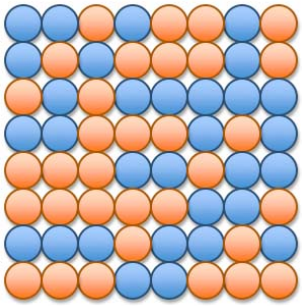


UNIT 3A
Ranges, Arrays, and Iterators

Ruby Data Types

- Numbers:
  - Integers: 3 -5
  - Floats: 1.557 3.05e+11
- Bools: true false
- Strings: “Woof” “jelly beans”
- Ranges: 1..5 -38400..65536
- Arrays: [3, 1, 5, 2] [“Moe”, “Larry”, “Curly”]

Type Conversion Is Easy in Ruby

- Convert to integer: to_i
- Convert to float: to_f
- Convert to string: to_s
- Convert to array: to_a

```
5.to_s        =>  "5"
"63".to_i     =>  63
"bam".to_i    =>  0
"12 spicy tacos" .to_i      =>  12
"12 warm churros" .to_f     =>  12.0
```


Extended Meanings For + And *

What should + and * mean for non-numbers?

- We can use + to mean concatenation:

```
"woof" + "meow"    =>  "woofmeow"
[3, 1] + [5, 2, 1]    =>  [3, 1, 5, 2, 1]
```

- We can use * to mean replication:

```
"woof" * 3   =>  "woofwoofwoof"
["k", 9] * 3   =>  ["k", 9, "k", 9, "k", 9]
```

Ranges

- A range is defined by *first* and *last* elements.

```
r1 = -5..3
r2 = “eel” .. “hippo”

r1.first   =>  -5
r2.last    =>  “hippo”
```


Range Inclusion

- The `include?` method returns a bool:

```
r1 = -5 .. 3                numerical order
r1.include?(4)   =>  false
r1.include?(-4)  =>  true

r2 = “eel” .. “hippo”    alphabetical order
r2.include?(“frog”)     =>  true
r2.include?(“zebra”)    =>  false
```

Successor and Predecessor

- `succ` and `pred` work for integers; only `succ` works for strings:

```
x = 5
x.succ   => 6
x.pred   =>  4

y = "cat"
y.succ   =>  "cau"
y.pred   =>  error!
```


Surprised?

- Ruby's interesting rules for string successor:

```
"99".succ   => "100"

"zz".succ   => "aaa"

"woof99".succ  => "woog00"

"9b9zzz".succ   => "9c0aaa"
```

Arrays

- Arrays can hold any kind of object:

  a = [8, “strawberry”, -5.062, false]

  a[0] => 8 *Ruby numbers items from 0!*
  a[1] => “strawberry”

  a.length => 4

- The empty array is written [ ]

Converting a Range to an Array

r = 3..8
r.to_a => [3, 4, 5, 6, 7, 8]

(8..3).to_a => [ ]

s = “gu” .. “he”
s.to_a => [“gu”, “gv”, “gw”, “gx”, “gy”,
“gz”, “ha”, “hb”, “hc”, “hd”, “he”]

The *to_a method uses succ to generate elements.*

Iteration Review

```
def test1 ()
  for i in 1..5 do
    puts “Woof”
  end
end
```

```
>> test1
Woof
Woof
Woof
Woof
Woof
=> 1..5
```


Iteration Review

```
def test2 ()
  for i in 1..5 do
    puts i
  end
end
```

```
>> test2
1
2
3
4
5
=> 1..5
```

Iteration Review

```
def test3 ()
  for i in 1..5 do
    puts “Woof” * i
  end
end
```

```
>> test3
Woof
WoofWoof
WoofWoofWoof
WoofWoofWoofWoof
WoofWoofWoofWoofWoof
=> 1..5
```


You Can Loop Over Arrays Too

```
fruits = [“apple”, “banana”, “cherry”, “date”]

for f in fruits do
   puts “Yummy ” + f + “ pie!”
end
```

```
Yummy apple pie!
Yummy banana pie!
Yummy cherry pie!
Yummy date pie!
=> [“apple”, “banana”, “cherry”, “date”]
```

Iterators

- Iterators are another way to operate on the elements of an array.
- The `each` iterator is similar to a for loop:

```
fruits.each { |f| puts “Yummy ” + f + “pie!” }
```

This { } thing is called a “block” in Ruby

- Ruby provides lots of other iterators that do cool and useful things.

Compare

Using a for loop:

```
for f in fruits do
  puts “Yummy ” + f
end
```

Using an iterator:

```
fruits.each { |f|
   puts “Yummy ” + f
  }
```

Fun With Iterators

- fruits.**collect** { |f| f.length }
  => [5, 6, 6, 4]

- fruits.**collect** { |f| f.reverse }
  => [“elppa”, “ananab”, “yrrehc”, “etad”]

- fruits.**select** { |f| (“b”..“cz”).include? f }
  => [“banana”, “cherry”]

“Destructive” Iterators

- Some iterators *modify* the array. Beware!

```
items = (1..10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

items.delete_if { |i| i.odd? }
=> [2, 4, 6, 8, 10]

items  =>  [2, 4, 6, 8, 10]
```

Ruby Has Hundreds of Methods, Classes, Keywords, Iterators, Etc.

Too much to remember!

That's why we have online documentation:
http://www.ruby-doc.org

AND NOW FOR SOMETHING COMPLETELY DIFFERENT

What Is a “Sieve” or “Sifter”?

Separates stuff you want from stuff you don’t:

A 2000 year old algorithm (procedure) for generating a table of prime numbers.

2, 3, 5, 7, 11, 13, 17, 23, 29, 31, …

Prime Numbers

- An integer is “prime” if it is not divisible by any smaller integers except 1.

- 10 is **not** prime because 10 = 2 × 5
- 11 **is** prime
- 12 is **not** prime because 12 = 2 × 6 = 2 × 2 × 3
- 13 **is** prime
- 15 is **not** prime because 15 = 3 × 5

Testing Divisibility in Ruby

- *x* is “divisible by” *y* if the remainder is 0

- 15 is divisible by 3 and 5, but not by 2:

```
15 % 3   => 0
15 % 5   => 0
15 % 2   => 1
```

Divisible By Three?

```
def  threezy? (n)
  return (n % 3) == 0
end
```

Equality test

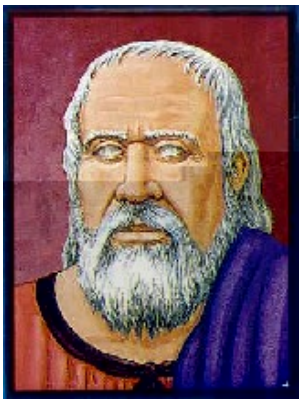
```
threezy?(5)   =>  false
threezy?(6)   =>  true
```


The Sieve of Eratosthenes

Start with a table of integers from 2 to N.

Cross out all the entries that are divisible by the primes known so far.

The first value remaining is the *next* prime.



Finding Primes Between 2 and 50

2 3 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50

2 is the first prime.

Finding Primes Between 2 and 50

2 **3** 4 5 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50

Filter out everything divisible by 2.

Now we see that 3 is the next prime.

Finding Primes Between 2 and 50

2 **3** 4 **5** 6 7 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50

Filter out everything divisible by 3.

Now we see that 5 is the next prime.

Finding Primes Between 2 and 50

2 **3** 4 **5** 6 **7** 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50

Filter out everything divisible by 5.

Now we see that 7 is the next prime.

Finding Primes Between 2 and 50

2 **3** 4 **5** 6 **7** 8 9 10
11 12 13 14 15 16 17 18 19 20
21 22 23 24 25 26 27 28 29 30
31 32 33 34 35 36 37 38 39 40
41 42 43 44 45 46 47 48 49 50

Filter out everything divisible by 7.

Now we see that 11 is the next prime.

Finding Primes Between 2 and 50

2 **3** 4 **5** 6 **7** 8 9 10
11 12 **13** 14 15 16 **17** 18 **19** 20
21 22 **23** 24 25 26 27 28 **29** 30
31 32 33 34 35 36 **37** 38 39 40
41 42 **43** 44 45 46 **47** 48 49 50

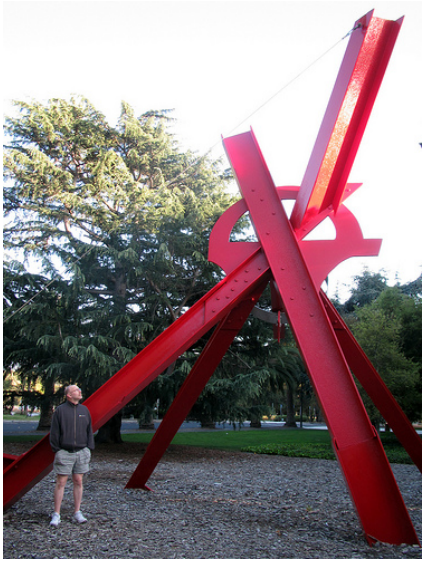
Since 11 × 11 > 50, all the remaining table entries must be prime.

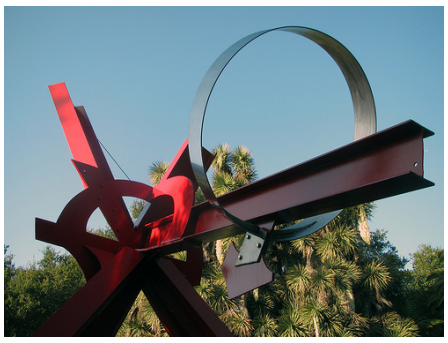
Doing It (Crudely) In Ruby

```
items = (2..50).to_a
items.delete_if { |i| (i>2) & (i%2 == 0) }
items.delete_if { |i| (i>3) & (i%3 == 0) }
items.delete_if { |i| (i>5) & (i%5 == 0) }
items.delete_if { |i| (i>7) & (i%7 == 0) }
```

- What if we wanted a table of the first 1000 primes? How would you automate this?

Algorithm-Inspired Sculpture





The Sieve of Eratosthenes, 1999 sculpture by Mark di Suvero. Displayed at Stanford University.