b.
- What is happening here?

Convert from decimal to base *b*

example 1

Convert $(239)_{10}$ to base 8:

$(239)_{10}$ / 8 = 29 remainder **7**

29 / 8 = 3, remainder **5**

3 / 8 = 0, remainder **3**

= $(357)_8$

Convert from decimal to base *b*

example 2

Convert $(239)_{10}$ to base 16:

$(239)_{10}$ / 16 = 14 remainder **15 (F)**

14 / 16 = 0, remainder **14 (E)**

= $(EF)_{16}$

Why?

Convert from base *x* to base *y*

- Actually the same algorithm as with decimal:
  - While number is not zero
    - Divide number by new base b
    - Note down remainder
    - Replace number by quotient
  - Read remainders in reverse

- However now we are doing division in base x.

- Can be significantly more difficult if we are not careful…

Convert from base x to base y

example 3

Convert $(A6C)_{14}$ to base $(12)_{10}$ (actually base $(C)_{14}$):

$(A6C)_{14}$ / C = C3 remainder **4**
C3 / C = 10, remainder **3**
10 / C = 1, remainder **2**
1 / C = 0, remainder **1**
= $(1234)_{12}$

Why?

Note that these numbers are all base 14!

How's your long division in base 14?

Requires thinking in bases which aren't intuitive to us.

Instead we usually just convert base x to decimal, and then convert from decimal to base y.

Cheats approach to convert between bases of power 2

- Octal:
  - Starting from the right, group digits into groups of 3.
  - Convert group by group to octal.

$(1001001101111)_2$				
1	001	001	101	111
1	1	1	5	7
$(11157)_8$				

Why does this work?

- Hexadecimal:
  - Starting from the right, group digits into groups of 4.
  - Convert group by group to hexadecimal.

$(1001001101111)_2$			
1	0010	0110	1111
1	2	6	F
$(126F)_{16}$			

- Lecture 2:
  - Arithmetic in binary
  - Importance of binary in computing
  - Looking ahead to using number representations

Binary Addition

Binary arithmetic - Addition
(more in pre-reading material)

- Arithmetic can be done directly on binary numbers as long as we remember the suitable addition and multiplication tables.

- In particular we must remember that **1 + 1 = 10** in binary. This gives a **carry.**

Why?

Binary Addition Table		
+	**0**	**1**
0	0	1
1	1	10

Binary Multiplication Table		
·	**0**	**1**
0	0	0
1	0	1

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

$$\begin{array}{r} 87_{10} \\ \underline{+\ \ 75}_{10} \end{array}$$

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

```
   1 0 1 0 1 1 1
+  1 0 0 1 0 1 1
```

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

$$\begin{array}{cccccccc} & & & & & & 1 & \\ & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\ + & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ \hline & & & & & & & 0 \end{array}$$

Carry Values

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

```
             1 1
    1 0 1 0 1 1 1
+   1 0 0 1 0 1 1
-----------------
              1 0
```

Carry Values

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

```
         1 1 1      ← Carry Values
   1 0 1 0 1 1 1
 + 1 0 0 1 0 1 1
 ---------------
           0 1 0
```

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

```
            1 1 1 1
    1 0 1 0 1 1 1
+   1 0 0 1 0 1 1
-----------------
            0 0 1 0
```

Carry Values

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

$$
\begin{array}{cccccccc}
 & & 1 & 1 & 1 & 1 & 1 & \\
 & 1 & 0 & 1 & 0 & 1 & 1 & 1 \\
+ & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\
\hline
 & & & 0 & 0 & 0 & 1 & 0
\end{array}
$$

Carry Values

Binary arithmetic - Addition

(more in pre-reading material)

- Example:

```
    1 1 1 1 1
  1 0 1 0 1 1 1
+ 1 0 0 1 0 1 1
---------------
    1 0 0 0 1 0
```

Carry Values

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

```
  1   1 1 1 1 1
    1 0 1 0 1 1 1
+   1 0 0 1 0 1 1
-----------------
    0 1 0 0 0 1 0
```

Carry Values

Binary arithmetic - Addition
(more in pre-reading material)

- Example:

```
 1   1 1 1 1 1
   1 0 1 0 1 1 1
 + 1 0 0 1 0 1 1
 ---------------
 1 0 1 0 0 0 1 0
```

Carry Values

Binary Subtraction

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of "**borrowing**" from a higher position.

$$\begin{array}{r} 87_{10} \\ -\ \ 59_{10} \\ \hline \end{array}$$

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of "**borrowing**" from a higher position.

$$
\begin{array}{r}
1\ 0\ 1\ 0\ 1\ 1\ 1 \\
-\quad 1\ 1\ 1\ 0\ 1\ 1 \\
\hline
\end{array}
$$

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of “**borrowing**” from a higher position.

	1	0	1	0	1	1	1
-		1	1	1	0	1	1
							0

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of “**borrowing**” from a higher position.

	1	0	1	0	1	1	1
-		1	1	1	0	1	1
						0	0

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of "**borrowing**" from a higher position.

Cannot subtract 1 from 0.
Need to borrow from
next position

	1	0	1	0	1	1	1
-		1	1	1	0	1	1
					1	0	0

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of “**borrowing**” from a higher position.

This is now 10 – 1 (in binary)

$$\begin{array}{cccccccc} & 1 & 0 & \overset{0}{\cancel{1}} & {}^{1}0 & 1 & 1 & 1 \\ - & & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline & & & & & 1 & 0 & 0 \end{array}$$

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of "**borrowing**" from a higher position.

Cannot subtract 1 from 0. Need to borrow from next position

But there is nothing to borrow from here.

So we need to go to the next position to borrow.

$$\begin{array}{cccccccc} & & & 0 & & & & \\ & 1 & 0 & \cancel{1} & {}^{1}0 & 1 & 1 & 1 \\ - & & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline & & & & 1 & 1 & 0 & 0 \end{array}$$

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of "**borrowing**" from a higher position.

$$
\begin{array}{cccccccc}
 & 0 & & 0 & & & & \\
 & \cancel{1} & {}^{1}0 & \cancel{1} & {}^{1}0 & 1 & 1 & 1 \\
- & & 1 & 1 & 1 & 0 & 1 & 1 \\
\hline
 & & & & 1 & 1 & 0 & 0
\end{array}
$$

Binary arithmetic - Subtraction
(more in pre-reading material)

For subtraction we have a concept of "**borrowing**" from a higher position.

This is now 10 – 1
(in binary)

	0	1	$^{1}0$				
	$\cancel{1}$	$^{1}\cancel{0}$	$\cancel{1}$	$^{1}0$	1	1	1
-		1	1	1	0	1	1
				1	1	0	0

Binary arithmetic - Subtraction

(more in pre-reading material)

For subtraction we have a concept of “**borrowing**” from a higher position.

$$\begin{array}{rccccccc} & 0 & 1 & {}^{1}0 & & & & \\ & \cancel{1} & {}^{1}\cancel{0} & \cancel{1} & {}^{1}0 & 1 & 1 & 1 \\ - & & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline & & & 1 & 1 & 1 & 0 & 0 \end{array}$$

Binary arithmetic - Subtraction

(more in pre-reading material)

For subtraction we have a concept of "**borrowing**" from a higher position.

$$
\begin{array}{cccccccc}
 & 0 & 1 & {}^{1}0 & & & & \\
 & \cancel{1} & {}^{1}\cancel{0} & \cancel{1} & {}^{1}0 & 1 & 1 & 1 \\
- & & 1 & 1 & 1 & 0 & 1 & 1 \\
\hline
 & & 0 & 1 & 1 & 1 & 0 & 0
\end{array}
$$

Binary arithmetic - Subtraction

(more in pre-reading material)

For subtraction we have a concept of "**borrowing**" from a higher position.

These two "leading zeros" are not needed, so can be discarded.

$$\begin{array}{cccccccc} & 0 & 1 & {}^{1}0 & & & & \\ & \cancel{1} & {}^{1}\cancel{0} & \cancel{1} & {}^{1}0 & 1 & 1 & 1 \\ - & & 1 & 1 & 1 & 0 & 1 & 1 \\ \hline & 0 & 0 & 1 & 1 & 1 & 0 & 0 \end{array} \quad = 28_{10}$$

Binary arithmetic - Multiplication

For multiplication, we multiply one of the numbers by each digit in the other number, and then sum them together.

$$
\begin{array}{rrrrr}
 & & 1 & 0 & 1 \\
\cdot & 1 & 0 & 1 & 1 \\
\hline
\end{array}
$$

Binary arithmetic - Multiplication

$$
\begin{array}{rrrrr}
 & & 1 & 0 & 1 \\
\cdot & 1 & 0 & 1 & 1 \\
\hline
\end{array}
$$

Binary arithmetic - Multiplication

$$
\begin{array}{rcccc}
 & & \boxed{1 \quad 0 \quad 1} \\
\cdot & 1 & 0 & 1 & \boxed{1} \\
\hline
 & & 1 & 0 & 1 \\
\hline
\end{array}
$$

Binary arithmetic - Multiplication

		$\boxed{1}$	$\boxed{0}$	$\boxed{1}$
·	1	0	$\boxed{1}$	1
		1	0	1
	1	0	1	

Binary arithmetic - Multiplication

			1	0	1
·		1	0	1	1
			1	0	1
		1	0	1	
	0	0	0		

Binary arithmetic - Multiplication

				1	0	1
·			1	0	1	1
				1	0	1
			1	0	1	
		0	0	0		
	1	0	1			

Binary arithmetic - Multiplication

Sum up these intermediate values

				1	0	1
·			1	0	1	1
				1	0	1
			1	0	1	
		0	0	0		
	1	0	1			

Binary arithmetic - Multiplication

					1	0	1
·				1	0	1	1
					1	0	1
				1	0	1	
		0	0	0			
+	1	0	1				

Binary arithmetic - Multiplication

				1	0	1
·			1	0	1	1
				1	0	1
			1	0	1	
		0	0	0		
+	1	0	1			
	1	1	0	1	1	1

Why does all this matter?

Why do we want this?

- Development of computers as we know them today required the ability to **store and manipulate information**.
- Being able to **reliably** do this is key to the success of modern computers, and developing hardware to do this is tricky.
- It’s much easier to differentiate **two** levels of electrical **voltage** (off vs on), which makes the **binary** system so attractive
- Often we treat low voltage as 0, and high voltage as 1.

Why do we want this?

- We call a single binary value a “**bit**” or “**b**inary dig**it**”
- We then group bits together into a “**byte**”, which often refers to 8 bits
- We often use some multiple of 8 bits to represent data in our machine, and modern computers are designed to carry out instructions on this. We call this a “**word**”. The “**word size**” of the computer defines the maximum number of bits the processor can operate on at a time.

Representation

- You may have noticed that all the numbers we have discussed so far have been Whole Numbers {0, 1, 2, 3, …}

- What about …
  - Negative numbers
  - Floating Point numbers

- These have special properties (a sign, a fractional part etc.) which we would want to represent

Representation

- Data can be quite complex, but we can utilise binary bits to build up a representation of these complex things from the atomic 0s and 1s.

- As long as we can agree on how to convert a collection of 0s and 1s to our “thing”, and vice-versa, then we can represent it on the machine and utilise it for computation.

- In the next session we will look at representation of data.

Why do we want this?

- We can go even further than representing the categories of numbers discussed earlier.

- We will use numbers to represent many different things:
  - Characters
  - Colors
  - Shapes
  - Objects
  - …

Have a think between now and then. How would you represent these things if all you have is binary?