Julia Essentials

HA Van Thao
Faculty of Math & Computer Science, HCMUS

Contents

- Common Data Types
- Input and Output
- Iterating
- User Defined Functions

Common Data Types

- Julia language defines and provides functions for operating on standard data types such as:
  - Integers
  - Floats
  - Strings
  - Arrays, …

Primitive Data Types - Boolean

- A Boolean value, which can be either true or false

```
julia> x = true
true

julia> typeof(x)
Bool

julia> y = 1 > 2  # Now y = false
false
```

- Under addition, true is converted to 1 and false is converted to 0

```
julia> true + false
1

julia> sum([true, false, false, true])
2
```


Ha Van Thao (hvthao@gmail.com)

Primitive Data Types - Numbers

- The two most common data types used to represent numbers are integers and floats
- Computers distinguish between floats and integers because arithmetic is handled in a different way

```
julia> typeof(1.0)
Float64

julia> typeof(1)
Int64
```

- If you're running a 32 bit system you'll still see Float64, but you will see Int32 instead of Int64

Primitive Data Types - Numbers

- You can use function (instead of infix) notation if you so desire

```
julia> +(10, 20)
30

julia> *(10, 20)
200
```

- Complex numbers are another primitive data type, with the imaginary part being specified by im

```
julia> x = 1 + 2im
1 + 2im

julia> y = 1 - 2im
1 - 2im
```

Strings

```
julia> x = "foobar"
"foobar"

julia> typeof(x)
ASCIIString (constructor with 2 methods)
```

- Julia's simple string formatting operations.
- To concatenate strings use *

```
julia> x = 10; y = 20
20

julia> "x = $x"
"x = 10"

julia> "x + y = $(x + y)"
"x + y = 30"
```


Ha Van Thao (hvthao@gmail.com)

Strings – functions

```
julia> s = "Charlie don't surf"
"Charlie don't surf"

julia> split(s)
3-element Array{SubString{ASCIIString},1}:
 "Charlie"
 "don't"
 "surf"

julia> replace(s, "surf", "ski")
"Charlie don't ski"

julia> split("fee,fi,fo", ",")
3-element Array{SubString{ASCIIString},1}:
 "fee"
 "fi"
 "fo"

julia> strip(" foobar ")  # Remove whitespace
"foobar"
```

Containers - Tuples

- Julia has several basic types for storing collections of data
- A related data type is **tuples**, which can act like "immutable" arrays

```
julia> x = ("foo", "bar")
("foo","bar")

julia> typeof(x)
(ASCIIString,ASCIIString)
```

- An immutable object is one that cannot be altered once it resides in memory

```
julia> x[1] = 42
ERROR: `setindex!` has no method matching setindex!
```


Ha Van Thao (hvthao@gmail.com)

Containers - Tuples

- Similar to Python, as is the fact that the parenthesis can be omitted

```
julia> x = "foo", "bar"
("foo","bar")
```

- Another similarity with Python is tuple unpacking

```
julia> x = ("foo", "bar")
("foo","bar")

julia> word1, word2 = x
("foo","bar")

julia> word1
"foo"

julia> word2
"bar"
```


Ha Van Thao (hvthao@gmail.com)

Containers - Referencing Items

- The last element of a sequence type can be accessed with the keyword end

```
julia> x = [10, 20, 30, 40]
4-element Array{Int64,1}:
 10
 20
 30
 40

julia> x[end]
40

julia> x[end-1]
30
```


Ha Van Thao (hvthao@gmail.com)

Containers - Referencing Items

- To access multiple elements of an array or tuple, you can use slice notation

```
julia> x[1:3]
3-element Array{Int64,1}:
 10
 20
 30

julia> x[2:end]
3-element Array{Int64,1}:
 20
 30
 40
```

```
julia> "foobar"[3:end]
"obar"
```


Ha Van Thao (hvthao@gmail.com)

Containers - Dictionaries

- Dictionaries are like arrays except that the items are named instead of numbered

```
julia> d = Dict("name" => "Frodo", "age" => 33)
Dict{ASCIIString,Any} with 2 entries:
  "name" => "Frodo"
  "age"  => 33

julia> d["age"]
33
```

- The strings name and age are called the **keys**
- The objects that the keys are mapped to ("Frodo" and 33) are called the **values**
- They can be accessed via keys(d) and values(d) respectively

Input and Output - Writing

```
julia> f = open("newfile.txt", "w")   # "w" for writing
IOStream(<file newfile.txt>)

julia> write(f, "testing\n")          # \n for newline
7

julia> write(f, "more testing\n")
12

julia> close(f)
```

- The effect of this is to create a file called newfile.txt in your present working directory with contents

```
testing
more testing
```

Input and Output - Reading

```
julia> f = open("newfile.txt", "r")   # Open for reading
IOStream(<file newfile.txt>)

julia> print(readall(f))
testing
more testing

julia> close(f)
```

- readall is deprecated, using readstring instead

Iterating

- One of the most important tasks in computing is stepping through a sequence of data and performing a given action
- **Iterables:** An iterable is something you can put on the right hand side of for and loop over

```
actions = ["surf", "ski"]
for action in actions
    println("Charlie don't $action")
end
```

- They also include so-called **iterators**

```
julia> for i in 1:3 print(i) end
123
```


Ha Van Thao (hvthao@gmail.com)

Iterating

- If you ask for the keys of dictionary you get an iterator

```
julia> d = Dict("name" => "Frodo", "age" => 33)
Dict{ASCIIString,Any} with 2 entries:
  "name" => "Frodo"
  "age"  => 33

julia> keys(d)
Base.KeyIterator for a Dict{ASCIIString,Any} with 2 entries. Keys:
  "name"
  "age"
```

- Should you need to transform an iterator into an array you can always use collect()

```
julia> collect(keys(d))
2-element Array{Any,1}:
 "name"
 "age"
```

Iterating – Looping without Indices

- You can loop over sequences without explicit indexing, which often leads to neater code

```
for x in x_values
    println(x * x)
end
```

```
for i in 1:length(x_values)
    println(x_values[i] * x_values[i])
end
```

Iterating – Looping without Indices

- zip(): stepping through pairs from two sequences

```
countries = ("Japan", "Korea", "China")
cities = ("Tokyo", "Seoul", "Beijing")
for (country, city) in zip(countries, cities)
    println("The capital of $country is $city")
end
```

- If we happen to need the index as well as the value, one option is to use enumerate()

```
countries = ("Japan", "Korea", "China")
cities = ("Tokyo", "Seoul", "Beijing")
for (i, country) in enumerate(countries)
    city = cities[i]
    println("The capital of $country is $city")
end
```

Iterating – Looping without Indices

- zip(): stepping through pairs from two sequences

```
countries = ("Japan", "Korea", "China")
cities = ("Tokyo", "Seoul", "Beijing")
for (country, city) in zip(countries, cities)
    println("The capital of $country is $city")
end
```

- If we happen to need the index as well as the value, one option is to use enumerate()

```
countries = ("Japan", "Korea", "China")
cities = ("Tokyo", "Seoul", "Beijing")
for (i, country) in enumerate(countries)
    city = cities[i]
    println("The capital of $country is $city")
end
```

Iterating – Comprehensions

- Comprehensions are an elegant tool for creating new arrays or dictionaries from iterables

```
julia> doubles = [2i for i in 1:4]
4-element Array{Int64,1}:
 2
 4
 6
 8

julia> animals = ["dog", "cat", "bird"];   # semicolon suppresses output

julia> plurals = [animal * "s" for animal in animals]
3-element Array{ByteString,1}:
 "dogs"
 "cats"
 "birds"
```

```
julia> [i + j for i in 1:3, j in 4:6]
3x3 Array{Int64,2}:
 5  6  7
 6  7  8
 7  8  9

julia> [i + j + k for i in 1:3, j in 4:6, k in 7:9]
3x3x3 Array{Int64,3}:
[:, :, 1] =
 12  13  14
 13  14  15
 14  15  16

[:, :, 2] =
 13  14  15
 14  15  16
 15  16  17

[:, :, 3] =
 14  15  16
 15  16  17
 16  17  18
```

User Defined Functions

- Any number of functions can be defined in a given file
- Any "value" can be passed to a function as an argument, including other functions
- Functions can be (and often are) defined inside other functions
- A function can return any kind of value, including functions

User Defined Functions
Return Statement

- In Julia, the return statement is optional, so that the following functions have identical behavior

```
function f1(a, b)
    return a * b
end

function f2(a, b)
    a * b
end
```

- When no return statement is present, the last value obtained when executing the code block is returned
- Although some prefer the second option, we often favor the former on the basis that explicit is better than implicit

User Defined Functions
Other Syntax for Defining Functions

- First, when the function body is a simple expression, it can be defined without the function keyword or end

```
julia> f(x) = sin(1 / x)
f (generic function with 2 methods)
```

- Julia also allows for you to define anonymous functions
- For example, to define f(x) = sin(1 / x) you can use x -> sin(1 / x)

```
julia> map(x -> sin(1 / x), randn(3))  # Apply function to each element
3-element Array{Float64,1}:
  0.744193
 -0.370506
 -0.458826
```


Ha Van Thao (hvthao@gmail.com)

User Defined Functions
Optional Arguments

- Function arguments can be given default values

```
function f(x, a=1)
    return exp(cos(a * x))
end
```

- If the argument is not supplied the default value is substituted

```
julia> f(pi)
0.36787944117144233

julia> f(pi, 2)
2.718281828459045
```

User Defined Functions
Keyword Arguments

- The difference between keyword and standard (positional) arguments is that they are parsed and bound by name rather than order in the function call

```
simulate(param1, param2, max_iterations=100, error_tolerance=0.01)
```

- To define a function with keyword arguments you need to use ; like so

```
function simulate(param1, param2; max_iterations=100, error_tolerance=0.01)
    # Function body here
end
```


Ha Van Thao (hvthao@gmail.com)

Preferences

- http://julialang.org