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

```ruby
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:

  \[
  x = 5 \\
  x.succ \rightarrow 6 \\
  x.pred \rightarrow 4 \\
  \]

  \[
  y = \text{“cat”} \\
  y.succ \rightarrow \text{“cau”} \\
  y.pred \rightarrow \text{error!} \\
  \]

Surprised?

- Ruby’s interesting rules for string successor:

  \[
  \text{“99”}.succ \rightarrow \text{“100”} \\
  \text{“zz”}.succ \rightarrow \text{“aaa”} \\
  \text{“woof99”}.succ \rightarrow \text{“woog00”} \\
  \text{“9b9zzz”}.succ \rightarrow \text{“9c0aaa”} \\
  \]
Arrays

• Arrays can hold any kind of object:

```ruby
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

```ruby
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

```ruby
def test1 ()
  for i in 1..5 do
    puts "Woof"
  end
end

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

Iteration Review

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

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

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

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

You Can Loop Over Arrays Too

```ruby
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:

  ```ruby
  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:**

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

**Using an iterator:**

```ruby
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.

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

Too much to remember!

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 $10 = 2 \times 5$
• 11 is prime
• 12 is not prime because $12 = 2 \times 6 = 2 \times 2 \times 3$
• 13 is prime
• 15 is not prime because $15 = 3 \times 5$

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:
  
  $15 \% 3 \Rightarrow 0$
  $15 \% 5 \Rightarrow 0$
  $15 \% 2 \Rightarrow 1$
Divisible By Three?

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

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

Equality test

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.

Filter out everything divisible by 2.
Now we see that 3 is the next prime.
### Finding Primes Between 2 and 50

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

Filter out everything divisible by 3.

Now we see that 5 is the next prime.
Finding Primes Between 2 and 50

<table>
<thead>
<tr>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>11</strong></td>
<td><strong>12</strong></td>
<td><strong>13</strong></td>
<td><strong>14</strong></td>
<td><strong>15</strong></td>
<td><strong>16</strong></td>
<td><strong>17</strong></td>
<td><strong>18</strong></td>
<td><strong>19</strong></td>
</tr>
<tr>
<td><strong>21</strong></td>
<td><strong>22</strong></td>
<td><strong>23</strong></td>
<td><strong>24</strong></td>
<td><strong>25</strong></td>
<td><strong>26</strong></td>
<td><strong>27</strong></td>
<td><strong>28</strong></td>
<td><strong>29</strong></td>
</tr>
<tr>
<td><strong>31</strong></td>
<td><strong>32</strong></td>
<td><strong>33</strong></td>
<td><strong>34</strong></td>
<td><strong>35</strong></td>
<td><strong>36</strong></td>
<td><strong>37</strong></td>
<td><strong>38</strong></td>
<td><strong>39</strong></td>
</tr>
<tr>
<td><strong>41</strong></td>
<td><strong>42</strong></td>
<td><strong>43</strong></td>
<td><strong>44</strong></td>
<td><strong>45</strong></td>
<td><strong>46</strong></td>
<td><strong>47</strong></td>
<td><strong>48</strong></td>
<td><strong>49</strong></td>
</tr>
</tbody>
</table>

Filter out everything divisible by 7.
Now we see that 11 is the next prime.

Since $11 \times 11 > 50$, all the remaining table entries must be prime.
Doing It (Crudely) In Ruby

```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