Learn Julia in Y minutes (2)

Get the code: learnjulia.jl

1 Control Flow

Tuples are immutable.

```
tup = (1, 2, 3) # => (1,2,3) # an (Int64,Int64,Int64) tuple.
tup[1] # => 1
try:
    tup[1] = 3 # => ERROR: no method setindex!((Int64,Int64,I
catch e
    println(e)
end

MethodError(setindex!,(:tup,3,1))
```

Many list functions also work on tuples

```
length(tup) # => 3
tup[1:2] # => (1,2)
in(2, tup) # => true
```

You can unpack tuples into variables

```
a, b, c = (1, 2, 3) # => (1,2,3)  # a is now 1, b is now 2 and
```

Tuples are created even if you leave out the parentheses

```
d, e, f = 4, 5, 6 # => (4,5,6)
```

A 1-element tuple is distinct from the value it contains

```
(1,) == 1 # => false
(1) == 1 # => true
```

Look how easy it is to swap two values

```
e, d = d, e  # => (5,4) # d is now 5 and e is now 4
```

Dictionaries store mappings

```
empty_dict = Dict() # => Dict{Any,Any}()
```

You can create a dictionary using a literal

```
filled_dict = Dict("one"=> 1, "two"=> 2, "three"=> 3)
# => Dict{ASCIIString,Int64}
```

Look up values with []

```
filled_dict["one"] # => 1
```

Get all keys

```
keys(filled_dict)
# => KeyIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=>
```

Note

dictionary keys are not sorted or in the order you inserted them.

Get all values

```
values(filled_dict)
# => ValueIterator{Dict{ASCIIString,Int64}}(["three"=>3,"one"=
```

Note - Same as above regarding key ordering.

Check for existence of keys in a dictionary with in, haskey

```
in(("one" => 1), filled_dict) # => true
in(("two" => 3), filled_dict) # => false
haskey(filled_dict, "one") # => true
haskey(filled_dict, 1) # => false
```

Trying to look up a non-existent key will raise an error

```
try
    filled_dict["four"] # => ERROR: key not found: four in ge
catch e
    println(e)
end

UndefVarError(:filled_dict)
```

Use the get method

to avoid that error by providing a default value

```
get(dictionary,key,default_value)
get(filled_dict,"one",4) # => 1
get(filled_dict,"four",4) # => 4
```

Use Sets to represent collections of unordered, unique values

```
empty_set = Set() # => Set{Any}()
```

Initialize a set with values

```
filled_set = Set([1,2,2,3,4]) # => Set{Int64}(1,2,3,4)
```

Add more values to a set

```
push!(filled_set,5) # => Set{Int64}(5,4,2,3,1)
```

Check if the values are in the set

```
in(2, filled_set) # => true
in(10, filled_set) # => false
```

There are functions for set intersection, union, and difference.

```
other_set = Set([3, 4, 5, 6]) # => Set{Int64}(6,4,5,3)
intersect(filled_set, other_set) # => Set{Int64}(3,4,5)
union(filled_set, other_set) # => Set{Int64}(1,2,3,4,5,6)
setdiff(Set([1,2,3,4]),Set([2,3,5])) # => Set{Int64}(1,4)
```

Control Flow

Let's make a variable

```
some_var = 5
```

Here is an if statement. Indentation is not meaningful in Julia.

```
if some_var > 10
    println("some_var is totally bigger than 10.")
elseif some_var < 10    # This elseif clause is optional.
    println("some_var is smaller than 10.")
else                    # The else clause is optional too.
    println("some_var is indeed 10.")
end
# => prints "some var is smaller than 10"
```

For loops iterate over iterables.

Iterable types include `Range`, `Array`, `Set`, `Dict`, and `AbstractString`.

```
julia> for animal=["dog", "cat", "mouse"]
    println("$animal is a mammal")
    # You can use $ to interpolate variables or expression in
end
dog is a mammal
cat is a mammal
mouse is a mammal
```

You can use 'in' instead of '='.

```
julia> for animal in ["dog", "cat", "mouse"]
    println("$animal is a mammal")
end
dog is a mammal
cat is a mammal
mouse is a mammal
```

Example

```
julia> for a in Dict("dog"=>"mammal","cat"=>"mammal","mouse"=>
    println("$(a[1]) is a $(a[2])")
end
mouse is a mammal
cat is a mammal
dog is a mammal
```

Example

```
julia> for (k,v) in Dict("dog"=>"mammal","cat"=>"mammal","mous
    println("$k is a $v")
end
mouse is a mammal
cat is a mammal
dog is a mammal
```

While loops loop while a condition is true

```
julia> x = 0
0
julia> while x < 4
    println(x)
    x += 1  # Shorthand for x = x + 1
end
0
1
2
3
```

Handle exceptions with a try/catch block

```
try
    error("help")
catch e
    println("caught it $e")
end
# => caught it ErrorException("help")
```