FINC-672 – WORKSHOP IN FINANCE: EMPIRICAL RESEARCH

Julia Data Structures IV

Prof. Matt Fleckenstein
University of Delaware

mflecken@udel.edu

Goals

- □ Pairs
- □ Dicts
- □ Dates

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> 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> 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.

```
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 to 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
```

Dates

- To work with dates in Julia, we import the `Dates` module from the Julia standard library.

```
julia> using Dates
```

- The `Dates` standard library module has two types for working with dates:
  - `Date`: representing time in days; and
  - `DateTime`: representing time in millisecond precision.
- We construct `Date` and `DateTime` with the default constructor by specifying an integer to represent year, month, day, hours and so on.
- Let's do a few examples.

Dates (cont'd)

```
julia> Date(1987) # year
1987-01-01

julia> Date(1987, 9) # month
1987-09-01

julia> Date(1987, 9, 13) # day
1987-09-13

julia> DateTime(1987, 9, 13, 21) # hour
1987-09-13T21:00:00

julia> DateTime(1987, 9, 13, 21, 21) # minute
1987-09-13T21:21:00
```

Dates (cont'd)

- In working with dates, it is useful to be able to use **Periods**.
- Julia defines the following types that we will use often in working with financial data.

```
julia> subtypes(DatePeriod)
Error: UndefVarError: subtypes not defined
```

Dates (cont'd)

- Next, we need to discuss *Parsing Dates.*
- This just means that when we are given a dataset where dates are written in a specific format (e.g. "20210132" or "01-31-2022"), we need to tell Julia how to interpret these date formats.
- Let's consider an example where our dataset has a date written as 20210131. How can we tell Julia that this number refers to January 31, 2021?

```
julia> Date("20210131","yyyymmdd")
2021-01-31
```

- We just use the `Date` constructor, and specify the date format as `"yyyymmdd`.
- Here, `yyyy` represents the year (i.e. 2021).
- `mm` represents the month (i.e. 01).
- `dd` represents the day (i.e. 31).

Dates (cont'd)

- We now know how to construct `Dates` in Julia.
- Next, we want to extract information such as the *year*, *month*, *day*, *weekday* etc. from a given `Date`.
- To illustrate some useful functions that Julia provides, let's suppose we have a Treasury bond with maturity date on May 15, 2025.

```
julia> maturityDate = Date("20250515",dateformat"yyyymmdd")
2025-05-15

julia> year(maturityDate)
2025

julia> month(maturityDate)
5

julia> day(maturityDate)
15
```

Dates (cont'd)

- We can also see the day of the week and other handy stuff.

```
julia> dayofweek(maturityDate)
4

julia> dayname(maturityDate)
"Thursday"
```

Dates (cont'd)

- We can perform operations in Dates instances.
- For example, we can add days to a `Date` or `DateTime` instance.
- Julias `Dates` will automatically perform the adjustments necessary for leap years, and for months with 30 or 31 days.

```
julia> maturityDate + Day(90)
2025-08-13

julia> maturityDate + Day(90) + Month(2) + Year(1)
2026-10-13
```

Dates (cont'd)

- To get **date duration**, we just use the subtraction `-` operator.
- To count the number of days between today and the maturity date of the bond, we can use the `today` function.

```
julia> maturityDate - today()
1325 days
```

Dates (cont'd)

- The last example, introduced the concept of **Date Intervals**.
- We can also easily construct date and time intervals.
- Suppose you want to create a `Day` interval. We do this with the colon `:` operator.

```
julia> Date("2022-01-01"):Day(1):Date("2022-01-07")
Dates.Date("2022-01-01"):Dates.Day(1):Dates.Date("2022-01-07")
```

Dates (cont'd)

- There is nothing special in using `Day(1)` as the interval.
- We can use whatever `Period` type as interval.
- For example, using 3 days as the interval

```
julia> Date("2022-01-01"):Day(3):Date("2022-01-07")
Dates.Date("2022-01-01"):Dates.Day(3):Dates.Date("2022-01-07")
```

Dates (cont'd)

- Months work just as well.

```
julia> Date("2021-01-01"):Month(1):Date("2021-03-01")
Dates.Date("2021-01-01"):Dates.Month(1):Dates.Date("2021-03-01")
```

Dates (cont'd)

- Note that in the previous examples, we created a `range` (actually a `StepRange`).
- We can convert this to a `vector` with the `collect` function.

```
julia> rng = Date("2021-01-01"):Month(1):Date("2021-03-01");

julia> vect = collect(rng)
3-element Vector{Dates.Date}:
 2021-01-01
 2021-02-01
 2021-03-01
```

Dates (cont'd)

- After we have materialized the `range` to a `Vector`, we have all the array functionalities available. For example, indexing:

```
julia> vect[end]
2021-03-01
```

- We can also broadcast date operations to our vector of `Dates`.
- We already know that we do this by using the dot-operator `.`

```
julia> vect .+ Day(10)
3-element Vector{Dates.Date}:
 2021-01-11
 2021-02-11
 2021-03-11
```

Wrap-Up

☑ Pairs
☑ Dicts
☑ Dates