JuliaTutorialExercises

February 3, 2024

1 Homework 0: Exercises for Julia Tutorial

After you have gone through the tutorial, you should be able to fill in these simple exercises. To submit your reponse, please take the following steps: 1. Make sure to run all the cells (such that we can see the output that you create). 2. Print a pdf of the notebook (make sure that it is easily readable). 3. Upload the pdf to Canvas by February 4.

1.0.1 Exercises

Get started by reading the following statements about your class work and signing that you understand and agree by typing your name at the bottom of this cell.

1. You are encouraged to discuss homework problems with classmates and even work in groups.
2. However, **the work you turn in must be your own**. You must not communicate files containing code or answers to homework questions to each other.
3. Many homeworks require the use of Julia and JuMP. We'll provide instructions to help you install this programming environment on your own computer, together with tutorials and exercises, but ultimate it is your responsibility to ensure that you have a stable platform on which to develop and run Julia code and jupyter notebooks.
4. Submission of all homeworks will be through Canvas. You will usually be required to submit a **pdf printout of your jupyter notebook**, NOT source code.
    a. Please submit the answers **in the same order as on the assignment**
    b. Please denote the start of each question in your notebook using a large font, for example: # Question 1a
5. You can learn to do optimization modeling only by doing it yourself, not by following along what others are doing!
6. **PLAGIARISM AND OTHER TYPES OF ACADEMIC MISCONDUCT IS NOT TOLERATED AND WILL HAVE CONSEQUENCES.** Please read this information about UW's definition of academic misconduct.

1.0 PLEASE ACKNOWLEDGE YOUR UNDERSTANDING AND ACCEPTANCE OF THESE RULES BY TYPING YOUR NAME HERE: Kefan Zheng

1.1 Finding help Look up docs for the function `convert`.

[1]:
```
# search: convert ConcurrencyViolationError code_native @code_native

#   convert(T, x)

#   Convert x to a value of type T.

#   If T is an Integer type, an InexactError will be raised if x is not␣
↪representable by T, for example if x is not integer-valued, or is
#   outside the range supported by T.

#   Examples
#

#   julia> convert(Int, 3.0)
#   3

#   julia> convert(Int, 3.5)
#   ERROR: InexactError: Int64(3.5)
#   Stacktrace:
#   [...]

#   If T is a AbstractFloat type, then it will return the closest value to x␣
↪representable by T.

#   julia> x = 1/3
#   0.3333333333333333

#   julia> convert(Float32, x)
#   0.33333334f0

#   julia> convert(BigFloat, x)
#   0.333333333333333314829616256247390992939472198486328125

#   If T is a collection type and x a collection, the result of convert(T, x)␣
↪may alias all or part of x.

#   julia> x = Int[1, 2, 3];

#   julia> y = convert(Vector{Int}, x);

#   julia> y === x
#   true

#   See also: round, trunc, oftype, reinterpret.
```

1.2 Assigning variables, checking and converting types Assign `365` to a variable named `days`.

[2]:
```
days = 365
```

[2]:
```
365
```

Use the `convert` function to change the variable `days` from an integer to a float, and assign it to a new variable `days_float`.

[3]:
```
days_float = convert(Float64, days)
```

[3]:
```
365.0
```

Check whether the types of `days` and `days_float` are the same.

[4]:
```
typeof(days) === typeof(days_float)
```

[4]:
```
false
```

1.3 Working with arrays and tuples

Define the array

`square = [1, 2, 3]`

and the tuple

`round = (4,5,6)`

[5]:
```
my_square = [1, 2, 3]
my_round = (4, 5, 6)
```

[5]:
```
(4, 5, 6)
```

Access the first element of the array and the tuple, and add them together.

[6]:
```
my_square[1] + my_round[1]
```

[6]:
```
5
```

Change the first element of the array to be equal to the first element of the tuple.

[7]:
```
my_square[1] = my_round[1]
```

[7]:
```
4
```

Try to change the third element of the tuple to be equal to that of the array.

Why will this not work?

[8]:
```
my_round[3] = my_square[3]
# This will not work, because tuple is a fixed-length container that can hold␣
↪any values, but cannot be modified (it is immutable).
```

```
MethodError: no method matching setindex!(::Tuple{Int64, Int64, Int64}, ::Int64␣
↪::Int64)
```

```
Stacktrace:
 [1] top-level scope
   @ In[8]:1
```

1.4 Dictionaries Create a dictionary which lists three of your favorite restaurants and their ranking (1, 2 or 3).

[9]:
```
favorite_restaurants = Dict("Gordon" => 1, "Subway" => 2, "Strada" => 3)
```

[9]:
```
Dict{String, Int64} with 3 entries:
  "Gordon" => 1
  "Strada" => 3
  "Subway" => 2
```

1.5 Matrix (two-dimension arrays) Create the following matrix:

$$B = \begin{bmatrix} 1 & 2 & 1 \\ 3 & 0 & 1 \\ 0 & 2 & 4 \end{bmatrix}$$

[10]:
```
B = [1 2 1; 3 0 1; 0 2 4]
```

[10]:
```
3×3 Matrix{Int64}:
 1  2  1
 3  0  1
 0  2  4
```

Change the first element in the first row of B into 5, and check if it is an even number. (Tip: Use "a%b == c", which means the remainder of a/b is c)

[11]:
```
B[1, 1] = 5
B[1, 1]%2 === 0
```

[11]:
```
false
```

1.6 Some basic math Check if 5 is an even number. (Tip: Use the modulo operation "a%b == c", which means the remainder of a/b is c)

[12]:
```
5 % 2 === 0
```

[12]:
```
false
```

1.7 For loops Write a for loop to print the integers from 1 to 5.

[13]:
```
for i = 1:5
    println(i)
end
```

```
1
2
3
4
5
```

Now write a for loop to go through every element in the above matrix, check if it is odd. If it is, then add 1 to that element. And print your matrix.

[14]:
```
for row in 1:size(B, 1)
    for col in 1:size(B, 2)
        if B[row, col] % 2 === 1
            B[row, col] += 1
        end
    end
end
print(B)
```

```
[6 2 2; 4 0 2; 0 2 4]
```

1.8 Functions Write a function called `my_func` which takes a number as an input, and return an array containing integers from 1 to n. And try your function with input 5.

[15]:
```
function my_func(n)
    res = [i for i = 1:n]
    return res
end

my_func(5)
```

[15]:
```
5-element Vector{Int64}:
 1
 2
 3
 4
 5
```

What happens to the output if you insted use the input 5.5 (which is a non-integer)?

[16]:
```
my_func(5.5)

#=
The difference between the output of my_func(5) and my_func(5.5) is:
The array element type returned by my_func(5) is integer, while the array␣
 ↪element type returned by my_func(5.5) is a floating point type.
Because in Julia, the type of the i variable in a for loop is determined by␣
 ↪the element type of the iterated object. In the range 1:5.5,
5.5 is floating point number. Therefore, i is inferred to be of type floating␣
 ↪point.
```

```
=#
```

[16]:
```
5-element Vector{Float64}:
 1.0
 2.0
 3.0
 4.0
 5.0
```

Now slightly modify your function, and create a new function `odd` which outputs an array of all the odd numbers in $[1,n]$. Test your function with input 7.

[17]:
```
function odd(n)
    res = [i for i = 1:n if i%2 === 1]
    return res
end

odd(7)
```

[17]:
```
4-element Vector{Int64}:
 1
 3
 5
 7
```

[ ]:
```

```

[ ]:
```

```