1 Ranges and Sets

Sequences of numbers are prevalent in math. A simple one is just counting by ones:

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \ldots$$

Or counting by sevens:

$$7, 14, 21, 28, 35, 42, 49, \ldots$$

More challenging for humans is counting backwards by 7:

$$100, 93, 86, 79, \ldots$$

These are examples of arithmetic sequences. The form of the first $n + 1$ terms in such a sequence is:

$$a_0, a_0 + h, a_0 + 2h, a_0 + 3h, \ldots, a_0 + nh$$

The formula for the a_nth term can be written in terms of a_0, or any other $0 < m \leq n$ with $a_n = a_m + (n - m) \cdot h$.

A typical question might be: The first term of an arithmetic sequence is equal to 200 and the common difference is equal to -10. Find the value of a_{20}. We could find this using $a_n = a_0 + n \cdot h$:

```
a0, h, n = 200, -10, 20
a0 + n * h
```

0

More complicated questions involve solving for an unknown first, as with: an arithmetic sequence has a common difference equal to 10 and its 6th term is equal to 52. Find its 15th term, a_{15}. Here we have to answer: $a_0 + 15 \cdot 10$, but to find a_0, we use $52 = a_0 + 6 \cdot (10)$. This yields $a_0 = -8$, and so:

```
a0, h, n = -8, 10, 15
a0 + n * h
```

142

Rather than express sequences by the a_0, h, and n, `Julia` uses the starting point (`a`), the difference (`h`) and a *suggested* stopping value (`b`). That is, we need three values to specify these ranges of numbers: a `start`, a `step`, and an `endof`. `Julia` gives a convenient syntax for this: `a:h:b`. When the difference is just 1, all numbers between the start and end are specified by `a:b`, as in

```
1:10
```

```
1:10
```

But wait, nothing printed? This is because `1:10` is efficiently stored. Basically, only the current state and a means to generate the next value are kept. To expand the values, you have to ask for them to be `collect`ed (though this typically isn't needed in practice):

```
collect(1:10)
```

```
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
```

When a non-default step size is needed, it goes in the middle, as in `a:h:b`. For example, counting by sevens from 1 to 50 is achieved by:

```
collect(1:7:50)
```

```
8-element Array{Int64,1}:
  1
  8
 15
 22
 29
 36
 43
 50
```

Or counting down from 100:

```
collect(100:-7:1)
```

```
15-element Array{Int64,1}:
 100
  93
  86
  79
  72
  65
  58
  51
  44
  37
 30
 23
 16
  9
  2
```

In this last example, we said end with 1, but it ended with 2. The ending value in the range is a suggestion to go up to, but not exceed. Negative values for `h` are used to make decreasing sequences.

1.0.1 The range function

For generating points to make graphs, a natural set of points to specify is n evenly spaced points between a and b. We can mimic creating this set with the range operation by solving for the correct step size. We have $a_0 = a$ and $a_0 + (n-1) \cdot h = b$. (Why $n-1$ and not n?) Solving yields $h = (b-a)/(n-1)$. To be concrete we might ask for 9 points between -1 and 1:

```
a, b, n = -1, 1, 9
h = (b-a)/(n-1)
collect(a:h:b)
```

```
9-element Array{Float64,1}:
 -1.0
 -0.75
 -0.5
 -0.25
  0.0
  0.25
  0.5
  0.75
  1.0
```

Pretty neat. If we were doing this many times - such as once per plot - we'd want to encapsulate this into a function, for example:

```
function evenly_spaced(a, b, n)
    h = (b-a)/(n-1)
    collect(a:h:b)
end
```

```
evenly_spaced (generic function with 1 method)
```

Great, let's try it out:

```
evenly_spaced(0, 2pi, 5)
```

```
5-element Array{Float64,1}:
 0.0
 1.5707963267948966
 3.141592653589793
 4.71238898038469
 6.283185307179586
```

Now, our implementation was straightforward, but only because it avoids somethings. Look at something simple:

```
evenly_spaced(1/5, 3/5, 3)
```

```
3-element Array{Float64,1}:
 0.2
 0.4
 0.6
```

It seems to work as expected. But looking just at the algorithm it isn't quite so clear:

```
1/5, 1/5 + 1*1/5, 1/5 + 2*1/5
```

```
(0.2, 0.4, 0.6000000000000001)
```

Floating point roundoff leads to the last value *exceeding* `0.6`, so should it be included? Well, here it is pretty clear it *should* be, but better to have something programmed that hits both `a` and `b` and adjusts `h` accordingly.

Enter the base function `range` which solves this seemingly simple - but not really - task. It can use `a`, `b`, and `n`. Like the range operation, this function returns a generator which can be collected to realize the values.

The number of points is specified with keyword arguments, as in:

```
xs = range(-1, 1, length=9)
```

```
-1.0:0.25:1.0
```

and

```
collect(xs)
```

```
9-element Array{Float64,1}:
 -1.0
 -0.75
 -0.5
 -0.25
  0.0
  0.25
  0.5
  0.75
  1.0
```

For `Julia` version `1.0` the stop value is also specified with a keyword, as in `range(-1, stop=1, length=9)`. An adjustment will need to be made if that version is used.

1.1 Filtering and modifying arithmetic progressions

Now we concentrate on some more general styles to modify a sequence to produce a new sequence.

For example, another way to get the values between 0 and 100 that are multiples of 7 is to start with all 101 values and throw out those that don't match. To check if a number is divisible by 7, we could use the `rem` function. It gives the remainder upon division. Multiples of 7 match `rem(m, 7) == 0`. Checking for divisibility by seven is unusual enough there is nothing built in for that, but checking for division by 2 is common, and for that, there is a built-in function `iseven`.

The act of throwing out elements of a collection based on some condition is called *filtering*. The `filter` function does this in `Julia`; the basic syntax being `filter(predicate_function, collection)`. The "`predicate_function`" is one that returns either `true` or `false`, such as `iseven`. The output of `filter` consists of the new collection of values - those where the predicate returns `true`.

To see it used, lets start with the numbers between `0` and `25` (inclusive) and filter out those that are even:

```
filter(iseven, 0:25)
```

```
13-element Array{Int64,1}:
  0
  2
  4
  6
  8
 10
 12
 14
 16
 18
 20
 22
 24
```

To get the numbers between 1 and 100 that are divisible by 7 requires us to write a function akin to `iseven`, which isn't hard (e.g., `is_seven(x)=x%7==0`), but isn't something we continue with just yet.

For another example, here is an inefficient way to list the prime numbers between 100 and 200. This uses an inefficient `isprime` function (which simply checks all possible factors do not divide `n` evenly), defined below. For real-world usage, the `Primes` package provides an alternative.

```
isprime(n) =  all(!iszero(rem(n, i)) for i in 2:floor(Int,sqrt(n)))
filter(isprime, 100:200)
```

```
21-element Array{Int64,1}:
 101
 103
 107
 109
 113
 127
 131
 137
 139
 149
   ⋮@*(163167173179181191193197199
```

Illustrating `filter` at this point is mainly a motivation to illustrate that we can start with a regular set of numbers and then modify or filter them. The function takes on more value once we discuss how to write predicate functions.

1.2 Comprehensions

Let's return to the case of the set of even numbers between 0 and 100. We have many ways to describe this set:

- The collection of numbers $0, 2, 4, 6 \ldots, 100$, or the arithmetic sequence with step size 2, which is returned by `0:2:100`.
- The numbers between 0 and 100 that are even, that is `filter(iseven, 0:100)`.
- The set of numbers $\{2k : k = 0, \ldots, 50\}$.

While `Julia` has a special type for dealing with sets, we will use a vector for such a set. (Unlike a set, vectors can have repeated values, but as vectors are more widely used, we demonstrate them.) Vectors are described more fully in a previous section, but as a reminder, vectors are constructed using square brackets: `[]` (a special syntax for concatenation). Square brackets are used in different contexts within `Julia`, in this case we use them to create a *collection*. If we separate single values in our collection by commas (or semicolons), we will create a vector:

```
x = [0, 2, 4, 6, 8, 10]
```

```
6-element Array{Int64,1}:
  0
  2
  4
  6
  8
 10
```

That is of course only part of the set of even numbers we want. Creating more might be tedious were we to type them all out, as above. In such cases, it is best to *generate* the values.

For this simple case, a range can be used, but more generally a comprehension provides this ability using a construct that closely mirrors a set definition, such as $\{2k : k = 0, \dots, 50\}$. The simplest use of a comprehension takes this form (as we described in the section on vectors):

`[expr for variable in collection]`

The expression typically involves the variable specified after the keyword `for`. The collection can be a range, a vector, or many other items that are *iterable*. Here is how the mathematical set $\{2k : k = 0, \dots, 50\}$ may be generated by a comprehension:

```
[2k for k in 0:50]
```

```
51-element Array{Int64,1}:
   0
   2
   4
   6
   8
  10
  12
  14
  16
  18
   ⋮@*(8486889092949698100
```

The expression is `2k`, the variable `k`, and the collection is the range of values, `0:50`. The syntax is basically identical to how the math expression is typically read aloud.

For some other examples, here is how we can create the first 10 numbers divisible by 7:

```
[7k for k in 1:10]
```

```
10-element Array{Int64,1}:
  7
 14
 21
 28
 35
 42
 49
 56
```

```
63
70
```

Here is how we can square the numbers between 1 and 10:

```
[x^2 for x in 1:10]
```

```
10-element Array{Int64,1}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

To generate other progressions, such as powers of 2, we could do:

```
[2^i for i in 1:10]
```

```
10-element Array{Int64,1}:
    2
    4
    8
   16
   32
   64
  128
  256
  512
 1024
```

Here are decreasing powers of 2:

```
[1/2^i for i in 1:10]
```

```
10-element Array{Float64,1}:
 0.5
 0.25
 0.125
 0.0625
 0.03125
 0.015625
 0.0078125
 0.00390625
 0.001953125
 0.0009765625
```

Sometimes, the comprehension does not produce the type of output that may be expected. This is related to `Julia`'s more limited abilities to infer types at the command line. If the output type is important, the extra prefix of `T[]` can be used, where `T` is the desired type. We will see that this will be needed at times with symbolic math.

1.3 Generators

A typical pattern would be to generate a collection of numbers and then apply a function to them. For example, here is one way to sum the powers of 2:

```
sum([2^i for i in 1:10])
```

2046

Conceptually this is easy to understand, but computationally it is a bit inefficient. The generator syntax allows this type of task to be done more efficiently. To use this syntax, we just need to drop the `[]`:

```
sum(2^i for i in 1:10)
```

2046

(The difference being no intermediate object is created to store the collection of all values specified by the generator.)

1.3.1 Filtering generated expressions - the "if" keyword in a generator

Both comprehensions and generators allow for filtering through the keyword `if`. The following shows *one* way to add the prime numbers in $[1, 100]$:

```
sum(p for p in 1:100 if isprime(p))
```

1061

The value on the other side of `if` should be an expression that evaluates to either `true` or `false` for a given `p` (like a predicate function, but here specified as an expression). The value returned by `isprime(p)` is such.

In this example, we use the fact that `rem(k, 7)` returns the remainder found from dividing `k` by `7`, and so is `0` when `k` is a multiple of `7`:

```
sum(k for k in 1:100 if rem(k,7) == 0)  ## add multiples of 7
```

735

Example This example of Stefan Karpinski comes from a blog post highlighting changes to the `Julia` language with version `v"0.5.0"`, which added features to comprehensions that made this example possible. It involves making change. First, a simple question: using pennies, nickels, dimes, and quarters how many different ways can we generate one dollar? Clearly 100 pennies, or 20 nickels, or 10 dimes, or 4 quarters will do this, so the answer is at least four, but how much more than four?

Well, we can use a comprehension to enumerate the possibilities. This example illustrates how comprehensions and generators can involve one or more variable for the iteration.

First, we either have $0, 1, 2, 3$, or 4 quarters, or 0, 25 cents, 50 cents, 75 cents, or a dollar's worth. If we have, say, 1 quarter, then we need to make up 75 cents with the rest. If we had 3 dimes, then we need to make up 45 cents out of nickels and pennies, if we then had 6 nickels, we know we must need 15 pennies.

The following expression shows how counting this can be done through enumeration. Here `q` is the amount contributed by quarters, `d` the amount from dimes, `n` the amount from nickels, and `p` the amount from pennies. `q` ranges over $0, 25, 50, 75, 100$ or `0:25:100`, etc. If we know that the sum of quarters, dimes, nickels contributes a certain amount, then the number of pennies must round things up to 100.

```
ways = [(q, d, n, p) for q = 0:25:100 for d = 0:10:(100 - q) for n = 0:5:(100 - q - d)
for p = (100 - q - d - n)]
length(ways)
```

242

We see 242 cases, each distinct. The first 3 are:

```
ways[1:3]
```

```
3-element Array{NTuple{4,Int64},1}:
 (0, 0, 0, 100)
 (0, 0, 5, 95)
 (0, 0, 10, 90)
```

The generating expression reads naturally. It introduces the use of multiple `for` statements, each subsequent one depending on the value of the previous (working left to right). Now suppose, like a shop keeper, we want to ensure that the amount of pennies is less than the amount from nickels, etc. We could use `filter` somehow to do this for our last answer, but using `if` allows for filtering while the events are generating. Here our condition is simply expressed: `q > d > n > p`:

```
[(q, d, n, p) for q = 0:25:100 for d = 0:10:(100 - q) for n = 0:5:(100 - q - d) for p =
(100 - q - d - n) if q > d > n > p]
```

```
4-element Array{NTuple{4,Int64},1}:
 (50, 30, 15, 5)
 (50, 30, 20, 0)
 (50, 40, 10, 0)
 (75, 20, 5, 0)
```

1.4 Random numbers

We have been discussing structured sets of numbers. On the opposite end of the spectrum are random numbers. `Julia` makes them easy to generate, especially random numbers chosen uniformly for $(0, 1)$.

- The `rand()` function returns a randomly chosen number in $(0, 1)$.
- The `rand(n)` function returns `n` randomly chosen numbers in $(0, 1)$.

To illustrate, this will return a single number

```
rand()
```

0 . 2 4 8 7 2 8 5 1 8 1 3 8 4 4 6

If the command is run again, it is almost certain that a different value will be returned:

```
rand()
```

0 . 8 2 7 9 8 2 4 5 0 9 4 4 8 7 0 3

This call will return a vector of 10 such random numbers:

```
rand(10)
```

```
10-element Array{Float64,1}:
 0.3171239488894584
 0.6610537440357966
 0.3561518467501743
 0.04102197231253735
 0.9546865324418858
 0.796781065831266
 0.8994826306943817
 0.7111315905172839
 0.572606966233651
 0.4213611088850344
```

Easy to use. The only common source of confusion is the subtle distinction between `rand()` and `rand(1)`, as the latter is a vector of 1 random number and the former just 1 random number.

The documentation for `rand` shows that the value is in $[0, 1)$, but in practice 0 doesn't come up with any frequency - about 1 out of every 10^{19} numbers - so we say $(0, 1)$.

1.5 Questions

⊛ Question
Which of these will produce the odd numbers between 1 and 99?

1. `1:3:99`
2. `1:99`
3. `1:2:99`

⊛ Question
Which of these will create the sequence $2, 9, 16, 23, \ldots, 72$?

1. `72:-7:2`
2. `2:72`
3. `2:7:72`
4. `2:9:72`

⊛ Question
How many numbers are in the sequence produced by `0:19:1000`?

⊛ Question
The range operation (`a:h:b`) can also be used to countdown. Which of these will do so, counting down from `10` to `1`? (You can call `collect` to visualize the generated numbers.)

1. `1:-1:10`
2. `10:-1:1`
3. `10:1`
4. `1:10`

⊛ Question
What is the last number generated by `1:4:7`?

⊛ Question
While the range operation can generate vectors by collecting, do the objects themselves act like vectors?

Does scalar multiplication work as expected? In particular, is the result of `2*(1:5)` *basically* the same as `2 * [1,2,3,4,5]`?

1. Yes
2. No

Does vector addition work? as expected? In particular, is the result of `(1:4) + (2:5)` *basically* the same as `[1,2,3,4]` + `[2,3,4,5]`?

1. Yes
2. No

What if parenthese are left off? Explain the output of `1:4 + 2:5`?

1. It is just random
2. It gives the correct answer, a generator for the vector `[3,5,7,9]`
3. Addition happens prior to the use of `:` so this is like `1:(4+2):5`

⊛ Question
How is `a:b-1` interpreted:

1. as `a:(b-1)`
2. as `(a:b) - 1`, which is `(a-1):(b-1)`

⊛ Question
Create the sequence $10, 100, 1000, \ldots, 1,000,000$ using a list comprehension. Which of these works?

1. `[i^10 for i in [1:6]]`
2. `[10^i for i in 1:6]`
3. `[10^i for i in [10, 100, 1000]]`

⊛ Question
Create the sequence $0.1, 0.01, 0.001, \ldots, 0.0000001$ using a list comprehension. Which of these will work:

1. `[i^(1/10) for i in 1:7]`
2. `[10^-i for i in 1:7]`
3. `[(1/10)^i for i in 1:7]`

⊛ Question
Evaluate the expression $x^3 - 2x + 3$ for each of the values $-5, -4, \ldots, 4, 5$ using a comprehension. Which of these will work?

1. `[x^3 - 2x + 3 for x in -(5:5)]`
2. `[x^3 - 2x + 3 for i in -5:5]`
3. `[x^3 - 2x + 3 for x in -5:5]`

⊛ Question
How many prime numbers are there between 1100 and 1200? (Use `filter` and `isprime`)

⊛ Question
Which has more prime numbers the range `1000:2000` or the range `11000:12000`?

1. `1000:2000`
2. `11000:12000`

⊛ Question
We can easily add an arithmetic progression with the `sum` function. For example, `sum(1:100)` will add the numbers $1, 2, ..., 100$.

What is the sum of the odd numbers between 0 and 100?

⊛ Question
The sum of the arithmetic progression $a, a+h, \dots, a+n\cdot h$ has a simple formula. Using a few cases, can you tell if this is the correct one:

$$(n+1)\cdot a + h\cdot n(n+1)/2$$

1. Yes, this is true
2. No, this is false

⊛ Question
A *geometric progression* is of the form $a^0, a^1, a^2, \dots, a^n$. These are easily generated by comprehensions of the form `[a^i for i in 0:n]`. Find the sum of the geometric progression $1, 2^1, 2^2, \dots, 2^{10}$.

Is your answer of the form $(1-a^{n+1})/(1-a)$?

1. Yes
2. No

⊛ Question
The product of the terms in an arithmetic progression has a known formula. The product can be found by an expression of the form `prod(a:h:b)`. Find the product of the terms in the sequence $1, 3, 5, \dots, 19$.