6c. Chaining Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

This section introduces two approaches to computing outputs that involve multiple intermediate steps. First, we introduce the so-called **let blocks**, which create a new scope that returns the last line as its output. Let blocks offer a concise way to wrap a sequence of operations, making them similar to functions but with less syntactic clutter. They also help maintain a tidy namespace, as all intermediate variables will be local and therefore inaccessible outside the block.

The second approach is based on **pipes**, which chain a series of operations and return the final output as result. As the built-in pipe can become unwieldy beyond single-argument functions, we also present an alternative based on the `Pipe` package.

LET BLOCKS

Let blocks are particularly helpful when we need to perform a series of operations, but only care about the ultimate result. To illustrate their utility, suppose we want to compute the rounded logarithm of `a`'s absolute value, formally expressed as $\text{round}\left(\ln\left(|a|\right)\right)$.

In Julia, this operation can be implemented using the expression `round(log(abs(a)))`, where `round(a)` returns the nearest integer to `a`. However, the readability of this expression is less than ideal due to the multiple parentheses, with the issue potentially exacerbated if variables and functions had long names.

To improve clarity, we could break the whole operation into multiple smaller steps: *i)* compute the absolute value of `a`, *ii)* compute the logarithm of the result, and *iii)* round the resulting output. An easy way to implement this is to create three intermediate variables to store the output in each step. Nonetheless, this approach would clutter our namespace and potentially obscure the nested nature of the operations.

A more elegant solution is to introduce a **let-block**, which resembles functions in several respects. This construct creates a new scope delimited by the `let` and `end` keywords, enabling multiple calculations to be performed within it. The result of the last calculation is then returned as the output of the let-block. Like functions, let-blocks also allow arguments to be passed by specifying them after the `let` keyword.

To highlight the benefits of let-blocks, the following examples compare various approaches to computing `round(log(abs(a)))`.

```
a      = -2

output = round(log(abs(a)))
```

```
julia> output
1.0
julia> temp1
julia> temp2
4
```

```
a      = -2

temp1  = abs(a)
temp2  = log(temp1)
output = round(temp2)
```

```
julia> output
1.0
```

```
a      = -2

output = let b = a          # 'b' is a local variable having the value of 'a'
   temp1 = abs(b)
   temp2 = log(temp1)
   round(temp2)
end
```

```
julia> output
1.0
julia> temp1 #local to let-block
ERROR: UndefVarError: `temp1` not defined
julia> temp2 #local to let-block
ERROR: UndefVarError: `temp2` not defined
```

```
a      = -2

output = let a = a          # the 'a' on the left of `=` defines a local variable
   temp1 = abs(a)
   temp2 = log(temp1)
   round(temp2)
end
```

```
julia> output
1.0
julia> temp1 #local to let-block
ERROR: UndefVarError: `temp1` not defined
julia> temp2 #local to let-block
ERROR: UndefVarError: `temp2` not defined
```

> **Let Blocks Can Mutate Variables**

Let blocks behave like functions regarding assignments and mutation. This means that you can mutate their arguments, but can't reassign variables.

```
x = [2,2,2]

output = let x = x
   x[1] = 0
end
```

```
julia> x
3-element Vector{Int64}:
 0
 2
 2
```

```
x = [2,2,2]

output = let x = x
   x = 0
end
```

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

Since mutations are possible within let-blocks, exercise caution to prevent unintended side effects in the global scope.

PIPES

Pipes provide an alternative to let-blocks for operations with multiple intermediate steps. Unlike let-blocks, they're specifically designed to chain operations together, with each step taking the output of the previous step as its input. These steps are separated through the `|>` keyword.

Pipes are particularly well-suited for sequential applications of single-argument functions. To illustrate this, let's revisit the example presented above for let blocks.

```
a      = -2

output = round(log(abs(a)))
```

```
julia> output
1.0
```

```
a      = -2

output = a |> abs |> log |> round
```

```
julia> output
1.0
```

Let Blocks and Pipes For Long Names

Both approaches facilitate the creation of temporary aliases for variables with lengthy names. In this way, users can assign meaningful names to variables, while preserving code readability.

```
variable_with_a_long_name = 2

output          =          variable_with_a_long_name          -
log(variable_with_a_long_name) / abs(variable_with_a_long_name)
```

```
julia> output
1.6534264097200273
```

```
variable_with_a_long_name = 2

temp   = variable_with_a_long_name
output = temp - log(temp) / abs(temp)
```

```
julia> output
1.6534264097200273
```

```
variable_with_a_long_name = 2

output = variable_with_a_long_name  |>
         a -> a - log(a) / abs(a)
```

```
julia> output
1.6534264097200273
```

```
variable_with_a_long_name = 2

output = let x = variable_with_a_long_name
    x - log(x) / abs(x)
end
```

```
julia> output
1.6534264097200273
```

BROADCASTING PIPES

Just like any other operator, pipes can be broadcasted by prefixing them with a dot `.`. Thus, `.|>` indicates that the subsequent operation must be applied element-wise to the preceding output. For example, the expression `x .|> abs` is equivalent to `abs.(x)`.

To demonstrate its use, suppose we want to transform `x` by taking the logarithm of its absolute values, and then sum the results.

```
x      = [-1,2,3]

output = sum(log.(abs.(x)))
```

```
julia> output
1.791759469228055
```

```
x      = [-1,2,3]

temp1  = abs.(x)
temp2  = log.(temp1)
output = sum(temp2)
```

```
julia> output
1.791759469228055
```

```
x      = [-1,2,3]

output = x .|> abs .|> log |> sum
```

```
julia> output
1.791759469228055
```

PIPES WITH MORE COMPLEX OPERATIONS

Our examples of pipes so far have followed a simple pattern, with each step consisting of a single-argument function. However, pipes applied in the current form preclude their application to multiple-argument functions or even operations. For example, it prevents the incorporation of expressions like `foo(x,y)` or `2 * x`.

To incorporate such cases, we can **combine pipes with anonymous functions**. In this way, the user can specify how the output of the previous step is incorporated into the subsequent operation. As shown below, the technique greatly expands the utility of pipes.

```
a      = -2

output = round(2 * abs(a))
```

```
a      = -2

temp1  = abs(a)
temp2  = 2 * temp1
output = round(temp2)
```

```
a      = -2

output = a |> abs |> (x -> 2 * x) |> round

#equivalent and more readable
output = a              |>
         abs            |>
         x -> 2 * x     |>
         round
```

PACKAGE PIPE

Combining pipes and anonymous functions can result in cumbersome code, defeating the very own purpose of using pipes to write clean and readable code.

The `Pipe` package provides a convenient solution, eliminating the need for anonymous functions. By prefixing the operation chain with the `@pipe` macro, you can reference the output of the previous step by the symbol `_`. Furthermore, for simple operations that don't require anonymous functions, and therefore don't need `_`, `@pipe` has the same syntax as built-in pipes.

To demonstrate its use, we revisit the last example.

```
#
a      = -2

output = a |> abs |> (x -> 2 * x) |> round

#equivalent and more readable
output =       a            |>
               abs          |>
               x -> 2 * x   |>
               round
```

```
using Pipe
a = -2

output = @pipe a |> abs |> 2 * _ |> round

#equivalent and more readable
output = @pipe a            |>
               abs          |>
               2 * _        |>
               round
```

FUNCTION COMPOSITION (*OPTIONAL*)

An alternative approach to nest functions is through the composition operator `∘`. This symbol can be inserted by tab completion through `\circ`, and its functionality is the same as in Mathematics. Specifically, for some functions `f` and `g`, `(f ∘ g)(x)` is equivalent to `f(g(x))`.

The operator `∘` can be considered as an alternative to piping, as it provides the same output as `x |> f |> g`. Moreover, `∘` is also available as a function, where `∘(f,g)(x)` is equivalent to `(f ∘ g)(x)`. The following examples demonstrate its use.

```
a        = -1



# all `output` are equivalent
output   = log(abs(a))
output   = a |> abs |> log
output   = (log ∘ abs)(a)
output   = ∘(log, abs)(a)
```

```
julia> output
0.0
```

```
a        = 2
outer(a) = a + 2
inner(a) = a / 2

# all `output` are equivalent
output   = (a / 2) + 2
output   = outer(inner(a))
output   = a |> inner |> outer
output   = (outer ∘ inner)(a)
output   = ∘(outer, inner)(a)
```

```
julia> output
3.0
```

The resulting function of the function composition can be broadcasted. The notation for implementing this is easier to understand by thinking of compositions as a new function $h := f \circ g$. This entails that $h(x) := (f \circ g)(x)$ and therefore $h(x) := (f \circ g)(x) = f[g(x)]$. Considering this, broadcasting `h` would require `h.(x)`, which is equivalent to `(f ∘ g).(x)` or `∘(f,g).(x)`.

```
x        = [1, 2, 3]



# all `output` are equivalent
output   = log.(abs.(x))
output   = x .|> abs .|> log
output   = (log ∘ abs).(x)
output   = ∘(log, abs).(x)
```

```
julia> output
3-element Vector{Float64}:
 0.0
 0.6931471805599453
 1.0986122886681098
```

```
x        = [1, 2, 3]
outer(a) = a + 2
inner(a) = a / 2

# all `output` are equivalent
output   = (x ./ 2) .+ 2
output   = outer.(inner.(x))
output   = x .|> inner .|> outer
output   = (outer ∘ inner).(x)
output   = ∘(outer, inner).(x)
```

```
julia> output
3-element Vector{Float64}:
 2.5
 3.0
 3.5
```

Lastly, we can broadcast the composition operator `∘` itself, allowing us to apply multiple functions to the same object. For instance, the following example ensures that each function takes the absolute value of its argument.

```julia
a            = -1

inners       = abs
outers       = [log, sqrt]
compositions = outers .∘ inners

# all `output` are equivalent
output       = [log(abs(a)), sqrt(abs(a))]
output       = [foo(a) for foo in compositions]
```

```
julia> compositions
2-element Vector{ComposedFunction{O, typeof(abs)} where O}:
 log ∘ abs
 sqrt ∘ abs
julia> output
2-element Vector{Float64}:
 0.0
 1.0
```