Week 2:

Control flow and functions.

Functions:

```
# Define a function that adds two numbers together

function add_numbers(x, y)

        return x + y

end


# Call the add_numbers function with arguments 2 and 3

result = add_numbers(2, 3)


# Print the result of the function call

println("The result of adding 2 and 3 is ", result)
```

Functions are a key part of programming in Julia, as they allow you to encapsulate and reuse blocks of code. When defining a function, you specify its name, any input arguments it takes (which can have default values), and the code to be executed when the function is called. You can then call the function by its name and pass in any required arguments.

Functions can also return values, which can be assigned to variables or used directly in other calculations. In this example, we return the sum of the input arguments by using return keyword. If we had not included a return statement, the function would still be executed but would not produce any output.

```
# Define a function that takes no arguments

function say_hello()

        println("Hello!")

end
```

```
# Define a function that takes one argument
function double(x)
    return 2 * x
end
# Define a function that takes multiple arguments
function add_numbers(x, y)
   return x + y
end
# Define an anonymous function (also called a lambda function)
anonymous_func = x -> x^2
# Call the functions
say_hello()
println(double(3))
println(add_numbers(2, 3))
println(anonymous_func(4))
```

- say_hello is a function that takes no arguments and simply prints out a greeting message.
- double is a function that takes one argument and returns twice that value.
- add_numbers is a function that takes two arguments and returns their sum.
- anonymous_func is an anonymous function (also known as a lambda function) that takes one argument and returns its square.

We then call each of these functions and print out the results. Note that we can define anonymous functions using the -> syntax, which specifies the input argument(s) on the left and the expression to be evaluated on the right.

Julia supports a wide variety of function types, including those that take no arguments, those that take one or more arguments, those that return values, and those

that don't. Additionally, Julia also supports higher-order functions, which are functions that take other functions as arguments or return functions as values.

Control Flow:

Compound Expressions:

Sometimes it is convenient to have a single expression which evaluates several subexpressions in order, returning the value of the last subexpression as its value. There are two Julia constructs that accomplish this: begin blocks and (;) chains. The value of both compound expression constructs is that of the last subexpression. Here's an example of a begin block:

```
julia> z = begin
           x = 1
           y = 2
           x + y
       end
3
```

Since these are fairly small, simple expressions, they could easily be placed onto a single line, which is where the (;) chain syntax comes in handy:

```
julia> z = (x = 1; y = 2; x + y)
3
```

This syntax is particularly useful with the terse single-line function definition form introduced in Functions. Although it is typical, there is no requirement that begin blocks be multiline or that (;) chains be single-line:

```
julia> begin x = 1; y = 2; x + y end
3

julia> (x = 1;
       y = 2;
       x + y)
3
```

Conditional Evaluation:

Conditional evaluation allows portions of code to be evaluated or not evaluated depending on the value of a boolean expression. Here is the anatomy of the if-elseif-else conditional syntax:

```
if x < y
  println("x is less than y")
elseif x > y
  println("x is greater than y")
else
  println("x is equal to y")
end
```

If the condition expression x < y is true, then the corresponding block is evaluated; otherwise the condition expression x > y is evaluated, and if it is true, the corresponding block is evaluated; if neither expression is true, the else block is evaluated. Here it is in action:

```
julia> function test(x, y)
  if x < y
    println("x is less than y")
  elseif x > y
    println("x is greater than y")
  else
    println("x is equal to y")
  end
end
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

```
julia> function test(x,y)
    if x < y
        relation = "less than"
    elseif x == y
        relation = "equal to"
    else
        relation = "greater than"
    end
    println("x is ", relation, " y.")
end
test (generic function with 1 method)

julia> test(2, 1)
x is greater than y.

julia> x = 3

3

julia> if x > 0
"positive!"
else
"negative..."
end
"positive!"
```

The easiest way to understand this behavior is to see an example. In the previous example, the println call is shared by all three branches: the only real choice is which literal string to print. This could be written more concisely using the ternary operator. For the sake of clarity, let's try a two-way version first:

```
julia> x = 1; y = 2;

julia> println(x < y ? "less than" : "not less than")
less than
```

```
julia> x = 1; y = 0;

julia> println(x < y ? "less than" : "not less than")
not less than
```

If the expression x < y is true, the entire ternary operator expression evaluates to the string "less than" and otherwise it evaluates to the string "not less than". The original three-way example requires chaining multiple uses of the ternary operator together:

```
julia> test(x, y) = println(x < y ? "x is less than y"    :
                            x > y ? "x is greater than y" : "x is          equal to y")
test (generic function with 1 method)

julia> test(1, 2)
x is less than y

julia> test(2, 1)
x is greater than y

julia> test(1, 1)
x is equal to y
```

Short-Circuit Evaluation:

Short-circuit evaluation is quite similar to conditional evaluation. The behavior is found in most imperative programming languages having the && and || boolean operators: in a series of boolean expressions connected by these operators, only the minimum number of expressions are evaluated as are necessary to determine the final boolean value of the entire chain. Explicitly, this means that:

In the expression a && b, the subexpression b is only evaluated if a evaluates to true.

In the expression a || b, the subexpression b is only evaluated if a evaluates to false.

The reasoning is that a && b must be false if a is false, regardless of the value of b, and likewise, the value of a || b must be true if a is true, regardless of the value of b. Both && and || associate to the right, but && has higher precedence than || does. It's easy to experiment with this behavior:

```
julia> t(x) = (println(x); true)
t (generic function with 1 method)
```

```
julia> f(x) = (println(x); false)
f (generic function with 1 method)

julia> t(1) && t(2)
1
2
true

julia> t(1) && f(2)
1
2
false

julia> f(1) && t(2)
1
false

julia> f(1) && f(2)
1
false

julia> t(1) || t(2)
1
true

julia> t(1) || f(2)
1
true

julia> f(1) || t(2)
1
2
true

julia> f(1) || f(2)
```

```
1
2
false
```

Boolean operations without short-circuit evaluation can be done with the bitwise boolean operators introduced in Mathematical Operations and Elementary Functions: & and |. These are normal functions, which happen to support infix operator syntax, but always evaluate their arguments:

```
julia> f(1) & t(2)
1
2
false

julia> t(1) | t(2)
1
2
true
```

On the other hand, any type of expression can be used at the end of a conditional chain. It will be evaluated and returned depending on the preceding conditionals:

```
julia> true && (x = (1, 2, 3))
(1, 2, 3)

julia> false && (x = (1, 2, 3))
false
```

Repeated Evaluation:

There are two constructs for repeated evaluation of expressions: the while loop and the for loop. Here is an example of a while loop:

```
julia> i = 1;

julia> while i <= 5
println(i)
```

```
    i += 1
end
1
2
3
4
5
```

The while loop evaluates the condition expression (i <= 5 in this case), and as long it remains true, keeps also evaluating the body of the while loop. If the condition expression is false when the while loop is first reached, the body is never evaluated.

The for loop makes common repeated evaluation idioms easier to write. Since counting up and down like the above while loop does is so common, it can be expressed more concisely with a for loop:

```
julia> for i = 1:5
    println(i)
end
1
2
3
4
5
```

Here the 1:5 is a Range object, representing the sequence of numbers 1, 2, 3, 4, 5. The for loop iterates through these values, assigning each one in turn to the variable i. One rather important distinction between the previous while loop form and the for loop form is the scope during which the variable is visible. If the variable i has not been introduced in an other scope, in the for loop form, it is visible only inside of the for loop, and not afterwards. You'll either need a new interactive session instance or a different variable name to test this:

```
julia> for j = 1:5
    println(j)
end
1
2
```

```
3
4
5

julia> j
ERROR: UndefVarError: j not defined
```

It is sometimes convenient to terminate the repetition of a while before the test condition is falsified or stop iterating in a for loop before the end of the iterable object is reached. This can be accomplished with the break keyword:

```
julia> i = 1;


julia> while true
println(i)
if i >= 5
   break
end
i += 1
end
1
2
3
4
5

julia> for i = 1:1000
println(i)
if i >= 5
   break
end
end
1
2
3
4
5
```

Without the break keyword, the above while loop would never terminate on its own, and the for loop would iterate up to 1000. These loops are both exited early by using break.

In other circumstances, it is handy to be able to stop an iteration and move on to the next one immediately. The continue keyword accomplishes this:

This is a somewhat contrived example since we could produce the same behavior more clearly by negating the condition and placing the println call inside the if block. In realistic usage there is more code to be evaluated after the continue, and often there are multiple points from which one calls continue.

```
julia> for i = 1:10
        if i % 3 != 0
            continue
        end
        println(i)
    end
3
6
9
```

Multiple nested for loops can be combined into a single outer loop, forming the cartesian product of its iterables:

```
julia> for i = 1:2, j = 3:4
        println((i, j))
    end
(1, 3)
(1, 4)
(2, 3)
(2, 4)
```

Basic Conditional Statement Example:

julia> bank_balance = 4583.11

4583.11

```
julia> withdraw_amount = 250
250

julia> if withdraw_amount <= bank_balance
            bank_balance -= withdraw_amount
           print("Withdrew ", withdraw_amount, " from your account")
 end
Withdrew 250 from your account
```

else/elseif

```
   julia> withdraw_amount = 4600
   4600

   julia> if withdraw_amount <= bank_balance
          bank_balance -= withdraw_amount
print("Withdrew ", withdraw_amount, " from your account")
   else
print("Insufficent balance")
   end
   Insufficent balance
```

Week 3

Data Structures (Dataframes, Arrays and Dictionaries)

Dictionaries

A dictionary (in computer programming) is much like an English dictionary. It takes a set of objects (the keys) and maps them to something else (the values). In an English dictionary the object/keys are words and they are mapped to values called definitions, but in programming the objects could be any type of data.

Dictionaries are also called associative arrays because we can conceptualize them as a generalization of a standard array. In a standard array the indexes are contiguous integers usually starting at either 0 or 1. In a dictionary, the indexes can be almost anything we like. The dictionary associates the key/index with the value.

In Julia a dictionary is called a ***Dict***. They're a core part of the language, so you don't need to include any special packages to use them. Critical things to know include:

1. The keys and values of a Dict can be almost anything but keys have to have an associated hash function so if you create a new data type, you may need to define this. My strategy has been to define hash functions recursively in terms of hashes that already exist. Hash functions are useful enough that I will do a post on these at some point.

2. If we restrict the types of the entries we can make the dictionary much more efficient, so it usually better to have some control, not just blurt stuff into it.

3. Dictionary elements are not sorted

Operations:

1. Creation of a new dictionary. Julia's uses constructor functions to create new variables, the standard syntax being the name of the datatype Dict following by the input arguments in round braces. Types specified in curly braces before the input arguments provide extra information so that a more specific constructor can be used to construct a more efficient variable.

```
julia> D = Dict{String, Int}()              # create an empty dictionary
julia> D = Dict("a" => 1, "b" => 2, "c" => 3) # start the dictionary with some pairs
julia> dict = Dict(string(i) => sind(i) for i = 0:5:360) # comprehension-like syntax
```

2. Addition of a (key,value) pair. Julia uses square braces for indexing into arrays (associative or otherwise) so this notation is pretty consistent. Unlike Matlab you can't just add a new element to a standard array, but you can to an associative array (a Dict). This is deliberate – to understand why, you should learn about the underlying implementations.

   julia> D["d"] = 59 # assign the value 59 to the key "d"

3. Removal of a (key,value) pair uses the delete! function. The exclamation mark in the name is a Julia idiom to indicate that the function operates on the input variable in place.

   julia> delete!(D, "d") # remove the key "d" from Dict D

4. Lookup the value associated with a key. There are a few approaches to find the value or test if it exists. I doubt I am listing them all below, but it's enough to get going.

   julia> D["a"] # get the value associated with key "a"
   julia> haskey(D, "a") # check if the key "a" is in the dictionary
   julia> get(D, "a", 0) # look up key "a", with fallback value of 0 it if is absent

5. Update of the value associated with a key. Often programming languages combine the update and insertion operations, as does Julia, for instance, we can update the value corresponding to key "d" using D["d"] = 60.

Sorting:

Dictionaries are not sorted, so one snippet of code I find often useful is to obtain dictionary elements in some order, in this case in the reverse order of the values of D.

julia> sort(collect(D), by = tuple -> last(tuple), rev=true)

The last argument says to sort in reverse order, and isn't needed. The by argument is specifying a function that is being used to output a sortable index (presuming that the values are such). The collect function converts the dictionary into an array of pairs/tuples1, which is then sorted based on the last elements of those tuples. The output is also as an array of Pairs.

Program:

CSV file with two columns, one of canonical character names, and the other a list of aliases. Here's the first few entries, and you can click on the link to get the whole file.
https://aleph-zero-heroes.netlify.com/csv/alias_list.csv

Converting the cast lists from IMDb requires us to read the aliases and generate a dictionary containing a mapping from each alias to the unique name we are going to use for each character. That is we create a dictionary called Aliases that has character aliases as keys, and the canonical character name as the value.

```
using DataFrames
using CSV
function read_aliases( file::String = "../Data/alias_list.csv")
    aliases = CSV.read( file; comment="#" )
    Aliases = Dict{String, String}()
    for r in eachrow(aliases)
        c = strip( r[:Character] )
        for a in split( r[:Aliases], "," )
            Aliases[ lowercase(strip(a)) ] = c
        end
    end
    return Aliases
end
```