Freestanding Methods

Basics

Where's the class?

Domain and range in Ruby

Varying numbers of arguments
Method definition

Here is a Ruby version of a simple method:

```ruby
def double(x)
    return x * 2
end
```

The keyword `def` indicates that a method definition follows. Next is the method name. The parameter list follows.

If the end of a method is reached without encountering a `return`, the value of the last expression becomes the return value. Here is an equivalent definition:

```ruby
def double(x)
    x * 2
end
```

If no arguments are required, the parameter list can be omitted

```ruby
def hello
    puts "Hello, world!"
end
```
Method definition, continued

One way to get a method into irb is to use load:

```ruby
% cat double.rb
def double(x)
  x * 2
end

% irb --prompt simple
>> load "double.rb"
=> true

>> double(5)
=> 10
```
Method definition, continued

Alternatively, we can type a definition directly into irb:

```ruby
% irb
irb(main):001:0> def double(x)
  x * 2
end
=> nil

irb(main):004:0> double(5)
=> 10

irb(main):005:0>
```

Note that `irb` was run without "--prompt simple". The default prompt includes a line counter and a nesting depth.
If `double` is a method, where's the class?

You may have noticed that even though we claim to be defining a method named `double`, there's no class in sight.

*In Ruby, methods can be added to a class at run-time.* A freestanding method defined in `irb` or found in a file is associated with an object referred to as "main", an instance of `Object`. At the top level, the name `self` references that object.

```ruby
>> [self.class, self.to_s]
=> [Object, "main"]          # The class of self and a string representation of it.

>> methods_b4 = self.methods
=> ["methods", "popb", ...lots more...]  

>> def double(x); x * 2 end
=> nil

>> self.methods - methods_b4
=> ["double"]
```

We can see that `self` has one more method (`double`) after `double` is defined.
Domain and range in Ruby

For reference:

```ruby
def double(x)
    x * 2
end
```

For the ML analog of `double` the domain and range are the integers. (int -> int)

What is the domain and range of `double` in Ruby?
Domain and range in Ruby, continued

Problem: Write a method `polysum(L)` that produces a "sum" of the values in `L`.

Examples:

```ruby
>> polysum([1,3,5])
=> 9

>> polysum([1.1,3.3,5.5])
=> 9.9

>> polysum(["one", "two"])  
=> "onetwo"

>> polysum(["one", [2,3,4], [[1],[1..10]]])
=> ["one", 2, 3, 4, [1], [1..10]]
```

How can we describe the domain and range of `polysum`?
Varying numbers of arguments

Unlike some scripting languages, Ruby considers it to be an error if the wrong number of arguments is supplied to a routine.

```ruby
def wrap(s, wrapper)
    wrapper[0,1] + s + wrapper[1,1]
end

>> wrap("testing", "<>")
=> "<testing>"

>> wrap("testing")
ArgumentError: wrong number of arguments (1 for 2)

>> wrap("testing", "<", ">")
ArgumentError: wrong number of arguments (3 for 2)
```

Contrast: Icon supplies `&null` (similar to Ruby's `nil`) for missing arguments. Extra arguments are ignored.
Varying numbers of arguments, continued

Ruby does not allow the methods of a class to be overloaded. Here's a Java-like approach that **DOES NOT WORK**:

```ruby
def wrap(s)
    wrap(s, "()")
end

def wrap(s, wrapper)
    wrapper[0,1] + s + wrapper[1,1]
end
```

The imagined intention is that if `wrap` is called with one argument it will call the two-argument `wrap` with "()" as a second argument.

In fact, the second definition of `wrap` simply replaces the first. (Last `def` wins!)

```ruby
>> wrap "x"
ArgumentError: wrong number of arguments (1 for 2)

>> wrap("testing", "[ "]")
=> "[testing]"
```
Varying numbers of arguments, continued

There's no intra-class method overloading but Ruby does allow default values to be specified for arguments:

```ruby
def wrap(s, wrapper = "()")
    wrapper[0,1] + s + wrapper[1,1]
end

>> wrap("x", "<>")
=> "<x>"

>> wrap("x")
=> "(x)"
```

W. H. Mitchell (whm@msweng.com)
Varying numbers of arguments, continued

Any number of defaulting arguments can be specified. Imagine a method that creates a window:

```ruby
def make_window(height = 500, width = 700,
                 font = "Roman/12", upper_left = 0, upper_right = 0)
  ...
end
```

A variety of calls are possible. Here are some:

```ruby
make_window
make_window(100, 200)
make_window(100, 200, "Courier/14")
```

Here's something that **DOES NOT WORK**:

```ruby
make_window( , , "Courier/14")  Leading arguments can't be omitted!
```
Sidebar: A study in contrast

Different languages approach overloading and default arguments in various ways. Here's a sampling:

Java    Overloading; no default arguments
C++     Overloading and default arguments
Ruby    No overloading; default arguments
Icon    No overloading; no default arguments; use an idiom

Here is wrap in Icon:

```
procedure wrap(s, wrapper)
    /wrapper := "()"   # if wrapper is &null, assign "()" to wrapper
end
```
Varying numbers of arguments, continued

It can be useful to have a method take an arbitrary number of arguments. `printf` is a good example.

Here's a Ruby method that accepts any number of arguments and simply prints them:

```ruby
def showargs(*args)
    # a...b is a to b-1
    printf("%d arguments:\n", args.size)
    for i in 0...args.size
        printf("#%d: %s\n", i, args[i])
    end
end
```

If a parameter is prefixed with an asterisk, a list is made of any remaining arguments.

```ruby
>> showargs(1, "two", 3.0)
3 arguments:
#0: 1
#1: two
#2: 3.0
```
Varying numbers of arguments, continued

Problem: Modify `polysum` so that this works:

```ruby
>> polysum(1,2,3,4)       # Instead of polysum([1,2,3,4])
=> 10
```

Problem: Write a method `printf0` that's like `printf` but simply interpolates argument values as a string (use `to_s`) where a percent sign is found:

```ruby
>> printf0("x = %, y = %, z = %\n", 10, "ten", "z")
x = 10, y = ten, z = z
=> 23

>> printf0("testing\n")
testing
=> 8
```
Varying numbers of arguments, continued

Sometimes we want to call a method with the values in a list:

```ruby
>> def add(x,y) x + y end

>> pair = [3,4]

>> add(pair[0], pair[1])
=> 7
```

Here's an alternative:

```ruby
>> add(*pair)
=> 7
```

In a method call, prefixing a list value with an asterisk causes the list values to be substituted for the parameters.

Speculate: What will be the result of `add(*[1,2,3])`?
Varying numbers of arguments, continued

Recall `make_window`:

```ruby
def make_window(height = 500, width = 700, font = "Roman/12", upper_left = 0, upper_right = 0)
    ...printf to echo the arguments...
end
```

Results of list-producing methods can be passed easily to `make_window`:

```ruby
>> where = get_loc(...whatever...)
=> [50, 50]

>> make_window(100, 200, "Arial/8", *where)
make_window(height = 100, width = 200, font = Arial/8, at = (50, 50)

>> win_spec = get_spec(...whatever...)
=> [100, 200, "Courier/9"]

>> make_window(*win_spec)
make_window(height = 100, width = 200, font = Courier/9, at = (0, 0)
```

Speculate: Will `make_window(*[300,400], "x", *[10,10])` work?