Chapter 2

Lab 2: Binary Numbers

2.1 Introduction

At the heart of numerical analysis, naturally, you will find numbers. In this lab, we will explore the basic data types in Python, with particular emphasis on the computer representation of integers and real numbers. All modern programming languages do an admirable job of hiding the limitations of the computer representations of these mathematical concepts. In this chapter, we will deliberately explore their limitations.

2.2 Preparation

This lab will rely on the material from Section 1.2.2 of the Scientific Python Lecture notes.

△ **Jupyter Notebook Exercise 2.1:** Enter the following code into a cell and check the output:

```
a = 121
print(type(a))
print(a)
```

Next, in the same cell, add a line at the end setting a to a real value, `a = 1.34`, and print the type and value again. Check the output. Next, set a to Boolean value, `a = False`, and print the type and value yet again.

In many languages, such as C and C++, variables are strictly typed: you would have to decide at the start whether you want a to be of integer, float, or boolean type, and then you would not be able to change to a different type later. Python variables are references to objects, which means they only point to memory locations that contain objects with all of the data and functionality associated with that object. When you write $a = 121$ it is interpreted as "set variable a to point to a location in memory that contains a class of type integer with the value 121".

△ **Jupyter Notebook Exercise 2.2:** Enter the following code into a cell and check the output:

```
a = 12
b = a
b = 5
print(a)
print(b)
```

Why is the output "12, 5" instead of "5, 5"? When you write `b = a`, the variable b points to the same Integer class that a points to. So when you write "b = 5" why doesn't the value of a change as well? This code snippet shows that it does not. The reason is that Integers are *immutable* objects in python... their values cannot be changed. So when you write `b = 5` it is interpreted as "variable b points to a (new) Integer with value 5." The variable a continues to point to the Integer with value 12. The "is" operator used like this:

```
print(a is b)
```

tells you if a references the same object as b. Add to compiles of this line to your snippet in order to clarify the situation. One call should return True and the other False.

△ **Jupyter Notebook Exercise 2.3:** If I set a variable a to an integer value and I set b to the same integer value, do a and b refer to the same object, or to two different objects with the same value? Write a snippet of code (three lines) to find out.

In this lab, it will be convenient to know how to calculate the absolute value of a number, which you can do with the python `abs` function:

```
a = -1.5
b = abs(a)
print(b)
```

2.3 Binary Representation of Integers

Computer hardware is based on digital logic: the electrical voltage of a signal is either high or low, which correspond to a mathematical zero or one. A digital clock is used to ensure that signals are only sampled at particular times, when they are guaranteed not to be in transition from a zero to one or vice versa. The Arithmetic-Logic Unit (ALU) uses digital logic gates (such as AND or OR) to perform calculations. For example, it is possible to build an adder that uses only NAND gates.

Because digital signals have only two states (zero and one), the most natural way to represet numbers in a computer is using the base two, which we call binary. In the familiar base ten, we have ten digits (from 0 to 9) and the place value increases by a factor of 10 with each digit moving toward the left. In binary, we have only two digits (0 and 1) and the place value increase by factors of two. A single digit in binary is referred to as a "bit". You add columns quickly when counting in binary: zero(0), one(1), two(10), three(11), four(100), five(101), and so on. For efficient operations, computers often group eight bits together to form a byte. Digital values are therefore commonly represented in hexidecimal (base 16) where two digits of hexidecimal describes one byte. See Table 2.1, which you can produce for yourself in Python like this:

```
print("dec: hex: bin:")
for d in range(16):
    print("{0:<2d}   0x{0:01x}   0b{0:04b}".format(d))
```

It is conventional to preprend binary numbers with "0b" and hexadecimal with "0x" otherwise we wouldn't know whether a "10" represents ten, sixteen, or two! Notice that the largest number that can be written with n bits is $2^n - 1$.

The mathematical notion of an integer can be naturally implemented by computer hardware. Although integers are represented in binary in the hardware, modern compilers and languages generally print them to screen as decimal by default. One caveat is that computers do not have

Table 2.1: The numbers 0 to 15 in decimal, hexadecimal, and binary.

dec:	hex:	bin:	dec:	hex:	bin:
0	0x0	0b0000	8	0x8	0b1000
1	0x1	0b0001	9	0x9	0b1001
2	0x2	0b0010	10	0xa	0b1010
3	0x3	0b0011	11	0xb	0b1011
4	0x4	0b0100	12	0xc	0b1100
5	0x5	0b0101	13	0xd	0b1101
6	0x6	0b0110	14	0xe	0b1110
7	0x7	0b0111	15	0xf	0b1111

an unlimited number of bits. Many computer languages use 64-bit integers, which means that only the integer values from 0 to 18446744073709551615 can be represented:

```
x = 2**64-1
print(x)
```

For signed integers, one bit is used to indicate the sign (positive or negative) and so a 64-bit signed integer can represent integer values from -9223372036854775808 to 9223372036854775807. As long as an integer value is within the range covered by the integer type, the integer value can be perfectly represented.

Python uses arbitrary sized integers: it simply adds more bits as needed to represent any number. For an extremely large number, you will eventually reach practical limitations on the amount of memory and processing time available in the computer, which will limit how large of an integer can be calculated.

△ **Jupyter Notebook Exercise 2.4:** See for yourself just how huge integers can be in python by entering:

```
x = 2**8000
print(x)
```

and checking the output.

△ **Jupyter Notebook Exercise 2.5:** Print the integer 65322 in decimal, hexadecimal, and binary. Hint: just reuse the carefully formatted print statement from the example above.

△ **Jupyter Notebook Exercise 2.6:** Suppose you are tasked with rewriting the firmware for a distant satellite which has just lost one line from an eight-bit digital communications bus due to radiation damage. You now have only seven working bits! What is the maximum sized unsigned integer which you could write on this degraded seven-bit bus? What range of signed integers could you write?

△ **Jupyter Notebook Exercise 2.7:** This is a paper and pencil exercise. You can do it on paper and pencil and submit a scanned PDF, or you can just record you steps as comments in a jupyter notebook cell. Let's consider a four-bit signed integer, so zero is 0000 and one is 0001. Suppose the upper bit is reserved for sign, so 1XXX is a negative number. An obvious choice for representing

-1 would be 1001, but there is a better choice. Consider that:

$$(-1) + 1 = 0$$

Well, if we simply ignore the last carry bit (5th bit):

$$1111 + 1 = 10000 = 0000$$

So if we define 1111 as -1, we can treat addition with negative numbers exactly the same as adding ordinary numbers. Find the representation for -2 such that

$$-2 + 2 = 10000 = 0000$$

then show that:

$$-1 + -1 = -2.$$

Python does not use this trick, but many other languages do.

2.4 Binary Representation of Real Numbers

Representing real numbers presents much more of challenge. There are an uncountably infinite number of real numbers between any two distinct rational numbers, but a computer has only finite memory and therefore a finite number of states. It is impossible for computers to exactly represent every real number. Instead, computers represent real numbers with an approximate floating point representation much like we use for scientific notation,

$$x = m \times B^n$$

where the significant m is a real number with a finite number of significant figures and the exponent n is an integer. The base B is ten for scientific notation but typically two in a floating point representation. The exponent n is typically chosen so that there is only one digit before the decimal point in the base B, e.g. 3.173×10^{-8} for scientific notation.

The limited precision of the discriminant can lead to challenges when using floating point numbers. The floating point precision is specified by the parameter ϵ (epsilon) which is the difference between one and the next highest number larger than one that can be represented. For scientific notation with four significant digits, $\epsilon = 0.001$, because we cannot represent anything between 1.000×10^0 and 1.001×10^0 with only four significant figures.

△ **Jupyter Notebook Exercise 2.8:** Determine the Python floating point ϵ by running

```
import sys
print(sys.float_info.epsilon)
```

△ **Jupyter Notebook Exercise 2.9:** Determine the Python floating point ϵ for yourself by running:

```
eps = 1.0
while eps + 1 > 1:
    eps = eps / 2
eps = eps * 2
print(eps)
```

Here the while loop continues running the indented code until the condition $\epsilon + 1 = \epsilon$ is met.

△ **Jupyter Notebook Exercise 2.10:** Python uses the IEEE 754 double-precision floating-point format. This format uses 64-bits overall, with 52 bits reserved for the significant. The standard uses a clever trick to save one bit, by requiring that the leading bit of the significant m is one, and defining 53 significant figures using 52 bits:

$$m = 1.m_1m_2m_3...m_{52}$$

Here each m_i represents an individual bit (0 or 1). Predict the parameter ϵ and compare with the above.

△ **Jupyter Notebook Exercise 2.11:** It seems like ϵ should be small enough to simply ignore it in most cases, but in fact it shows up quite clearly if you apply strict equality to floating point quanties. To see the problem, run this code, checking if $\sin(\pi)$ is zero:

```
x = np.sin(np.pi)
print(x)
print(x==0)
```

The strict equality condition $x == 0$ is not met because of limited floating point precision. Instead of strict equality, check that x is near zero with a condition like:

$$|x| < 10\epsilon$$

where ϵ is the machine precision and the factor of 10 is a conservative factor to allow for round-off errors that might be a bit larger than the best possible precision ϵ. Add such a condition to the code above and show that this looser definition of equality now holds. In general, when using approximations (like floating point numbers) you can only check things to within the accuracy of the approximation!

△ **Jupyter Notebook Exercise 2.12:** Here is another case where floating point precision joins the chat uninvited:

```
x = 0.1
y = x+x+x
print(y == 0.3)
```

Devise an alternative to `y==0.3` that properly accounts for floating point precision.

△ **Jupyter Notebook Exercise 2.13:** Personally, I prefer my zero's to look like zero. When floating point limitations are making them look non-zero, I like to clean then up with rounding, like this:

```
x = np.sin(2*np.pi)
print(x)
x = np.around(x,15)
print(x)
```

Yeck! What is -0??!! IEEE 754 defines two zeros -0 and 0. -0 is used to indicate that 0 was reached by rounding a negative number. This is so that $1/-0$ can be interpreted as $-\infty$ and $1/0$ as $+\infty$. If you just want to make this go away, add "+0":

```
x = np.around(x,15)+0
```

Show that this works.

△ **Jupyter Notebook Exercise 2.14:** Consider the following code:

```
a = 3;
b = 1.2343E-17;
sum = 0
sum += a;
for i in range(1000000):
    sum = sum + b
print(sum)
```

Is there any problem here? If there is, fix it by changing only the *order* of the lines of code.

2.5 Other Data Types

Integers and floating point numbers are the real work horses of computational physics. We'll add numpy arrays in a future lab. This section will briefly introduce the remaining types.

Python includes strings as a basic type:

```
s = "hello world"
s = s + " (it's been a strange few years)"
print(s)
print(type(s))
print(s[6],s[4],s[6])
print(type(s[0]))
```

Strings are *immutable* objects that contain textual data. If you have used other languages, you might expect `s[0]` to be a "character" but in Python it is a string of length one. There is no built-in character type.

Python includes complex numbers as a basic type:

```
z = 1 + 2j
print(type(z))
print(z.real)
print(z.imag)
print(z.conjugate())
```

This is our first example of an object oriented programming (OOP) class interface. To compute the complex conjugate of z, an ordinary function would need to be passed z as an argument, or else it would not know which complex value to use for the computation. But `z.conjugate()` is a *method* of the class complex. The method is tied to the instance of complex number z by the "." and has access to all of the data it needs from z. Similarly, the `real` and `imag` are member data of class complex: they are the integers that contain the real and imaginary parts of z. Objects play a central role in Python, but in a refeshingly understated and reserved manner. It is enough for now to understand that `z.conjugate()` is much like a function that already has z as a parameter, and `z.imag` and `r.real` are just ways to access the data contained in z.

△ **Jupyter Notebook Exercise 2.15:** Define a complex number with value of $1+i$ and multiply it by it's complex conjugate. Show that the resulting complex number has zero for it's imaginary part.

Python provides Lists as a native container of python objects. We'll make much more use of numpy arrays, which are better suited to numerical analysis, but Python Lists occassionally play a role for various bookkeeping tasks:

```
L = ["hello", 1, 2, 3+2j, 3.45, "green"]
print(L[0])
print(L[1])
print(L[5])
L[0] = "goodbye"
print(L)
```

Here the List L contains a variety of (admittedly rather useless) objects. These objects can be referred to individually by their index. One place where lists really shine is in looping over a custom list of values:

```
for i in [1,5,10,50,100]:
    print(i)
```

△ **Jupyter Notebook Exercise 2.16:** Run the following code:

```
a = [1,2,3,4,5]
b = a
b[0] = 3
print(a[0])
print(a is b)
```

What happened here? This illustrates a very important concept in Python. When you equate something to an *immutable* object, like an integer or string, it creates a *new and separate* variable. This is known as "pass by value". In fact, if you try to change the value of an immutable object, it will actually delete the old object and create a new one with the same name. The following is a list of the most common immutable objects in Python

- int
- float
- complex
- book
- str
- tuple
- bytes
- frozenset

On the other hand, almost all other objects are *mutable*. In the example above, the first line created an array. When you equated "b" to "a" on the second line, it didn't copy the array, but rather passed a pointer to the original array. This is knows as "pass by object reference" or "pass by assignment.

△ **Jupyter Notebook Exercise 2.17:** Copying a mutable object.

There's a good chance that what you did in the previous example was not what you wanted to do.

Redo the previous example, but change the second line to

```
    b = a.copy()
```

and see how it changes the behavior.

Note, both Java and Python use the default pass by reference behavior for complex objects and require you to explicitly make a copy if that's what you want. This is generally considered a big improvement over C++, which defaults to passing by value, which can result in a minor typo causing you to accidentally copy *enormous* amounts of data.

△ **Jupyter Notebook Exercise 2.18:** It's good to read the documentation, but it's a useful skill to figure things out for yourself too! Without looking up the documentation, write a snippet of code to determine for yourself if complex numbers are mutable (like lists) or immutable (like integers). We are able to change just a part of a list (like `a[0] = 3`). But can we change part of a complex number (like `z.real`). Try it and find out!

(It's OK if your code throws an error here, but you can also comment it out if you are the sort of person that can't possibly leave it alone)