1.3 Representing Data I: Numbers

Data is all around us and the amount of data stored increases every single day. In today's world, decisions must be data-driven and so it is imperative that we be able to process, analyze, and understand the data we collect. Other important factors include the security and privacy of data. Businesses and governments need to answer important questions such as "Where should this data be stored?"; "How should this data be stored?"; and even, "Should this data be stored at all?". The answers to these questions for Health Canada and personal health data is very different from the answers Nintendo might come up with for the next Animal Crossing game. If decisions must be data-driven then computers are an excellent tool for processing that data, especially when we consider that computers are several orders of magnitude faster at computing with data than a human.

We begin our study of computer science in earnest by defining different categories of data, and seeing how to represent them in the Python programming language. In over the next three sections, we'll review the common data types that we'll make great use of in this course: numeric data, boolean data, textual data, and various forms of compound data that combine multiple pieces of data into a single entity. We'll both discuss these data types independent of computers or programming language, and then learn about subtle, but crucial, differences between our theoretical conceptualizations of data and what can actually be represented in Python.

What is a data type?

Given the amount and different varieties of data in the world, it is useful for both humans and computers to have ways to categorize data. A **data type** is a precise description of a category of data that contains two parts:

1. The allowed *values* for a piece of data of that type.
2. The allowed *operations* we can perform on a piece of data of that type.

For example, we could say that a person's age is a natural number, which would tell us that values like 25 and 100 would be expected, while an age of -2 or "David" would be nonsensical. Knowing that a person's age is a natural number also tells us what operations we could perform (e.g., "add 1 to the age"), and rules out other operations (e.g., "sort these ages alphabetically").

Importantly, these data types exist ouside of program languages—we've had natural numbers for much, much longer than we've had computers, after all! At the same time, every programming language has a way of representing data types, so that programs can differentiate between data of various types when performing computations. So in this section, we'll present both **abstract** versions of common data types (sometimes called **abstract data types**), which are independent of programming language, and the corresponding **concrete data types** that existing in the Python programming language. Many terms and definitions may be review from your past studies, but be careful—they may differ slightly from what you've learned before, and it will be important to get these definitions exactly right.

Numeric data: natural numbers and integers

We'll start with two of the most common forms of numeric data, which represent numbers that have no fractional component:

- A **natural number** is a value from the set $\{0, 1, 2, \dots\}$. We use the symbol $\mathbb{N}$ to denote the set of natural numbers.[1]
- An **integer** is a value from the set $\{\dots, -2, -1, 0, 1, 2, \dots\}$. We use the symbol $\mathbb{Z}$ to denote the set of integers.

[1] Note that our convention in computer science is to consider 0 a natural number!

Of course, the natural numbers are a subset of the integers: every natural number is an integer, but not vice versa. The Python programming language defines the data type `int` to represent natural numbers and integers.[2] In Python, an `int` literal is simply the number as a sequence of digits with an optional `-` sign, like `110` or `-3421`.

[2] You might wonder why we care about natural numbers at all, and why we don't just talk about integers like Python seems to. The answer is that it is often useful to consider only integers that are ≥ 0, and so we define a special name for that category of integer.

```
>>> 110
110
>>> -3421
-3421
```

Arithmetic operations on natural numbers and integers

All integers support the familiar arithmetic operations: addition ($4 + 5$), subtraction ($4 - 5$), multiplication 4×5), exponentiation (4^5). They also support division, but that's a special case we'll discuss in more detail down below.

One additional arithmetic operation that may be less familiar to you is the **modulo operation**, which produces the *remainder* when one integer is divided by another. We'll use the percent symbol $\%$ to denote the modulo operation, writing $a \% b$ to mean "the remainder when a is divided by b". For example, $10 \% 4 = 2$ and $30 \% 3 = 0$.

Now, Python! It should not be surprising that the Python programming language supports all of these arithmetic operations, using various operators that mimic their mathematical counterparts:

```
>>> 2 + 3
5
>>> 2 - 5
-3
>>> -2 * 10
-20
>>> 2 ** 5  # This is exponentiation, "2 to the power of 5
32
>>> 10 % 4
2
```

In the second-last prompt, we included some additional text: `# This is exponentiation, "2 to the power of 5"`. In Python, we use the character `#` in code to begin a **comment**, which is code that is ignored by the Python interpreter. Comments are only meant for humans to read, and are a useful way of providing additional information about some Python code. We used it above to explain the meaning of the `**` operator.

Python supports the standard precedence rules for arithmetic operations,[3] performing exponentiation before multiplication, and multiplication before addition and subtraction:

[3] sometimes referred to as "BEDMAS" or "PEMDAS"

```
>>> 1 + 2 ** 3 * 5  # Equal to "1 plus ((2 to the power of 3) times 5)"
41
```

Just like in mathematics, long expressions like this one can be hard to read. So Python also allows you to use parentheses to group expressions together:

```
>>> 1 + ((2 ** 3) * 5)  # Equivalent to the previous expression
41
>>> (1 + 2) ** (3 * 5)  # Different grouping: "(1 plus 2) to the power of (3 times 5)"
14348907
```

Division

When we add, subtract, multiply, and use exponentiation on two integers, the result is always an integer, and so Python always produces an `int` value for these operations. But *dividing* two integers certainly doesn't always produce an integer. This is fine in mathematics, since we know how to represent fractions. But how does this affect what Python does?

It turns out that Python has two different division operators. The first is the operator `//`, and is called **floored division** (or sometimes **integer division**). For two integers `x` and `y`, the result of `x // y` is equal to the fraction $\frac{x}{y}$, *rounded down* to the nearest integer; this is also called the **quotient** of dividing `x` by `y`. Here are some examples in Python:

```
>>> 6 // 2
3
>>> 15 // 2  # 15 ÷ 2 = 7.5, and // rounds down
7
>>> -15 // 2  # Careful! -15 ÷ 2 = -7.5, which rounds down to -8
-8
```

But what about "real" division to represent a statement like $15 \div 2 = 7.5$? This is done using the exact division operator `/`:

```
>>> 15 / 2
7.5
```

The output in this case is *not* an integer, but rather a value of a different data type called `float` that Python uses to represent arbitrary real numbers, including fractional values. We'll discuss fractional values more in a little bit, but first we'll wrap up our discussion of operations on numbers with the *comparison* operators.

Arithmetic operation summary

To help you review, here is a table summarizing the seven arithmetic operators we've seen in Python so far:

Operation	Description
`a + b`	Produces the sum of `a` and `b`
`a - b`	Produces the result of subtracting `b` from `a`
`a * b`	Produces the result of multiplying `a` by `b`
`a / b`	Produces the result of dividing `a` by `b`
`a // b`	Produces the quotient when `a` is divided by `b`
`a % b`	Produces the remainder when `a` is divided by `b`
`a ** b`	Produces the result of `a` raised to the power of `b`

Comparisons

When comparing two numbers, we have the standard mathematical symbols $=$ and $\neq$ for stating whether two numbers are equal or not, as well as the symbols $<, \leq, >, \geq$ to describe which of two numbers is larger.

As with arithmetic operations, each of these mathematical symbols has a corresponding Python operator:

Operation	Description
`a == b`	Produces whether `a` and `b` are equal.
`a != b`	Produces whether `a` and `b` are *not* equal (opposite of `==`).
`a > b`	Produces whether `a` is greater than `b`.
`a < b`	Produces whether `a` is less than `b`.
`a >= b`	Produces whether `a` is greater than or equal to `b`.
`a <= b`	Produces whether `a` is less than or equal to `b`.

Here are a few examples:

```
>>> 4 == 4
True
>>> 4 != 6
True
>>> 4 < 2
False
>>> 4 >= 1
True
```

Fractional and real numbers

Now let's talk about non-integer numbers. There are a few number sets that you will be familiar with from your earlier studies:

- A **rational number** is a value from the set $\{\frac{p}{q} \mid p, q \in \mathbb{Z} \text{ and } q \neq 0\}$—that is, the set of possible fractions. This includes numbers like $\frac{3}{2}$ and $-\frac{4}{7}$, but also integers, since (for example) $3 = \frac{3}{1}$. We use the symbol $\mathbb{Q}$ to denote the set of rational numbers.
- An **irrational number** is a number with a infinite and non-repeating decimal expansion. Examples are π, e, and $\sqrt{2}$. We use the symbol $\overline{\mathbb{Q}}$ to denote the set of irrational numbers.
- A **real number** is either a rational or irrational number. We use the symbol $\mathbb{R}$ to denote the set of real numbers.

Python uses a separate data type called `float` to represent non-integer numbers. A `float` literal is written as a sequence of digits followed by a decimal point (`.`) and then another sequence of digits. Here are some examples of `float` literals:

```
>>> 7.5
7.5
>>> .123
0.123
>>> -1000.00000001
-1000.00000001
```

Operations on fractional and real numbers

From a mathematical standpoint, all of the arithmetic and comparison operations we described for integers work with `float` values as well. Here are some examples:

```
>>> 3.5 + 2.4
5.9
>>> 3.5 - 20.9
-17.4
>>> 3.2 > 1.5
True
>>> -3.2 > -1.5
False
>>> 3.5 / 2.5
1.4
```

The limitations of `float`

Here's an interesting example. Recall that the square root of a number x is equal to raising x to the power of $\frac{1}{2}$. Let's see what happens if we use what we've learned to calculate the square root of 2 in Python:

```
>>> 2 ** 0.5
1.4142135623730951
```

See the problem? $\sqrt{2}$ is an irrational number and its decimal expansion is infinite and non-repeating. But the Python interpreter, as a program run on your computer, has only a finite amount of computer memory to work with, and so cannot represent $\sqrt{2}$ exactly, just as you would not be able to write down all of the decimal places of $\sqrt{2}$ on any finite amount of paper.[4]

[4] More precisely, computers use a binary system where all data, including numbers, are represented as a sequence of 0s and 1s. This sequence of 0s and 1s is finite since computer memory is finite, and so cannot exactly represent $\sqrt{2}$. We will discuss this binary representation of numbers later this year.

So the `float` value that the Python interpreter output, `1.4142135623730951`, is an inexact approximation of $\sqrt{2}$. Let's see what happens if we take this number and square it:

```
>>> 1.4142135623730951 ** 2
2.0000000000000004
```

Or, more dramatically (and practicing with our comparison operators):

```
>>> (2 ** 0.5) ** 2 == 2
False
```

😔 This illustrates a fundamental limitation of `float`: this data type is used to represent real numbers in Python programs, but cannot always represent them exactly. Rather, a `float` value *approximates* the value of the real number; sometimes that approximation is exact, like `2.5`, but most of the time it isn't.

Mixing `int`s and `float`s

Here's an oddity:

```
>>> 6 // 2
3
>>> 6 / 2
3.0
```

Even though $\frac{6}{2}$ is mathematically an integer, the results of the division using `//` and `/` are subtly different in Python. When `x` and `y` are `int`s, `x // y` *always* evaluates to an `int`, and `x / y` *always* evaluates to a `float`, even if the value of $\frac{x}{y}$ is an integer! So `6 // 2` has value `3`, but `6 / 2` has value `3.0`. These two values represent the same mathematical quantity—the number 3—but are stored as different data types in Python, something we'll explore more later in this course when we study how `int`s and `float`s are stored in computer memory.

However, even though `3` and `3.0` are of different data types, Python does recognize them as having equal *values*:

```
>>> 3.0 == 3
True
```

Arithmetic operations with `int`s and `float`s

So to summarize: for two `int`s `x` and `y`, all of the expressions `x + y`, `x - y`, `x * y`, `x // y`, `x % y`, and `x ** y` produce `int`s, and `x / y` always produces a `float`. For two `float`s, it's even simpler: all seven of these arithmetic operations produce a `float`.[5]

[5] Even `//`. Try it!

But what happens when we mix these two data types? *An arithmetic operation that is given one `int` and one `float` always produces a `float`.* Even in long arithmetic expressions where only one value is a `float`, the whole expression will evaluate to a `float`.[6]

[6] This is true even when the resulting value is mathematically an integer, as shown in this example.

```
>>> 12 - 4 * 5 // (3.0 ** 2) + 100
110.0
```

References

- CSC108 videos: Python as a Calculator (Part 1, Part 2, Part 3)
- Appendix A.2 Python Built-In Data Types Reference