FINC-672 – WORKSHOP IN FINANCE: EMPIRICAL RESEARCH

Julia Data Structures

Prof. Matt Fleckenstein
University of Delaware

mflecken@udel.edu

Goals

- ☐ Broadcasting Operators and Functions
- ☐ String
- ☐ Tuple, NamedTuple
- ☐ UnitRange,
- ☐ Array
- ☐ Pair, Dict

Broadcasting

- For mathematical operations, like `*` (multiplication) or `+` (addition), we can broadcast it using the dot operator `.`
- For example, broadcasted addition would imply in changing the `+` to `.+`

```
julia> [1, 2, 3] .+ 1
3-element Vector{Int64}:
 2
 3
 4
```

Broadcasting

- It also works with functions automatically.
- Let's use the `logarithm` function.

```
julia> log.([1, 2, 3])
3-element Vector{Float64}:
 0.0
 0.6931471805599453
 1.0986122886681098
```

Comprehensions

- It is often useful to use a *comprehension* as a basic programming construct.
- A typical form of a comprehension is.

```
[f(x) for x in A]
```

- For example

```
julia> [(2n+1)^2 for n in 1:5]
5-element Vector{Int64}:
   9
  25
  49
  81
 121
```

Strings

- **Strings** are represented delimited by double quotes.

```
julia> typeof("This is a string")
String
```

Strings

- We can also write a multiline string.

```
julia> text = "
This is a big multiline string.
As you can see.
It is still a String to Julia.
";

julia> println(text)

This is a big multiline string.
As you can see.
It is still a String to Julia.
```

Strings

- But, it is, typically, more clear to use triple quotation marks.[1]

```
julia> s = """
    This is a big multiline string with a nested "quotation".
    As you can see.
    It is still a String to Julia.
    """;

julia> println(s)
This is a big multiline string with a nested "quotation".
As you can see.
It is still a String to Julia.
```

[1] When using triple-backticks, the indentation and newline at the start is ignored by Julia. This improves code readability because you can indent the block in your source code without those spaces ending up in your string.

String Concatenation

- A common string operation is **string concatenation**.
- Suppose that you want to construct a new string that is the concatenation of two or more strings.
- This is accomplished in julia either with the `*` operator or the `join` function.

```
julia> hello = "Hello";

julia> goodbye = "Goodbye";

julia>
hello * goodbye
"HelloGoodbye"
```

String Concatenation (cont'd)

- As you can see, we are missing a space between `hello` and `goodbye`.
- We could concatenate an additional `" "` string with the `*`, but that would be cumbersome for more than two strings.
- That's when the `join` function is useful.
- We just pass as arguments the strings inside the brackets `[]` and the separator.

```
julia> join([hello, goodbye], " ")
"Hello Goodbye"
```

String Interpolation

- Concatenating strings can be convoluted.
- We can be much more expressive with **string interpolation**.
- It works like this: you specify whatever you want to be included in your string with the dollar sign `$`.
- Here's the example before but now using interpolation.

```
julia> s = "$hello $goodbye"
"Hello Goodbye"
```

String Interpolation (cont'd)

- It works even inside functions. Let's create a function to illustrate the idea.

```
julia> function test_interpolated(a, b)
    if a < b
        "$a is less than $b"
    elseif a > b
        "$a is greater than $b"
    else
        "$a is equal to $b"
    end
end;

julia>
test_interpolated(3.14, 3.14)
"3.14 is equal to 3.14"
```

String Manipulations

- There are several functions to manipulate strings in Julia. We will demonstrate the most common ones.
- Also, note that most of these functions accepts a Regular Expression (RegEx) as arguments.
- We won't cover RegEx in this course, but you are encouraged to learn about them, especially if most of your work uses textual data.
- First, let us define a string for us to work with.

```
julia> julia_string = "Julia is an amazing opensource programming language"
"Julia is an amazing opensource programming language"
```

String Manipulations (cont'd)

- 1. `occursin`, `startswith` and `endswith`: A conditional (returns either `true` or `false`) if the first argument is a.
    - **substring** of the second argument
    - **prefix** of the second argument
    - **suffix** of the second argument

```
julia> occursin("Julia", julia_string)
true
```

```
julia> startswith("Julia", julia_string)
false
```

```
julia> endswith("Julia", julia_string)
false
```

String Manipulations (cont'd)

- 2. `lowercase`, `uppercase`, `titlecase` and `lowercasefirst`.

```
julia> lowercase(julia_string)
"julia is an amazing opensource programming language"

julia> uppercase(julia_string)
"JULIA IS AN AMAZING OPENSOURCE PROGRAMMING LANGUAGE"

julia> titlecase(julia_string)
"Julia Is An Amazing Opensource Programming Language"

julia> lowercasefirst(julia_string)
"julia is an amazing opensource programming language"
```

String Manipulations (cont'd)

- 3. `replace`: introduces a new syntax, called the **`Pair`**
- 4. `split`: breaks up a string by a delimiter.

```
julia> replace(julia_string, "amazing" => "awesome")
"Julia is an awesome opensource programming language"
```

```
julia> split(julia_string, " ")
7-element Vector{SubString{String}}:
 "Julia"
 "is"
 "an"
 "amazing"
 "opensource"
 "programming"
 "language"
```

String Conversions

- Often, we need to convert between types in Julia.
- We can use the `string` function.

```
julia> my_number = 123;

julia> typeof(string(my_number))
String
```

- Sometimes, we want the opposite: convert a string to a number.
- Julia has a handy function for that: `parse`

```
julia> typeof(parse(Int64, "123"))
Int64
```

String Conversions (cont'd)

- Sometimes, we want to play safe with these convertions.
- That's when `tryparse` function steps in.
- It has the same functionality as `parse` but returns either a value of the requested type, or `nothing`.
- That makes `tryparse` handy when we want to avoid errors.
- Of course, you would need to deal with all those `nothing` values afterwards.

```
julia> tryparse(Int64, "A very non-numeric string")
```

Tuples

- Julia has a data structure called **tuple**. They are really *special* in Julia because they are often used in relation to functions.
- A tuple is a **fixed-length container that can hold multiple different types**.
- A tuple is an **imutable object**, meaning that it cannot be modified after instantiation.
- To construct a tuple, use parentheses `()` to delimitate the beginning and end, along with commas `,` as value's delimiters.

Tuples (cont'd)

```
julia> my_tuple = (1, 3.14, "Julia")
(1, 3.14, "Julia")
```

- Here, we are creating a tuple with three values.
- Each one of the values is a different type. We can access them via indexing.

```
julia> my_tuple[2]
3.14
```

- We can also loop over tuples with the `for` keyword. And even apply functions to tuples.
- But we can **never change any value of a tuple** since they are **immutable**.

Tuples (cont'd)

- Recall that functions can return multiple values.
- Let's inspect what our `add_multiply` function returns.

```
julia> function add_multiply(x, y)
    addition = x + y
    multiplication = x * y
    return addition, multiplication
end;

julia>
return_multiple = add_multiply(1, 2)
(3, 2)

julia> typeof(return_multiple)
Tuple{Int64, Int64}
```

- This is because `return a, b` is the same as `return (a, b)`.

Tuples (cont'd)

- One more thing about tuples.
- When you want to pass more than one variable to an anonymous function, what would you need to use?
- Answer: tuples

```
julia> map((x, y) -> x^y, 2, 3)
8
```

- Or, even more than two arguments.

```
julia> map((x, y, z) -> x^y + z, 2, 3, 1)
9
```

Named Tuple

- Sometimes, you want to name the values in tuples. That's when **named tuples** comes in.
- Their functionality is pretty much same the same as tuples. they are **immutable** and can hold **any type of value**.
- Named tuple's construction are slightly different from tuples.
- You have the familiar parentheses `()` and comma `,` value separator.
- But now you **name the values**.

```
julia> my_namedtuple = (i=1, f=3.14, s="Julia")
(i = 1, f = 3.14, s = "Julia")
```

Named Tuple (cont'd)

- We can access a named tuple's values via indexing like regular tuples or, alternatively, **access by their names** with `.`

```
julia> my_namedtuple.s
"Julia"
```

Named Tuple (cont'd)

- Often Julia users create a named tuple by using the familiar parenthesis `()` and commas `,`, but without naming the values.
- To do so you begin the named tuple construction by specifying first a semicolon `;` before the values.
- This is especially useful when the values that would compose the named tuple are already defined in variables or when you want to avoid long lines.

```
julia> i = 1;

julia> f = 3.14;

julia> s = "Julia";

julia>
my_quick_namedtuple = (; i, f, s)
(i = 1, f = 3.14, s = "Julia")
```

Ranges

- A range in Julia represents an interval between a start and stop boundaries.
- The syntax is `start:stop`

```
julia> 1:10
1:10
```

- As you can see, our instantiated range is of type `UnitRange{T}` where `T` is the type inside the `UnitRange`.

```
julia> typeof(1:10)
UnitRange{Int64}
```

Ranges (cont'd)

- And, if we gather all the values, we get.

```
julia> [x for x in 1:10]
10-element Vector{Int64}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10
```

Ranges (cont'd)

- We can construct ranges also for other types:

```
julia> typeof(1.0:10.0)
StepRangeLen{Float64, Base.TwicePrecision{Float64}, Base.TwicePrecision{Float64}}
```

- Sometimes, we want to change the default interval stepsize behavior.
- We can do that by adding a stepsize in the range syntax `start:step:stop`.
- For example, suppose we want a range of `Float64` from 0 to 1 with steps of size 0.2.

```
julia> 0.0:0.2:1.0
0.0:0.2:1.0
```

- If you want to "materialize" a `UnitRange` into a collection, you can use the function `collect`.
- We have an array of the type specified in the `UnitRange` between the boundaries that we've set.

Arrays

- Arrays are a **systematic arrangement of similar objects**, usually in *rows* and *columns*.
- Let's start with arrays types. There are several, but we will focus on two.
  - `Vector{T}`: **one-dimensional** array. Alias for `Array{T, 1}`.
  - `Matrix{T}`: **two-dimensional** array. Alias for `Array{T, 2}`.[2]
- For example, `Vector{Int64}` is a `Vector` which all elements are `Int64`s and `Matrix{AbstractFloat}` is a `Matrix` which all elements are subtypes of `AbstractFloat`.
- Most of the time, especially when dealing with tabular data, we are using either one- or two-dimensional arrays.
- We can use the aliases `Vector` and `Matrix` for clear and concise syntax.

[2] Note here that `T` is the type of the underlying array.

Array Construction

- How do we construct an array? The simplest answer is to use the *default constructor*.
- It accepts the element type as the type parameter inside the `{}` brackets and inside the constructor you pass the element type followed by the dimensions.
- It is common to initialize vector and matrices with undefined elements by using the `undef` argument for type.

Array Construction (cont'd)

- For example, a vector of 10 `undef` `Float64` elements can be constructed as.

```
julia> my_vector = Vector{Float64}(undef, 10)
10-element Vector{Float64}:
 2.379519675e-315
 2.37951999e-315
 1.675265855e-315
 2.379520307e-315
 2.379520465e-315
 2.37952078e-315
 2.37952094e-315
 2.3795211e-315
 2.33538167e-315
 0.0
```

Array Construction (cont'd)

- For matrices, we need to pass two dimensions arguments inside the constructor: one for **rows** and another for **columns**.
- For example, a matrix with 10 rows, 2 columns is instantiated as.

```
julia> my_matrix = Matrix{Float64}(undef, 10, 2)
10×2 Matrix{Float64}:
 2.29504e-315  1.19738e-315
 2.29504e-315  2.29505e-315
 2.29504e-315  1.19739e-315
 2.29504e-315  1.19738e-315
 2.29504e-315  1.19739e-315
 2.29504e-315  1.19739e-315
 2.29504e-315  2.29505e-315
 2.29504e-315  2.29505e-315
 2.29504e-315  2.29505e-315
 1.19739e-315  1.1974e-315
```

Array Construction (cont'd)

- We also have some **syntax aliases** for the most common elements in array construction.
- `zeros` for all elements being initialized to value zero.

```
julia> my_vector_zeros = zeros(10)
10-element Vector{Float64}:
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
 0.0
```

Array Construction (cont'd)

```
julia> my_matrix_zeros = zeros(Int64, 10, 2)
10×2 Matrix{Int64}:
 0  0
 0  0
 0  0
 0  0
 0  0
 0  0
 0  0
 0  0
 0  0
 0  0
```

Array Construction (cont'd)

- `ones` for all elements being initialized to value one.

```
julia> my_vector_ones = ones(Int64, 10)
10-element Vector{Int64}:
 1
 1
 1
 1
 1
 1
 1
 1
 1
 1
```

Array Construction (cont'd)

```
julia> my_matrix_ones = ones(10, 2)
10×2 Matrix{Float64}:
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
 1.0  1.0
```

Array Construction (cont'd)

- For other elements we can first intantiate an array with `undef` elements and use the `fill!` function to fill all elements of an array with the desired element.
- Here's an example with `3.14` (π).

```
julia> my_matrix_π = Matrix{Float64}(undef, 2, 2);

julia> fill!(my_matrix_π, 3.14)
2×2 Matrix{Float64}:
 3.14  3.14
 3.14  3.14
```

Array Construction (cont'd)

- We can also create arrays with `arrays literals`.
- For example a 2x2 matrix of integers.

```
julia> [[1 2]
[3 4]]
2×2 Matrix{Int64}:
 1  2
 3  4
```

Array Construction (cont'd)

- Array literals also accept a type specification before the `[]` brackets.
- So, if we want the same 2x2 array as before but now as floats, we can do so.

```
julia> Float64[[1 2]
        [3 4]]
2×2 Matrix{Float64}:
 1.0  2.0
 3.0  4.0
```

- It also works for vectors.

```
julia> Bool[0, 1, 0, 1]
4-element Vector{Bool}:
 0
 1
 0
 1
```

Array Construction (cont'd)

- You can even **mix and match** array literals with the constructors.

```
julia> [ones(Int, 2, 2) zeros(Int, 2, 2)]
2×4 Matrix{Int64}:
 1  1  0  0
 1  1  0  0
```

```
julia> [zeros(Int, 2, 2)
 ones(Int, 2, 2)]
4×2 Matrix{Int64}:
 0  0
 0  0
 1  1
 1  1
```

Array Construction (cont'd)

```
julia> [ones(Int, 2, 2) [1; 2]
 [3 4]             5]
3×3 Matrix{Int64}:
 1  1  1
 1  1  2
 3  4  5
```

Array Comprehensions

- Another powerful way to create arrays are **array comprehensions**.
- You specify what you want to do inside the `[]` brackets.
- For example, say we want to create a vector of squares from 1 to 100.

```
julia> [x^2 for x in 1:10]
10-element Vector{Int64}:
   1
   4
   9
  16
  25
  36
  49
  64
  81
 100
```

Array Comprehensions (cont'd)

- They also support multiple inputs.

```
julia> [x*y for x in 1:7 for y in 1:2]
14-element Vector{Int64}:
  1
  2
  2
  4
  3
  6
  4
  8
  5
 10
  6
 12
  7
 14
```

Array Comprehensions (cont'd)

- And conditionals.

```
julia> [x^2 for x in 1:10 if isodd(x)]
5-element Vector{Int64}:
  1
  9
 25
 49
 81
```

Array Comprehensions (cont'd)

- As with array literals you can specify your desired type before the `[]` brackets.

```
julia> Float64[x^2 for x in 1:10 if isodd(x)]
5-element Vector{Float64}:
  1.0
  9.0
 25.0
 49.0
 81.0
```

Array Concatenation

- Finally, we can also create arrays with **concatenation functions**.
- `cat`: concatenate input arrays along a specific dimension `dims`

```
julia> cat(ones(2), zeros(2), dims=1)
4-element Vector{Float64}:
 1.0
 1.0
 0.0
 0.0

julia> cat(ones(2), zeros(2), dims=2)
2×2 Matrix{Float64}:
 1.0  0.0
 1.0  0.0
```

Array Concatenation (cont'd)

- `vcat`: vertical concatenation, a shorthand for `cat(...; dims=1)`

```
julia> vcat(ones(2), zeros(2))
4-element Vector{Float64}:
 1.0
 1.0
 0.0
 0.0
```

Array Concatenation (cont'd)

- `hcat`: horizontal concatenation, a shorthand for `cat(...; dims=2)`

```
julia> hcat(ones(2), zeros(2))
2×2 Matrix{Float64}:
 1.0  0.0
 1.0  0.0
```

Array Inspection

- Once we have arrays, the next logical step is to inspect them.
- There are a lot of handy functions that allows the user to have an inner insight into any array.
- It is most useful to know what *type of elements* are inside an array.
- We can do this with `eltype`:

```
julia> eltype(my_matrix_π)
Float64
```

Array Inspection (cont'd)

- `size` is a little tricky.
- By default it will return a tuple containing the array's dimensions.

```
julia> size(my_matrix_π)
(2, 2)
```

- You can get a specific dimension with a second argument to `size`

```
julia> size(my_matrix_π, 2) # columns
2
```

Array Indexing and Slicing

- Sometimes we want to only inspect certain parts of an array.
- This is called **indexing** and **slicing**.
- If you want a particular observation of a vector, or a row or column of a matrix; you'll probably need to *index an array*.
- First, let's create an example vector and matrix.

```
julia> my_example_vector = [1, 2, 3, 4, 5];

julia>
my_example_matrix = [[1 2 3]
                     [4 5 6]
                     [7 8 9]];
```

Array Indexing and Slicing (cont'd)

- Let's see first an example with vectors.
- Suppose you want the second element of a vector.
- You append `[]` brackets with the desired **index** inside.

```
julia> my_example_vector[2]
2
```

Array Indexing and Slicing (cont'd)

- The same syntax follows with matrices.
- But, since matrices are 2-dimensional arrays, we have to specify **both** rows and columns.
- Let's retrieve the element from the second row (first dimension) and first column (second dimension).

```
julia> my_example_matrix[2, 1]
4
```

Array Indexing and Slicing (cont'd)

- Julia also have conventional keywords for the first and last elements of an array: `begin` and `end`.
- For example, the second to last element of a vector can be retrieved as.

```
julia> my_example_vector[end-1]
4
```

- It also works for matrices.
- Let's retrieve the element of the last row and second column.

```
julia> my_example_matrix[end, begin+1]
8
```

Array Indexing and Slicing (cont'd)

- Often, we are not only interested in just one array element, but in a whole **subset of array elements**.
- We can accomplish this by **slicing** an array.
- It uses the same index syntax, but with the added colon `:` to denote the boundaries that we are slicing through the array.
- For example, suppose we want to get the 2nd to 4th element of a vector.

```
julia> my_example_vector[2:4]
3-element Vector{Int64}:
 2
 3
 4
```

Array Indexing and Slicing (cont'd)

- We could do the same with matrices.
- Particularly with matrices if we want to select all elements in a following dimension we can do so with just a colon `:`.
- For example, all elements in the second row.

```
julia> my_example_matrix[2, :]
3-element Vector{Int64}:
 4
 5
 6
```

Array Indexing and Slicing (cont'd)

- You can interpret this with something like "take 2nd row and all columns".
- It also supports `begin` and `end`.

```
julia> my_example_matrix[begin+1:end, end]
2-element Vector{Int64}:
 6
 9
```

Array Manipulation

- There are several ways we could manipulate an array.
- The first would be to manipulate a **singular element** of the array.
- We just index the array by the element and proceed with an assignment =

```
julia> my_example_matrix[2, 2] = 42;

julia> my_example_matrix
3×3 Matrix{Int64}:
 1   2  3
 4  42  6
 7   8  9
```

Array Manipulation

- Or you can manipulate a certain **subset** of elements of the array.
- In this case, we need to slice the array and then assign with `=`

```
julia> my_example_matrix[3, :] = [17, 16, 15];

julia> my_example_matrix
3×3 Matrix{Int64}:
  1   2   3
  4  42   6
 17  16  15
```

Array Manipulation (cont'd)

- Note that we had to assign a vector because we our sliced array is of type **`Vector`**.

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

Array Manipulation (cont'd)

- The second way we could manipulate an array is to alter its shape.
- Suppose you have a 6-element vector and you want to make it a 3x2 matrix.
- You can do so with `reshape`, by using the array as first argument and a tuple of dimensions as second argument.

```
julia> six_vector = [1, 2, 3, 4, 5, 6];

julia> tree_two_matrix = reshape(six_vector, (3, 2));

julia> tree_two_matrix
3×2 Matrix{Int64}:
 1  4
 2  5
 3  6
```

Array Manipulation (cont'd)

- You can do the reverse, convert it back to a vector, by specifying a tuple with only one dimension as second argument.

```
julia> reshape(tree_two_matrix, (6, ))
6-element Vector{Int64}:
 1
 2
 3
 4
 5
 6
```

Array Manipulation (cont'd)

- The third way we could manipulate an array is to `apply a function over every array element.`
- This is where the familiar broadcasting "dot" operator `.` comes in.

```
julia> log.(my_example_matrix)
3×3 Matrix{Float64}:
 0.0      0.693147  1.09861
 1.38629  3.73767   1.79176
 2.83321  2.77259   2.70805
```

Array Manipulation (cont'd)

- We can broadcast operators.

```
julia> my_example_matrix .+ 100
3×3 Matrix{Int64}:
 101  102  103
 104  142  106
 117  116  115
```

Array Manipulation (cont'd)

- We can use also `map` to apply a function to every element of an array.

```
julia> map(log, my_example_matrix)
3×3 Matrix{Float64}:
 0.0      0.693147  1.09861
 1.38629  3.73767   1.79176
 2.83321  2.77259   2.70805
```

Array Manipulation (cont'd)

- It also accepts an anonymous function.

```
julia> map(x -> x*3, my_example_matrix)
3×3 Matrix{Int64}:
  3    6   9
 12  126  18
 51   48  45
```

Array Manipulation (cont'd)

- It also works with slicing.

```
julia> map(x -> x + 100, my_example_matrix[:, 3])
3-element Vector{Int64}:
 103
 106
 115
```

Array Manipulation (cont'd)

- Finally, sometimes, and specially when dealing with tabular data, we want to apply a *function over all elements in a specific array dimension*.
- This can be done with the `mapslices` function.
- Similar to `map`, the first argument is the function and the second argument is the array.
- The only change is that we need to specify the `dims` argument to flag what dimension we want to transform the elements.
- For example let's use `mapslice` with the `sum` function on both rows (`dims=1`) and columns (`dims=2`).

Array Manipulation (cont'd)

```
julia> mapslices(sum, my_example_matrix; dims=1)
1×3 Matrix{Int64}:
 22  60  24

julia> mapslices(sum, my_example_matrix; dims=2)
3×1 Matrix{Int64}:
  6
 52
 48
```

Array Iteration

- One common operation is to iterate over an array with a `for` loop.
- The regular `for` loop over an array returns each element.

```
julia> simple_vector = [1, 2, 3];

julia> empty_vector = Int64[];

julia>
for i in simple_vector
    push!(empty_vector, i + 1);
end

julia>
empty_vector
3-element Vector{Int64}:
 2
 3
 4
```

Array Iteration (cont'd)

- Sometimes you don't want to loop over each element, but actually over each array index.
- We can the `eachindex` function combined with a `for` loop to iterate over each array index.
- Let's look at an example on the next slide.

Array Iteration (cont'd)

```
julia> forty_two_vector = [42, 42, 42];

julia> empty_vector = Int64[];

julia>
for i in eachindex(forty_two_vector)
    push!(empty_vector, i)
end

julia>
empty_vector
3-element Vector{Int64}:
 1
 2
 3
```

Array Iteration (cont'd)

- In the last example the `eachindex(forty_two_vector)` iterator inside the `for` loop returns not `forty_two_vector`'s values but its indices: `[1, 2, 3]`.

Array Iteration (cont'd)

- Iterating over matrices involves more details.
- The standard `for` loop goes first over columns then over rows.
- It will first traverse all elements in column 1, from the first row to the last row, then it will move to column 2 in a similar fashion until it has covered all columns.[3]
- Let's show this in an example.

[3] Those familiar with other programming languages, Julia, like most scientific programming languages, is "column-major". This means that arrays are stored contiguously using a column orientation.

Array Iteration (cont'd)

```
julia> column_major = [[1 2]
                       [3 4]];

julia>
row_major = [[1 3]
             [2 4]];

julia>
empty_vector = Int64[];
```

- Example continues on next slide.

Array Iteration (cont'd)

```
julia> for i in column_major
    push!(empty_vector, i)
end;

julia>
empty_vector
4-element Vector{Int64}:
 1
 3
 2
 4
```

Array Iteration (cont'd)

```
julia> empty_vector = Int64[];

julia>
for i in row_major
    push!(empty_vector, i)
end;

julia>
empty_vector
4-element Vector{Int64}:
 1
 2
 3
 4
```

Array Iteration (cont'd)

- There are some handy functions to iterate over matrices.
- `eachcol`: iterates over an array column first
- `eachrow`: iterates over an array row first

```
julia> first(eachcol(column_major))
2-element view(::Matrix{Int64}, :, 1) with eltype Int64:
 1
 3

julia> first(eachrow(column_major))
2-element view(::Matrix{Int64}, 1, :) with eltype Int64:
 1
 2
```

Pairs

- `Pair` is a data structure that holds two types.
- How we construct a pair in Julia is using the following syntax.

```
julia> my_pair = Pair("Julia", 42)
"Julia" => 42
```

- Alternatively, we can create a pair by specifying both values and in between we use the pair ‘=>‘ operator.

```
julia> my_pair = "Julia" => 42
"Julia" => 42
```

Pairs (cont'd)

- The elements are stored in the fields `first` and `second`.

```
julia> my_pair.first
"Julia"
```

```
julia> my_pair.second
42
```

Dict

- `Dict` in Julia is just a "hash table" with pairs of `key` and `value`.
- `keys` and `values` can be of any type, but generally you'll see `keys` as strings.
- There are two ways to construct `Dict`s in Julia.
- The first is using the default constructor `Dict` and passing a vector of tuples composed of `(key, value)`.

```julia
julia> my_dict = Dict([("one", 1), ("two", 2)])
Dict{String, Int64} with 2 entries:
  "two" => 2
  "one" => 1
```

Dict (cont'd)

- The second way of constructing `Dict`s is more elegant because it has a more expressive syntax.
- You use the same default constructor `Dict`, but now you pass pairs of `key` and `value`.

```
julia> my_dict = Dict("one" => 1, "two" => 2)
Dict{String, Int64} with 2 entries:
  "two" => 2
  "one" => 1
```

- You can retrieve a `Dict`s `value` by indexing it by the corresponding `key`.

```
julia> my_dict["one"]
1
```

Dict (cont'd)

- Similarly, to add a new entry you index the `Dict` by the desired `key` and assign a `value` with the assignment `=` operator.

```
julia> my_dict["three"] = 3
3
```

- If you want to check if a `Dict` has a certain `key` you can use the `haskey` function.

```
julia> haskey(my_dict, "two")
true
```

Dict (cont'd)

- To delete a `key` you can use either the `delete!` function.

```
julia> delete!(my_dict, "three")
Dict{String, Int64} with 2 entries:
  "two" => 2
  "one" => 1
```

- Or to delete a `key` while retuning its `value` you can use the `pop!` function.

```
julia> popped_value = pop!(my_dict, "two")
2
```

Dict (cont'd)

- Now our `my_dict` has only one `key`.

```
julia> length(my_dict)
1

julia> my_dict
Dict{String, Int64} with 1 entry:
  "one" => 1
```

Dict (cont'd)

- There is one useful `Dict` constructor that we use a lot.
- Suppose you have two vectors and you want to construct a `Dict` with one of them as `keys` and the other as `values`.
- You can do that with the `zip` function which "glues" together two objects just like a zipper.

```
julia> A = ["one", "two", "three"];

julia> B = [1, 2, 3];

julia>
dic = Dict(zip(A, B))
Dict{String, Int64} with 3 entries:
  "two"   => 2
  "one"   => 1
  "three" => 3
```

Dict (cont'd)

- For instance, we can now get the number 3 via.

```
julia> dic["three"]
3
```

Splat Operator

- In Julia we have the "splat" operator `...` which is mainly used in function calls as a **sequence of arguments**.
- The most intuitive way to learn about splatting is with an example.
- The `add_elements` function below takes three arguments to be adeed together.

```
julia> add_elements(a, b, c) = a + b + c;
```

Splat Operator (cont'd)

- Now suppose that I have a collection with three elements.
- The naïve way to this would be to supply the function with all three elements as function arguments like this.

```
julia> my_collection = [1, 2, 3];

julia>
add_elements(my_collection[1], my_collection[2], my_collection[3])
6
```

Splat Operator (cont'd)

- Here is where we use the "splat" operator `...` which takes a collection (often an array, vector, tuple or range) and converts into a sequence of arguments.

```julia
julia> add_elements(my_collection...) # and splat!
6
```

- The `...` is included after the collection that we want to "splat" into a sequence of arguments.
- In the example above, syntactically speaking, the following are the same.

```julia
collection = [x, y, z]

function(collection...) = function(x, y, z)
```

Splat Operator (cont'd)

- Anytime Julia sees a splatting operator inside a function call, it will be converted on a sequence of arguments for all elements of the collection separated by commas.
- It also works for ranges.

```
julia> add_elements(1:3...) # and splat!
6
```

Wrap-Up

- ☑ Broadcasting Operators and Functions
- ☑ String
- ☑ Tuple, NamedTuple
- ☑ UnitRange,
- ☑ Array
- ☑ Pair, Dict