

Number Representation, Functions and Procedures

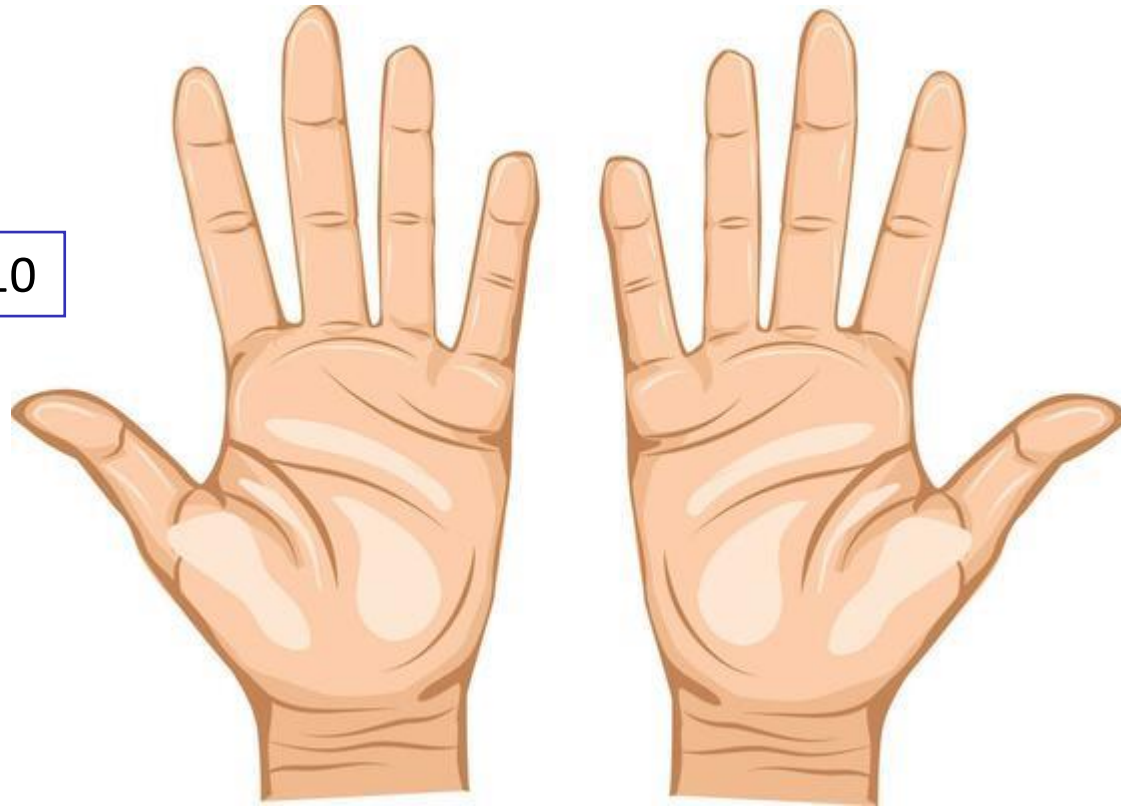
Alexander Lam

Why do we have 10 digits?

0 1 2 3 4 5 6 7 8 9

10 11 12 13 14 15 16 17 18 19

We count from 1 to 10



Decimal (Base-10) System

How does Bart Simpson count?

0 1 2 3 4 5 6 7

10 11 12 13 14 15 16 17

20 ...

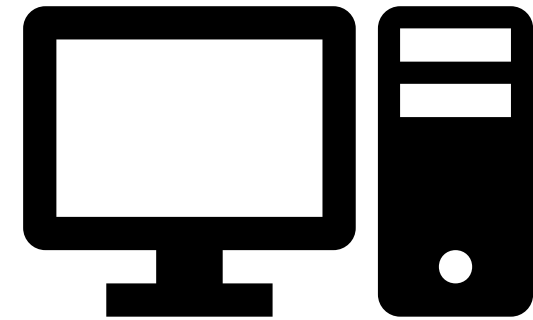
He counts from 1 to 8

Octal (Base-8) System



How does a Computer count?

0 1
10 11
100 101 110 111



It counts from 1 to 2

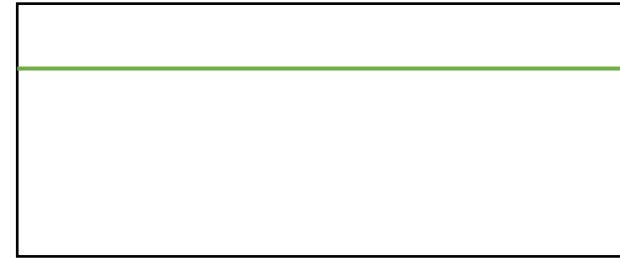
Binary (Base-2) System

Why Binary?

- In a binary electrical circuit, we either have a current or we don't.

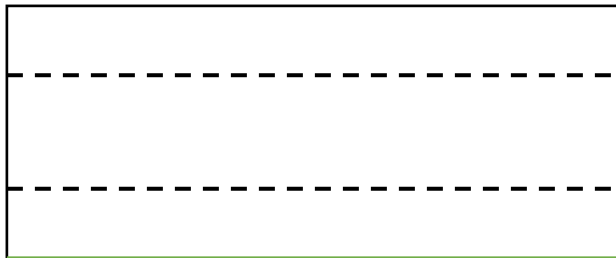


0

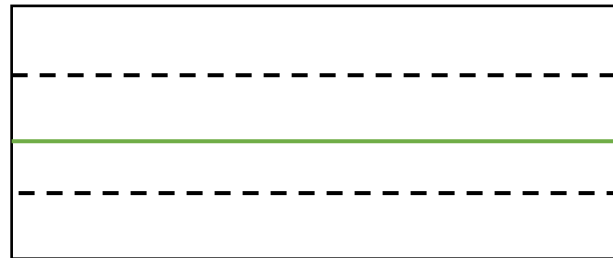


1

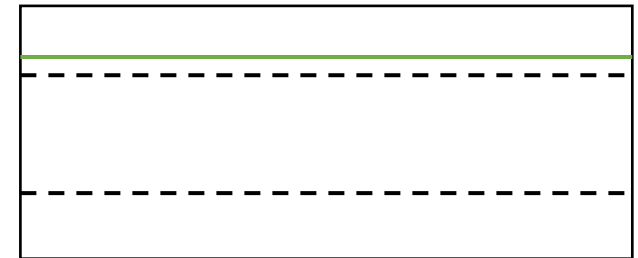
- What about a base-3 electrical circuit?



0



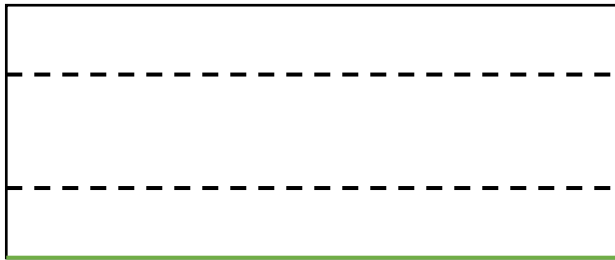
1



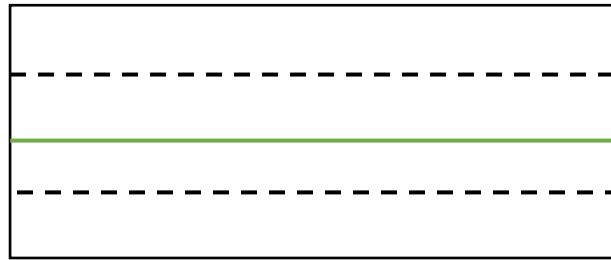
2

Why Binary?

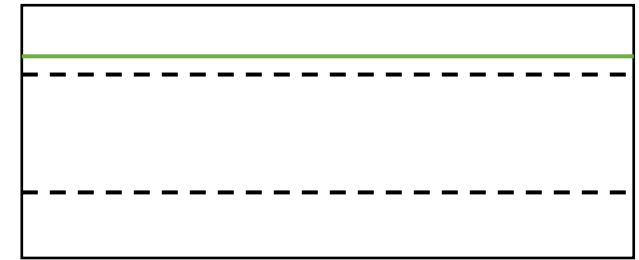
- What about a base-3 electrical circuit?



0

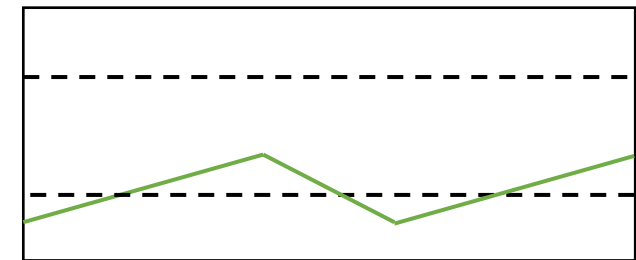


1



2

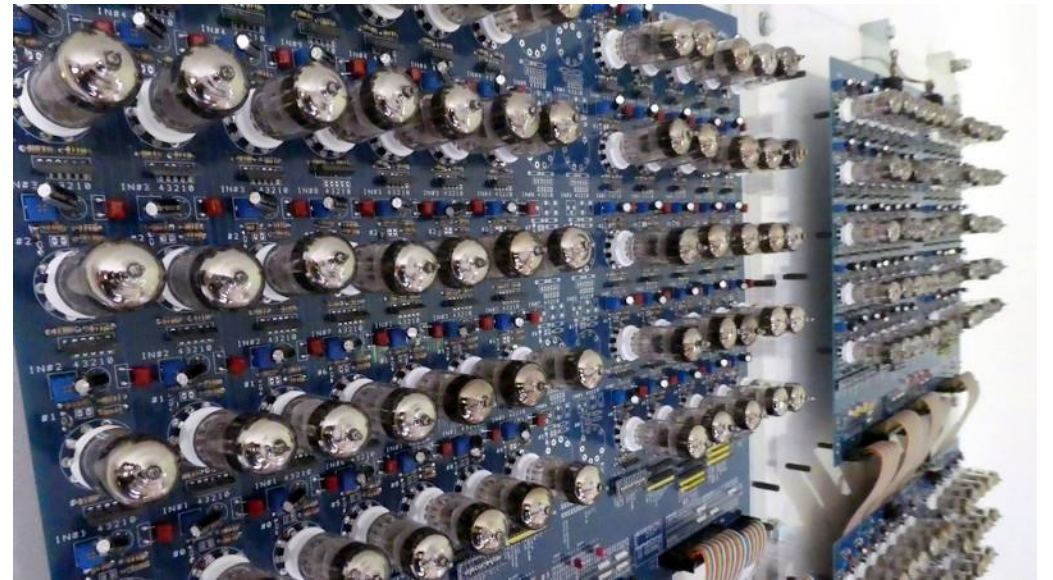
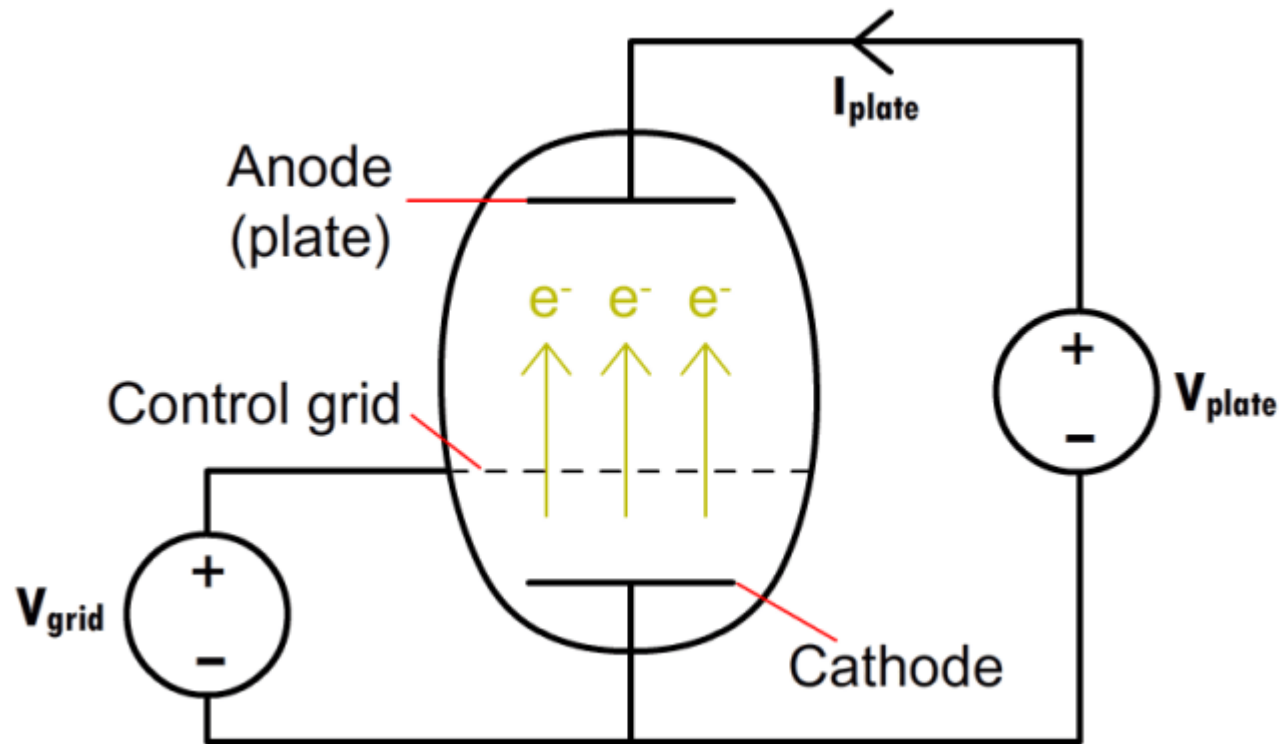
- **Issues:**
 - Signal noise
 - Signal degradation from old parts
 - Consistency across multiple components



- These issues are particularly problematic in **transistors**

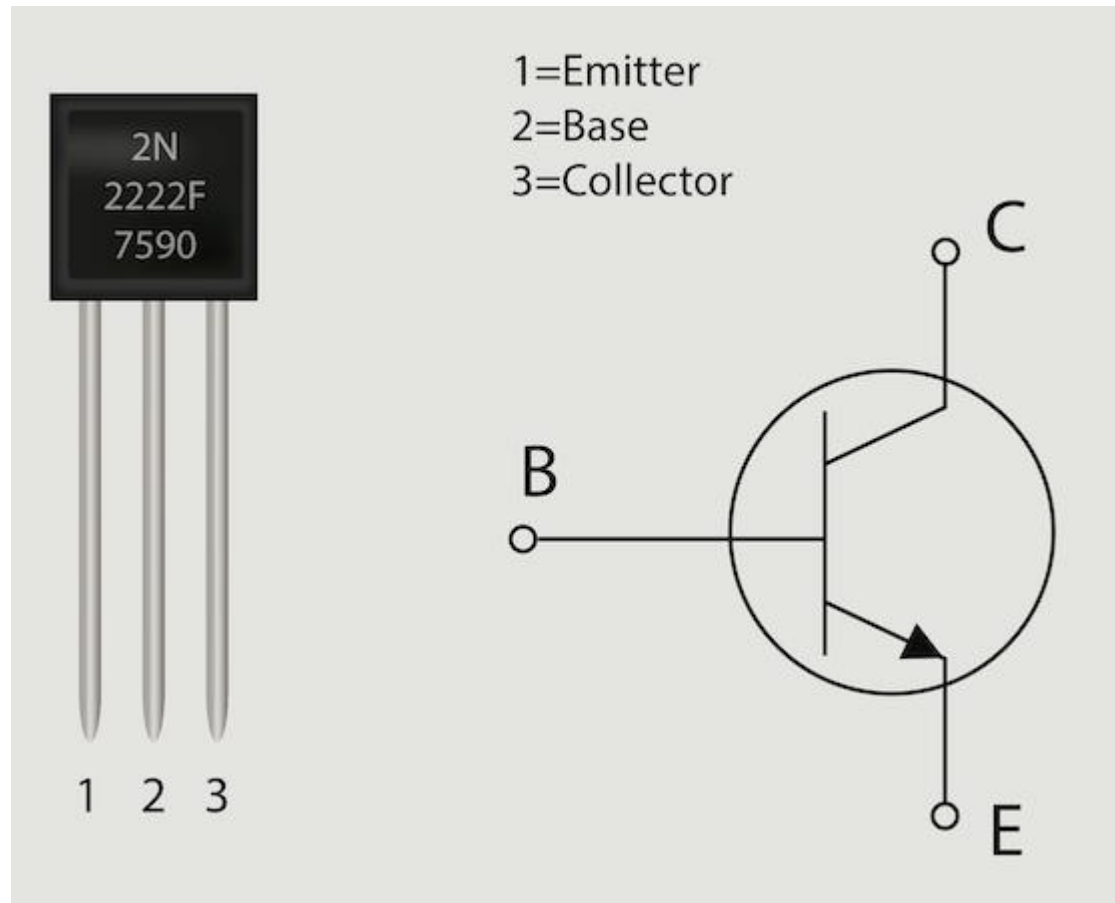
Transistors

- A transistor is a device that can modify electrical signals.
 - We can use it to switch signals on (1) or off (0).



Transistors

- A transistor is a device that can modify electrical signals.
 - We can use it to switch signals on (1) or off (0).



Back to the Decimal System

- $123 = 1 \times 100 + 2 \times 10 + 3 \times 1$
 $= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$
- $1234 = 1 \times 1000 + 2 \times 100 + 3 \times 10 + 4 \times 1$
 $= 1 \times 10^3 + 2 \times 10^2 + 3 \times 10^1 + 4 \times 10^0$

What about the binary system?

- $1011 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 8 + 0 + 2 + 1 = 11$
- Each **binary digit** is called a **bit**
- 10101 21
- 110110 54
- 10100101 165
- 111010000000110 29702

Binary to Decimal

- 10101

Binary to Decimal

- 110110

Binary to Decimal

- 10100101

Binary to Decimal

- 111010000000110

Decimal to Binary

- $11_{10} = 8 + 2 + 1$

$$= 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 1011_2$$

First bit may as well be a 1

- $50_{10} = 32 + 16 + 2$

50 is between 32 (2^5) and 64 (2^6)

$$= 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 110010_2$$

- 178_{10} in binary?

178 is between 128 (2^7) and 256 (2^8)

- $178_{10} = 128 + 50$

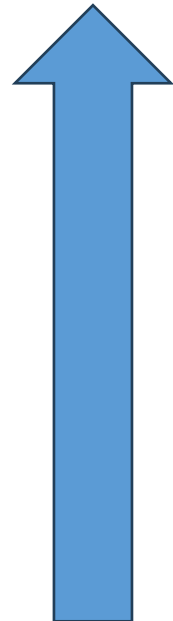
- $178_{10} = 128 + 32 + 16 + 2$

$$= 1 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$$
$$= 10110010_2$$

Division Method for Conversion

We can convert a base 10 number to base k by repeatedly dividing the number by k and taking the remainders in reverse.

- $50 / 2 = 25 \text{ R } 0$
- $25 / 2 = 12 \text{ R } 1$
- $12 / 2 = 6 \text{ R } 0$
- $6 / 2 = 3 \text{ R } 0$
- $3 / 2 = 1 \text{ R } 1$
- $1 / 2 = 0 \text{ R } 1$



110010

Why does this work?

Why does the Division Method work? (Optional)

I will give you 90% of the answer.

- We can convert a base 10 number to base 2 by repeatedly dividing the number by 2 and taking the remainders in reverse.
- **Fact:** If a decimal number ends in 0 and we delete that 0, the number is divided by 10.
- **Fact:** If a binary number ends in 0 and we delete that 0, the number is halved. (This is also known as **bit-shift**)
- **Fact:** If a binary number ends in 1 and we delete that 1, the number is subtracted by 1 **then** halved.

Why does the Division Method work? (Optional)

We can convert a base 10 number to base 2 by repeatedly dividing the number by 2 and taking the remainders in reverse.

- **Fact:** If a binary number ends in 0 and we delete that 0, the number is halved.
- **Fact:** If a binary number ends in 1 and we delete that 1, the number is subtracted by 1 **then** halved.

- $110010_2 = 50_{10}$
- $11001_2 = 25_{10}$
- $1100_2 = ((25-1)/2)=12_{10}$
- $110_2 = 6_{10}$
- $11_2 = 3_{10}$
- $1_2 = ((3-1)/2)=1_{10}$

Challenge: Why does it work for converting base 10 to base k?

Octal Number System

Octal (base 8)

- $7623_8 = 7 \times 8^3 + 6 \times 8^2 + 2 \times 8^1 + 3 \times 8^0$
 $= 3584 + 384 + 16 + 3 = 3987_{10}$
- $29702_{10} = 7 \times 4096 + 2 \times 512 + 6$
 $= 7 \times 8^4 + 2 \times 8^3 + 0 \times 8^2 + 0 \times 8^1 + 6 \times 8^0$
 $= 72006_8$

Octal Number System

Octal (base 8)

- $29702_{10} / 8 = 3712 \text{ R } 6$
- $3712 / 8 = 464 \text{ R } 0$
- $464 / 8 = 58 \text{ R } 0$
- $58 / 8 = 7 \text{ R } 2$

72006_8

Hexadecimal Number System

Hexadecimal (base 16)

We only have 10 digits from 0 to 9. What do we use for remaining 6 digits?

- A = 10, B = 11, C = 12, D = 13, E = 14, F = 15
- $29702_{10} = 7 \times 4096 + 4 \times 256 + 6$
$$= 7 \times 16^3 + 5 \times 16^2 + 0 \times 16^1 + 6 \times 16^0 = 7506_{16}$$
- $ADD_{16} = 10 \times 16^2 + 13 \times 16^1 + 13 \times 16^0 = 2781_{10}$

Hexadecimal Number System

Hexadecimal (base 16)

- $29702_{10} / 16 = 1856 \text{ R } 6$
- $1856 / 16 = 116 \text{ R } 0$
- $116 / 16 = 7 \text{ R } 4$

7406_{16}

Binary, Octal, and Hexadecimal

$$29702_{10} =$$

- 111010000000110_2
- 72006_8
- 7406_{16}

The “decimal” value of the binary group becomes the octal/hexadecimal digit!

Make sure you group the bits from right to left!

111	010	000	000	110
7	2	0	0	6

0111	0100	0000	0110
7	4	0	6

Binary, Octal, and Hexadecimal

- $1110100000001100101110111100101_2 = 1946574309_{10}$

Octal

- 1 110 100 000 001 100 101 110 111 100 101₂
- 16401456745₈

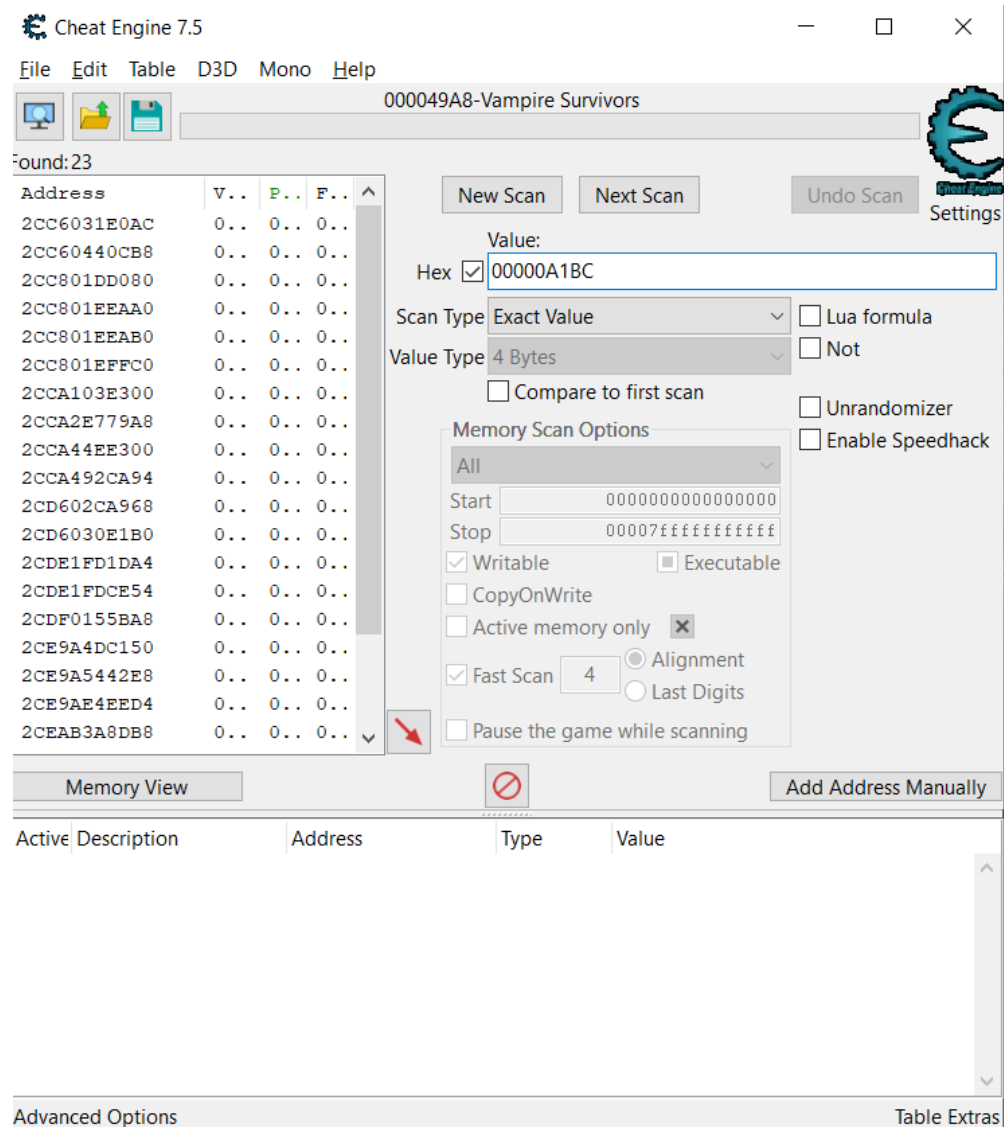
Hexadecimal

- 111 0100 0000 0110 0101 1101 1110 0101₂
- 74065DE5

Binary, Octal, and Hexadecimal

- Converting between binary, octal, and hexadecimal is easy
 - 111 010 000 000 110
 - 111 0100 0000 0110
- Converting between decimal and these bases is not easy
 - Decimal is often not ideal for computers
- Hexadecimal and octal are used for representing binary data compactly
 - **Hexadecimal:** Colours, memory addresses, cryptography, HTML colours
 - **Octal:** Setting access privileges in Linux systems

Where you may actually use Hexadecimal



#66CC66		#33CC66		#00CC66	
#66CC33		#33CC33		#00CC33	
#66CC00		#33CC00		#00CC00	
#6699FF		#3399FF		#0099FF	
#6699CC		#3399CC		#0099CC	
#669999		#339999		#009999	
#669966		#339966		#009966	
#669933		#339933		#009933	
#669900		#339900		#009900	

Exercise

Convert the following decimal numbers to octal, and hexadecimal

- 51966
- 64206
- 712173
- 14600926

Decimal to Hexadecimal

- 51966

Computer Memory

- Data needs to be stored somewhere before being manipulated by a computer.
 - Data in use is typically stored in the computer **memory**.
- The typical unit of digital information is a **byte** (B).
 - A byte is normally made of 8 bits.
 - 8 bits: 00000000_2 to 11111111_2
 - 0_{10} to 255_{10}
- Another unit of digital information is a **word**
 - A word may be either 2 bytes (16 bits) or 4 bytes (32 bits)
 - 16 bits: 0_{10} to 65535_{10} , 32 bits: 0_{10} to 4294967295_{10}

Computer Memory

- KB = kilobyte = 1 000 bytes (text files)
- MB = megabyte = 1 000 000 bytes (image files)
- GB = 1 000 000 000 bytes (memory size)
- TB = 1 000 000 000 000 (hard-disk size)
- PB = 1 000 000 000 000 000 (cloud storage size)

Note: K, M, etc used to mean 1024 and 1024 x 1024 etc. due to their use in RAM.
It is now standard notation for a kilobyte to always be 1000 bytes etc.

- For 1024 bytes, we have KiB (kibibyte).

Remember:

- $2^1 = 2$
- $2^2 = 4$
- $2^3 = 8$
- $2^4 = 16$
- $2^5 = 32$
- $2^6 = 64$
- $2^7 = 128$
- $2^8 = 256$

- $2^9 = 512$
- $2^{10} = 1024$
- $2^{11} = 2048$
- $2^{12} = 4096$
- $2^{13} = 8192$
- $2^{14} = 16384$
- $2^{15} = 32768$
- $2^{16} = 65536$

Number Representation

How many bits do we need to store a number in a program?

It depends on:

- The length of a word (hardware)
- The programming language (software)
- You (the programmer)
- The problem

Older system architectures support 16-bit words (0 to 65535)

Less older ones support 32-bit words (0 to 4294967295)

Newer ones support 64-bit words (I'm not showing this range)

Number Representation

- Let's multiply $123456789 * 987654321$

- How many digits do we need?

- Excel says:

- $123456789 * 987654321 = 1.21933E+17$

- 121933000000000000

- C says:

- $123456789 * 987654321 = 4227814277$

- Java says:

- $123456789 * 987654321 = -67153019$

- If we use a `long` (64 bits) in Java:

- $123456789 * 987654321 = 121932631112635269$

Integer overflow!

If the number of output bits exceeds the maximum number of bits stored in the variable, the leftmost bits are removed.

But why does this turn negative?

Number Representation

- Let's multiply $123456789 * 987654321$

- Python says:

```
>>> 123456789*987654321
121932631112635269
>>>
```

- Is this correct?

- Number of digits: 18
- Last digit: 9
- Multiply it yourself by hand?
- Check by division

```
>>> 121932631112635269 / 123456789
987654321.0
>>> |
```

Number Representation

- Even bigger numbers

Python has **unlimited size** for integers

- 123456789^3 ?

- Around 25-27 digits

- Java says (even with long): **-2204193661661244627**

- Python says:

```
>>> 123456789**3
1881676371789154860897069
```

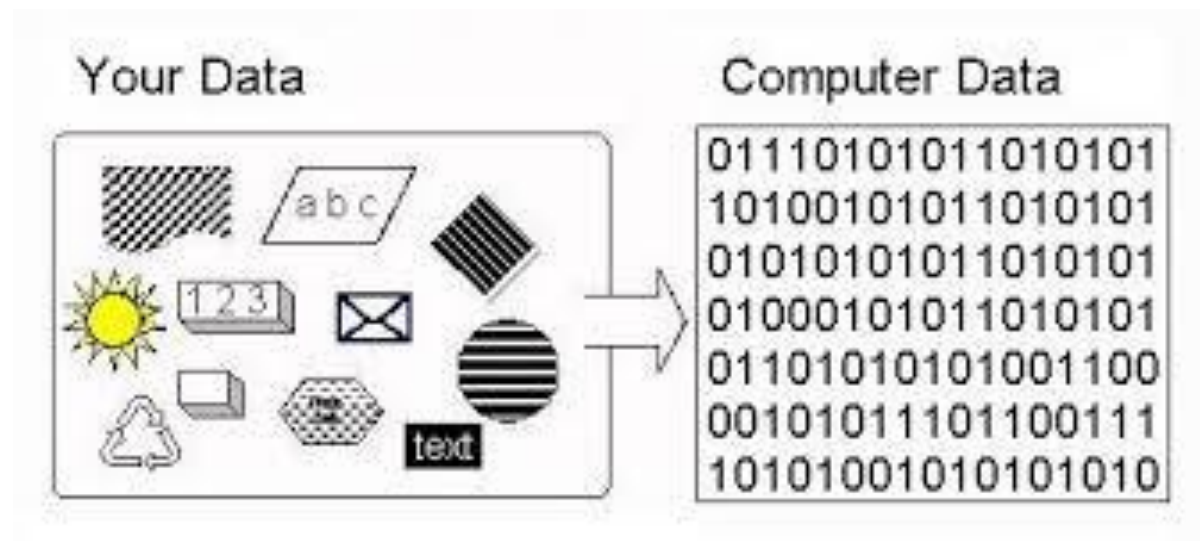
- **Even bigger:** 123456789^{100} ?

```
>>> 123456789**100
14174172601035587702142524239761426685023098432892168330190482375947577082389861
82489372231899746980921982728329402793285767462862882464121635860400730716254039
94235108484654701851813111412522017073436551977468182566355508096008844818770030
66625910338354975470658498293933938513368505135167166549545948424070710591295654
67264692831108320130483612672587797723547589358874240495338978427064891700798459
02824088981773992924362039029250020367962086497153391426082783460157920931418912
06269019044584869367276229055823673888183254671596267470545995689537867035621227
99941680845141148189896305104641344839457223830507901762718528867396981775965176
55470069835676583069071363091251912629058336230389234503573930897227480759410337
69593485936785871479329670603921014307898298170610105986211966740731734671893744
3597566001
```



Information Representation

- We know how (positive) integers can be stored in binary.
- What about:
 - Negative numbers
 - Real numbers
 - Complex numbers
 - Text
 - Images
 - Audio
 - Video



Information Representation – Signed vs Unsigned nums

- Numbers stored in memory can be **signed** or **unsigned**
 - Unsigned numbers can only be zero or positive
 - E.g. 8-bit unsigned integers range from 0 to 255
 - Signed numbers can be positive or negative or zero
 - E.g. 8-bit signed integers range from -127 to 127 **or** -128 to 127

Information Representation – Negative Numbers

In negative numbers, the first bit represents the **sign** (+/-)

- **Sign-and-magnitude** representation:

3 = **0**0000011

-3 = **1**0000011

- leading sign bit 0 means positive, leading sign bit 1 means negative.

- **Two's complement** representation:

- -N is represented as the binary of $(2^n - N)_{10}$ where n is number of bits

- Or, invert bits then add 1.

52 = **0**0110100 3 = **0**0000011

-52 = **1**1001100 -3 = **1**1111101

- **One's complement** representation:

- Simply invert the bits 3 = **0**0000011

-3 = **1**1111100

Information Representation – Negative Numbers

- **Sign-and-magnitude** representation: leading sign bit 0 means positive, leading sign bit 1 means negative.
- Let's consider 4-bit signed integers.
- $2_{10} = 0010_2$. What is -2_{10} ? 1010_2
- $7_{10} = 0111_2$. What is -7_{10} ? 1111_2
- $0_{10} = 0000_2$. What is -0_{10} ? 1000_2
- $7_{10} + 1_{10} = 0111_2 + 0001_2 = 1000_2$
- Is 1000_2 equal to 8_{10} or -0_{10} ?
- What happens when we try to make -8_{10} ?

Two representations of 0!

Range of 4-bit signed integers using sign-and-magnitude is -7 to 7

Information Representation – Negative Numbers

- **One's complement** representation: simply invert the bits.
- Let's consider 4-bit signed integers.
- $1_{10} = 0001_2$. What is -1_{10} ? 1110_2
- $7_{10} = 0111_2$. What is -7_{10} ? 1000_2
- $0_{10} = 0000_2$. What is -0_{10} ? 1111_2
- $7_{10} + 1_{10} = 0111_2 + 0001_2 = 1000_2$
- Is 1000_2 equal to 8_{10} or -7_{10} ?
- What happens when we try to make -8_{10} ?

Again, two representations of 0!

Range of 4-bit signed integers using one's complement is -7 to 7

Information Representation – Negative Numbers

- **Two's complement** representation:
 - $-N$ is represented as the binary of $(2^n - N)_{10}$ where n is number of bits
 - Or, invert bits then add 1.
- Let's consider 4-bit signed integers.
- $3_{10} = 0011_2$. What is -3_{10} ? 1101_2
- $7_{10} = 0111_2$. What is -7_{10} ? 1001_2
- $0_{10} = 0000_2$. What is -0_{10} ? 0000_2 Only one representation of 0!
- $7_{10} + 1_{10} = 0111_2 + 0001_2 = 1000_2$ Range of 4-bit signed integers using two's complement is -8 to 7
- Is 1000_2 equal to 8_{10} or some other number?
- What happens when we try to make -8_{10} ?

Information Representation – Negative Numbers

- **Two's complement** representation:
 - $-N$ is represented as $2^n - N$ where n is number of bits
 - Or, invert bits then add 1.
- Two's complement is the most common method

- **Why?**

- Only one representation of zero (00000000)
 - For sign-and-magnitude, we have (10000000) and (00000000)
 - For one's complement, we have (11111111) and (00000000)

8-bit Two's complement ranges from -128 to 127

- **Subtraction by addition**

More on this in a future lecture (approx. Wk 5)

$$\begin{aligned}x - y \\&= x - y + 2^n && (n \text{ bit encoding}) \\&= x + (2^n - y) \\&= x + (-y)\end{aligned}$$

8-bit One's complement and sign-and-magnitude ranges from -127 to 127

Information Representation – Other numbers

- **Decimals** can be represented in **fixed point representation**
 - “Decimal point” separates integer part and fractional part
 - Each decimal place is a negative power of 2.
 - $101.11_2 = 5 + 1/2 + 1/4 = 5.75_{10}$
 - $0.10101_2 = 1/2 + 1/8 + 1/32 = 0.65625_{10}$
- **Real numbers** can be represented in **floating point representation**
 - 1.0111×2^2 <https://www.h-schmidt.net/FloatConverter/IEEE754.html>
 - Exponent is represented in binary (and can be negative)
- **Complex numbers** are not supported in most languages but are supported in python

Information Representation - Text

Text is a sequence of characters

- Each character is typically one byte
 - 0 to 255 (256 characters)
- Characters include letters, digits, punctuation symbols and more
- Most common standard is ASCII (American Standard Code for Information Interchange).
 - 128 characters using 7 bits

ASCII Table

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Information Representation – Text Example

- Hey, Jude!
- Each character is represented by a byte.
 - Decimal: 0 to 255, Hexadecimal: 0 to FF
- H = 72, e = 101, y = 121, “,” = 44, “ “ = 32, J = 74, u = 117, d = 100, e = 101, “!” = 33
- In decimal
 - <72 101 121 44 32 74 117 100 101 33>
- In hexadecimal
 - <48 65 51 2C 20 4A 75 64 65 21>

Information Representation

- Multimedia data such as **images**, **audio**, and **videos** are represented via standard encoding **methods**.
 - E.g. GIF, PNG, JPEG, MP3, MP4
 - ASCII is a simple encoding method
- ASCII using 7 bits and encodes the Latin alphabet
- Extended ASCII uses 8 bits
- Other languages?
 - Chinese?
 - Japanese?
 - Korean?
 - CJK Character set

Information Representation – Character Encoding

- There are standards to encode characters in different languages
 - [ISO/IEC 10646](#) (International Organization for Standardization)
 - [Unicode](#) (4 bytes): 143859 characters
- Python uses [UTF-8 encoding](#) (Unicode Transformation Format)
 - 8 to 32 bits per character (First 128 ASCII characters use a single byte)
 - 16 bits suffice for all CJK characters

Information Representation – `chr` function

- In Python, the `chr(n)` function outputs a character represented by the number n

```
>>> chr(72)+chr(101)+chr(108)+chr(108)+chr(111)
'Hello'
```

- $111010000000110_2 = 29702_{10}$ in Unicode?

```
>>> chr(29702)  >>> chr(24037)
'理'            '工'
```

理工大學

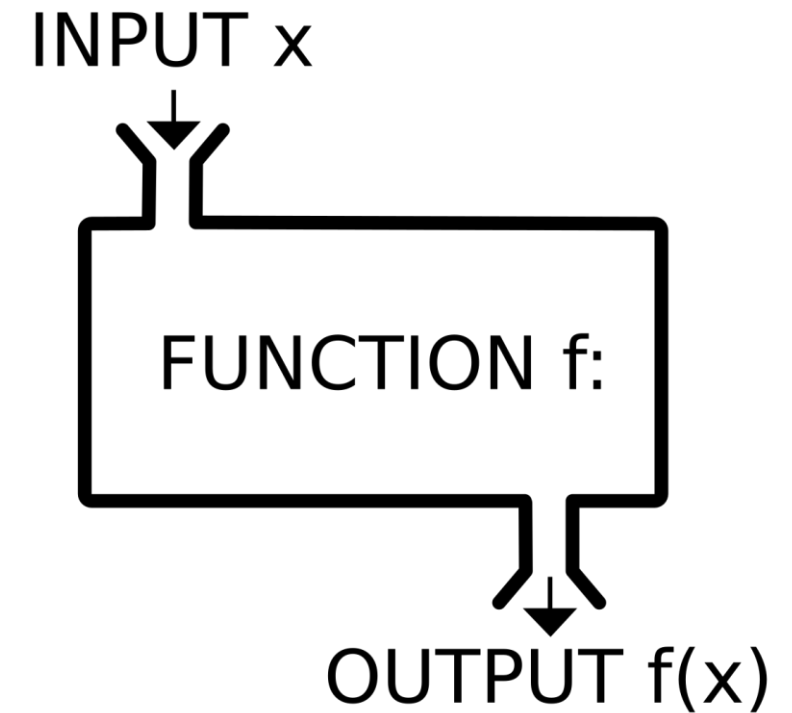
- CJK characters run from 19968 to 40956
- `chr(49340) + chr(49457)`
 - Korean Hangul characters from 44032 to 55171

```
chr(49340) + chr(49457)
```

```
'삼성'
```

Recap: Mathematical Functions

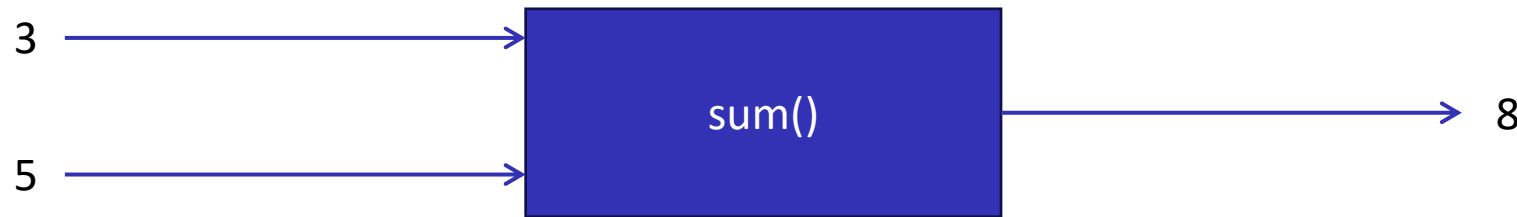
- In mathematics, a **function** from a domain X to a range Y maps **each** element of X to **one** element of Y .
 - E.g. $f: \mathbb{R} \rightarrow \mathbb{R}, f(x) = 2x^3 + 5x - 8$
 - E.g. $g: \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}, g(x, y) = 2xy - x$



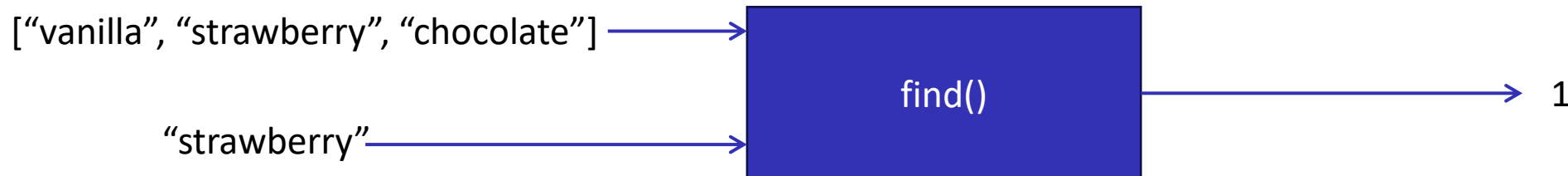
Programming Functions

- In programming, a **function** performs a **certain task** with some inputs, and provides the result as an output.

- E.g. `answer = sum(x, y)`

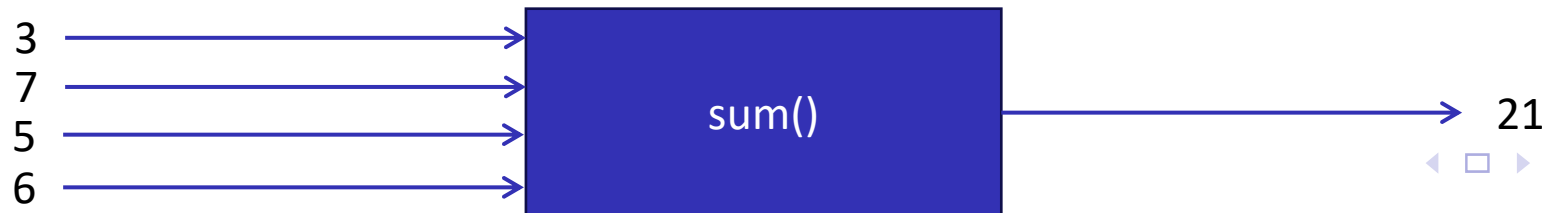


- E.g. `position = find(List, item)`



Programming Functions

- Function **inputs** and **outputs** are clearly specified
 - E.g. `sum(x, y)`
 - Input: Integers x and y
 - Output: sum of x and y
- For many functions, we don't need to know how it is implemented, we just use it.
 - **Black box**
- Can have more inputs, typically only one output

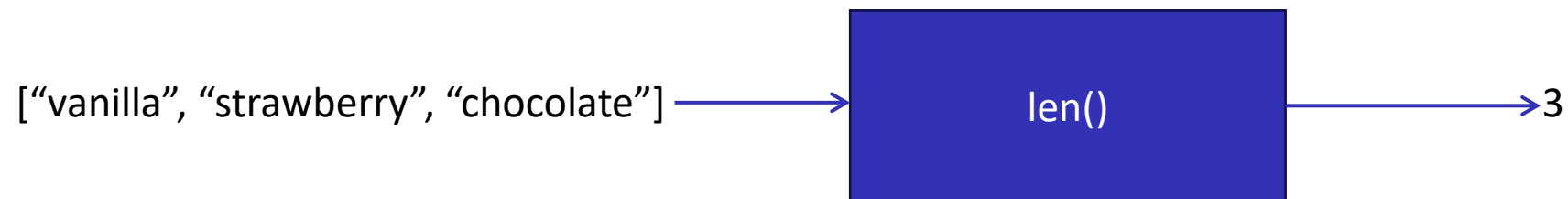


Programming Functions

- E.g. `sortedList = sort(List)`



- E.g. `length = len(List)`



Programming Functions

- In mathematics, each function represents a calculation that would lead to a value.
- In programming, each function **returns a value**.
- Technically speaking, a programming function can do more than a mathematical function.
 - The output can vary depending on external factors:
 - System time, network conditions
 - Programming functions can have side effects:
 - Writing to a file, modifying external variables

Programming Functions

- Can we have a function that doesn't return anything?
- In many languages, a function that does not return a value is called a **procedure** or other name.
 - Called a **subroutine** in Basic and Fortran.
 - In C, it is still called a **function**, returning a data type **void**.
 - In Python, it is still called a **function**, returning nothing (**None**).
 - In AppInventor, all functions are called **procedures** instead.
 - In Java, all functions and procedures are called **methods**.
- Can a function return more than one value?
 - In most languages, that is not allowed: some tricks are needed.
 - In Python, **multiple values** could be returned.

Procedures

- Programming functions can return nothing.
 - A function that returns nothing is often called a **procedure**.
- Besides output, a program is designed to perform computation, as reflected by changes **in memory content**, or external storage like **file / database**.
 - The collection of memory content and the current position of program execution is called the **program** (or system) **state**.
 - A procedure will lead to a **change in the state**.
 - Example: a procedure may **sort** a list of numbers stored in the memory, without giving any output. The list printed out before and after executing this procedure will be different.

Programming Functions

- E.g. `sort(List)`

`["vanilla", "strawberry", "chocolate"]`



`sort()`

Example Python Functions

- Mathematical functions

```
>>> sum([3,5])  
8
```
- Text manipulation functions
 - `join("Good", "Morning")` -> "GoodMorning"
- Drawing functions
 - `drawLine((0,0), (1,1))` -> draws a line from coordinate (0,0) to (1,1)
- Input/Output functions

```
print("COMP1010")  
COMP1010
```
- Object creation functions
 - `range(2,7)` -> [2,3,4,5,6]

`main()` function in Python

- `main()`
 - Default function in a Python program.
 - Contains the body of the program.
 - Normally included at the bottom of a Python program (under other functions).
 - A call to `main()` is used to start a program.

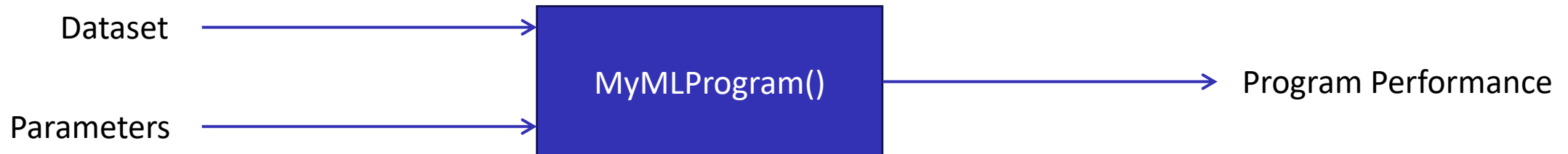
```
def main():  
    celsius = eval(input("What is the Celsius temperature? "))  
    fahrenheit = 9/5 * celsius + 32  
    print("The temperature is", fahrenheit, "degrees Fahrenheit.")
```

```
main()
```

Programs can be functions too

True story of my first real job: Data Science Intern (ML)

- **Job Part 1:** Deconstruct and update existing program to handle wider ranges of data.
- **Job Part 2:** Run hours and hours of experiments on various datasets to find optimal program parameters and measure program performance.
 - Parameters are values that affect how different parts of the program work.



Programs can be functions too

Job Part 2: Run hours and hours of experiments on various datasets to find optimal program parameters and measure program performance.

- I wrote a **program** that treats MyMLProgram as a function, and runs it multiple times with different Datasets and Parameter combinations.
 - All program performance details saved to my computer
- Average workday:
 - 10 am: Get to work, turn on **program**
 - 10 am – noon: Listen to podcasts, self-learn pure mathematics while **program** runs
 - Noon – 2 pm: **Program** finished its round, run it again. 2 hr lunch break
 - 2 pm – 4 pm: **Program** finished its round, run it again. Write up report on today's Program Performance. Wait for boss to step outside then go home.

