mio Documentation

Release 0.1.9.dev

James Mills

June 24, 2015

Contents

Release 0.1.9.dev

Date June 24, 2015

CHAPTER 1

About

Note: As of the 7th Sep 2014 mio is being redesigned and redeveloped from the ground up using the RPython toolchain from the PyPy project with clean design, performance and modern features in mind. Please see: https://bitbucket.org/miolang/mio

mio is a minimalistic IO programming language written in the Python Programming Language based on MIo (*a port from Ruby to Python*) in the book How To Create Your Own Freaking Awesome Programming Language by Marc-Andre Cournoye.

This project is being developed for educational purposes only and should serve as a teaching tool for others wanting to learn how to implement your own programming language (*albeit in the style of Smalltalk, Io, etc*). Many thanks go to Marc-Andre Cournoye and his wonderful book which was a great refresher and overview of the overall processing and techniques involved in programming language design and implementation. Thanks also go to the guys in the `#io` channel on the FreeNode IRC Network specifically **jer** nad **locks** for their many valuable tips and help.

The overall goal for this project is to create a fully useable and working programming language implementation of a langauge quite similar to Io with heavy influence from Python (*because Python is awesome!*). This has already largely been achived in the current version. See the RoadMap for what might be coming up next.

Warning: mio is a new programming language in early **Planning** and **Development**. DO NOT USE IN PRODUCTION! USE AT YOUR OWN RISK!

- Visit the Project Website
- Read the Docs
- Read the Tutorial
- Download it from the Downloads Page

1.1 Examples

Factorial:

```
Number fact = method(reduce(block(a, x, a * x), range(1, self)))
```

Hello World:

```
print("Hello World!")
```

1.2 Features

- Homoiconic
- Message Passing
- Higher Order Messages
- Higher Order Functions
- Full support for Traits
- Object Orienated Language
- Written in an easy to understand language
- Supports Imperative, Functional, Object Oriented and Behavior Driven Development styles.

1.3 Installation

The simplest and recommended way to install mio is with pip. You may install the latest stable release from PyPI with pip:

```
> pip install mio-lang
```

If you do not have pip, you may use easy_install:

```
> easy_install mio-lang
```

Alternatively, you may download the source package from the PyPI Page or the Downloads page on the Project Website; extract it and install using:

```
> python setup.py install
```

You can also install the latest-development version by using `pip` or `easy_install`:

```
> pip install mio-lang==dev
```

or:

```
> easy_install mio-lang==dev
```

For further information see the mio documentation.

CHAPTER 2

Documentation

2.1 mio Tutorial

2.1.1 Expressions

```
mio> 1 + 2
===> 3
mio> 1 + 2 * 3
===> 9
mio> 1 + (2 * 3)
===> 7
```

Note: mio has no operator precedence (*in fact no operators*). You **must** use explicit grouping with parenthesis where appropriate in expressions.

2.1.2 Variables

```
mio> a = 1
===> 1
mio> a
===> 1
mio> b = 2 * 3
===> 6
mio> a + b
===> 7
```

2.1.3 Conditionals

```
mio> a = 2
===> 2
mio> (a == 1) ifTrue(print("a is one")) ifFalse(print("a is not one"))
a is not one
```

2.1.4 Lists

```
mio> xs = [30, 10, 5, 20]
===> [30, 10, 5, 20]
mio> len(xs)
===> 4
mio> print(xs)
[30, 10, 5, 20]
mio> xs sort()
===> [5, 10, 20, 30]
mio> xs[0]
===> 5
mio> xs[-1]
===> 30
mio> xs[2]
===> 20
mio> xs remove(30)
===> [5, 10, 20]
mio> xs insert(1, 123)
===> [5, 123, 10, 20]
```

2.1.5 Iteration

```
mio> xs = [1, 2, 3]
===> [1, 2, 3]
mio> xs foreach(x, print(x))
1
2
3
mio> it = iter(xs)
===> it(Object) at 0x7ff68693b9a8:
       N               = 2
       i               = -1
       iterable        = [1, 2, 3]
mio> next(it)
===> 1
mio> next(it)
===> 2
mio> next(it)
===> 3
mio> next(it)

  StopIteration:
  --------------
  raise(StopIteration)

  ifFalse(
 raise(StopIteration)
)

  __next__

  next(it)
```

2.1.6 Strings

```
mio> a = "foo"
===> u"foo"
mio> b = "bar"
===> u"bar"
mio> c = a + b
===> u"foobar"
mio> c[0]
===> u'f'
```

```
mio> s = "this is a test"
===> u"this is a test"
mio> words = s split()
===> [u"this", u"is", u"a", u"test"]
mio> s find("is")
===> 2
mio> s find("test")
===> 10
```

2.1.7 Functions

```
mio> foo = block(
....     print"foo"
.... )
===> block():
       args            = args()
       body            = body()
       kwargs          = kwargs()
mio> foo()
===> u"foo"
mio> add = block(x, y,
....     x + y
.... )
===> block(x, y):
       args            = args()
       body            = body()
       kwargs          = kwargs()
mio> add(1, 2)
===> 3
```

Note: Functions in mio do not have access to any outside state or globals (*there are no globals in mio*) with the only exception to the rule being closures.

2.1.8 Objects

```
mio> World = Object clone()
===> World(Object) at 0x7fd6b12799a8
mio> World
===> World(Object) at 0x7fd6b12799a8
```

Attributes

```
mio> World = Object clone()
===> World(Object) at 0x7fe141f429a8
mio> World
===> World(Object) at 0x7fe141f429a8
mio> World name = "World!"
===> u"World!"
mio> World name
===> u"World!"
```

Methods

```
mio> World = Object clone()
===> World(Object) at 0x7fe2994a19a8
mio> World
===> World(Object) at 0x7fe2994a19a8
mio> World name = "World!"
===> u"World!"
mio> World name
===> u"World!"
mio> World hello = method(
....     print("Hello", self name)
.... )
===> method():
       args            = args()
       body            = body()
       kwargs          = kwargs()
mio> World hello()
Hello World!
```

Note: Methods implicitly get the receiving object as the first argument self passed.

2.1.9 Traits

```
mio> TGreetable = Object clone() do (
....     hello = method(
....         print("Hello", self name)
....     )
.... )
===> TGreetable(Object) at 0x7f42e3cc19a8:
       hello           = method()
mio> World = Object clone() do (
....     uses(TGreetable)
....
....     name = "World!"
.... )

  AttributeError: Object has no attribute 'uses'
  ---------------
  uses(TGreetable)

 set(name, World!)
```

```
  do(
 uses(TGreetable)

 set(name, World!)
)
  set(World, Object clone do(
 uses(TGreetable)

 set(name, World!)
))

mio> World

  AttributeError: Object has no attribute 'World'
  ---------------
  World

mio> World traits

  AttributeError: Object has no attribute 'World'
  ---------------
  World traits

mio> World behaviors

  AttributeError: Object has no attribute 'World'
  ---------------
  World behaviors

mio> World hello()

  AttributeError: Object has no attribute 'World'
  ---------------
  World hello
```

2.2 Grammar

The following EBNF Grammar defines the Syntax for mio:

```
 operator    ::=   "**" | "++" | "-" | "+=" | "-=" | "*=" | "/=" | "<<" | ">>" |
                   "==" | "!=" | "<=" | ">=" | "+" | "-" | "*" | "/" | "=" |
                   "<" | ">" | "!"  | "%" | "|" | "^" | "&" | "is" | "or" |
                   "and" | "not" |
                   "return"
 comment     ::=   r"^#.*$"
 whitespace  ::=   r"[ \t]+"
 string      ::=   r"'[^"]*"'
 number      ::=   r'-?([0-9]+(\.[0-9]*)?)'
 identifier  ::=   r'[A-Za-z_][A-Za-z0-9_]*'
 terminator  ::=   ";" | "\r" | "\n"
 expression  ::=   (message | terminator)*
 message     ::=   (symbol, [ arguments ]) | arguments
 opening     ::=   "(" | "{" | "["
```

```
closing     ::=   ")" | "}" | "]"
arguments   ::=   opening , ( expression, ( "," , expression )* )* , closing
symbol      ::=   identifier | number | operator | string
```

2.3 API Documentation

2.4 Changes

2.4.1 mio 0.1.9.dev

- Added support for testing with Jenkins for Continuous Integration
- Fixed Traits test

2.4.2 mio 0.1.8 (*2013-12-24*)

- Implemented `caller()` [**functools**]
- Fixed parser support for `[1] + [2]` –> `[](1} +([](2))`
- Fixed `List extend()` to take another list, not `*args`.
- Implemented + List operator.
- Implemented `min` and `max` and added to the mio std. lib.

2.4.3 mio 0.1.7 (*2013-12-09*)

- Fixed `Object evalArg()` so it can take zero arguments and return `None`. e.g: `() a`
- Implemented `identity()` and `constantly()` in `functools` std. lib and added unit tests.
- Fixed typos in `operators` module.
- Moved all trait related user methods to `TAdaptable` core trait.
- Added `adapt()` method to `TAdaptable`.

Note: This is more or less an alias of `use()` except the object being adapted is cloned and the original left in tact. The adaptation is therefore temporary.

- Added support for contextually aware tab completion.
- Test Coverage back up to 100%

2.4.4 mio 0.1.6 (*2013-12-04*)

- Added support for packaging Python wheels.
- Define a default `init()` method on `Object` and just make `TCloneable init()` just call `init()`.
- Added `String strip()` method.
- Make `String` use `TIterable`.

- Implemented `input()` builtin.
- Implemented a special object `Trait` which all traits are cloend from.
  - This means traits now cannot contain state and a `TypeError` is raised.
  - Objects can only use or add traits that are inherited from the `Trait` object.

A simple trait thus looks like:

```
TFoo = Trait clone() do (
    foo = method(
        print("Foo!")
    )
)
```

- Implemented `Trait requires()` so that traits can declare the names of methods and attribute they depend on.

e.g:

```
TGreetable = Trait clone() do (
    requires("name")

    hello = method(
        print("Hello ", self name)
    )
)

World = Object clone() do (
    name = "World!"
    uses(TGreetable)
)

World hello()
```

- Added conflict resolution for traits.
- Remvoed `uses` in favor of single `use(trait, resolution)` where **resolution** is a dict of key->value pairs that rename conflicting methods of the trait being used.
- Make `Object hasTrait()` lookup the parent chain.
- Abuse the `is` method of `TComparble` to be used as short-cut for: `foo hasTrait(TFoo)`.

2.4.5 mio 0.1.5 (*2013-11-28*)

- Don't look into `builtins` module for the tab-completer function of the REPL if there are no builtins. i.e: `mio -S`
- Added a nice big shiny red warning about mio's status.

2.4.6 mio 0.1.4 (*2013-11-27*)

- Added "Functions" to the tutorial.
- Added "Objects" to the tutorial.
- Added "Traits" to the tutorial.
- Added the start of a small `functools` library to the mio std. lib.

- Added `test_builtins` to mio. std library as a tests package.
- Added operators module to mio std. lib.
- Rought cut of generators implemented in CPython and mio.
- Implemented `KeyError` exception type and used this to guard non-existent key lookups on dicts.
- Somewhat improved the error handling and tracebacks.
- Improved the way generators work.
- Added `yield` to the mio std lib builtins.
- Added `TypeError` guard around opening non-files with the `Path` object.
- Rewrote the tutorial to utilize `sphinxcontrib-autorun` extension with custom `miointerpreter.py` module.
- Improved REPL continuation output of unclosed parens.
- Removed docs dependencies of unused sphinx extensions.
- Added `mio.state.State.runsource()` which simplifies `mio.state.State.repl()` a bit and allows our custom `sphinxcontrib-autorun` ext to work.
- Added API Docs.

2.4.7 mio 0.1.3 (*2013-11-19*)

- Fixed tutorial by writing a bash script that generates it (*Read the Docs forbids program-output extension*).

2.4.8 mio 0.1.2 (*2013-11-19*)

- Moved `File` from types to core.
- Allow multiple `-e expr` options to be given on the CLI.
- Implemented `len` builtin.
- Implemented `List remove()` method.
- Implemented `String split()` method.
- Implemented `String __getitem__()` and `String __len__()` methods.
- Implemented `Bytes __getitem__()` and `Bytes __len__()` methods.
- Added `[]` syntactic sugar to `Bytes` and `String` objects.
- Wrote a basic mio tutorial
- Updated the factorial example to be a little clearer.
- Improved Test Coverage of new features and objects.

2.4.9 mio 0.1.1 (*2013-11-18*)

- The beginnings of a testing framework.
- Implemented unit tests for mio builtins: `abs`, `all` and `any`.

- Added paren detection to the REPL so you start write long functions over multiple lines. (*Borrowed from: https://bitbucket.org/pypy/lang-scheme/src/b1d5a1b8744f3c7c844775cb420c1a5d4c584592/scheme/interactive.py?at=default*).
- Added basic tab completion support to the REPL.

Note: This is not context aware and build up a list of known objects from `Root`, `Types`, `Core` and `builtins`.

- Many RPython compilation issues fixed.
- Implemented `Path` object.

2.4.10 mio 0.1 (*2013-11-14*)

- Updated factorial examples
- Added support for and a `fab compile` task for compiling mio with RPython

Warning: This does not work yet!

- Changed the way results are printed on the REPL by implementing a `format_value(...)` utility function.
- Added `assert` as a special name (*operator*).
- Added optional message argument to `assert`.
- Improved repr of `Core` and `Types` objects.
- Fixed a bug in the parser so that we can use `[]` and `{}` as methods.

```
xs = [1, 2, 3, 4]
xs[0]
```

- Implemented dict literals.

```
{"a": 1, "b": 2}
```

Note: This only works with keys as strings for the moment.

- Added a `hash` builtin.
- Improved `{}` dict literal so that any hashable keys can be used.

Note: Like Python this means any object whoose `__hash__()` method returns a non-`None` value.

- Fixed `any` and `all` builtins.
- IMplemented `in` method of `TComparable`

```
1 in(1, 0)
```

- Changed the semantics of closures.
  - `this` is a new attribute of `Locals` that always references the current block scope.
  - `self` is a reference to the current object in scope (*if there is one*).

Warning: This behavior may change as I'm not 100% happy with this.

Note: This is sort of a work-around to allow blocks to access the currently scoped object `self` inside the scope of a method or nested blocks within a method.

2.4.11 mio 0.0.9 (*2013-11-10*)

- Fixed `[]` syntax for creating lists.
- Fixed `TIterable foreac` to work more like a for loop.
- Introduced properties for internal Python Functions exposed to mio.
- Adopted calling with `(...)` 's for all methods.
- Also format functions in `format_object(...)`.
- Added `__call__` to `Error` objects. This allows:

```
raise TypeError("foo")
```

- Added iterator support for `Range` object.
- Renamed `str` and `repr` methods of `Object` to `__str__` and `__str__` respectively and implemtned `str` and `repr` builtins.
- Optimized the tokenizer
- Added `ifError`, `ifNonError` and `catch` to the `Object` object to deal with non-errors.
- Improved and fixed a lot of the builtins.
- Improved the way `*args` is handled (*still needs more work*).
- Implemented `assert` builtin.

2.4.12 mio 0.0.8 (*2013-11-07*)

- Removed operator precedence parsing.
  - Operator precedence is **HARD**
  - Operator precedence rules hare **HARD** to remember
  - Operator precedence is not the main goal of mio right now.
- Tidied up the builtins module.
- `from foo import *` works again (*operator precedence parsing broke it*).

2.4.13 mio 0.0.7 (*2013-11-06*)

- Added rudamentary stack trace support to errors. A somewhat "okay" stack trace is displayed on error(s).
- Added `String format` method for performing string iterpolation. Only supports `{0}`, `{1}`, etc.
- Implemented `ListIterator` iterable object with `iter` added to mio std. lib. This works similiarly to Python's iterators:

```
mio> xs = [1, 2, 3, 4]
===> list(1, 2, 3, 4)
mio> it = iter(xs)
===> ListIterator(list(1, 2, 3, 4))
mio> it next()
===> 1
mio> it next()
===> 2
mio> it next()
===> 3
mio> it next()
===> 4
```

A further iteration would result in:

```
mio> it next()

  StopIteration:
  --------------
  next
ifFalse(
 raise(StopIteration)
)

raise(StopIteration)
```

- Re-implemented `return` function as part of the mio std. lib.
- Don't allow `return` to be called outside of a `Block` (*block/method*) as this is illegal.
- Implemented `while` builtin as part of the mio std. lib. (*no break or continue support yet*)
- Implemented `loop` builtin as part of the mio std. lib. (*no break or continue support yet*)
- Implemented basic support for reshuffling messages before chaining to support `x is not None` –> `not(x is None)`.
- Finally implemented operator precedence support (*which seems to cover most edge cases*).

Note: Need to write lots of unit tests for this!

- Fixed all found edge cases with the new operator precedence lexer/parser.
- Improved `Error` object and added `Error catch` method for catching errors.
- Implemented `reduce` builtin.
- Implemented TComparable trait
- Implemented TCloneable trait
- Iterpret `call message args` to mean "pass all args to the callable"
- Imroved Dict and List objects.
- Implemented `__call__` calling semantics whereby an object can implement this as a method and `Foo()` will invoke `Foo __call__` if it exists.
- IMplemented the `__get__` part of the Data Descriptor protocol.

2.4.14 mio 0.0.6 (*2013-11-02*)

- Allow an optional object to be passed to the `Object id` method.
- Implemented `hex` builtin.
- Implemented `Bytes` and `Tuple` objects.
- Implemented `State` core object and sample `loop` builtin (*in testing*).
- Refactored all of the context state management code (*stopStatus*) and exposed it to the end user.
  - This means we can now write flow based constructs such as loops directly in mio.
- Fixed a minor bug in the parser where `not(0) ifTrue(print("foo"))` would parse as `not(0, ifTrue(print("foo")))`
- Fixed a minor bug in the parser where `isError` would parse as `is(Error)`. Parse identifiers before operators.
- Implemented basic excpetion handling and error object(s) (*no stack traces yet*).
- Moved `exit` to builtins.
- Moved the setting of `.binding` attribute to `Object set/del` methods.
- Added support for `..` operator and added this to `Number`. This allows you to write:

```
x = 1 .. 5  # a Range from 1 to 5
```

- Added + and − operators to the `Range` object so you can do things like:

```
x = (1 .. 5) + 2  # a Range from 1 to 5 in increment of 2
```

- Changed default REPL prompt to:

```
$ mio
mio 0.0.6.dev
mio>
```

- Implemented `sum` builtin.
- Added `try` and `raise` builtins. (*''raise'' is not implemented yet...*).
- Added support for User level Error(s) and implemented `Exception raise`
- Replaced `Continuation call` with activatable object semantics. This means:

```
c = Continuation current()
print("foo")
c()
```

- `Object evalArg` should evaluate the argument with context as the receiver.
- Added `List __getitem__` and `List __len__` methods.
- Added `TIterable` trait to the mio bootstrap library and added this to `List`.
- Removed `foreach`, `whilte`, `continue`, `break` and `return Object` methods. These will be re-implemented as traits and builtins.
- Changed the way the parser parses and treats operators. They are no longer parsed in a deep right tree.

Example:

```
1 + 2 * 3
```

OLD:

```
1 +(2 *(3))
```

NEW:

```
1 +(2) *(3)
```

- This will probably make reshuffling and therefore implementing operator precedence a lot easier.
- This also makes the following expressions possible (*used in the builtins module*):

```
from foo import *
```

- Added `TypeError`, `KeyError` and `AttributeError` to the mio std. lib.
- Made it possible to import members from a module with: `from foo import bar`

2.4.15 mio 0.0.5 (*2013-10-29*)

- Split up core into core and types.
- Re-implemented `True`, `False` and `None` in mio.
- Implemented `bin` builtin.
- Implemented `bool` builtin.
- Implemented `callable` builtin.
- Implemented `cha` builtin.
- IMplemented `from` and `import` builtins.
- Make the `Object pimitive :foo` method return the internal Python data type.
- Fixed the `abs` builtin to return an newly cloned Number.
- Implemented support for packages ala Python.
- Restructured the mio std. lib
- mio nwo bootstraps itself via an import of the "bootstrap" package.
- Reimplemented `Object clone` in the mio std. lib.

2.4.16 mio 0.0.4 (*2013-10-27*)

- Moved the implementation of `super` to the mio std. lib
- Only set _ as the last result in the Root object (*the Lobby*)
- Added support for `()`, `[]` and `{}` special messages that can be used to define syntactic suguar for lists, dicts, etc.
- Implemented `Dict` object type and `{a=1, b=2}` syntactic sugar to the builtint (*mio std. lib*) `dict()` method.
- Refactored the `File` object implementation and made it's repr more consistent with other objects in mio.
- Fixed keyword argument support.
- Fixed a few minor bugs in the `Message` object and improved test coverage.

- Added `?` as a valid operator and an implementation of `Object ?message` in the mio std. lib.
- Fixed a bug with `Range`'s internal iterator causing `Range asList` not to work.
- Fixed a bug with `Object foreach` and `continue`.
- **Achived 100% test coverage!**
- Implemented `*args` and `**kwargs` support for methods and blocks.
- Removed `Object` methods `print`, `println`, `write` and `writeln` in favor of the new builtin `print` function in the mio std. lib
- Added an implemenation of `map` to the mio std. lib
- Fixed a bug with the parser where an argument's previous attribute was not getting set correctly.
- Reimplemented `not` in the mio std. lib and added `-=`, `*=` and `/=` operators.
- Added a `Object :foo` (*primitive*) method using the `:` operator. This allows us to dig into the host object's internal methods.
- Added an implementation of `abs` builtin using the primitive method.
- Changed the `import` function to return the imported module (*instead of ''None''*) so you can bind imported modules to explicitly bound names.
- Added `from` an alias to `import` and `Module import` so you can do:

```
bar = from(foo) import(bar)
```

- Fixed some minor bugs in `Object foreach` and `Object while` where a ReturnState was not passed up to the callee.
- Added implementations of `all` and `any` to the mio std. lib.
- Added this.mio (The Zen of mio ala Zen of Python)
- Added List insert method and internal __len__.
- Moved the implementations of the `Importer` and `Module` objects to the host language (*Python*).
- Added support for modifying the `Importer` search path.
- Restructured the mio std. library and moved all bootstrap modules into ./lib/bootstrap
- Added (almost) Python-style string literal support. Triple Quote, Double, Quote, Single Quote, Short and Long Strings
- Added support for exponents with number literals.
- Added internal `tomio` and `frommio` type converion function.
- Added an `FFI` implementation that hooks directly into the host language (*Python*).
- Implemented the `antigravity` module in mio.
- Added support for exposing builtin functions as well in the FFI.
- Simplified the two examples used in the docs and readme and write a simple bash script to profile the factorial example.
- Changed the calling semantics so that calls to methods/blocks are explicitly made with `()`.
- Added a new internal attribute to `Object` called `binding` used to show the binding of a bound object in repr(s).

2.4.17 mio 0.0.3 (*2013-10-20*)

- Improved test coverage
- Improved the `Range` object
- Fixed the scoping of `block` (s).
- Fixed the `write` and `writeln` methods of `Object` to not join arguments by a single space.
- Don't display `None` results in the REPL.
- Improved the `__repr__` of the `File` object.
- Added `open` and `with` builtins to the mio standard library.
- Implemented a basic import system in the mio standard library.
- Implemented `Dict items` method.

2.4.18 mio 0.0.2 (*2013-10-19*)

- Include lib as package data
- Allow mio modules to be loaded from anywhere so mio can be more usefully run from anywhere
- Added bool type converion
- Improved the documentation and added docs for the grammar
- Changed Lobby object to be called Root
- Added an -S option (don't load system libraries).
- Added unit test around testing for last value with return
- Refactored Message.eval to be non-recursive
- Set _ in the context as the last valeu
- Implemented Blocks and Methods
- Fixed return/state issue by implementing Object evalArg and Object evalArgAndReturnSelf in Python (not sure why this doesn't work in mio itself)
- Implemented Object evalArgAndReturnNone

2.4.19 mio 0.0.1 (*2013-10-19*)

- Initial Release

2.5 Road Map

- Better semantics and validation of traits
- Write a compiler and bytecode virtual machine
- Add coroutine support and multiprocessing
- Rewrite in RPython to take advantage of PyPy's JIT and STM

2.6 TODO

- Update the `docs/source/grammar.rst` and try to auto-generate it from the parser via `.ebnf()`
- Fix `property.mio` example and basic data descriptors.
- Fix `Object super()`. Make it a builtin.
- Deal with this better and raise a better error instead of crashing.

```
x = method(1 + 1, a, b,
    a + b
)
```

- Implement `__slice__`, `slice(...)` and `xs[start:[end][:step]`
- Change the way methods and blocks are bound and unifiy them into a single entity.
  - Unified method: `method(...)`.
  - Can be dynamically bound to objects.
  - Is always passed it's bound object as the first parameter `self`.
  - Are by default bound to the context they are created in.
- Fix keyword argument(s) parameters.
- Figure out a way to avoid recursion so `loop(print("foo"))` works as expected.
- Write a testing framework for mio in mio.
- Implement a "trace" hook into the interpreter. i.e: Python's `sys.settrace()`
- Implement a basic debugger.
- Implement a basic coverage tool.
- Add `__doc__` (*doc strings*) support.
- Implement a basic help system.
- Do a refresher on how to write an interpreter in RPython and write a really really simple one:
  - http://doc.pypy.org/en/latest/coding-guide.html#restricted-python
  - http://morepypy.blogspot.com.au/2011/04/tutorial-writing-interpreter-with-pypy.html

CHAPTER 3

Indices and tables

- genindex
- modindex
- search