6c. Chaining Operations

Martin Alfaro
PhD in Economics

INTRODUCTION

This section introduces two approaches to computing operations 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, rendering 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 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 convenient when performing a series of operations, but only the final result is of interest. To illustrate their utility, consider the task of computing the rounded logarithm of `a`'s absolute value, formally expressed as $\mathrm{round}\left(\ln\left(\left|a\right|\right)\right)$.

In Julia, this operation can be implemented evaluating `round(log(abs(a)))`, where `round(a)` returns the nearest integer to `a`. However, the expression is hard to read because of the nested parentheses, with the issue potentially exacerbated if variables or functions had long names.

One straightforward way to improve clarity is to break the whole operation into multiple steps: *i)* compute the absolute value of `a`, *ii)* compute the logarithm of the result, and *iii)* round the resulting output. While this can be implemented through three intermediate variables that store the output in each step, this approach would clutter our namespace and potentially obscure the nested nature of the operations.

A more refined solution is to introduce a **let-block**. This construct resembles functions in several respects. It introduces a new scope delimited by the `let` and `end` keywords, enabling multiple calculations to be performed locally. The result of the last calculation is then returned as

the output. Like functions, let-blocks also allow arguments to be passed by adding 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 varia
   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
   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 you can mutate their arguments, but can't reassign global 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
```

Because mutations can occur within let-blocks, you should exercise caution to prevent unintended consequences in the global scope.

PIPES

For operations consisting of multiple intermediate step, pipes constitutes an alternative to let-blocks. 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 with the `|>` keyword.

Pipes are well-suited for sequential applications of single-argument functions. To illustrate, 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 are useful to create temporary aliases for variables with lengthy names. This allows users to assign meaningful names to variables, while preserving code readability.

```
variable_with_a_long_name = 2

output = variable_with_a_long_name - log(variable
```

```
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 this use, suppose we want to transform each element of `x` with 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

So far, our examples of pipes have followed a simple pattern, with each step consisting of a single-argument function. However, this form precludes the application of pipes to multiple-argument functions or even operations. For example, it prevents the inclusion of expressions like `foo(x,y)` or `2 * x`.

To address this limitation, we can **combine pipes with anonymous functions**. This enables users to specify how the output of the previous step is integrated into the subsequent

operation. By doing so, the utility of pipes is significantly expanded, as demonstrated below.

```
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 in the first place.

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 `_`. Additionally, for single-argument operations that don't require anonymous functions, `@pipe` maintains the same syntax as built-in pipes.

To illustrate its convenience, below we reimplement the last code snippet.

```
#
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 nesting 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, given 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
```

Importantly, the resulting function from function composition can be broadcasted. To understand this notation more clearly, you should think of compositions as defining a new function: \(h:=f\circ g\). This entails that \(h\left(x\right):=\left(f\circ

g\right)\left(x\right)\), and therefore \(h\left(x\right):=\left(f\circ g\right)\left(x\right)=f\left[g\left(x\right)\right]\). Given this, broadcasting `h` requires `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
```

We can also broadcast the composition operator `∘` itself, enabling the simultaneous application of multiple functions to the same object. For instance, the following implementation ensures that each function takes the absolute value of its argument.

```
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
```