Execution time and measurement

INTERMEDIATE JULIA

Anthony Markham
Quantitative Developer

datacamp

Initial function definition

- While problems have multiple solutions, there is often an 'optimal' solution
- Large memory usage and long execution times can frustrate both users and developers
- Loops are a common cause of longer execution times and inefficiency

```
function my_function()
  x = Vector{Char}()
  for i in "Anthony"
    push!(x, i)
  end
  println(x)
end
```

```
['A', 'n', 't', 'h', 'o', 'n', 'y']
```

Base @time macro

- We use macros to time our functions.

Advantages:

- Part of Julia's base package
- Simple and easy to call
- Output is easy to interpret
- Call the macro on our function

```
@time my_function()
```

```
@time my_function()
```

Disadvantages:

- Limited flexibility
- Only times the function once
- First macro call has compilation overhead
- Return values

```
0.278266 seconds (110.66 k allocations:
6.343 MiB, 99.79% compilation time)
```

```
0.000493 seconds (100 allocations: 3.281
KiB)
```

BenchmarkTools - @benchmark

Why use `@benchmark` over `@time` ?

- The most flexible option, lots of parameters to tweak, such as number of `samples` .
- In-depth runtime statistics (range, median, mean).

```
@benchmark my_function
```

```
BenchmarkTools.Trial: 6178 samples with 1 evaluation.
 Range (min … max):  135.830 μs … 61.304 ms  | GC (min … max): 0.00% … 0.00%
 Time  (median):     333.030 μs              | GC (median):    0.00%
 Time  (mean ± σ):   787.335 μs ±  3.978 ms  | GC (mean ± σ):  0.00% ± 0.00%

  136 μs           Histogram: frequency by time          8.29 ms <
```

Let's practice!

INTERMEDIATE JULIA

Positional and Default Arguments, Type Declarations

INTERMEDIATE JULIA

Anthony Markham
Quantitative Developer

datacamp

Function arguments overview

- When declaring a function, we specify parameters.

```
function my_function(param1, param2)
    return param1, param2
end
```

- When calling a function, we pass arguments in to these parameters.

```
my_function(1, 2)
```

```
(1, 2)
```

Positional arguments

- The function arguments that we have seen so far are positional.
- Positional arguments rely on the order that they are specified in.

```
function my_function(param1, param2)
    return param1, param2
end
```

```
my_function(2, 1)
```

```
(2, 1)
```

Default arguments

- Default arguments provide a default value if an argument is not specified.

```
function my_function(param1, param2=2)
    return param1, param2
end
```

```
my_function(1)
```

```
(1, 2)
```

Type declarations

- Type declarations allow us to control the type of data passed into a function.
- Allows us to safeguard our code against incorrect values passed to functions.
- Each parameter can have a data type specified. Use double colon `::` syntax.

```
function my_function(param1::String, param2::Integer=2)
    return param1, param2
end
```

```
my_function(1)
```

```
MethodError: no method matching my_function(::Int64)
```

Let's practice!

INTERMEDIATE JULIA

Keyword Arguments

INTERMEDIATE JULIA

Anthony Markham
Quantitative Developer

Keyword arguments - overview

- Assigns a keyword to a function parameter.
- When passing an argument into a function we use the assigned keyword.
- We can mix keyword arguments with other types, but keyword arguments must come last.

```
function person(; location)
    return location
end
```

Keyword arguments - syntax

- Use the semicolon `;` operator to denote keyword arguments.

```
function person(; location)
    return location
end
```

```
person(location="Sydney")
```

```
"Sydney"
```

Keyword arguments - mixing argument types

- Remember, positional arguments need to be before keyword arguments.

```
function person(name, ; location)
    return name, location
end
```

```
person("Anthony", location="Sydney")
```

```
("Anthony", "Sydney")
```

Variable number of arguments

- Variable number of arguments (varargs) allows us to pass an arbitrary number of arguments.
- Use the ellipsis `...` operator to mark a parameter as accepting varargs.

```
function names(name...)
    println(name)
end

names("Anthony", "Ben", "Hannah", "Julia")
```

```
("Anthony", "Ben", "Hannah", "Julia")
```

Variable number of argument types

- We can mix positional, keyword, and varargs all together in a function definition.

```
function person(name, education... ; location)
    return name, education, location
end

anthony = person("Anthony", "BE", "BS", "MComm", location="Sydney")
```

```
("Anthony", ("BE", "BS", "MComm"), "Sydney")
```

```
println(anthony[2][1])
```

```
BE
```

Let's practice!

INTERMEDIATE JULIA

Writing Your Own Function

INTERMEDIATE JULIA

Anthony Markham
Quantitative Developer

datacamp

Structs recap

- Structs are an application of composite types.
- These are like other types, but with as many fields as you want.

```
mutable struct House
    bedrooms::Int64
    bathrooms::Int64
    location::String
    price::Float64
end
```

```
my_house = House(3, 2, "Sydney", 1500000)
```

- Common in many programming languages for solving real-world problems.

Functions and custom structs

```
mutable struct Person
    age::Int64
    height::Int64
    location::String

    function Person(age, height)
        new(age, height, "London")
    end
end

steve = Person(19, 180)
```

```
Person(19, 180, "London")
```

Functions recap

- How to pass different argument types into a function

```
function arg_types(pos, ; key)
    return pos, key
end
```

- how to return output from a function

```
function return_x_times_y(x, y)
    return x * y
end
```

- how to pass a variable amount of arguments into a function

```
function vararg_names(names...)
    return names
end
```

- Functions are flexible and writing your own functions is an essential part of programming.
- Practical, real-world problem solving is a key part of learning to code.

Let's practice!

INTERMEDIATE JULIA