Australian National University

EMSC4033/8033

Semester 1 2020

Computational Geoscience

Andrew Valentine

Research School of Earth Sciences

andrew.valentine@anu.edu.au

Who, what, where?

- The course team:
  - Andrew Valentine
  - Rebecca McGirr
- For the next three weeks:
  - Mon/Wed/Thur/Fri: 10-12
  - Wed/Fri: 2-5
- All classes will be online via Zoom
- Classes will be focused on practical exercises

Assessment

- In-class assessment (20%)
  - 10-12 Thurs 7th May
  - *Please see me today if there is any chance you cannot attend for this*
- Assessed programming project
  - More details towards the end of the course
  - Due: 5pm Friday 5th June
- Four **optional** drop-in sessions will be held:
  - Dates/times tbc...
  - Opportunity to ask questions and get help
  - No need to attend if you have no questions!

Assessment

- Assessed programming project
  - More details towards the end of the course
  - Due: 5pm Friday 5th June
- When you get stuck, you are encouraged to:
  - Talk to friends/colleagues
  - Make use of online resources
- However, you **must**
  - **Write your code yourself**: don't copy and paste from a friend's solution or a website
  - **Understand** how your solution works
  - Be able to **explain** why you chose to solve the problem the way you did

Assessment

- Assessed programming project
  - More details towards the end of the course
  - Due: 5pm Friday 5th June

- Graded based on:
  - Submitted solution to the task, **and**
  - Oral examination (via Zoom; 15-20 minutes)

- **If you do not attend the oral examination you will receive no credit!**

- Oral examinations will be arranged for the week commencing 8th June (dates and times to be confirmed). *If you expect to be unavailable during this week please talk to me today.*

Please give us feedback…!

- If you have any comments or suggestions about the course (or its online delivery), please get in touch!
  - Talk to/email me or Rebecca
  - Send a message via the AD Honours/Masters (Dr Rhodri Davies)
  - SELT? – Not running during COVID-19?

"I think it would be worthwhile to mention in the first lecture that learning programming can be a **steep learning curve** for some people and that **it does get easier** the more you use it and once you are able to debug code. Initially, I was doubtful at my ability to learn python and do well in the course which temporarily decreased my productivity and motivation to succeed."

Student, 2019

Why?

Why does a geoscientist need to know about programming?

What is a programming language?

What is an algorithm?

How do you make a cup of coffee?

- Need to spell everything out carefully – no shortcuts or assumptions
- Task can be broken into sub-tasks, e.g. “boil kettle”, provided they are well-defined
    - Sub-task may appear in other contexts, e.g. to make tea or hot chocolate

How do you make dinner?

- Our ‘recipe’ for coffee works with *any* mug

- ‘Mug’ in the recipe is an *abstract* object assumed to have certain properties

Can you make coffee in a bucket?

Can you make coffee in a cat?

A *type definition* is a set of rules defining the properties and behaviours we can expect a certain object to have.

Every object is an *instance* of some abstract type. It is guaranteed to have all the properties and behaviours set out in the type definition.

- Our ‘recipe’ for coffee works with *any* mug

- ‘Mug’ in the recipe is an *abstract* object assumed to have certain properties

- When we execute the recipe, we select a real, physical mug and let it ‘stand in for’ the abstract entity

A *function* or *subroutine* is a sequence of instructions designed to perform a particular task, packaged into readily-reusable form.

- To use a function, we only need to understand its *interface* – we don't care about how it works internally
  - What information do we have to give to the function? ('*Arguments*')
  - What information does it give us back? ('*Return value*')

Programming is like Lego - we connect lots of simple pieces together to make something much more sophisticated

$$1+2+3+\ldots+10 = ?$$

Recap

- To tackle a complex problem, we **break it down** into manageable pieces.
- First understand the **algorithm** we want to create, then worry about how to **implement** that in a particular **programming language**.
- Important to spell everything out in detail – computers are **dumb**.
- Write generic ‘recipes’ or **functions** that work for **all** objects of a certain **type**
- Then **call** the function passing it specific **instances** as **arguments**

I am thinking of a number between 1 and 10…

A *variable* is a named ‘slot’ in which a piece of information can be stored temporarily, accessed and updated.

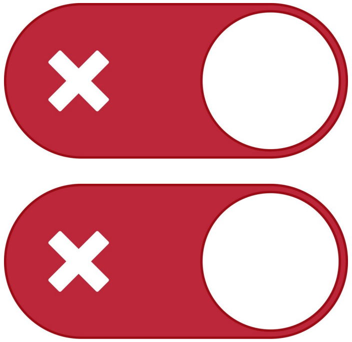
A variable has an associated type.

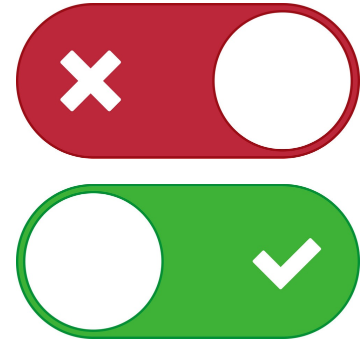
A *constant* is a piece of information which is built into a program and cannot be changed.

Variables and constants are stored in your computer’s RAM (random access memory)

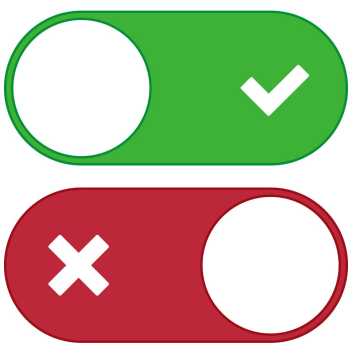
Computing is built around the **binary** system

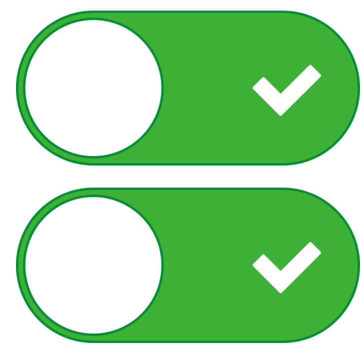
- Information is stored using **bits** (binary digits)
- A bit can take the value 0 or 1 – it’s essentially a switch
- To represent more complex information we combine bits

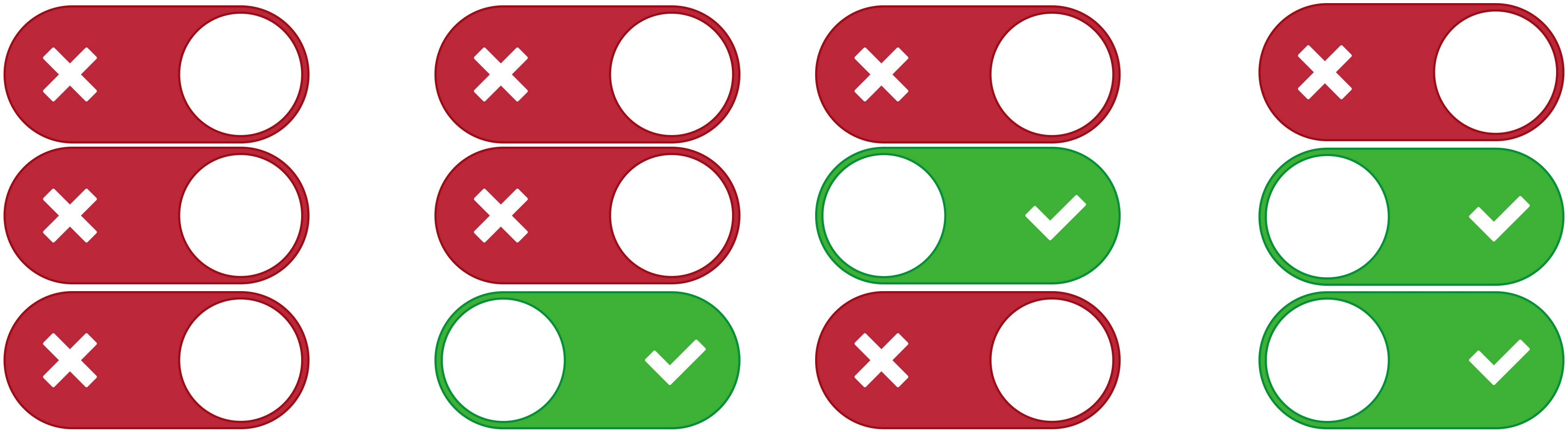




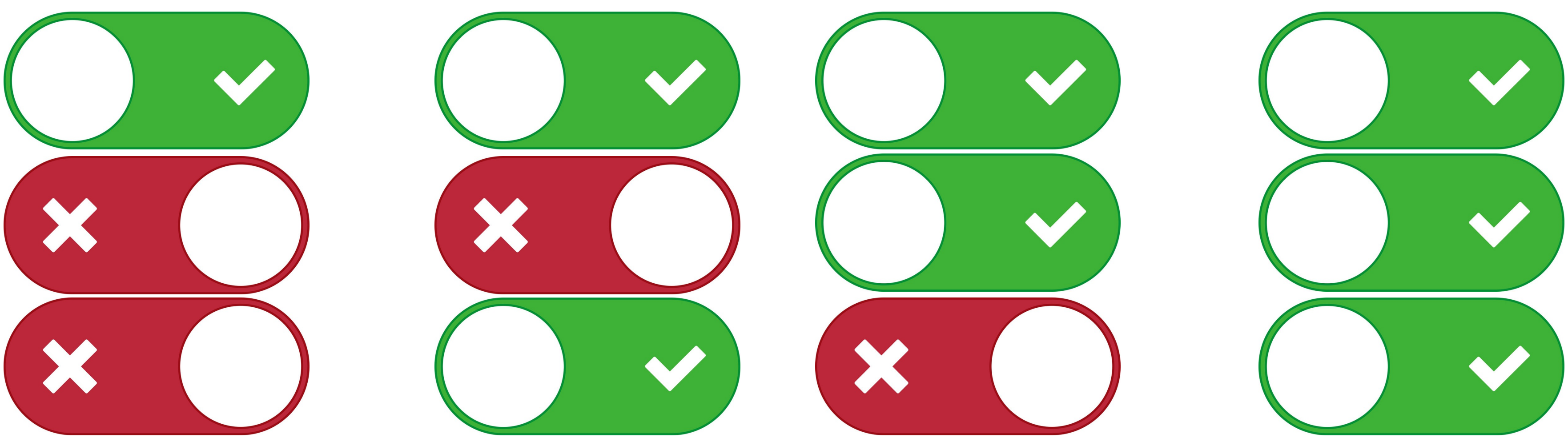
A 2-bit system has 4 states







A 3-bit system has 8 states



In general, a set of N bits can represent 2^N different states

- We can only represent a finite, discrete (countable) set of quantities
- Every kind of information must be mapped onto this discrete set

N	2^N
1	2
2	4
3	8
4	16
5	32
6	64
7	128
8	256
16	65536
32	4294967296
64	$\approx 1.845 \times 10^{19}$

A *byte* is a unit of 8 bits

A ***kilo****byte* is 2^{10}=1024 bytes
A ***mega****byte* is 2^{20} bytes (1024 kilobytes)
A ***giga****byte* is 2^{30} bytes (1024 megabytes)
And so on: terabytes, petabytes, exabytes…

NB: Sometimes kilo-/mega-/giga- etc are used in their usual SI sense, i.e. 10^3/10^6/10^9 bytes. An SI terabyte is only 90% the size of a base-2 terabyte!

Three main ‘building blocks’ of information:

- Integers (1,2,3,…)
- Real/floating-point numbers (1.326, -2.33162, π,...)
- Alphanumeric characters (‘a’, ‘b’, ‘1’,...

Integers are already discrete, so can easily be mapped onto binary system

1. Choose how many bits you are going to use (typically 32 bits/4 bytes)
2. Decide whether you need to represent negative numbers
3. Decide whether to read the bits from left-to-right or right-to-left ('big endian or little endian')

Unsigned	Signed
0 0 0 → 0	0 0 0 → 0
0 0 1 → 1	0 0 1 → 1
0 1 0 → 2	0 1 0 → 2
0 1 1 → 3	0 1 1 → 3
1 0 0 → 4	1 0 0 → ?
1 0 1 → 5	1 0 1 → -1
1 1 0 → 6	1 1 0 → -2
1 1 1 → 7	1 1 1 → -3

Alphanumeric data is also discrete and easy to map, by defining a standard ordering:

0 0 0 → a
0 0 1 → b
0 1 0 → c
0 1 1 → d
1 0 0 → A
1 0 1 → B
1 1 0 → C
1 1 1 → D

Usually the ASCII (American Standard Code for Information Interchange) table is used…

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	`
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	|
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Real numbers are harder: we have to **discretize** them

- Cannot represent most numbers exactly
- Common source of ‘numerical error’ in calculations

Represent a.bcde as $p \times 2^q$ where p and q are integers

$$0.13 \rightarrow 8724152 \times 2^{-26} = 0.12999999952 \approx 0.13$$

$$\overbrace{0}^{+}\quad \underbrace{100001010001111010111000}_{8724152\text{ (24 bits)}}\quad \overbrace{1}^{-}\quad \underbrace{011010}_{26\text{ (6 bits)}}$$

Floating point arithmetic can give weird results…

0.1+0.1+ ... + 0.1 = 0.999999999999

Any number ‘too large’ to be representable is assumed to be infinity

Any number ‘too small’ to be representable is assumed to be 0

You will sometimes see values of ‘NaN’, meaning ‘**N**ot **A** **N**umber’ – this means something weird happened in your calculation