Data representation in computers

Bryan Ruddy

9 May 2022

Now that you have become familiar with common computational techniques, it is helpful to go back and think about what it is we actually work with in computing. Everything we do with computers is based on data: collecting it, processing it, and/or using it to make some change in the world. But what, exactly, is data?

Data represents some piece of information. From a physicist's perspective, information is a deviation from perfect randomness in the universe. From a human perspective, it can be a set of numbers, a recorded snippet of human language, a functional procedure, or even a representation of abstract art. In computing, these all have a common storage method and medium: binary numbers.

The physicists' definition can be used for interesting calculations of the theoretical limits of computing, in terms of energy and power, but has little practical relevance.

The objective of this module is to provide a refresher on binary numbers, and some discussion of how they are used in practice to represent numerical and non-numerical data on computers. By the end, you should be familiar with the different formats used for representation of numerical and text data, and should be able to explain the major properties of each format. You should also have a general understanding of how other types of data are represented, and of the key considerations in choosing a data representation.

A brief review of number representations

In English and many other languages, we most often think about numbers in terms of a *decimal* system: a sequence of digits that can take one of ten values, from 0 to 9. We can think of the decimal representation, mathematically, as a collection of n individual digits d of *base* $b = 10$, where the overall number represented is computed as a sum of digit values times powers of the base b as follows:

$$d_{n-1}d_{n-2}\dots d_1 d_0 \equiv \sum_{0}^{n-1} d_k b^k = d_{n-1}b^{n-1} + \dots + d_1 b^1 + d_0 b^0 \qquad (1)$$

While the use of base 10 is convenient for humans, who mostly have 10 fingers (including thumbs), it is a challenging representation for electrical and mechanical devices. A small amount of noise or misalignment could cause an error, and great precision would be required to represent the digit values. Instead, what is the simplest possible representation?

Other cultures use different number systems: many cultures use base 20, the ancient Sumerians used base 60, and some societies in Papua New Guinea use base 23!

Most reliable is the use of just two values, 0 and 1, which can be represented by simply the presence or absence of something. In an electronic computer, this signal is a voltage - the presence of voltage means 1, while the absence means 0. This *binary* representation is a bit lengthier than decimal, but admits many simple calculations. For instance, consider the number 42, represented in binary as 101010_b. (To distinguish between binary numbers and those in

other bases, we will append a "b" to all binary numbers.) To divide it in half, just shift all the bits one place right: $10101b$. To multiply it by two, just shift them one place left: $1010100b$. Operations involving powers of 2 are very simple to perform.

In this course, we will not ask you to manually do arithmetic on binary numbers, but you should practice converting between binary and decimal notations. For instance, to convert $N = 42$ to binary, first find the largest power of 2 that is smaller than it: 32, or 2^5. Place a 1 as digit d_5, and subtract 32 from N to give the intermediate result 10. Repeat the process, finding the powers $2^3 = 8$ and $2^1 = 2$. Using 0 for all the other digit positions, we find

The answer to life, the universe, and everything!

$$42 \equiv d_5 d_4 d_3 d_2 d_1 d_0 = 101010b. \tag{2}$$

This process can also be done by dividing by two and taking remainders; in this case you build the binary number from right to left, using the remainder of each division as the digit.

Similarly, to convert back, add up the appropriate powers of 2:

$$101010b \equiv 1 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = 42. \tag{3}$$

There is one further representation with which you should become familiar, because it can serve as a more compact way to write binary numbers. This is the base-16, or *hexadecimal* representation, which is written with the digits 0–9 and the letters A–F. By convention, when writing a hexadecimal (or hex, for short) number it is preceded by "0x", so that for instance 0x10 is immediately recognizable as hex, with a decimal equivalent of 16, rather than decimal 10. Each hexadecimal digit represents exactly four binary digits, like so:

$0x0 = 0000b = 0$	$0x8 = 1000b = 8$
$0x1 = 0001b = 1$	$0x9 = 1001b = 9$
$0x2 = 0010b = 2$	$0xA = 1010b = 10$
$0x3 = 0011b = 3$	$0xB = 1011b = 11$
$0x4 = 0100b = 4$	$0xC = 1100b = 12$
$0x5 = 0101b = 5$	$0xD = 1101b = 13$
$0x6 = 0110b = 6$	$0xE = 1110b = 14$
$0x7 = 0111b = 7$	$0xF = 1111b = 15$

Get skilled in reading hexadecimal, and you too will be able to see through the Matrix...[1]

Decimal-hexadecimal conversion is done in the same way as decimal-binary conversion, but using powers of 16 rather than powers of 2. For instance, we can find that

$$42 = 2 \cdot 16^1 + 10 \cdot 16^0 \equiv 0x2A \tag{4}$$

One way to practice this is to use the Windows calculator, with has a *programmer* mode that allows you to examine numbers in all of the common bases used by computers. Try converting your favourite number, and use the calculator to check your results!

There are also a number of "fun" hex representations: try finding the hex representations of 48,879 or 65,261, for instance.

Storage of numerical data

It is all well and good to have binary positive integers, but how do we represent other kinds of numbers? Also, how do we standardize

the amount of storage needed for a number? We will begin with the second question.

Bits, bytes, and ends

For convenience in handling the mathematics, computers usually work with binary numbers in multiples of 8 digits, which are called *bits* in computing parlance. One group of 8 bits (a.k.a. an 8-bit number) is called a *byte*. By using a number of bits equal to a power of two, it becomes easy to represent operations performed on numbers themselves as binary numbers. As modern computers have become more powerful, it has become common to handle several bytes together. A typical modern computer has a 64-bit processor, meaning it works with 8-byte numbers; smaller and cheaper processors might work with 32-bit or 16-bit numbers instead. These 4 sizes are most common: 8-bit (1 byte), 16-bit (2 byte), 32-bit (4 byte), and 64-bit (8 byte). Larger sizes than this are only used in specialized applications like cryptography, where very large numbers must be manipulated precisely.

We often group digits outside of computing - we write with thousands separators in many languages, and when you recall a long number from your brain, like a phone number or credit card number, you probably do so with groups of digits rather than reciting the whole number without pause.

This notation may be familiar to those who followed the history of video games, where console generations were defined by the bit size of their processors.

With the move to processors handling larger numbers has come a problem: how is the representation constructed? There are two different ways to build a larger number from multiple bytes: either the first byte can be taken as the part describing the left-most digits in the number, or it can be taken as the right-most digits. Most personal computers use processors that treat the first byte as the right-most digits, called *little-endian*, but the processors in most mobile phones treat the first byte as the left-most digits, called *big-endian*. Thus, the number 48,813, which has the hex representation 0xBEAD, could be represented internally as the byte sequence 0xBE 0xAD on some computers, but 0xAD 0xBE on others.

8-bit adventure![2]

There is no practical difference between these approaches when working on a single computer; your programming language hides this detail, and you only need to think about the overall number. However, when storing information for use on different devices, you need to keep track of how the data are represented, so that the bytes don't get mixed up!

Integers

For instance, many file types start with a sequence of two different bytes, so that the value when interpreted as a 16-bit number can be used to determine whether big-endian or little-endian storage was used.

Binary representation works very naturally for integer numbers. For a positive integer, we simply use the mathematical representation directly. However, some complications arise when we introduce fixed-size binary integers. When manipulating an integer of a fixed bit size, what happens if you perform a mathematical operation on it that needs more space for its representation?

This situation is called an *overflow*, and when one happens the value wraps around from the largest value to the smallest one. For instance, the largest value an 8-bit integer can have is $2^8 - 1$, or 255. If you add one to this, the result is zero, not 256! Likewise, subtracting one from zero yields 255, not a negative number. It is

usually possible to find out if the calculation overflowed, so that your code can handle the situation as appropriate—you might need to change to a larger representation, or alter your calculations elsewhere to account for the overflow.

The standard mathematical representation breaks down if we want to represent negative integers, however, because it contains no way to represent the sign. An integer can be either positive or negative, and so we need one extra bit of information to represent the sign. There are several ways to do this, but the most common method is called *two's complement*: the value of the left-most bit is negated, so that the value of an n-bit signed integer is computed from the binary representation as follows:

One can also represent the sign by explicitly using a sign bit, called a *sign-magnitude* representation, or by subtracting an *offset* value - we will see these methods used for non-integer numbers shortly.

$$d_{n-1}d_{n-2}\ldots d_1 d_0 \equiv -d_{n-1}2^{n-1} + \sum_{0}^{n-2} d_k 2^k \tag{5}$$

$$\equiv -d_{n-1}2^{n-1} + d_{n-2}2^{n-2}\ldots + d_1 b^1 + d_0 b^0 \tag{6}$$

For instance, the binary number $10000001b$ represents a signed integer as follows:

$$10000001b \equiv -2^7 + 2^0 = -128 + 1 = -127 \tag{7}$$

If you need to change the sign of a two's-complement binary number, there is a quick shortcut: flip all the individial bit values, then add one to the result.

This representation has some important consequences. The smallest value an n-bit signed integer can have is -2^{n-1}, if the left-most bit is one and the rest are all zero. The largest value, however, is only $2^{n-1} - 1$, represented by a zero in the leftmost bit and all remaining bits set to one. The representation is asymmetric; negating the smallest allowable value causes an overflow, rather than giving the highest allowable value. Also, binary and hex representations of negative numbers cannot readily be distinguished from unsigned representations of larger positive numbers—you must know independently whether the number is signed or unsigned. For instance, the hex value 0xBEAD could represent -16,273, if it is a signed 16-bit integer, or it could represent the unsigned value 48,813. This kind of ambiguity will become increasingly apparent when representing more complex data.

What happens if you want to represent a very large integer? Python takes care of this automatically, by representing it as a list of digits in base 2^{32}. You don't need to think about how this works, though - Python will handle it, and you can rest assured that your Python integers will simply never overflow.

Fixed-point numbers

Just as is done with decimal data in everyday life, one common way to represent non-integer real numbers in computers is by using a decimal point to indicate the fractional part of a number; in computing this is called *fixed-point*. Digits to the right of the

Imaginary numbers are represented with a pair of real numbers, though the details vary with the programming language.

decimal point represent negative powers of two, i.e. 1/2, 1/4, 1/8, etc. Alternatively, you can think of a fixed-point number as being an integer divided by two taken to the power of the number of digits to the right of the decimal point.

The main difference here between computing and everyday life is that fixed-point representations don't actually have a decimal point; the location of the decimal point must be tracked by the programmer and/or programming language, in much the same way as whether a number is signed or unsigned must be tracked. For instance, a 4.4 fixed point representation is an 8-bit unsigned integer divided by 2^4; the value 5.625 would be represented as $01011010b$.

Can you see why? Try the calculation for yourself.

A number with a short decimal representation may not be representable using a finite-size fixed-point binary number. For instance, just as the fraction 1/3 is represented by an infinite series of decimal digits, the decimal number 0.1 requires an infinite number of binary digits for its representation.

Operations involving fixed-point numbers with different decimal positions can be complex to think about, because overflow becomes more likely. *Underflow* is also possible, where information is lost because the value is too small to be represented. For example, going back to the 4.4 representation we used earlier, what is $00000001b \times 00000001b$? The exact result in decimal would be 1/256, but the smallest value we can represent is 1/16; thus, the fixed-point result must be zero. If this effect is not carefully considered when writing fixed-point code, the information loss can have serious consequences.

Fixed point numbers are most commonly encountered in electromechanical control systems, where calculations must be done very quickly and underflow errors don't cause problems. Special programming languages or software libraries can be used to help keep track of the decimal points and manage the limitations of the computing hardware used in these control systems.

Floating-point numbers

The most common form of representation for non-integer real numbers in computing is the binary version of scientific notation, called *floating-point*. This name is used because these numbers are represented using a decimal point that can be moved relative to zero using an exponent. We can formally represent this as follows, for a number x:

$$x = \underbrace{\pm}_{\text{sign}} \underbrace{d_1.d_2d_3\cdots d_n}_{\text{significand}} \times b^{\underbrace{\pm e_1e_2\cdots e_m}_{\text{exponent}}} \tag{8}$$

Here, there are n digits d in the significand, m digits e in the exponent, and a base b. The degree of precision available is governed by n, while the maximum range of numbers that can be represented is governed by m. The sign of the significand is marked separately, due to the way it is represented in practice. For instance, consider a

simple representation of the number 7680, in both decimal scientific notation and in floating point (with an unsigned significand and exponent):

$$7680 \longleftrightarrow 1111000000000b \longleftrightarrow 7.68 \times 10^3 \longleftrightarrow 1.111b \times 2^{1100b} \quad (9)$$

Floating-point allows for compact representations of large (and small) numbers, but at the cost of limited precision. If the significand is kept to four bits, as per above, then the number 7681 has an identical representation to 7680; the numbers are indistinguishable. You have already experienced the practical consequences of the limited precision of floating-point numbers in your study of numerical error.

As with any other representation, we can only successfully interpret a floating-point number if we know the rules that have been used to construct the representation. There is an international standard, IEEE 754, that describes what is now the most common way to construct a floating-point number. There are two common floating-point representations in the standard: *single-precision*, which uses 32 bits, and *double-precision*, which uses 64 bits to construct the representation. IEEE 754 uses a separate *sign bit* to represent the sign of the significand, but uses an offset with the exponent to achieve positive and negative values. To find the actual exponent, subtract $2^{m-1} - 1$ from the value stored. It also always assumes that the first (most-significant) digit of the significand is 1, and therefore only stores the fractional part (i.e. digits d_2 to d_n).

Confusingly, floating-point representations abandon the two's-complement approach used to representing negative integers.

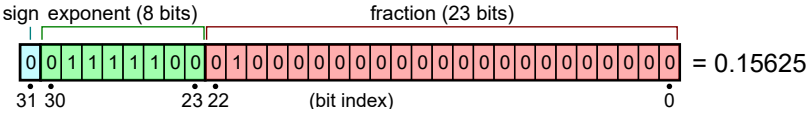


Figure 1: The IEEE 754 single-precision floating-point representation.[3]

As shown above, a single-precision floating-point number uses one sign bit, 8 bits for the exponent, and 24 bits for the significand (of which 23 bits are actually stored). A double-precision number instead uses an 11-bit exponent and a 53-bit significand.

A final property of the IEEE 754 standard is that there are some special values for the exponent that change the way the number is interpreted. If the exponent is all zeros, then the first bit of the significand is assumed to be zero, instead of 1, but the exponent should be interpreted as $2 - 2^{m-1}$. If the significand is also zero, then the number is zero. If the exponent is instead all ones, but the significand is zero, the number is infinity, positive or negative depending on the sign bit. If the significand is instead nonzero, the number is treated as invalid, and propagated through calculations as not-a-number (NaN).

The sign bit still matters - positive zero and negative zero are different! This can be useful when accounting for rounding error.

You might see this if you try to calculate 0/0, for instance.

Most computers have special hardware for working with floating-point numbers, so you don't have to worry about the details of the representation when writing code. However, from the complexity of the representation, you can see that it would be much more complicated to perform calculations with floating-point numbers

than with integers. This is reflected by slower calculations with floating-point numbers than with integers.

Storage of non-numerical data

The various binary representations of numbers have a certain logic to them, as they are numbers representing numbers. How can we represent data that is not a number?

Text data

For textual data, there are several established standards. One common standard is called ASCII, which stands for American Standard Code for Information Interchange, and was developed in the 1960s. At this time, the communication protocols used with computers and telegraph systems worked best with 7-bit representations, and so there are 128 different characters represented in ASCII, as shown in the table below. For instance, the character "A" is represented in ASCII as hex 0x41, or decimal 65. The first 32 ASCII characters are *control characters* intended to control printers, rather than represent printed characters; for example, 0x0A causes a printer to advance one line of text, while 0x07 ("BEL") causes the device reading it to emit a beep or ring a bell.

Some of these control characters still work! You can enter the BEL character at a command prompt by pressing `Ctrl-G` on the keyboard.

ASCII Code Chart

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Figure 2: All the characters fit to represent in 7 bits. The rows correspond to the first hex digit, while the columns correspond to the second hex digit.[4]

ASCII allows for convenient representations of text strings with one byte per character, but there are several limitations. The main one is that it is an "American" standard—it was intended to represent only the English language, using the most common punctuation. Also, its 1960s heritage means that the control characters are mostly long-obsolete, and now waste space in the representation. A variety of text encodings were historically developed to work with or replace ASCII for other languages, but the modern method is a standard called Unicode.

Unicode aims to represent all known forms of text in a single standard, and as such has room for over a million different *code points*, most of which correspond to characters. While this allows it to represent languages from Armenian to Zanabazar Square, the way many of you will be most familiar with it is its inclusion of emoji.

Working with emoji in code can sometimes feel like this...[5]

Of course, it is impossible to represent a million different characters using only one byte. Instead, there are several methods in use to represent Unicode with multiple bytes per character. The first is to simply use 4-byte characters, which is easy to do but very wasteful of space. The other two approaches in common use are *variable-width* encodings—a single character may use extra bytes if necessary. Most commonly used is an encoding called UTF-8, which allows ASCII characters to still be represented in one byte, but uses up to 4 bytes for other Unicode characters, such as emoji or Chinese text. Internally, Microsoft Windows uses a different encoding, UTF-16, which represents most characters in 2 bytes but requires 4 bytes for others (such as emoji or Chinese text). In Python, a different encoding is used: every character is the same size as the one requiring the most storage. Thus, if a text string contains a single pile-of-poo emoji, every character in the string will then occupy 4 bytes.

Some emoji require multiple Unicode characters to represent them, such as ones including gender and skin colour, so even knowing the number of Unicode characters won't tell you the number of characters displayed on the screen.

Beyond the character representation itself, there are different ways to store *metadata* about a string of text, such as the length. The traditional approach, used in the C language and others, is to place a special charcater (ASCII 0x00, or NUL) at the end of the string; this is called a *null-terminated string*. Any code processing the text string starts at the beginning and continues until a NUL character is found. This approach is simple, but can lead to dangerous errors if the NUL character is missing. It also requires programmers to remember that every string has to be stored with enough extra space for the NUL character. Instead, some programming languages, like Python, store strings alongside their length and other metadata explicitly, simplifying their use and security at the expense of more storage space.

Pointers

As you learned in ENGGEN 131, sometimes a variable doesn't actually contain data, but instead just points to a different memory location that has the desired data. In C these are called *pointers*, and they are the same size as the processor bit-size; i.e. on a modern desktop PC with a 64-bit processor the pointers are also 64-bit. You will usually not want to store the memory addresses in pointer variables for later, because the correct place in memory for them to point to may be different every time the program is run, and may depend on many other factors. However, on a tiny processor embedded in a sensor or a similar device, the memory address of interest might be constant, and therefore useful to store. This is typically done as an unsigned integer of the same size as the processor bit-size.

Other data

What about more complex data? If your intent is only to store data for your own use, you can invent whatever format you like!

You already have experience with complex data representations in Python, including objects, lists, and NumPy arrays. Depending on the needs of your own code, you can use these to construct even more complex representations, like lists of lists or objects with code and multiple different pieces of labelled data. At a low level, a *struct* in the C language allows you to split up the bits of a binary number however you like between several other quantities. For instance, a single 32-bit value could represent 5 integers, each with 6 bits, or 32 one-bit true/false (Boolean) values. However, you will have to carefully document your data for your own future reference and to share with others who may wish to work with your code. Without the documentation, no matter what was intended for those 32 bits it just looks like one integer. Even if you use a higher-level representation as is common in Python, you still need to keep careful documentation in order to interpret complex data structures successfully.

Python has several ways to store these data structures for later; the simplest approach involves the *pickle* package and associated tools from the standard library.

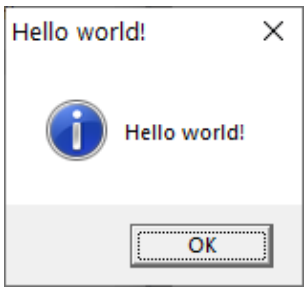
We regularly work with very complex data structures, but deep down these are all just sequences of binary digits that are nothing but numbers without context. For example, consider the decimal integer

29127159951297396070515571495922530238 0 . . .

369686167047087686032666294295649621 56.

This may look like just a long number, but Windows can interpret this as a script file that prints “Hello, world!” on the screen! Internally, this binary sequence represents a string of ASCII text, containing commands interpreted by the operating system. The rather longer number

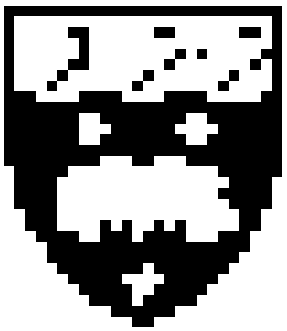
44551001405542418286834633109968756717906072802800642 . . .
55765732335646221215566764644512723286322194960098612 . . .
77404591214326552699455701558339438591673680647599414 . . .
03021220927267007286733922657653529220524553893107816 . . .
64670249031964572091725049048268631510168894064200443 . . .
47850921908474007149871263343719139181664357439733947 . . .
96732069048098643799874171137943602404190852794642209 . . .
47821047240372723920007296398983652689869009929576306 . . .
75877034744559780099242078360804791193553647801027212 . . .
17704717625872550481404034135568646263772131539910853 . . .
86922261520935291158868881808001637905680505119953044 . . .
30037799492389975344357619435098342641147400680645782 . . .
40521020916311421527641271595721432361740758791230514 . . .
41012278888023286086669451452038668461143294362344879 . . .
71202335089166179543324753253170654846329124176451644 . . .
26297711272192



can be interpreted as an actual executable program on Windows, which pops up a dialog box that says "Hello, world!". Internally, this binary sequence represents both a structured set of parameters for how the code should be executed and a series of machine code instructions to be run by the processor. Finally, the integer

The problem of identifying executable numbers like this one has been closely linked to protests against copy prevention mechanisms; have a look at `https://en.wikipedia.org/wiki/Illegal_prime` if you're curious.

95255723516189124388258642527343979802562670455251020 . . .
66880152315564094830666068406800899246027593934777589 . . .
09895320746420103140747714846408068734212653394829141 . . .
41958169351248792265286404473233289882986413917024604 . . .
22708888220014769182523319629082749871574228579629168 . . .
89311484857994069111353491103970444699903425673181414 . . .
36979721630794495196114225820345988015063344602409031 . . .
97075998555205869627270380492324652918606640703577080 . . .
708573749086618277739702874275824



A very small bitmap file!

can be interpreted as a (low-quality) bitmap image of the University of Auckland seal, as shown.

When storing other kinds of data for exchange with others, it is best to use a standardized format. Typically, there will be a public document describing the format. You most often can use code written by others to follow the required format. For instance, you have already used Python code from the standard library to handle importing data from comma-separated value files (the .dat files from Lab 3). In some cases, you may need to refer to the specifications to write your own code to handle a format—someday you may even be involved in writing the specs for a new data format!

The bitmap image format is described by Microsoft, its creator, at `https://docs.microsoft.com/en-us/windows/desktop/gdi/bitmap-storage`.

One common feature of these standard formats is a *magic number*, a special value located near the start of the data that is always present for that data type. These magic numbers can also be useful in determining whether there has been a change in endianness between the computer that created the file and the one that is reading it. For example, bitmap files like the one above always start with the ASCII text sequence `BM`, short for bitmap, while Windows executable files start with the ASCII sequence `MZ`, after the initials of the format's author. Older Microsoft Office documents all start with an eight-byte magic number, `0xD0CF11E0A1B11AE1`, which allows endianness to be detected but also has a hex representation that looks a bit like "DOCFILE".

I recommend the free program XVI32 to look at the raw hex data contained in files under Windows; similar tools are available for MacOS and Linux.

Figure sources:

[1]By Jamie Zawinski, from `https://commons.wikimedia.org/wiki/File:Xmatrix.png` (MIT license)

[2]From `https://en.wikipedia.org/wiki/Super_Mario_Bros.`

[3]By user Stannered, from `https://en.wikipedia.org/wiki/File:Float_example.svg` (CC BY-SA 3.0)

[4]From `https://commons.wikimedia.org/wiki/File:ASCII_Code_Chart.svg` (Public domain)

[5]By Google, from `https://commons.wikimedia.org/wiki/File:Emoji_u1f4a9.svg` (Apache license)