Program Structure

- Defining your own methods
- Design strategies

Context

- At this point in the term you have a working knowledge of Ruby
  - you know everything you need to write a complete program
- Main concepts in Ruby:
  - **variables** are references to **objects**
  - the value of a variable is set by an assignment statement that evaluates an **expression**
  - an object is an instance of a **class**
  - an object's class determines what you can do with it
  - operations on objects are defined by **methods**
- The topics for the next three weeks will help you become a more effective programmer
  - organizing your program so it is easier to write, easier to maintain
  - interacting with the rest of the world (3 Rs: readin’, writin’, runnin’)
  - testing and debugging strategies
- All of these will be presented along with many more examples of Ruby code

Program Organization

- Large programs need to be broken down into smaller pieces
- The smaller parts have lots of different names
  - subroutines
  - modules
  - procedures
  - methods
- In each case the goal is the same:
  - complex programs are much easier to write if you can break the task into smaller, more manageable pieces
  - the programs are also easier to read and maintain
- You will be reading your own code -- give yourself a break and structure your large programs as collections of smaller pieces
  - you will need to change your programs when there are new releases of the language or the OS
  - plan on spending time to remember what you were thinking when you wrote the program the first time

A Big Program

- As an example of a big program consider a bioinformatics project
  - a scientist has a large collection of sequence data in a text file
  - each sequence is identified by a unique name
  - the project starts by scanning the file to locate two sequences by name
  - next the sequences are “aligned” to show regions of similarity
  - the program counts the differences in the aligned sequences
  - If this is a command line program, it might be run as
    ```bash
    % compare.rb acidocaldaris_solfataricus ef1a.fa
    ```

Aside: FASTA Files

- A common format for sequences files is known as FASTA
  - pronounced “fast-A” or “fast-uh”
- Each sequence has a definition line (“defline”) that starts with a > symbol
- The sequence starts on the next line
- Example: efla.fa
  >Sulfolobus.acidocaldaris
  STLGRLLMDRGFLDEKTVKEAAEAAKLGDEKSYAFLM
  DRLKEERERGVTINLTFMRFETKTTYFTVIDAPGHRDFVK
  ...
  >Sulfolobus.tokodaii
  STLGRLLMDRGFLDEKTVKEAAEAAKLGDEKSYAFLL
  DRLKEERERGVTINLTFMRFETKTTYFTVIDAPGHRDFVK
  ...
  >Sulfolobus.solfataricus
  STLGRLLMDRGFLDEKTVKEAAEAAKLGDEKSYAFLL
  DRLKEERERGVTINLTFMRFETKTTYFTVIDAPGHRDFVK
  ...

Outline of compare.rb

- This is a "sketch" of the main parts of the program:
  read ids from ARGV
  while lines remain in file
    read a defline
    while line is not a defline
      read sequence letters
      save sequence as seq1 if the defline matches id1
      save sequence as seq2 if the defline matches id2
      make a “dynamic programming matrix" object
      initialize the top row and left column of the matrix
      for each row j
          for each col i
              compute cost of aligning seq1[j] with seq2[i]
      for i = 0..seq1.length
          compare seq1[i] to seq2[i]
      print results

Structure of compare.rb

- The complete program will probably be over 100 lines of Ruby code
  - 200 lines if you add comments (notes to yourself and others)
  - in Ruby comment line begin with #
- This program won’t be too hard to write or understand (as long as you know the basic algorithms involved)
- But consider what will happen over the next few months and years:
  - you might want to change the alignment parameters (add more code to the middle step)
  - the method works so well you want it to do many more pairs
  - compare.rb pairs.txt efla.fa
    - read pairs of ids from pairs.txt, do the alignment and compare steps for each pair
  - this new version puts almost the entire program inside a new while loop
  - you might want to add some code that summarizes differences between pairs
  - you decide you need to create a phylogenetic tree for all the sequences
- The program will become much harder to maintain and extend

Spaghetti

- The technical term for programs that are just one big pile of statements that hop all over the place is spaghetti code
- To understand what these programs do you need to mentally trace the flow of execution
- The term was coined for old programs written in low-level machine languages
- It’s possible to write spaghetti code in any language
Better Structure for compare.rb

- From the very start of this project the code would be improved if the programmer broke it into smaller pieces.
- Write methods for each piece:
  ```ruby
  def read_sequences
  ...
  end
  def align
  ...
  end
  def compare
  ...
  end
  read_sequences
  align
  compare
  ```

Motivation

- Is this new organization really any better?
  - it has all the same Ruby expressions as the original
  - it has additional new code (def..end) and will be longer than the original
- In the new organization it will be easier to focus on parts of the code:
  - if you want to change the alignment algorithm you know just where to start.
  - if you suspect there is a bug in the code that reads sequences you know where to search.
  - if you want to process several sequences it’s easier to create the loop:
    ```ruby
    read_sequences
    while pair = read_pair_names
    align
    compare
    end
    ```

Parameters

- Most methods have **parameters**
  - values passed into the method from the code that calls the method.
- To tell Ruby your method takes parameters, just list the parameter names in parentheses after the method name:
  ```ruby
  def sieve(n)
  ...
  end
  ... puts sieve(100)
  ```

Parameters (cont’d)

- Parameters are names of **local variables**
  - these variables are created when the method is called.
  - they exist only when the expressions in the body of the method are evaluated.
  - they are deleted when the method completes:
    ```ruby
    def sieve(n)
    ...
    end
    ... puts sieve(100)
    ```
Return Values

- Methods often create values that should be returned to the code that calls the method.
- Example: the sieve method should return the list of prime numbers.

```ruby
def sieve(n, mode = nil)
  worklist = (2..n).to_a
  primes = []
  while worklist.first < Math.sqrt(n)
    primes << worklist.first
    worklist.delete_if { |x| x % primes.last == 0 }
    puts primes.inspect + " " + worklist.inspect if mode == :trace
  end
  return primes + worklist
end
```

p primes(100)

Important Aside: Copy vs Reference

- Ruby keeps track of all the variables you define.
  - it uses a large container sometimes referred to as the "object store".
- Each time a new variable is defined, Ruby makes an entry in the storage area.
  - think of the value of a variable as a "pointer" to the object that is the variable's value.
  - the pointer is a reference to the variable.

```ruby
def test(str)
  str.gsub!("l","*")
end
```

```
> x = 26
=> 26
> s = "hello"
=> "hello"
> test(s)
```

Note: in this example, the variable `str` is erased from the object store when the method exits.

Copy vs Reference (cont'd)

- When you assign a new variable based on the value of an existing variable, Ruby copies the reference.
  - it does not copy the variable itself.
- Here's an experiment (using irb) that verifies this claim.
  ```ruby
  >> x = 26
  => 26
  >> s = "hello"
  => "hello"
  >> str = s
  => "hello"
  >> str.gsub!("l","*")
  => "he**o"
  >> str
  => "he**o"
  >> s
  => "he**o"
  ```

s and str are aliases of the same object

Copy vs Reference (cont’d)

- The same thing happens when you pass a variable as a parameter to a method.
  - Ruby passes a reference to the variable.
    - it does not copy the object itself.
```
def test(str)
  str.gsub!("l","*")
end
```

```ruby
s = "hello"
test(s)
puts s
```

Note: in this example, the variable `str` is erased from the object store when the method exits.

s and str are aliases of the same object
Clones

There are times when you want to make a copy of a variable
- e.g. copy a piece of “DNA” before mutating it
- pass a copy to a method so the original is preserved
Every class has a method named clone
It is a constructor that makes a new object that is an exact copy of an existing object

```ruby
>> s = 'hello'
=> 'hello'
>> str = s.clone
=> 'hello'
>> str.gsub!('l','*')
=> 'he**o'
>> s
=> 'hello'
```

Many bugs in programs can be traced to wrong assumptions about when a variable is an alias vs when it is a copy
- sprinkle `puts` or `printf` in your code, before and after method calls, to see what’s going on
- you may find something is being modified when you didn’t expect it
Most, but not all, methods in predefined classes have `@` in their names when they modify a variable

```ruby
>> a = ['one', 'two', 'three']
=> ['one', 'two', 'three']
>> a.sort
=> ['one', 'three', 'two']
>> a
=> ['one', 'two', 'three']
>> a.sort!
=> ['one', 'three', 'two']
>> a
=> ['three', 'two']
```

Parentheses

In many cases you can leave out the parentheses surrounding parameter values when you call a method
The expressions on the left are equivalent to the ones on the right:

```ruby
>> puts("Hello")
Hello
>> puts "Hello"
Hello

>> s.gsub("l","*")
=> "He**o"
>> s.gsub!("l","*")
=> "He**o"

>> sieve(100)
=> [2,3,...]
>> sieve 100
=> [2,3,...]
```

Parentheses (cont’d)

Making parentheses optional is something Ruby borrowed from Perl
- I’m not sure I approve
- Leaving out parentheses can make an expression ambiguous
  - this usually leads to a syntax error, but maybe not the one you expected
  ```ruby
  >> s.gsub("l","*") .length
  => 5
  >> s.gsub("l","*") .length
  TypeError: can’t convert Fixnum into String
  ```
- My advice:
  - it’s OK to leave out parentheses when an expression uses an “idiom”
  - examples: parameters to `puts`, `load`, and other kernel methods that have one argument
  - do whatever seems more natural to you -- it’s your program
  - be consistent (within any one program)
Examples

- The next set of slides includes several examples of methods
- The goal for these slides:
  - show examples of pieces of code that are good candidates to put in methods
  - introduce a few new Ruby constructs (e.g. “parallel assignment”)
  - show many more examples of loops, conditionals, and other Ruby constructs introduced in the last two weeks
- The programs here will be available via the class web site
  - except for one or two small ones which I’ll want you to type in yourself as part of the homework project

Example: gpa

```ruby
#! /usr/bin/env ruby
# gp(s) is the number of points for a letter grade s

def gp(s)
  val = case s
  when "A": 4
  when "B": 3
  when "C": 2
  when "D": 1
  when "F": 0
  else
    nil
  end
  puts "gp(#{s}) = #{val}"
  return val
end
```

This method is part of the grade point average program.
The rest of the program is on the next slide.

Example: gpa (cont’d)

```ruby
sum = 0.0
ARGV.each do |grade|
  sum += gp(grade)
end
puts "sum = " + sum.to_s

gpa = sum / ARGV.length
puts "gpa = " + gpa.to_s
```

The gp method (previous slide) is an example of abstraction

As I wrote this program I thought “if I had a function that mapped letter grades to grade points the rest of this program would be very straightforward”

Example: factorial

```ruby
#! /usr/bin/env ruby
# Calculates, prints the factorial of N

def factorial(n)
  f = 1
  for i in 2..n
    f *= i
  end
  return f
end

abort "Usage: factorial.rb N" unless ARGV.length == 1
n = ARGV.shift.to_i
f = factorial(n)
puts "#{n}! = #{f}"
```

Recall from math classes that

A common idiom: test at the beginning of the program to make sure the command line arguments are acceptable
Example: choose

- Now that we have a method for computing factorials, we can use it to make a method for computing combinations:

\[
\binom{n}{k} = \frac{n!}{k!(n-k)!}
\]

```ruby
# Compute the number of ways of choosing k out of n items
# (unordered and without repetition)

def choose(k, n)
    return factorial(n) / (factorial(k) * factorial(n-k))
end
```

Abort "Usage: choose.rb K N" unless ARGV.length == 2

k = ARGV.shift.to_i
n = ARGV.shift.to_i
puts "There are #{choose(k,n)} ways to choose #{k} out of #{n} items"

Example: diffs

```ruby
#! /usr/bin/env ruby
# Count the number of differences between two sequences

def diffs(seq1, seq2)
    ndiffs = 0
    seq1.length.times do |i|
        if seq1[i] != seq2[i]
            ndiffs += 1
        end
    end
    return ndiffs
end

gene1 = gets
gene2 = gets
abort "Error: genes must be the same length" unless gene1 && gene2 && gene1.length == gene2.length
puts "Number of differences: #{diffs(gene1, gene2)}"
```

Example: compare.rb

- This next example shows the first few steps in the development of compare.rb, the hypothetical program that will analyze differences in pairs of sequences

- Outline of this version:
  - read a set of sequences, save them in a Hash object named `seq` (key = sequence id)
  - each line in the input file will have two ids, `x` and `y`
  - align `seq[x]` and `seq[y]`
  - count the differences in the aligned sequences

- The command line that invokes this program will look something like this:

  ```
  % compare.rb pairs.txt ef1a.fa
  ```

- The first argument is the name of the file with names of sequences to compare, the second is the (FASTA) file with the sequence data

```ruby
#!/usr/bin/env ruby

def read_sequences
    puts "reading sequences..."
end

def align
    puts "aligning..."
end

def diffs
    puts "counting differences..."
end

read_sequences
align
diffs
```

Example: compare.rb (cont’d)

- The first step is to create the file for the application

- Copy hello.rb or some other simple program, change it so it has one method for each of the three main operations:

```ruby
#!/usr/bin/env ruby

It may seem silly, but take the time to run this program.

It’s very easy to make a simple mistake like forgetting the end of a method definition or a closing quote.

Find and fix these mistakes now -- they’re annoying and distracting to fix when you’re working on other things later.

```
The next step is to decide how to represent sequences

The outline says we’ll use a hash container to hold each sequence

Somewhere in read_sequences we’ll see something like this:

```ruby
h = Hash.new
...  
  h[x] = <...>
...  
return h
```

The assignment shows that we plan on adding a new item to the hash, using the id of a sequence

What we haven’t figured out yet is what the sequence itself will look like, i.e. what goes between the < and >

A common technique: use a hash for the sequences, also

Use key/value pairs for the main attributes of a sequence

Example:
```ruby
>> seq1 = {:def => "Sulfolobus.acidocaldaris", :seq => "STLIGRLL"}
>> seq2 = {:def => "Sulfolobus.tokodaii", :seq => "STLVGRLL"}
>> seq1[:def]
=> "Sulfolobus.acidocaldaris"
>> seq1[:seq]
=> "STLIGRLL"
>> h = Hash.new
=> {}
>> h["acidocaldaris"] = seq1
>> h["acidocaldaris"][:def]
=> "Sulfolobus.acidocaldaris"
Recall from the slides on hashes things that start with a colon (:) are symbols
```

At this point you have a choice -- you can work on any of the three main components

or if you’re working in a group you can divide up the tasks and work on them in parallel

How can you work on the alignment or difference counting method if you haven’t read the sequences?

write a stub -- a short temporary method that is a place-holder for the real method

Why would you do this?

the stub is a way of introducing test data -- short sequences where you know what the final answer will be

you may be concerned about the later two steps, and want to work on them first -- if they won’t work there is no reason to spend time getting the others working

boredom -- you really want to get going on the other steps

Here is a stub for the read_sequences method

It returns a test data set with two short sequences:

```ruby
def read_sequences
  h = Hash.new
  h["acidocaldaris"] = { :def => "Sulfolobus.acidocaldaris", :seq => "STLIGRLL" }
  h["tokodaii"] = { :def => "Sulfolobus.tokodaii", :seq => "STLVGRLL" }
  return h
end
```

Note these two sequences are already aligned (each has 8 letters) and there is only one difference (I vs V)
Example: compare.rb (cont’d)

- Next add some code to the main program to save the hash created by the call to `read_sequences`
- At this point we should also add some code that verifies we have a hash object with the expected contents
- Here is the new code for the main part of the program:

```ruby
seqs = read_sequences
puts "read sequences: #{seqs.keys.sort.join(" ")}"
puts seqs["acidocaldaris"]["seq"]
align
diffs
```

Example: compare.rb (cont’d)

- And here is the output when we test the new version:

```bash
% compare.rb pairs.txt ef1a.fa
read sequences: acidocaldaris, tokodaii
STLIGRLL
STLIGRLL
aligning...
counting differences...
```

- Note: even though the program isn’t checking the command line arguments yet it’s a good habit to include them when we run the tests
- One of the next steps will be to verify there are two command line arguments and then add code that verifies there are files with these names and print an error message if the files don’t exist

Example: compare.rb (cont’d)

- The current to-do list for this project
  1. check there are two command line arguments
  2. make sure the arguments are names of files that can be opened
  3. read FASTA sequences, store in a hash
  4. implement the alignment algorithm
  5. copy the diffs method from diffs.fa, adapt it for this application

- There are some dependences (e.g. it would be best to complete 1 and 2 before 3) but otherwise work on these according to your interest and energy level
- One of the homework projects: do step 5

Example: overlap

- A common operation in bioinformatics is to test if two items overlap
- We may be given the locations of two “features” (e.g. genes) on complementary strands of DNA
  - example: the pairs labeled b and c overlap, but pairs a and d do not:

```
  X
  a b c d
  Y
```

- In another situation we may be given the coordinates of a gene and want to know if it lies within a certain region
Example: overlap (cont’d)

- To define a boolean expression for overlaps, it might be easier to first ask the opposite question: when are two items nonoverlapping?
- notation: $X_s$ is “start of $X$”, $X_e$ is “end of $X$”, $Y_s$ is “start of $Y$”, $Y_e$ is “end of $Y$”
- $X$ and $Y$ do not overlap if $Y_e < X_s$ (case a) or $X_e < Y_s$ (case d)

From DeMorgan’s law (a fundamental theorem of boolean algebra):
- $\neg (P \lor Q) = \neg P \land \neg Q$
- $X$ and $Y$ overlap if $Y_e > X_s$ and $X_e > Y_s$

Why do we need the “and” here? Can we test just one condition?

Example: overlap (cont’d)

- Implementation in Ruby:

```ruby
# X and Y overlap if Y_e > X_s and X_e > Y_s
def overlap(xe, ye, xs, ys)
    return ye > xs && xe > ys
end
```

Example: overlap (cont’d)

- The original justification I gave for writing your own methods was to break large programs into manageable pieces
- This example shows another reason: abstraction
- This is a very short method -- just one line -- but compare these two programs

```ruby
xs, xe = loc(x)
ys, ye = loc(y)
if ye > xs && ye > ys
    puts "skipped..."
else
    process(x, y)
end
```

Claim: the one on the right is no harder to write, and much easier to understand

Which would you rather come back to in two months?

Example: permute

- A permutation of a list is a re-ordering of the elements in the list
- for strings of letters, a permutation is an anagram
- Many applications need random permutations of lists or strings
- A simple (but not very elegant) way to make a random permutation is to exchange each element with a random element to its right
- In “pseudo-code” notation, to permute a string $a$ with $n$ elements:

```
for i = 0 to n-2
    j = a random number between i and n-1
    exchange a[i] with a[j]
```
Example: permute (cont’d)

- A Ruby construct that will be very helpful when we write this method: parallel assignment
  ```ruby
  >> a, b = 1, 2
  => [1, 2]
  >> a
  => 1
  >> b
  => 2
  >> b, a = a, b
  => [1, 2]
  >> b
  => 1
  >> a
  => 2
  ```

Parallel assignment gives us a way to exchange the values of two variables.

Example: permute (cont’d)

- The last time we used `rand` it was to generate a random number between 0 and 1
  ```ruby
  >> rand(10)
  => 3
  >> rand(10)
  => 6
  >> rand(10)
  => 9
  >> rand(10)
  => 9
  >> rand(10)
  => 7
  ```

- A variation that takes a single integer parameter `n` returns a random integer between 0 and `n-1`
  ```ruby
  >> rand(10)
  => 3
  >> rand(10)
  => 6
  >> rand(10)
  => 9
  >> rand(10)
  => 9
  >> rand(10)
  => 7
  ```

Example: permute (cont’d)

```ruby
def permute(s)
    str = s.clone
    (str.length-2).times do |i|
        r = rand(str.length-i) + i
        # i <= r < length
        str[i], str[r] = str[r], str[i]
    end
    return str
end
```

```ruby
> load "permute.rb"  # A more idiomatic way to write
> s = "california"  # this is to use a "for loop" and
>                  # a range:
> permute(s)  # for i in 0..str.length-2
> "california"
> permute(s)  # ...
> "clfaoarini"
> permute(s)  # end
> "oarfnilcia"
> permute(s)  # for i in 0..str.length-2
> "lriaifcona"
```

Design

- Defining new methods for your programs is where a sense of design or programming style is important
- A good design is what separates an elegant program from spaghetti code
- You will develop your own style over time, with experience
- I will try to give rules of thumb and suggestions, but feel free to experiment
- But if you deviate from standards or idioms document what you do
- Most of what I’m looking for in your program documentation will be a high-level description of the main modules and how they interact
Design Strategies

- Use a top-down approach, at least initially
  - think of the main parts of the program, write an outline, implement stubs for main pieces as you start development
  - if necessary, repeat the top-down design on the major parts
- KISS (keep it simple)
  - if the program gets to be more than N lines, break it into pieces
  - for me, N < 10
- “Refactor”
  - if you find yourself writing a piece of code that is the same, or very similar to, something you’ve already done turn it into a method
  - example: looking for overlapping segments of DNA
  - the first time it may not seem complex enough to warrant making a method
  - very frustrating to fix a bug once, but forget the same logic was used elsewhere

Design Strategies (cont’d)

- Use local variables
  - this will need more explanation, when we talk about global variables (variables accessible to all methods)
  - global variables have a place, but if over-used are a good source of bugs
- Avoid side effects
  - the best approach is to write functions: a method that gets its input from parameters, computes something, and returns the result via a return statement
  - there are times when modifying a parameter is a good approach, but use it sparingly
  - another type of side effect is modifying a global variable

refactor: CS jargon, means “restructure the program, but don’t change the interface”, i.e. I/O is identical