Ch2_notes

March 8, 2024

1 Chapter/Part 2 Notes - Getting started with Julia

This chapter, whilst being very elementary, is actually a terrific refresher on Julia for those who see themselves as somewhat seasoned

See the bit representation of a values using **bitstring()**

[1]:
```
bitstring(1)
```

[1]:
```
"0000000000000000000000000000000000000000000000000000000000000001"
```

A vector in julia is a one dimensional array - it would be defined as `Array{Int, 1}` whereby the parenthesis indicate the type followed by the dimensions - in the row, cols format

[3]:
```
vec = [1, 2, 3] ; typeof(vec)
```

[3]:
```
Vector{Int64} (alias for Array{Int64, 1})
```

[6]:
```
# This should tell us true
isa(vec, Array)
```

[6]:
```
true
```

Be careful when binding variables to other variables, for the modification of one will propagate to the other and also change it. What you want to do instead is perform a **copy()** or a **deepcopy** - the differences between them being the physical allocation of the deepcopy is different, so an equivalence evalution will come up as false as they are in fact "different" entities in an ontological sense even though they may have equal values. E.g. you and I are both humans but are not copies of one another. A bit of a tickly subject but I believe it becomes more relevant when handling memory strictly etc.

Due to round off error in floating point numbers, `1.0 + 2.0 = 3.0` is going to be evaluated as false. Technically we cannot put a truth statement to this as floats are approximiations of intergers and so they are only precise to a certain decimal place, and so using them in our code like this is bound to cause errors. I we insist on using floats, we can use the **isapprox()** function to evaluate the statement. We can provide arguments to the function indicating the level of precision we're after

[7]:
```
isapprox(1.0 + 2.0, 3.0)
```

```
[7]: true
```

1.1 Ternary operator

The ternary operator is super handy for creating concise statements and evaluating conditions within creating an explicit loop. The basic structure of the ternary is `x > y ? println("x is greater than y") : println("x is less than or equal to y")` - the combination of `?` and `:` with the else statement

```
[8]: x, y = 5, 6
     x > y ? println("x is greater than y") : println("x is less than or equal to y")
```

```
x is less than or equal to y
```

Something which eluded me all this time is the ability to assign variables to the outcome of a conditional statement - for instance, assign to y the outcome of an if-else statement.

```
[14]: z = 15
      y = if z % 3 == 0
              sqrt(z)
          else
              sqrt(-z)
      end
```

```
[14]: 3.872983346207417
```

Using the ternary operator this would be condensed too

```
[15]: y = z % 3 == 0 ? sqrt(z) : sqrt(-z)
```

```
[15]: 3.872983346207417
```

We can even cleverly embed the ternary operator within a loop, in the println() block

```
[16]: for i in [1, 2, 3]
          println(i, " is ", isodd(i) ? "odd" : "even")
      end
```

```
1 is odd
2 is even
3 is odd
```

Now using a while loop - this helps to avoid infinite regress

```
[18]: i = 0
      while i < 4
          println(i, " is ", isodd(i) ? "odd" : "even")
          global i += 1
      end
```

```
0 is even
1 is odd
```

```
2 is even
3 is odd
```

The *continiue* and *break* statements will either skip an iteration of the current object if it doesn't meet a certain condition, proceeding onto the next element, or it will break the whole loop and finish the evaluation

[19]:
```
while true
    global i += 1
    i > 6 && break
    isodd(i) && continue
    println(i, " is even")
end
```

```
6 is even
```

"Observe that we write while true to set up the loop. Since this condition is always true, unless we have another means to interrupt the loop, it would run infinitely many times. This is exactly what the break keyword achieves."

1.2 Compound expressions

Compound expressions, as the name hints at are several expressioned compounded and wrapped together. We do create them either using begin-end blocks, or using the semicolon ; , the latter being slicker and smaller but sometimes less obvious.

Let's try a begin block - notice that since the begin block is within a conditional, if the first part of the condition is not met then the expression won't be executed

[27]:
```
x = 4
x < 5 && begin
    println(x)
    x += 1
    println(x)
    x * 2
end
```

```
4
5
```

[27]:
```
10
```

The same thing chained using *;*

[29]:
```
x = 4 ; x < 5 ; println(x) ; x +=1 ; println(x) ; x * 2
```

```
4
5
```

[29]:
```
10
```

Here is an example of combining chaining with the ternary operator - pay attention to the parenthesis

```
[34]: x = 2 ; x > 3 ? println(x) : (x += 1 ; println(x * 2))
```

```
6
```

1.3 Creating the function - windsorized mean

Input data - basic array of numbers

```
[1]: inputarray = [12, 13, 11, 9, 444, 213, 33, 120, 330, 129, 78, 300, 55]
```

```
[1]: 13-element Vector{Int64}:
       12
       13
       11
        9
      444
      213
       33
      120
      330
      129
       78
      300
       55
```

Sort the array

```
[3]: sort!(inputarray)
```

```
[3]: 13-element Vector{Int64}:
        9
       11
       12
       13
       33
       55
       78
      120
      129
      213
      300
      330
      444
```

```
[ ]: y = inputarray
```

Replace *k* smallest values by the *k + 1* smallest value in vector y by using a loop. Similarly, replace *k* largest values by the *k + 1* largest value. For example, if *k=3*, and y = 1,2,3,4,6,7,8,9,10,11, we would take the 3 smallest values, namely, 1,2,3, and replace them by 3 + 1 smallest value, this would be 4. On the other end, we would replace 9,10,11 with 12. This would create the vector, 4,4,4,4,6,7,8,8,8,8 giving us a mean of

[8]:
```
using StatsBase
```

[107]:
```
q = collect(1:11)
```

[107]:
```
11-element Vector{Int64}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
 11
```

[26]:
```
mean(q)
```

[26]:
```
6.0
```

[31]:
```
qq = [4,4,4,4,5,6,7,8,8,8,8]
```

[31]:
```
11-element Vector{Int64}:
 4
 4
 4
 4
 5
 6
 7
 8
 8
 8
 8
```

[32]:
```
mean(qq)
```

[32]:
```
6.0
```

[37]:
```
k = 3
```

[37]:
```
3
```

[52]:
```
q[(k+1):(end-4)]
```

[52]:
```
4-element Vector{Int64}:
 4
 5
 6
 7
```

Create the function! - my insane first attempt - look at how many lines of code this is! laughable!

[78]:
```
function winsorised(vector::Vector, k::Int)
    y = sort!(vector)
    veclength = length(y)
    middle_vec = y[(k+1):(end-3)]
    small_k = y[k+1]
    large_k = y[end-(k)]
    newvec = []
    for i in 1:k
        push!(newvec, small_k)
    end
    for i in middle_vec
        push!(newvec, i)
    end
    for i in 1:k
        push!(newvec, large_k)
    end
    mean(newvec)
end
```

[78]:
```
winsorised (generic function with 1 method)
```

[80]:
```
winsorised(inputarray, 3)
```

[80]:
```
101.46153846153847
```

My modified second attempt after having a peep at Bogumils example

[108]:
```
function winzy(y, k)
    for i in 1:k
        y[i] = y[k] + 1
        y[end-i + 1] = y[end-k]
    end
    mean(y)
end
```

[108]:
```
winzy (generic function with 2 methods)
```

[ ]:
```
winzy(q, k)
```

Answer is 6.0 as expected

Key take away is not to rush ahead, to thinking through functions…. and not to be arrogant

1.4 Positional and keyword arguments

When defining functions, we may specify which sort of arguments are required to be provided to the execute the function (if any) - these take two general forms. 1) Positional arguments are those that both provided by the user, and those which cannot change, and keyword arguments, when often have defaults in the function definition, but can also be different based upon the user provided values. In the case of a function which three open positional arguments, the user must provide all three to allow execution. If say, the third argument, had a default value, it would only requre the user to provide at minimum the first two arguments. Remember, since the argument is positional, it cannot be changed IF it has a default value. On the other hand, keyword arguments can have both defaults, and can also be modified by the user from these defaults if they are present. A bit of a tirade no.. let's see some examples

Three positional arguments, no defaults

[110]:
```
function positionalone(x, y, z)
end
```

[110]:
```
positionalone (generic function with 1 method)
```

Three positional arguments, one default which cannot be changed by the user

[111]:
```
function positionaltwo(x, y, z = 10)
end
```

[111]:
```
positionaltwo (generic function with 2 methods)
```

Two positional arguments, and two keyword arguments which need to be provided by the user - Now! notice that after we define the positional arguments, we have to separate the following keyword arguments with a semi-colon. This let's julia know that the next arguments are keyword form.

[112]:
```
function positionalkeywordone(x, y; z, q)
end
```

[112]:
```
positionalkeywordone (generic function with 1 method)
```

Two position arguments, one with a default, and two keyword arguments, both with defaults.

[117]:
```
function  positionalkeywordtwo(x, y = 10 ; z = 2, q = 4)
    x + y + z + q
end
```

[117]:
```
positionalkeywordtwo (generic function with 2 methods)
```

Another thing to remember, we providing default values, they have to be entered at the end of the argument block, so in the positional block, if x = 10 and y was unassigned, we would have to put x after y.

When we call the function, we don't need to provide the name of the position argument to the function, but when calling keyword arguments we need to provide the name followed by the value as per `z = 3`

[123]:
```
positionalkeywordtwo(4; z=3, q=10)
```

[123]:
```
27
```

As Bogumil says "When calling a function that takes both positional and keyword arguments, it is good practice to separate them by using a semicolon (;), just as when you define a function. This is the convention that I use in this book; however, using a comma is also allowed."

1.5 Compact or short syntax for function definitions

Often times we can create a function in a one liner. Imagine we just want a function that performs a simple operation on an argument, such as divinding by 10

[124]:
```
divideten(x::Int) = x / 10
```

[124]:
```
divideten (generic function with 1 method)
```

1.6 Anonymous function

Another compact form of functional programming is anonymous functions - those that don't even have a definition, but often perform certain controls flows such as iteration. We often use the **filter()** function in an anonymous way

Some examples - use map to map each element in an array to the function

[125]:
```
map(x -> x * 2, [1,2,3])
```

[125]:
```
3-element Vector{Int64}:
 2
 4
 6
```

Notice the general structure of the anon function - the iteration is specified whereby x is the iterator, and the elements are provided in the last argument, this time it being an array. The map function will then iterater over the array :)

1.7 Do blocks

"These blocks are used if (1) you use a function that accepts another function as its first positional argument and (2) you want to pass an anonymous function composed of several expressions (so a standard anonymous function definition style is not convenient)." There we go - use the do-end blocks if there are more than one expressions that you would like to evaluate, so a typical anonymous function of the form `x->x,[]` is inadequate. And ofcourse, if the first argument of our function takes another function as input.

[134]:
```
# Sum the values of x*2 for each element in the arry
# 1^2 + 2^2 + 3^3
```

```
sum([1, 2, 3]) do x
    println("processing ", x)
    return x ^ 2
end
```

```
processing 1
processing 2
processing 3
```

[134]:

```
14
```

Aside from printing the value of x at each iteraction, the core of the square root operation is as below

[132]:

```
sum(x -> x^2, [1,2,3])
```

[132]:

```
14
```

Perhaps a filter function may look something like this

```
filter(df) do row
    println("this row")
    row == "somevalue"
end
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: