Conditional Evaluation

Conditional evaluation
if statements
case statements
Conditional expressions

Definitions

- In the slides on iteration we saw examples loops that did something with every element in an array
- Most loops, however, do some sort of test so that operations are performed on some steps and not others
- skip 0’s when counting elements in an array
- print lines from a file that match a certain pattern
- count the differences between two strings (add 1 to a counter only if corresponding letters are different)
- In each case there are many different ways to write the statement in the body of the loop
- What they all have in common is the general notion of conditional evaluation
  - test whether a certain condition is true
  - if so, evaluate the expression, otherwise skip it

Statement Modifiers

- The simplest way to do a conditional evaluation is to attach a “modifier” to the end of a statement
- A modifier is just the word “if” or “unless” followed by an expression that evaluates to true or false
- Example: counting the nonzero elements in an array

```
>> a = [10, 9, 0, 10, 7, 0, 9]
=> [10, 9, 0, 10, 7, 0, 9]
>> n = 0
=> 0
>> a.each do |x|
  n += 1 if x > 0
end
=> [10, 9, 0, 10, 7, 0, 9]
>> n
=> 5
```

What is True?

- No, we’re not poets or philosophers
  “Beauty is truth, truth beauty,” - that is all
  Ye know on earth, and all ye need to know
- We’re not even logicians
  a proposition is true if it is provable
- As Ruby programmers, we just want to know what will cause a conditional expression to be evaluated
- The two common techniques (illustrated on the following slides) are
  - evaluating a boolean expression to get a value that is true or false
  - testing whether an object exists, i.e. test to see if it is nil
Boolean Expressions

- Boolean algebra is a simple form of logic where expressions evaluate to true or false
- In Ruby, a boolean expression is (usually) an expression using comparison operators like `<`, `>`, `==`, etc
- **Note!** do not use the single `=` sign when testing for equality! (why not?)

```
>> x = 5
=> 5
>> y = 7
=> 7
>> x == y
=> false
>> x < y
=> true
>> b = x < y
=> true
>> b
=> true
```

What is the difference between these two expressions?

```
>> b = x == 7
>> b = x = 7
```

They are objects, and can be stored in variables

Boolean Operators

- You can create more complex expressions using operators for AND, OR, and NOT

```
>> s = "Hello, World!"
=> "Hello, World!"
>> s.length > 0 && s.index(",")) < 10
=> true
>> s.index(",")) < 10 || s.index("!")) < 10
=> true
>> b = (x == 7)
=> false
>> !b
=> true
```

nil

- nil is a special object used to mean “no object” or “nothing”
- It is often used as a return value from a method to indicate a special situation or an exception
- Example: this loop reads lines from the terminal, and ends when the user types the “end of stream” character (typically ^D):

```
while s = gets
  puts "input: " + s
end
```

- The same loop, except reading from a file (but not the best way to read every line):

```
f = File.open("data.txt")
while s = f.gets
  puts "input: " + s
end
f.close
```

nil (cont’d)

- Another example: the index method returns nil if the string does not contain the character it is looking for:

```
=> s = "Hello, World!"
>> s.index("")
=> nil
```

- This is a better way to check if a character occurs before the 10th location in a string:

```
>> s.index(",") && s.index(",")) < 10
=> true
```

- Why is this better? Look what happens if you try to compare s.index(x) with 10 when x is not in the string:

```
>> s.index("x") < 10
NoMethodError: undefined method `<' for nil:NilClass
```
if Statements

- The previous examples using if are fine when there is only one thing to do when the condition evaluates to true
- When you want to do more than one action you'll want to use an if statement

```ruby
while s = gets
  puts s
  nwords += s.split.length
  nchars += s.length
end
```

- The statement looks a lot like a while statement:
  - “if” and an expression on the first line
  - any number of statements,
  - last line has the word end

- split is the opposite of join -- it breaks s at word boundaries

Note: one end for while, another for if

Nested ifs and elsens

- A common situation is where there is a series of tests
  - example: print a string depending on the GPA

```ruby
if gpa >= 3.9
  puts "dean's list"
else
  if gpa >= 3.5
    puts "department honors"
  else
    if gpa >= 3.0
      puts "burgerville"
    else
      puts "grad school"
    end
  end
end
```

- This kind of code is hard to read and very hard to maintain (imagine adding another case, e.g. gpa > 2.8)

else

- An if statement can have an else clause
  - the code following else is evaluated if the expression is false
- Example: the formula for BMI (project 2) is \( 703 \times \frac{w}{h^2} \)
  - the constant 703 is used when units are inches and pounds
  - for the metric system (meters and kilograms) the constant is 1
  - you can add a “-m” switch to your command line to tell the program the units are metric

```ruby
if ARGV[0] == "-m"
  bmi = w / h**2
else
  bmi = 703 * w / h**2
end
```

- Note there is only one end for the entire statement

elsif

- For situations like this, Ruby allows an elsif clause for an if statement
- The same example is much easier to understand when written with elsif:

```ruby
if gpa >= 3.9
  puts "dean's list"
elsif gpa >= 3.5
  puts "department honors"
elsif gpa >= 3.0
  puts "burgerville"
else
  puts "grad school"
end
```

- Note there is only one end for the entire statement
case Statements

- Here is part of the code from the gpa program
- it assigns a numeric value for a grade

```ruby
if grade == "A"
  val = 4
elsif grade == "B"
  val = 3
elsif ...
end
```

- Note that every boolean expression has the same form
- they all test to see if a variable (named grade) has a particular value

case Statements (cont’d)

- For situations like this the code is often more readable (and more reliable) if it is written with a case statement

```ruby
case grade
  when "A": val = 4
  when "B": val = 3
  when "C": val = 2
  else val = 0
end
```

- the statement has an expression that evaluates to some type (e.g. integer, string, ...)
- a series of when clauses have labels of the same type
- the program finds the first when clause with a label that matches the value of the expression
- the code immediately following the matching label is evaluated
- any number of expressions can be given after a label

Conditional Expressions

- Recall this example from a previous slide:

```ruby
if ARGV[0] == "-m"
  bmi = w / h**2
else
  bmi = 703 * w / h**2
end
```

- The only thing being done in each branch of the if statement is an assignment, and the target of the assignment is the same in each case
- A better way to write this code is with an if expression:

```ruby
bmi = if ARGV[0] == "-m" then w / h**2 else 703 * w / h**2 end
```

Conditional Expressions (cont’d)

- A short-hand notation uses ? and : instead of if and else

```ruby
bmi = if ARGV[0] == "-m" then w / h**2 else 703 * w / h**2 end
bmi = (ARGV[0] == "-m") ? (w / h**2) : (703 * w / h**2)
or even better yet
bmi = (ARGV[0] == "-m" ? 1 : 703) * w / h**2
```

- Why are these “better” than the original?
- it makes it clear that this statement is assigning a value to a variable
- the reader’s attention is drawn to the variable name at the beginning of the line
- the final form emphasizes the similarities between the two cases
- it’s always a good idea to “factor out” the common code and write it once
A similar construct can be used when there are several different options.

```ruby
case grade
  when "A": val = 4
  when "B": val = 3
  when "C": val = 2
  when "D": val = 1
  else: val = 0
end
```