CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2016
Doc 3 Julia Control & Compound Types
Sep 1, 2016

Control Flow

Compound Expressions

begin end block - evaluate the blockexpressions, return value of last expression

```
z = begin                  # z now is 3
      x = 1
      y = 2
      x + y
   end
```

(;) chains - evaluate expressions in the chain, return the value of the last expression

```
z = (x = 1; y = 2; x + y)          # z == 3
```

If

```
x = 5
x < 3 ? "yes" : "no"
result = x >= 4 ? "good" : "bad"
```

```
if name == "Jeeves"
  println("Very Good Jeeves")
elseif name == "Brinkley"
  println("Thank you, Brinkley")
else
  println("Fine, just ignore me")
end
```

```
if name == "Jeeves"
  println("Very Good Jeeves")
end
```

```
x = if (y < 3)
          5
    else
          4
    end
```

```
x = if (y < 3)  5 else 4  end
```

Boolean Values

true, false

```
1 == 2          # false
2 == 2          # true


if 1            # TypeError
  2
else
  1
end
```

Short-Circuit Evaluations

&& and	a && b	# b is evaluated only if a is true
|| or	a || b	# b is evaluated only if a is false

```
if (a < 5 ) || (y > 3)
      z = 9
end
```

```
function factorial(n::Int)
      n >= 0 || error("n must be non-negative")
      n == 0 && return 1
      n * factorial(n-1)
end
```

for

```
for i in 1:5
  println(i)
end
```

```
1
2
3
4
5
```

```
for i = 1:5
  println(i)
end
```

```
for i = 1:3:5
  println(i)
end
```

```
1
4
```

```
for color in ["red", "green", "blue"]
     print(color, " ")
end
```

```
for letter in "julia"
    print(letter, " ")
end
```

```
for i in Dict("A"=>1, "B"=>2)
  println(i)
end
```

```
"B"=>2
"A"=>1
```

```
for (index,value) in enumerate(1:4)
  println("The $index-th element is $value")
end
```

```
The 1-th element is 1
The 2-th element is 2
The 3-th element is 3
The 4-th element is 4
```

Nested For

```
for k in 1:10
    for j in 1:5
        println(k+j)
    end
end
```

```
for k in 1:10, j in 1:5
    println(k+j)
end
```

Continue

```
for i = 1:10
    if i % 3 != 0
        continue
    end
    println(i)
end
```

```
3
6
9
```

UnitRange - Data Type

```
a = 1:10
b = 1:2:100_000_000
length(a)              # 10
length(b)              # 50_000_000
```

```
whos()
```

In Console

```
a     16 bytes  10-element UnitRange{Int64}
b     24 bytes  50000000-element StepRange{Int64,I…
```

Finding the Size of Data

```
a = 1:10
b = 1:2:100_000_000

Base.summarysize(12)          # 8    in bytes

Base.summarysize(a)           # 16
Base.summarysize(b)           # 24

sizeof(UnitRange{Int64})      # 16
```

Memory allocation becomes important with big data

while

```
while i <= 5
    println(i)
    i += 1
end
```

```
while true
    println(i)
    if i >= 5
        break
    end
    i += 1
end
```

Function Basics

```
function fibonacci(n)
  if n < 2
    return 1
  end
  return fibonacci(n-1) + fibonacci(n-1)
end
```

```
function fibonacci(n)
  if n < 2
    return 1
  end
  fibonacci(n-1) + fibonacci(n-1)
end
```

```
fibonacci(n) = n < 2 ? 1:fibonacci(n-1) + fibonacci(n-2)
```

```
f(x) = 2x^2 - 3x + 5
```

Declaring Types

```
increment1(n) = n + 1

increment1(3)           # 4
increment1("cat")       MethodError: `+` has no method matching +(::ASCIIString, ::Int6

increment2(n::Int64) = n + 1

increment2("cat")

    MethodError: `increment2` has no method matching increment2(::ASCIIString)
```

Declaring types of Variables

```
function increment3(n)
  result::Int = n + 1
  return result
end
```

```
bar = 4
foo::Int = 3
```

UndefVarError: foo not defined

Can not declare types of variables at top level

Return Types

```
increment4(n::Int64)::Int64 = n + 1
```

Add in Julia 0.5

Exceptions

Raising Exception
Catching Exceptions
Defining Exceptions

Raising Exceptions

error() - raises a generic exception

```
fussy_sqrt(x) = x >= 0 ? sqrt(x) : error("negative x not allowed")
```

throw() - raises a specific exception

```
fussy_sqrt(x) = x >= 0 ? sqrt(x) : throw(DomainError())
```

Builtin Exceptions

ArgumentError
BoundsError
CompositeException
DivideError
DomainError
EOFError
ErrorException
InexactError
InitError
InterruptException
InvalidStateException

KeyError
LoadError
OutOfMemoryError
ReadOnlyMemoryError
RemoteException
MethodError
OverflowError
ParseError
SystemError
TypeError
UndefRefError
UndefVarError
UnicodeError

try/catch

```
f(x) = try
        sqrt(x)
      catch
        sqrt(complex(x, 0))
      end
```

```
f = open("file")
try
    # operate on file f
finally
    close(f)
end
```

```
sqrt_second(x) = try
        sqrt(x[2])
      catch y
        if isa(y, DomainError)
          sqrt(complex(x[2], 0))
        elseif isa(y, BoundsError)
          sqrt(x)
        end
      end
```

You can use catch and finally together

Composite Types

Type

```
type Point
  x::Float64
  y::Float64
end

a = Point(12.3,2.1)
b = Point(1,2)
a.x                   # 12.3
b.x                   # 1.0

function +(a::Point, b::Point)
  Point(a.x + b.x, a.y + b.y)
end

a + b                 # Point(13.3,4.1)
+(a,b)
```

Two constructors are created
With exact types in order
With types converted via convert

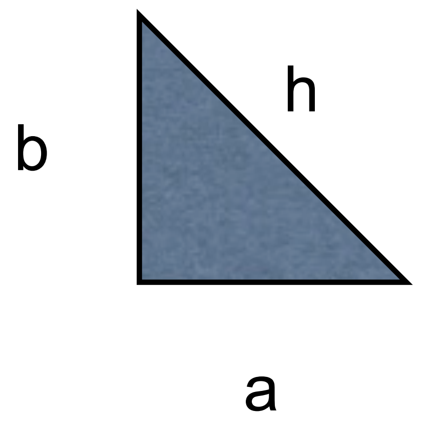
```
type Point
  x::Float64
  y::Float64
end

import Base.length
length(p::Point) = hypot(p.x, p.y)

a = Point(12.3,2.1)

length(a)
```

Numerical Aside



h = sqrt(a*a + b*b)

What if a is large?

```
a = typemax(Int64)÷2
b = 1
a*a                      # -9223372036854775807
sqrt(a*a + b*b)          # DomainError
hypot(a,b)               # 4.611686018427388e18
```

Here's how to compute sqrt(x*x + y*y) without risking overflow

```
max = maximum(|x|, |y|)
min = minimum(|x|, |y|)
r = min / max
return max*sqrt(1 + r*r)
```

```
sqrt(x*x + y*y) = sqrt(max*max + min*min)
                = sqrt(max*max + max*max*min*min/(max*max))
                = sqrt(max*max + max*max*(min/max)*(min/max))
                = sqrt(max*max + max*max*r*r)
                = sqrt(max*max*(1 + r*r)
                = max*sqrt(1 + r*r)
```

http://www.johndcook.com/blog/2010/06/02/whats-so-hard-about-finding-a-hypotenuse/

Objects & Julia

```
class Point {
    Float64 x;
    Float64 y;

    public Float64 length() {
        return Math.sqrt(x*x+y*y);
    }
}

Point a = new Point(1,1);


a.length();
```

```
type Point {
    Float64 x;
    Float64 y;
}

length(p::Point)::Float64 {
    return Math.sqrt(p.x*p.x+p.y*p.y);
}

a::Point = Point(1,1)


length(a)
```

Python Class

```
class Point:

  def __init__(self, x = 0, y = 0):
    self.x = x
    self.t = y

  def length(self):
   return sqrt(self.x*self.x + self.y*self.y)


 p = Point(1,2)

 p.length()      ——————————>      length(p)

              Compiler transforms into
```

Why This Matters

OO langauges have a virtual method table (vtable) stored in each class

- At run time have to find the proper vtable to find the proper method to call

Julia stores a vtable for each function name

- The proper vtable is known at compile time
- Makes Julia much faster

Immutable Types

```
immutable Point2
  x::Float64
  y::Float64
end

d = Point2(1,1)
d.x                    # 1.0
d.x = 2                # Error
```

Subtypes

```
abstract Point

type Point2D <: Point
  x::Float64
  y::Float64
end

type Point3D <: Point
  x::Float64
  y::Float64
  z::Float64
end
```

All parent types must be abstract

```
function foo(x::Point)
  "Point"
end

function foo(x::Point3D)
  "3D Point"
end

function bar()
  a::Point = Point2D(1,1)
  b::Point = Point3D(1,2,3)
  (foo(a), foo(b))
end

bar()          # ("Point", "3D Point")
```