Ruby
Our topic sequence:

• Functional programming with Haskell (Done!)

• Imperative and object-oriented programming using dynamic typing with Ruby

• Logic programming with Prolog

• Whatever else in the realm of programming languages that we find interesting and have time for.
Introduction
whm wrote:
> I ran into John Cropper in the mailroom a few minutes ago. He said
> he was out at your place today and that you're doing well. I
> understand you've got a meeting coming up regarding math in your
> weaving book -- sounds like fun!?

Hi, William

I'm doing well in the sense of surviving longer than expected. But
I'm still a sick person without much energy and with a lot of pain.
>
> My first lecture on Ruby is tomorrow. Ruby was cooked up by a
> Japanese fellow. Judging by the number of different ways to do the
> same thing, I wonder if Japanese has a word like "no".

Interesting. I know nothing about Ruby, but I've noticed it's
getting a lot of press, so there must be something to it.
What is Ruby?

"A dynamic, open source programming language with a focus on simplicity and productivity. It has an elegant syntax that is natural to read and easy to write." — ruby-lang.org

Ruby is commonly described as an "object-oriented scripting language".
• I don't like the term "scripting language"!
• I describe Ruby as a dynamically typed object-oriented language.

Ruby on Rails, a web application framework, has largely driven Ruby's popularity.

Ruby was invented by Yukihiro Matsumoto ("Matz"), a "Japanese amateur language designer", in his own words.
Here is a second-hand excerpt of a posting by Matz:
"Well, Ruby was born on February 24, 1993. I was talking with my colleague about the possibility of an object-oriented scripting language. I knew Perl (Perl4, not Perl5), but I didn't like it really, because it had smell of toy language (it still has). The object-oriented scripting language seemed very promising."

Another quote from Matz:
"I believe that the purpose of life is, at least in part, to be happy. Based on this belief, Ruby is designed to make programming not only easy but also fun. It allows you to concentrate on the creative side of programming, with less stress. If you don’t believe me, read this book [the "pickaxe" book] and try Ruby. I’m sure you’ll find out for yourself."
ISO/IEC 30170:2012 is an international standard for Ruby but the language is effectively defined by MRI—Matz' Ruby Implementation.

The most recent stable version of MRI is 2.5.0.

On lectura we'll use rvm (the Ruby Version Manager) to run version 2.2.4.

macOS from Mavericks through Sierra has Ruby 2.0.0. High Sierra has Ruby 2.3.3.

The last major upheaval in Ruby occurred between 1.8 and 1.9. (2007)

In general, there are few incompatibilities between 1.9.3 (2011) and the latest version.

   Notable: In 2.4, Fixnum and Bignum were unified to Integer.
Resources

The Ruby Programming Language by David Flanagan and Matz
– Perhaps the best book on Safari that covers 1.9 (along with 1.8)
– I'll call it "RPL".

Programming Ruby 1.9 & 2.0 (4th edition): The Pragmatic Programmers' Guide by Dave Thomas, with Chad Fowler and Andy Hunt
– Known as the "Pickaxe book"
– $28 for a DRM-free PDF at pragprog.com.
– I'll call it "PA".
– First edition is here: http://ruby-doc.com/docs/ProgrammingRuby/

O'Reilly Safari has:
• Many relatively new Ruby books
  One recommendation: The Ruby Way, 3rd edition by Hal Fulton
• Lots of books on Ruby on Rails
• Lots of pre-1.9 Ruby books
ruby-lang.org
  • Ruby's home page

ruby-doc.org
  • Documentation

  • Here's a sample URL, for the **String** class in 2.2.4:
    [http://ruby-doc.org/core-2.2.4/String.html](http://ruby-doc.org/core-2.2.4/String.html)

  • Suggestion: Create a Chrome "search engine" named rc ("Ruby class") with the following URL template:
    [http://www.ruby-doc.org/core-2.2.4/%s.html](http://www.ruby-doc.org/core-2.2.4/%s.html)
    (See [http://cs.arizona.edu/~whm/o1nav.pdf](http://cs.arizona.edu/~whm/o1nav.pdf))
Getting and running Ruby
Getting Ruby for OS X

Ruby as supplied by Apple with recent versions of macOS, should be fine for our purposes.

I installed Ruby 2.2.0 on my Mac using MacPorts. The "port" is ruby22.

Lots of people install Ruby versions using the Homebrew package manager.
Getting Ruby for Windows

Go to http://rubyinstaller.org/downloads and get "Ruby 2.2.6" (not x64)

When installing, I recommend these selections:
   Add Ruby executables to your PATH
   Associate .rb and .rbw files with this Ruby installation
**Running Ruby on lectura**

**rvm** is the Ruby Version Manager. It lets one easily select a particular version of Ruby to work with.

On lectura, we can select Ruby 2.2.4 and then check the version like this:

```
% rvm 2.2.4
% ruby --version
ruby 2.2.4p230 (2015-12-16 revision 53155) [x86_64-linux]
```

Depending on your Bash configuration, **rvm** may produce a message like "Warning! PATH is not properly set up..." but if **ruby --version** shows 2.2.4, all is well.

Note: **rvm** does not work with **ksh**. If you're running **ksh**, let us know.
IMPORTANT: you must either

1. Do `rvm 2.2.4` each time you login on lectura.
   —OR—

2. Add the command `rvm 2.2.4` to one of your Bash start-up files.

There are a variety of ways in which Bash start-up files can be configured.

- With the default configuration for CS accounts, add the line
  `rvm 2.2.4>& /dev/null`
  at the end of your `~/.profile`.

- If you're using the configuration suggested in my Fall 2015 352 slides, put
  `rvm 2.2.4>& /dev/null`
  at the end of your `~/.bashrc`.

- Let us know if you have trouble with this.
**irb**—Interactive Ruby Shell

**irb**, *Interactive Ruby Shell*, provides a REPL for Ruby.

**irb** evaluates expressions as they are typed.

```
$ irb
>> "abc" + "12"
=> "abc12"
```

If an expression is definitely incomplete, **irb** displays an alternate prompt:

```
>> 1.23 +
?> 2e3
=> 2001.23
```

Note: To save space on the slides, the result line (=> ...) won't be shown when it's uninteresting.

Control-D terminates **irb**.
Collaborative Learning Exercise

cs.arizona.edu/classes/cs372/spring18/cle-7.html
Note to self: push-cle 7
I use a Bash alias for `irb` that requests a simple prompt and activates auto-completion:

```
alias irb="irb --prompt simple -r irb/completion"
```

When `irb` starts up, it first processes `~/.irbrc`, if present.

`spring18/ruby/dotirbrc` is a recommended starter `~/.irbrc` file.

```
% cp /cs/www/classes/cs372/spring18/ruby/dotirbrc ~/.irbrc
```

On Windows you might use a batch file named `irbs.bat` to start with those options. Example:

```
W:\372\ruby> type irbs.bat
irb --prompt simple -r irb/completion
```

Run it by typing `irbs` (not just `irb`).
I like "it" better than underscore for referencing the previous result.

With the ~/.irbrc suggested on the previous slide, I can use "it" to reference the last result:

```
>> 3 + 4
=> 7

>> it * 3
=> 21

>> it.to_s
=> "21"
```

Observation: Ruby's flexibility lets me use a convention I like.
Ruby basics
In Ruby, every value is an object.

Methods can be invoked using `receiver.method(parameters...)

```ruby
>> "testing".count("t")  # How many "t"s are there?
=> 2

>> "ostentatious".tr("aeiou","12345")

>> "testing".length()
```

Repeat: In Ruby, every value is an object.

What are some values in Java that are not objects?
Of course, "everything" includes numbers:

```ruby
>> 1.2.class()
=> Float
```

```ruby
>> (10-20).class()
=> Fixnum
```

```ruby
>> 17**25
=> 5770627412348402378939569991057
```

```ruby
>> it.succ()  # Remember: the custom \texttt{.irbrc} is needed to use "it"
=> 5770627412348402378939569991058
```

```ruby
>> it.class()
=> Bignum
```
Everything is an object, continued

The TAB key can be used to show completions in irb:

>> 100.<TAB><TAB>
Display all 107 possibilities? (y or n)
100.__id__                100.display
100.__send__               100.div
100.abs                    100.divmod
100.abs2                   100.downto
100.angle                  100.dup
100.arg                    100.enum_for
100.between?               100.eql?
100.ceil                   100.equal?
100.chr                    100.even?
100.class                  100.extend
100.clone                  100.fdiv
100.coerce                 100.floor
100.conj                   100.freeze
100.conjugate              100.frozen?
100.define_singleton_method 100.gcd
100.denominator            100.gcdlcm
Parentheses are optional, sometimes

Parentheses are often optional in method invocations:

```ruby
>> 1.2.class
=> Float

>> "testing".count "aeiou"
=> 2
```

But, the following case fails. (Why?)

```ruby
>> "testing".count "aeiou".class
TypeError: no implicit conversion of Class into String
from (irb):17:in `count'
```

Solution:

```ruby
>> "testing".count("aeiou").class
=> Fixnum
```

I usually omit parentheses in simple method invocations.
A post-Haskell hazard!

Don't let the optional parentheses make you have a Haskell moment and leave out a comma between arguments:

```bash
>> "testing".slice 2 3
SyntaxError: (irb):20: syntax error, unexpected tINTEGER,
expecting end-of-input
```

Commas are required between arguments!

```bash
>> "testing".slice 2,3
=> "sti"
```

I almost always use parentheses when there's more than one argument:

```bash
>> "testing".slice(2,3)
=> "sti"
```
Operators are methods, too

Ruby operators are methods with symbolic names.

In general,

\[ expr1 \ op \ expr2 \]

means

\[ expr1.op(expr2) \]

Example:

\[
\begin{align*}
>> & 3 + 4 \\
& => 7 \\
>>& 3.+(4) \\
& => 7 \\
>>& "abc".==(97.chr.+("bc"))
\end{align*}
\]
Kernel methods

The **Kernel module** has methods for I/O and more. Methods in **Kernel** can be invoked with only the method name.

```ruby
>> puts "hello"
hello
=> nil

>> printf("sum = %d, product = %d\n", 3+4, 3 * 4)
sum = 7, product = 12
=> nil

>> puts gets.inspect
testing
"testing\n"
=> nil
```

What can say about value, type and side-effect for **puts** and **printf**?

See [http://ruby-doc.org/core-2.2.4/Kernel.html](http://ruby-doc.org/core-2.2.4/Kernel.html)
Extra Credit Assignment 2

For two assignment points of extra credit:

1. Run `irb` somewhere and try ten Ruby expressions with some degree of variety.

2. Capture the output and put it in a plain text file, `eca2.txt`. No need for your name, NetID, etc. in the file. No need to edit out errors.

3. On lectura, turn in `eca2.txt` with the following command:
   
   ```
   % turnin 372-eca2 eca2.txt
   ```

Due: At the start of the next lecture after we hit this slide.

Needless to say, feel free to read ahead in the slides and show experimentation with the following material, too.
A LHtLaL suggestion:
Start accumulating a file of brief notes on Ruby. Example:

```bash
$ cat ~/notes/ruby.txt
#running
rvm 2.2.4 to select version on lectura
irb is REPL, reads ~/.irbrc

#irb
_ is last value

#misc
Every value is an object
Can often omit parens on methods:
   3.class, "testing".count "t"
Operators are methods: 3+4 is really 3.+(4)

#i/o
gets, puts, printf (in Kernel module)
```
Executing Ruby code in a file

The **ruby** command can be used to execute Ruby source code contained in a file.

By convention, Ruby files have the suffix `.rb`.

Here is "Hello" in Ruby:

```ruby
% cat hello.rb
puts "Hello, world!"
```

```bash
% ruby hello.rb
Hello, world!
```

Note that the code does not need to be enclosed in a method—"top level" expressions are evaluated when encountered.

There is no evident compilation step or artifact produced. It just runs!
Executing Ruby code in a file, continued

Alternatively, code can be placed in a method that is invoked by an expression at the top level:

```ruby
% cat hello2.rb
def say_hello
  puts "Hello, world!"
end

say_hello

% ruby hello2.rb
Hello, world!
```

The definition of `say_hello` must precede the call.

We'll see later that Ruby is somewhat sensitive to newlines.
A line-numbering program

The program below reads lines from standard input and writes each, with a line number, to standard output:

```ruby
line_num = 1  # numlines.rb

while line = gets
    printf("%3d: %s", line_num, line)
    line_num += 1  # Ruby does not have ++ and --
end
```

Execution:

```
% ruby numlines.rb < hello2.rb
 1: def say_hello
 2:     puts "Hello, world!"
 3: end
 4:
 5: say_hello
```
Problem: Write a program that reads lines from standard input and writes them in reverse order to standard output. Use only the Ruby you've seen.

For reference, here's the line-numbering program:

```ruby
line_num = 1
while line = gets
  printf("%3d: %s", line_num, line)
  line_num += 1
end
```

Solution:
Some basic types
The value `nil`

`nil` is Ruby's "no value" value. The name `nil` references the only instance of the class.

```ruby
>> nil
=> nil
```

```
>> nil.class
=> NilClass
```

```
>> nil.object_id
=> 4
```

We'll see that Ruby uses `nil` in a variety of ways.

Speculate: What happens if we use a variable that hasn't been assigned to?
Strings and string literals

Instances of Ruby's `String` class represent character strings.

A variety of "escapes" are recognized in double-quoted string literals:

```ruby
>> puts "newline >\n< and tab >\t<"
newline >
< and tab > <

>> "\n\t".length
=> 3

>> "Newlines: octal \012, hex \xa, control-j \cj"
=> "Newlines: octal \n, hex \n, control-j \n"
```

Section 3.2, page 49 in RPL has the full list of escapes.
String literals, continued

In single-quoted string literals only \' and \" are recognized as escapes:

```ruby
>> puts '\n\t'
\n\t
=> nil

>> '\n\t'.length  # Four chars: backslash, n, backslash, t
=> 4

>> puts '\'\\'
'\n
=> nil

>> '\"\"'.length  # Two characters: apostrophe, backslash
=> 2
```
The `public_methods` method shows the public methods that are available for an object. Here are some of the methods for `String`:

```ruby
>> "abc".public_methods.sort

>> "abc".public_methods.length
=> 169
```
Unlike Java, Haskell, and many other languages, strings in Ruby are mutable.

If two variables reference a string and the string is changed, the change is reflected by both variables:

```ruby
>> x = "testing"

>> y = x  # x and y now reference the same instance of String

>> y << " this"  # the << operator appends a string

>> y

>> x
```

Is it a good idea to have mutable strings?
Strings are mutable, continued

The `dup` method produces a copy of a string.

```ruby
>> x = "testing"
>> y = x.dup
=> "testing"

>> y << "...more"

>> y

>> x
```

Some objects that hold strings `dup` the string when the string is added to the object.
Sidebar: Applicative vs. imperative methods

Some methods have both an *applicative* and an *imperative* form.

**String**'s `upcase` method is *applicative*—it produces a new **String** but doesn't change its *receiver*, the instance of **String** on which it's called:

```ruby
>> s = "testing"
=> "testing"

>> s.upcase
=> "TESTING"

>> s
```

CSC 372 Spring 2018, Ruby Slide 40
Applicative vs. imperative methods, continued

In contrast, an *imperative* method potentially changes its receiver.

String's `upcase!` method is the imperative counterpart to `upcase`:

```ruby
>> s.upcase!
=> "TESTING"
```

```ruby
>> s
```

A Ruby *convention*:

When methods have both an applicative and an imperative form, the imperative form ends with an exclamation mark.
String comparisons

Strings can be compared with a typical set of operators:

```ruby
>> s1 = "apple"

>> s2 = "testing"

>> s1 == s2
=> false

>> s1 != s2
=> true

>> s1 < s2
=> true
```

These operators work with most other types, too!

We'll talk about details of `true` and `false` later.
String comparisons, continued

There is also a comparison operator: 

Behavior:

```ruby
>> "apple" <=> "testing"
>> "testing" <=> "apple"
>> "x" <=> "x"
```

Speculate: How is the operator `<>` read aloud by some programmers?

What are the Java and C analogs for `<>` when applied to strings?
Substrings

Subscripting a string with a number produces a one-character string.

```ruby
>> s="abcd"

>> s[0]  # Positions are zero-based

>> s[1]

>> s[-1]  # Negative positions are counted from the right

>> s[100]
```

Historical note: With Ruby versions prior to 1.9, "abc"[0] is 97.

Why doesn't Java provide s[n] instead of s.charAt(n)?
A subscripted string can be the target of an assignment. A string of any length can be assigned.

```ruby
>> s = "abc"
=> "abc"

>> s[0] = 65.chr

>> s[1] = "tomi"

>> s

>> s[-3] = ""

>> s
```
A substring can be referenced with 
\[ s[start, length] \]

\[
\begin{align*}
&>> s = "replace" \\
&\quad r e p l a c e \\
&\quad 0 1 2 3 4 5 6 \\
&\quad 7 6 5 4 3 2 1 \quad (negative)
\end{align*}
\]

\[
\begin{align*}
&>> s[2,3] \\
&\quad r e p l a c e \\
&\quad 0 1 2 3 4 5 6 \\
&\quad 7 6 5 4 3 2 1 \quad (negative)
\end{align*}
\]

\[
\begin{align*}
&>> s[3,100] \\
&\quad r e p l a c e \\
&\quad 0 1 2 3 4 5 6 \\
&\quad 7 6 5 4 3 2 1 \quad (negative)
\end{align*}
\]

\[
\begin{align*}
&>> s[-4,3] \\
&\quad r e p l a c e \\
&\quad 0 1 2 3 4 5 6 \\
&\quad 7 6 5 4 3 2 1 \quad (negative)
\end{align*}
\]

\[
\begin{align*}
&>> s[10,10] \\
&\quad r e p l a c e \\
&\quad 0 1 2 3 4 5 6 \\
&\quad 7 6 5 4 3 2 1 \quad (negative)
\end{align*}
\]
Instances of Ruby's `Range` class represent a range of values. A `Range` can be used to reference a substring.

```ruby
>> r = 2..-2
=> 2..-2

>> r.class
=> Range

>> s = "replaced"

>> s[r]

>> s[r] = ""

>> s
```
It's more common to use literal ranges with strings:

```ruby
>> s = "rebuilding"
>> s[2..-1]

>> s[2..-4]

>> s[2...-4]

>> s[-8..-4]

>> s[-4..-8]
```

Substrings with ranges, continued
A substring can be the target of an assignment:

```ruby
>> s = "replace"

>> s[0,2] = ""

>> s

>> s[3..-1] = "naria"

>> s

>> s["aria"] = "kton"

>> s
```

```
replace
0 1 2 3 4 5 6
7 6 5 4 3 2 1 (negative)

place
0 1 2 3 4
5 4 3 2 1 (negative)
```
Interpolation in string literals

In a string literal enclosed with double quotes the sequence `#{expr}` causes interpolation of `expr`, an arbitrary Ruby expression.

```ruby
>> x = 10

>> y = "twenty"

>> s = "x = #{x}, y + y = #{y + y}"
=> "x = 10, y + y = twentytwenty"

>> puts "There are #{"".public_methods.length} string methods"
There are 169 string methods

>> "test #{"#{"abc".length*4}"}"    # Arbitrary nesting works
```

It's idiomatic to use interpolation rather than concatenation to build a string from multiple values.
With 2.2.4 on lectura, integers in the range \(-2^{62}\) to \(2^{62}-1\) are represented by instances of \texttt{Fixnum}. If an operation produces a number outside of that range, the value is represented with a \texttt{Bignum}.

\begin{verbatim}
>> x = 2**62-1
=> 4611686018427387903

>> x.class
=> Fixnum

>> x += 1
=> 4611686018427387904

>> x.class
=> Bignum

>> x -= 1
=> 4611686018427387903

>> x.class
=> Fixnum
\end{verbatim}

Is this automatic transitioning between \texttt{Fixnum} and \texttt{Bignum} a good idea? How do other languages handle this?
The **Float** class represents floating point numbers that can be represented by a double-precision floating point number on the host architecture.

```ruby
>> x = 123.456
=> 123.456

>> x.class
=> Float

>> x ** 0.5
=> 11.111075555498667

>> x = x / 0.0

>> (0.0/0.0).nan?
```
Arithmetic on two \textbf{Fixnum}s produces a \textbf{Fixnum}.
\begin{verbatim}
>> 2/3
=> 0

>> it.class
=> Fixnum
\end{verbatim}

\textbf{Fixnum}s and \textbf{Float}s can be mixed. The result is a \textbf{Float}.
\begin{verbatim}
>> 10 / 5.1
=> 1.9607843137254903

>> 10 % 4.5
=> 1.0

>> it.class
=> Float
\end{verbatim}
Ruby has a `Complex` type.

```ruby
>> x = Complex(2,3)
=> (2+3i)

>> x * 2 + 7
=> (11+6i)

>> Complex 'i'

>> it ** 2
```
There's Rational, too.

```ruby
>> Rational(1,3)
=> (1/3)

>> it * 300

>> Rational 0.5

>> Rational 0.6

>> Rational 0.015625
```
Unlike some languages, Ruby does not automatically convert strings to numbers and numbers to strings as needed.

>> 10 + "20"
TypeError: String can't be coerced into Fixnum

The methods `to_i`, `to_f`, and `to_s` are used to convert values to Fixnums, Floats and Strings, respectively.

>> 10.to_s + "20"

>> 10 + "20".to_f

>> 10 + 20.9.to_i
Arrays

A sequence of values is typically represented in Ruby by an instance of `Array`.

An array can be created by enclosing a comma-separated sequence of values in square brackets:

```ruby
>> a1 = [10, 20, 30]
=> [10, 20, 30]

>> a2 = ["ten", 20, 30.0, 2**40]
=> ["ten", 20, 30.0, 1099511627776]

>> a3 = [a1, a2, [[a1]]]
=> [[10, 20, 30], ["ten", 20, 30.0, 1099511627776], [[[10, 20, 30]]]]
```

What's a difference between Ruby arrays and Haskell lists?
Array elements and subarrays (sometimes called slices) are specified with a notation like that used for strings.

```ruby
>> a = [1, "two", 3.0, %w{a b c d}]
=> [1, "two", 3.0, ['a', 'b', 'c', 'd']]

>> a[0]

>> a[1,2]       # a[start, length]

>> a[-1]

>> a[-1][-2]
```
Elements and subarrays can be assigned to. Ruby accommodates a variety of cases; here are some:

```ruby
>> a = [10, 20, 30, 40, 50, 60]

>> a[1] = "twenty"; a

>> a[2..4] = %w{a b c d e}; a

>> a[1..-1] = []; a
```

Semicolon separates expressions. We make a change and show new value.
A few more:

```ruby
>> a
=> [10]

>> a[0] = [1,2,3]; a

>> a[4] = [5,6]; a

>> a[0,2] = %w{a bb ccc}; a
```

What's important to retain from the examples above?

- Elements of arrays and subarrays can be assigned to.
- Rules are complex.
A variety of operations are provided for arrays. Here's a sampling:

```ruby
>> a = []

>> a << 1; a

>> a << [2,3,4]; a

>> a.reverse; a

>> a.reverse!; a
```
A few more:

```ruby
>> a
=> [[2, 3, 4], 1]

>> a[0].shift

>> a

>> a.unshift "a","b","c"

>> a.shuffle.shuffle
```
Even more!

```ruby
>> a = [1,2,3,4]; b = [1,3,5]

>> a + b

>> a - b

>> a & b

>> a | b

>> ('a..'-'zzz').to_a.size
```
Comparing arrays

We can compare arrays with == and !=. Elements are compared in turn, possibly recursively.

```ruby
>> [1,2,3] == [1,2]

>> [1,2,[3,"bcd"]]) == [1,2] + [[3, "abcde"]]

>> [1,2,[3,"bcd"]) == [1,2] + [[3, "abcde"[1..-2]]]
```
Comparing arrays

Comparison of arrays with \(\leq\) is lexicographic.

\[
\begin{align*}
&> [1,2,3,4] \leq [1,2,10] \\
&=> -1 \\
&> [[10,20],[2,30],[5,"x"]].sort \\
&=> [[2,30],[5,"x"],[10,20]]
\end{align*}
\]
Comparing arrays

Comparison with \(\leq\) produces \texttt{nil} if differing types are encountered.

\[
\texttt{\[1,2,3,4\] \(\leq\) \[1,2,3,"four"\]} \quad \Rightarrow \texttt{nil}
\]

\[
\texttt{\[[10,20],[5,30],[5,"x"]\].sort} \quad \text{ArgumentError: comparison of Array with Array failed}
\]

Here's a simpler failing case. Should it be allowed?

\[
\texttt{\["sixty",20,"two"\].sort} \quad \text{ArgumentError: comparison of String with 20 failed}
\]
Comparing arrays, continued

At hand:
    >> ['sixty', 20, 'two'].sort
    ArgumentError: comparison of String with 20 failed

Contrast with Icon:
        ][ sort(['sixty', 20, 'two'])
        r := [20, 'sixty', 'two'] (list)

        ][ sort([3.0, 7, 2, 'a', 'A', ':', [2], [1], -1.0])
        r := [2, 7, -1.0, 3.0, ':', 'A', 'a', [2], [1]] (list)

What does Icon do better? What does Icon do worse?

Here's Python 2:
    >>> sorted([3.0, 7, 2, 'a', 'A', ':', [2], [1], -1.0])
    [-1.0, 2, 3.0, 7, [1], [2], ':', 'A', 'a']
An array can hold a reference to itself:

```ruby
>> a = [1, 2, 3]

>> a.push a

>> a.size

>> a[-1]

>> a[-1][-1][-1]
```

Arrays can be cyclic

```ruby
>> a << 10

>> a[-2][-1]

>> a[-1][-1][-1]
```