CS 696 Intro to Big Data: Tools and Methods
Fall Semester, 2016
Doc 6 Functions, Map, Reduce
Sep 13, 2016

Equality

== Are the values the same

=== (≡) Can any program tell them apart, stronger then ==

isequal Used in Dictionaries to determine if keys should be considered the same

```
5 == 5.0          # true
5 === 5.0         # false
5 === 5           # true
isequal(5, 5.0)   # true
```

```
x = [1,2,3]
y = [1,2,3]
z = [1,2,3.0]

x == y          # true
x === y         # false
isequal(x, y) # true

x == z              # true
x === z         # false
isequal(x, z) # true
```

Comprehensions

```
[n for n in [1:4]]
```

```
4-element Array{Int64,1}:  [1,2,3,4]
```

```
[n^2 for n in [1:4]]
```

```
4-element Array{Int64,1}: [1, 4, 9, 16]
```

```
[row+col for row in [1:2], col in [1:3]]
```

```
2x3 Array{Int64,2}:
 [2  3  4
 3  4  5]
```

```
[n => n^2 for n in [1:3]]
```

```
Dict{Int64,Int64} with 3 entries:
  2 => 4
  3 => 9
  1 => 1
```

Functions

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

```
function mult2(x, y)
    x * y
end
```

```
mult3(x, y) =  x * y
```

```
function sum_product(x, y)
    x + y, x * y
end
```

```
a, b = sum_product(2, 4)
a                  # 6
b                  # 8
```

```
(a, b) = sum_product(2, 4)
```

```
poly(x) = 3x^2 - 4x + 1
poly(2)                    # 5
```

Pass-by-Sharing

Parameters are not copied

```
function foo(x)
  x[1] = 4
end

a = [1,2]

foo(a)

a[1]  == 4
```

```
function bar(x)
  x = 10
end

b = 2

bar(b)

b == 2
```

Splicing Arguments

```
args(x, y) = x + y

args(2, 3)          # 5

x = (2, 3)
args(x...)          # 5

y = [3, 4]
args(y...)              # 7

z = [1, 2, 3]
args(z...)          # MethodError
```

Variable Arguments

```
varargs(x, foo...) = println("x = $x, foo= $foo" )


varargs(1)          # X = 1, foo= ()

varargs(1, 2)       # X = 1, foo= (2,)

varargs(1, 2, 3)    # X = 1, foo= (2, 3)
```

```
function var_sum(args...)
  sum = zero(Int64)
  for k in args
    sum += k
  end
  sum
end
```

Optional arguments

```
f(a, b = 5, c= 6) = "a=$a, b=$b, c=$c"

f(1,2,3)  # a=1, b=2, c=3
f(1,2)          # a=1, b=2, c=6
f(1)            # a=1, b=5, c=6


g(a::Int64 = 1, b::ASCIIString = "cat") = "$a $b"

g(1)            # 1 cat
g(“a”)          # MethodError
```

Optional arguments come after positional arguments

Optional arguments must be given in order

Optional Keyword

```
h(a; b = 2, c = 5) = "$a $b $c"

h(1, c = 7, b = 0)          # 1 0 7
h(1)                            # 1 2 5
h(3,c =4)                   # 3 2 4
```

```
k(;a = 1,b = 2) = "$a $b"
k()
k(b = 5)
```

Optional keyword arguments come after all optional arguments

Nested & Recursive

```
function a(x)
    z = x * 2
    function b(z)
        z += 1
    end
    b(z)
end
```

```
sum(n) =  n > 1 ? sum(n-1) + n : n
```

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

Anonymous Functions

Functions with no names

```
function (x)
      x + 2
end
```

```
(x, y) -> x + 3*y
```

```
foo = (x::Int64) -> x + 1
```

```
x -> x^2
```

```
() -> “No arguments”
```

Anonymous functions in Julia 0.4

Much slower than regular functions

Julia 0.5 corrects this

Lambda

```
function counter(n)
  return (k) ->  n += k
end
```

```
a = counter(10)
b = counter(5)
b(1)  == 6
a(2) == 12
a(1) == 11
```

More hidden State

```
function counter(start = 0)
  n = start
  return () -> n += 1, () -> n = start
end


(plus_a, reset_a) = counter(10)
(plus_b, reset_b) = counter()
```

```
plus_a()        # 11
plus_a()        # 12
reset_a()       # 10
plus_b()        # 1
plus_a()        # 11
```

Single Dispatch

```
function foo(n::Integer)
  "Integer $n"
end

function foo(n::Int32)
  "Int32 $n"
end

function foo(x::AbstractFloat)
  "Float $x"
end
```

```
test = 2

foo(test)                Integer 2


test = 2.9

foo(test)                Float 2.9


test = Int32(2)

foo(test)                Int32 2


test = Int16(2)

foo(test)                Integer 2
```

Java/C++/C# Example

```
public class Parent {
}

public class Child extends Parent {
}

public Class Bar {

    public foo(Parent x) {
        return “Parent”;
    }

    public foo(Child x) {
        return “Child”;
    }
```

```
Bar test = new Bar();

Parent x = new Parent();
test.foo(x);                    Parent

x = new Child();
test.foo(x);                    Parent

Child y = new Child();
test.foo(y);                    Child
```

Multiple Dispatch

```
foo(a::Integer,b::Integer) = "Integer,Integer"
foo(a::Integer,b::Number) = "Integer,Number"
foo(a::Number,b::Integer) = "Number,Integer"
foo(a::Number,b::Number) = "Number,Number"
foo(a::Number,b::Complex) = "Number,Complex"
```

foo(1,2)	Integer,Integer
foo(1,2.3)	Integer,Number
foo(2//3,1)	Number,Integer
foo(2.3,2im + 2)	Number,Complex

Why Important

```
function power(n::Number,exponent::Real)
    Some complicated process for float exponent
end

function power(n::Number,exponent::Complex)
    Deal with complex exponent
end

function power(n::Number, exponent::Integer)
  if exponent == 0
    return 1
  result = n
  for k in 1:n-1
    result *= n
  end
  result
end
```

Open & Closed

A module is open if can

Add/remove fields
Add/remove methods

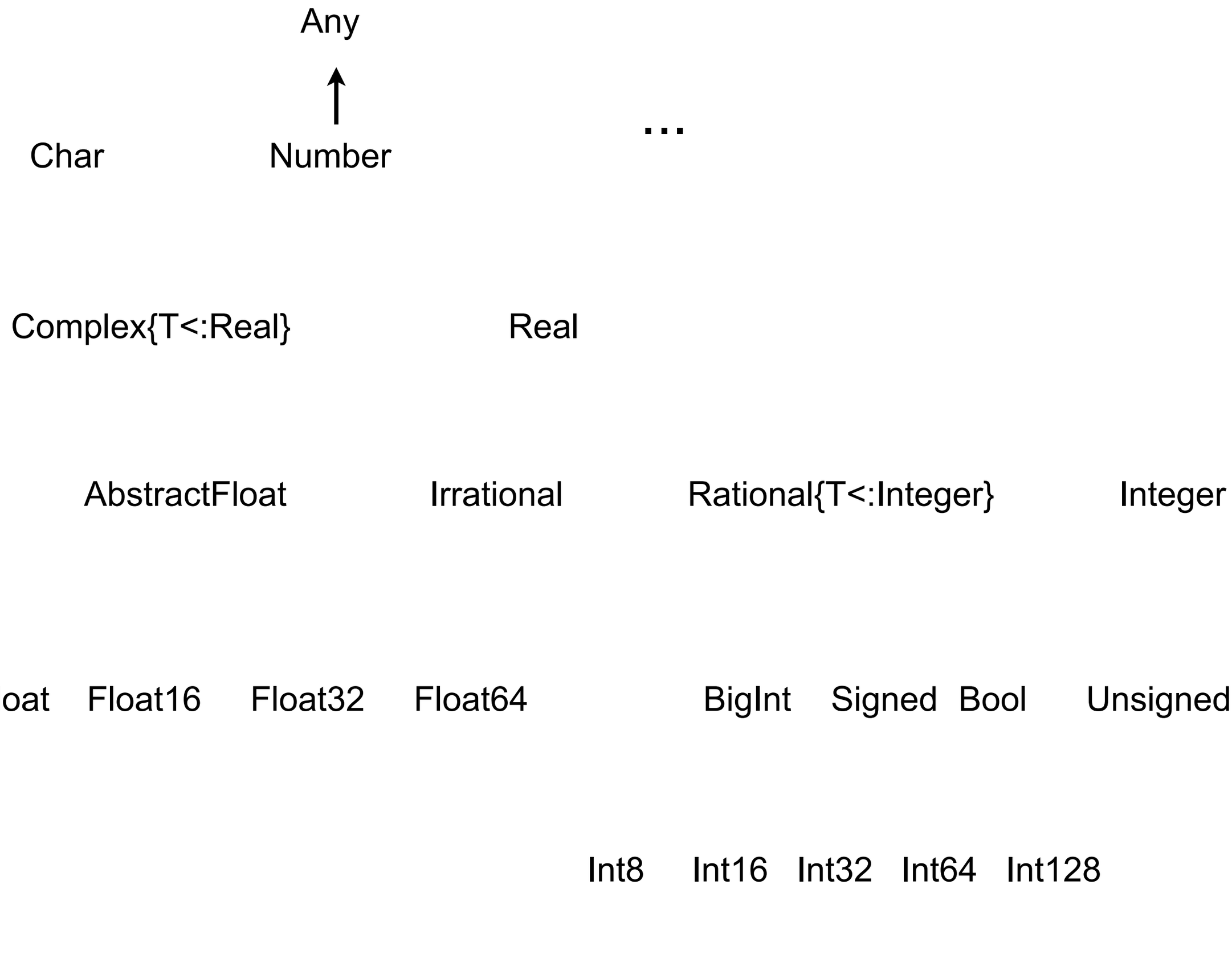
A module is closed if

It can be used by other modules

Open-Close Principle

Module should be open for extension

But closed for modification


Any
Char
Number
...
Complex{T<:Real}
Real
AbstractFloat
Irrational
Rational{T<:Integer}
Integer
BigFloat
Float16
Float32
Float64
BigInt
Signed
Bool
Unsigned
Int8
Int16
Int32
Int64
Int128

Map, Reduce, Filter

Higher order functions

Very important

Map

Apply a function to each element of a collection, return resulting collection

Ruby - collect, map

Smalltalk - collect

Filter

Returns elements of collection that make function true

Reduce

Applies function to combine collection into one value

Map

```
inc(x) = x + 1

map(inc,[1 2 3])              # [2 3 4]

map(inc,[1 2; 3 4])           # [2 3
                                        4 5]

map(+, [1 2 3], [4 4 4])      # [5 6 7]

map(pair -> pair[2] + 10, Dict("a"=> 1,"b"=>2))        # 2-element Array{Any,1}:
                                                                12
                                                                11

map(inc,Set([1,2,3]))         # 3-element Array{Any,1}:
                                        3
                                        4
                                        2
```

Map

```
inc(x) = x + 1


function by_hand(array, fn)
  result = Array(Int64, length(array))
  for k in 1:length(array)
    result[k] = fn(array[k])
  end
  result
end
```

```
map(inc,sample)
```

```
sample = Array(Int64,100_000_000)
@time by_hand(sample, inc)          #  7.2 seconds
@time map(inc,sample)               #  3.7 seconds
```

Long Methods in Map

```
map(x-> if x == 0 return 0
        elseif iseven(x) return 2
        elseif isodd(x) return 1
        end
       ,[-3:3])
```

```
map([-3:3]) do x
  if x == 0 return 0
  elseif iseven(x) return 2
  elseif isodd(x) return 1
  end
end
```

```
7-element Array{Int64,1}:
 1
 2
 1
 0
 1
 2
 1
```

Example

```
function substrings(string,n)
  map(k -> string[k:k+n-1], 1:length(string) - n + 1)
end
```

```
substrings("12345",2)              # [“12”, “23”, “34”, “45”]
```

Maps & List Compressions

```
map(inc,[1 2 3])   # [2 3 4]
```

```
[inc(x) for x in [1 2 3]]  # [2, 3, 4]
```

```
map(inc,[1, 2, 3])       # [2, 3, 4]
```

```
[inc(x) for x in [1, 2, 3]]  # [2, 3, 4]
```

```
map(pair -> pair[2] + 10, Dict("a"=> 1,"b"=>2))
```

[12, 11]

```
[p[2]+10 for p in Dict("a"=> 1,"b"=>2)]
```

```
map(inc,Set([1,2,3]))
```

[3, 4, 2]

```
[inc(n) for n in Set([1 2 3])]
```

map! - In place map

```
x = [1, 2, 3]

map(inc,x)
x                       # [1, 2, 3]

map!(inc,x)
x                       # [2, 3, 4]
```

map! is slower that map

```
result = Array(Int64,100_000)
map!(inc,result,x)

result                  # [3, 4, 5]
```

Map - Why Important

Less code

Faster

Hide details of how to traverse collection

Main part of Hadoop & Spark

Filter

```
filter(iseven, 1:10)                # [2, 4, 6, 8, 10]

filter(iseven,[1 2 3;4 5 6 ])       # [4, 2, 6]

filter(iseven,Set([1 2 3]))         # Set([2])
```

```
lines = readlines("Collections.jl")
filter(x->in('#',x),lines)
```

```
filter(lines) do x
  in('#',x)
end
```

Reduce

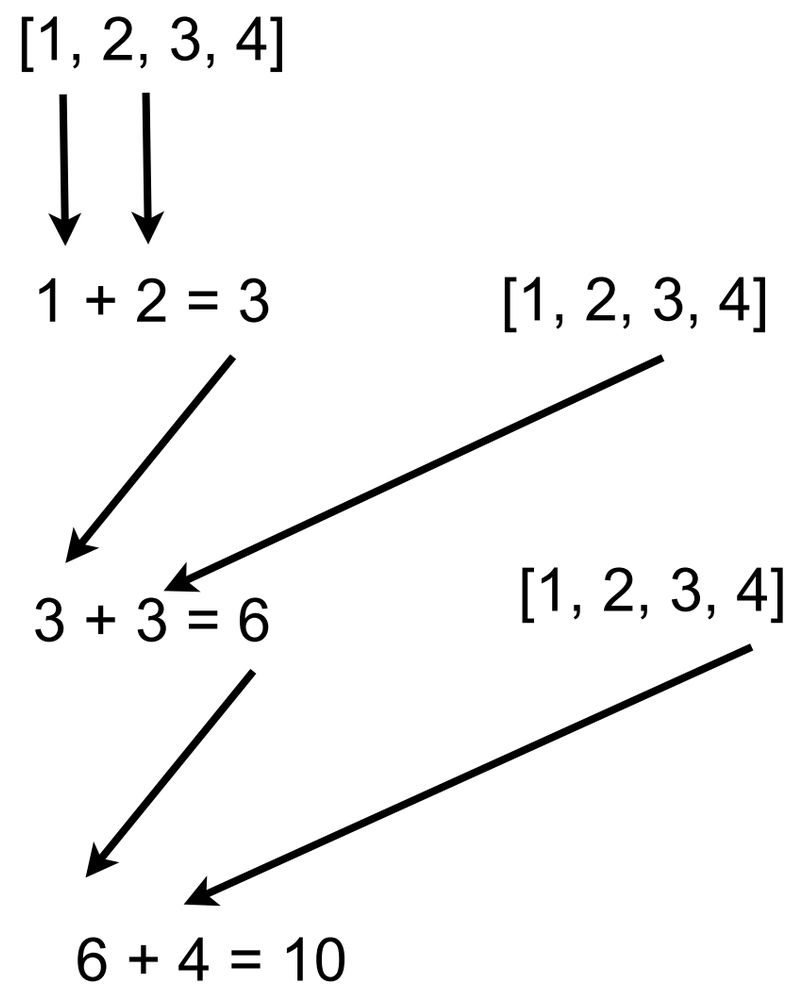
```
reduce(+,[1, 2, 3, 4])        # 10

reduce(+,[1 2; 3 4])          # 10

reduce(+,Set([1 2 3]))        # 6

start_value = 10
reduce(+, start_value ,[1, 2, 3, 4])  # 20
```

reduce(function,collection)
reduce(function,start,collection)

function needs two arguments

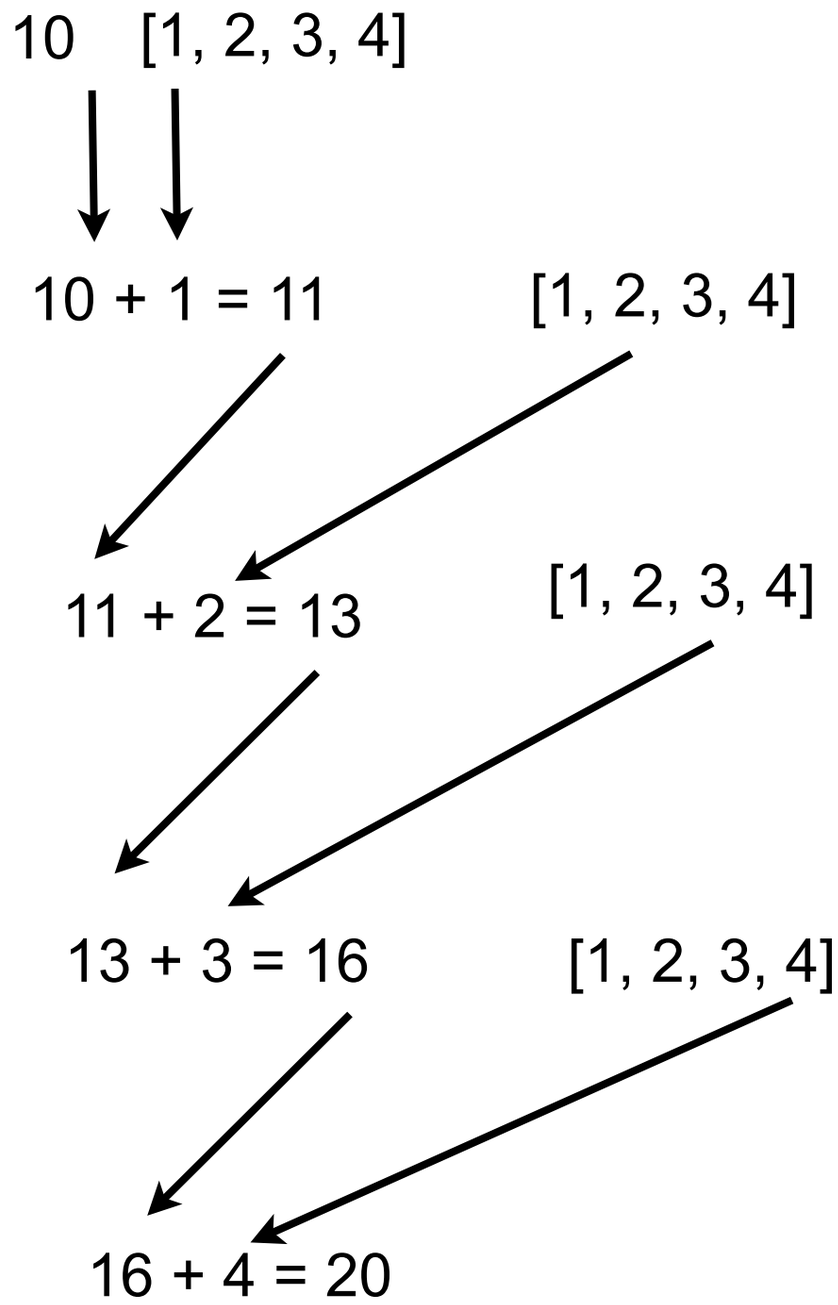
How it Works

reduce(+,[1, 2, 3, 4])



How it Works

reduce(+,10[1, 2, 3, 4])



```
function verbose_plus(x,y)
  println("x = $x, y = $y")
  x + y
end
```

```
reduce(verbose_plus, [1, 2, 3, 4])
```

```
x = 1, y = 2
x = 3, y = 3
x = 6, y = 4
```

```
reduce(verbose_plus,10,[1, 2, 3, 4])
```

```
x = 10, y = 1
x = 11, y = 2
x = 13, y = 3
x = 16, y = 4
```

Using do

```
reduce([1, 2, 3, 4]) do x, y
  println("x = $x, y = $y")
  x + y
end
```

```
reduce(10, [1, 2, 3, 4]) do x, y
  println("x = $x, y = $y")
  x + y
end
```

Specialized reducers

```
sum([1, 2, 3])

product([1, 2, 3])

maximim([1, 2, 3])

minimim([1,2, 3])

extrema([2,1,3,5,4])        # (1,5)

indmax([2,1,3,5,4])         # 4

indmin([2,1,3,5,4])         # 2

findmax([2,1,3,5,4])        # (5,4)

maxabs([2,-3,1,5,-7])       # 7

sumabs([1,-2,3,-4])         # 10

sumabs2([1,-2,3,-4])        # 30

count(isprime, 1:100)       # 25
```

```
all(i->(4<=i<=6), [4,5,6])

all([4,5,6]) do i
  4<=i<=6
end
```

```
any(isprime,200:230)
```

filter using reduce

```
filter(iseven, 1:10)
```

```
reduce([],1:10) do accumulator, item
  if iseven(item)
    push!(accumulator,item)
  end
  accumulator
end
```

map using reduce

```
map(inc, [1,2,3])

reduce((accum, n)-> push!(accum, inc(n)),[],[1,2,3])

reduce([],[1,2,3]) do accum, n
  push!(accum, inc(n))
end
```

Map, Reduce, Hadoop

Hadoop & Spark use map and reduce operators

You break down a problem to using map & reduce

So need to get used to thinking in terms of map & reduce

Hadoop add a sort and grouping between map & reduce

Defining an Iteration

Define a type and the following methods

Required methods		Brief description
start(iter)		Returns the initial iteration state
next(iter, state)		Returns the current item and the next state
done(iter, state)		Tests if there are any items remaining
Optional methods	Default definition	Brief description
eltype(IterType)	Any	The type the items returned by next()
length(iter)	(undefined)	The number of items, if known

Defining an Iteration - Pairs

Given a one dimensional array iterate through it returning Tuples of two items

```
immutable Pairs{I}
  xs::I
end

Base.start(::Pairs) = 1
Base.next(P::Pairs, state) = ((P.xs[state],P.xs[state+1]), state + 2)
Base.done(P::Pairs, state) = state >= length(P.xs)
Base.length(P::Pairs) = length(P.xs)÷2
Base.eltype{I}(::Type{Pair{I}}) = Tuple{I,I}
```

```
for x in Pairs([1 2 3 4])
  println(x)
end
```

Output
(1, 2)
(3, 4)

Defining an Iteration - Pairs

```
for x in Pairs([1 2 3 4])
  println(x)
end
```

```
iter = Pairs([1 2 3 4])
state = start(iter)
while !done(iter, state)
      (x, state) = next(iter, state)
      println(x)
end
```

```
map( x -> println(x), Pairs([1 2 3 4]))
```

```
reduce((y,x) -> push!(y, x),[], Pairs([1 2 3 4]))
```