1 Number systems

In mathematics, there are many different number systems in common use. For example by the end of pre-calculus, all of the following have been introduced:

- The integers, $\{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}$;
- The rational numbers, $\{p/q : p, q \text{ are integers}, q \neq 0\}$;
- The real numbers, $\{x : -\infty < x < \infty\}$;
- The complex numbers, $\{a + bi : a, b \text{ are real numbers and } i^2 = -1\}$.

On top of these, we have special subsets, such as the natural numbers $\{0, 1, 2, \dots\}$, the even numbers, the odd numbers, positive numbers, non-negative numbers, etc.

Mathematically, these number systems are naturally nested within each other as integers are rational numbers which are real numbers, which can be viewed as part of the complex numbers.

Calculators typically have just one type of number - floating point values. These model the real numbers. `Julia`, on other other hand, has a rich type system, and within that has many different number types. There are types that model each of the four main systems above, and within each type, specializations for how these values are stored.

Most of the details will not be of interest to all, and will be described later.

For now, let's consider the number 1. It can be viewed as either an integer, rational, real, or complex number. To construct "1" in each type within `Julia` we have these different styles:

```
1, 1.0, 1//1, 1 + 0im
```

```
(1, 1.0, 1//1, 1 + 0im)
```

The basic number types in `Julia` are `Int`, `Float64`, `Rational` and `Complex`, though in fact there are many more, and the last two aren't even *concrete* types. This distinction is important, as the type of number dictates how it will be stored and how precisely the stored value can be expected to be to the mathematical value it models.

Though there are explicit constructors for these types, these notes avoid them unless necessary, as `Julia`'s parser can distinguish these types through an easy to understand syntax:

- integers have no decimal point;
- floating point numbers have a decimal point (or are in scientific notation);
- rationals are constructed from integers using the double division operator, `//`; and
- complex numbers are formed by including a term with the imaginary unit, `im`.

Heads up, the difference between `1` and `1.0` is subtle (and even more so, as `1.` will parse as `1.0`).

Similarly, each type is printed slightly differently.

The key distinction is between integers and floating points. While floating point values include integers, and so can be used exclusively on the calculator, the difference is that an integer is guaranteed to be an exact value, whereas a floating point value is typically just an *approximate* value - used to advantage, as floating point values can model a much wider set of numbers.

Now in nearly all cases, the differences are not noticed. Take for instance this simple calculation involving mixed types.

```
1 + 1.25 + 3//2
```

3 . 7 5

The sum of an integer, a floating point number and rational number returns a floating point number without a complaint.

This is because behind the scenes, `Julia` will often "promote" a type to match, so for example to compute `1 + 1.25` the integer `1` will be promoted to a floating point value and the two values are then added. Similarly, with `2.25 + 3//2`, where the fraction is promoted to the floating point value `1.5` and addition is carried out.

As floating point numbers are approximations, some values are not quite what they would be mathematically:

```
sqrt(2) * sqrt(2) - 2, sin(pi)
```

```
(4.440892098500626e-16, 1.2246467991473532e-16)
```

These values are *very* small numbers, but not exactly 0, as they are mathematically.

The only common issue is with powers. `Julia` tries to keep a predictable output from the input types (not their values). Here are the two main cases that arise where this can cause unexpected results:

- integer bases and integer exponents can *easily* overflow. Not only `m^n` is always an integer, it is always an integer with a fixed storage size computed from the sizes of `m` and `n`. So the powers can quickly get too big. This can be especially noticeable on older 32-bit machines, where too big is $2^{32} = 4,294,967,296$. On 64-bit machines, this limit is present but much bigger.

Rather than give an error though, `Julia` gives seemingly arbitrary answers, as can be seen in this example on a 64-bit machine:

```
2^62, 2^63
```

```
(4611686018427387904, -9223372036854775808)
```

This could be worked around, but it isn't, as it would slow down this basic computation. So, it is up to the user to be aware of cases where their integer values can grow to big. Again, use floating point numbers in this domain, as they have more room, at the cost of often being approximate values.

- the `sqrt` function will give a domain error for negative values:

```
sqrt(-1.0)
```

```
Error: DomainError with -1.0:
sqrt will only return a complex result if called with a complex argument. T
ry sqrt(Complex(x)).
```

This is because for real-valued inputs `Julia` expects to return a real-valued output. Of course, this is true in mathematics until the complex numbers are introduced. Similarly in `Julia` - to take square roots of negative numbers, start with complex numbers:

```
sqrt(-1.0 + 0im)
```

```
0.0 + 1.0im
```

At one point, `Julia` had an issue with a third type of power: integer bases and negative integer exponents. For example `2^(-1)`. This is now special cased. Historically, the desire to keep a predictable type for the output (integer) led to defining this case as a domain error.

1.1 Some more details.

What follows is only needed for those seeking more background.

Julia has *abstract* number types `Integer`, `Real`, and `Number`. All four types described above are of type `Number`, but `Complex` is not of type `Real`.

However, a specific value is an instance of a *concrete* type. A concrete type will also include information about how the value is stored. For example, the *integer* `1` could be stored using 64 bits as a signed integers, or, should storage be a concern, as an 8 bits signed or even unsigned integer, etc.. If storage isn't an issue, but exactness at all scales is, then it can be stored in a manner that allows for the storage to grow using "big" numbers.

These distinctions can be seen in how `Julia` parses these three values:

- `1234567890` will be a 64-bit integer (on newer machines), `Int64`
- `12345678901234567890` will be a 128 bit integer, `Int128`
- `123456789012345678901234567890123456789012345678901234567890` will be a big integer, `BigInt`

Having abstract types allows programmers to write functions that will work over a wide range of input values that are similar, but have different implementation details.

1.1.1 Integers

Integers are often used casually, as they come about from parsing. As with a calculator, floating point numbers *could* be used for integers, but in `Julia` - and other languages - it proves useful to have numbers known to have *exact* values. In `Julia` there are built-in number types for integers stored in 8, 16, 32, 64, and 128 bits and `BigInt`s if the previous aren't large enough. (8 bits can hold 8 binary values representing 1 of $256 = 2^8$ possibilities, whereas the larger 128 bit can hold one of 2^{128} possibilities.) Smaller values can be more efficiently used, and this is leveraged at the system level, but not a necessary distinction with calculus where the default size along with an occasional usage of `BigInt` suffice.

1.1.2 Floating point numbers

Floating point numbers are a computational model for the real numbers. For floating point numbers, 64 bits are used by default for both 32- and 64-bit systems, though other storage sizes can be requested. This gives a large - but still finite - set of real numbers that can be represented. However, there are infinitely many real numbers just between 0 and 1, so there is no chance that all can be represented exactly on the computer with a floating point value. Floating point then is *necessarily* an approximation for all but a subset of the real numbers. Floating point values can be viewed in normalized scientific notation as $a \cdot 2^b$ where a is the *significand* and b is the *exponent*. Save for special values, the significand a is normalized to satisfy $1 \leq |a| < 2$, the exponent can be taken to be an integer, possibly negative.

As per IEEE Standard 754, the `Float64` type gives 52 bits to the precision (with an additional implied one), 11 bits to the exponent and the other bit is used to represent the sign. Positive, finite, floating point numbers have a range approximately between 10^{-308} and 10^{308}, as 308 is about $\log_{10} \cdot 2^{1023}$. The numbers are not evenly spread out over this range, but, rather, are much more concentrated closer to 0.

You can discover more about the range of floating point values provided by calling a few different functions.

- `typemax(0.0)` gives the largest value for the type (`Inf` in this case).
- `prevfloat(Inf)` gives the largest finite one, in general `prevfloat` is the next smallest floating point value.
- `nextfloat(-Inf)`, similarly, gives the smallest finite floating point value, and in general returns the next largest floating point value.
- `nextfloat(0.0)` gives the closest positive value to 0.

- `eps()` gives the distance to the next floating point number bigger than `1.0`. This is sometimes referred to as machine precision.

Scientific notation Floating point numbers may print in a familiar manner:

```
x = 1.23
```

1 . 2 3

or may be represented in scientific notation:

```
6.23 * 10.0^23
```

6 . 2 3 e 2 3

The special coding `aeb` (or if the exponent is negative `ae-b`) is used to represent the number $a \cdot 10^b$ ($1 \leq a < 10$). This notation can be used directly to specify a floating point value:

```
x = 6.23e23
```

6 . 2 3 e 2 3

The first way of representing this number required using `10.0` and not `10` as the integer power will return an integer and even for 64-bit systems is only valid up to `10^18`. Using scientific notation avoids having to concentrate on such limitations.

Example Floating point values in scientific notation will always be normalized. This is easy for the computer to do, but tedious to do by hand. Here we see:

```
4e30 * 3e40, 3e40 / 4e30
```

```
(1.2000000000000001e71, 7.5e9)
```

The power in the first is 71, not 70 = 30+40, as the product of 3 and 4 as 12 or `1.2e^1`. (We also see the artifact of `1.2` not being exactly representable in floating point.)

Special values: Inf, -Inf, NaN The coding of floating point numbers also allows for the special values of `Inf`, `-Inf` to represent positive and negative infinity. As well, a special value `NaN` ("not a number") is used to represent a value that arises when an operation is not closed (e.g., 0.0/0.0 yields `NaN`). Except for negative bases, the floating point numbers with the addition of `Inf` and `NaN` are closed under the operations `+`, `-`, `*`, `/`, and `^`. Here are some computations that produce `NaN`:

```
0/0, Inf/Inf, Inf - Inf, 0 * Inf
```

```
(NaN, NaN, NaN, NaN)
```

Whereas, these produce an infinity

```
1/0, Inf + Inf, 1 * Inf
```

```
(Inf, Inf, Inf)
```

Finally, these are mathematically undefined, but still yield a finite value with `Julia`:

```
0^0, Inf^0
```

```
(1, 1.0)
```

Floating point numbers and real numbers Floating point numbers are an abstraction for the real numbers. For the most part this abstraction works in the background, though there are cases where one needs to have it in mind. Here are a few:

- For real and rational numbers, between any two numbers $a < b$, there is another real number in between. This is not so for floating point numbers which have a finite precision. (Julia has some functions for working with this distinction.)
- Floating point numbers are approximations for most values, even simple rational ones like $1/3$. This leads to oddities such as this value not being 0:

```
sqrt(2)*sqrt(2) - 2
```

4 . 4 4 0 8 9 2 0 9 8 5 0 0 6 2 6 e - 1 6

It is no surprise that an irrational number, like $\sqrt{2}$, can't be represented **exactly** within floating point, but it is perhaps surprising that simple numbers can not be, so $1/3$, $1/5$, ... are approximated. Here is a surprising-at-first consequence:

```
1/10 + 2/10 == 3/10
```

false

That is adding `1/10` and `2/10` is not exactly `3/10`, as expected mathematically. Such differences are usually very small and are generally attributed to rounding error. The user needs to be mindful when testing for equality, as is done above with the `==` operator.

- Floating point addition is not necessarily associative, that is the property $a + (b + c) = (a + b) + c$ may not hold exactly. For example: