hy

Release 0.17.0+139.g6cc4c23

Feb 02, 2020

Contents

PyPI https://pypi.python.org/pypi/hy

Source https://github.com/hylang/hy

List hylang-discuss

IRC irc://chat.freenode.net/hy

Stack Overflow The [hy] tag

Hy is a Lisp dialect that's embedded in Python. Since Hy transforms its Lisp code into Python abstract syntax tree (AST) objects, you have the whole beautiful world of Python at your fingertips, in Lisp form.

To install the latest stable release of Hy, just use the command `pip3 install --user hy`. Then you can start an interactive read-eval-print loop (REPL) with the command `hy`, or run a Hy program with `hy myprogram.hy`.

CHAPTER 1

Why Hy?

Hy is a multi-paradigm general-purpose programming language in the Lisp family. It's implemented as a kind of alternative syntax for Python. Compared to Python, Hy offers a variety of extra features, generalizations, and syntactic simplifications, as would be expected of a Lisp. Compared to other Lisps, Hy provides direct access to Python's built-ins and third-party Python libraries, while allowing you to freely mix imperative, functional, and object-oriented styles of programming.

1.1 Hy versus Python

The first thing a Python programmer will notice about Hy is that it has Lisp's traditional parenthesis-heavy prefix syntax in place of Python's C-like infix syntax. For example, `print("The answer is", 2 + object.method(arg))` could be written `(print "The answer is" (+ 2 (.method object arg)))` in Hy. Consequently, Hy is free-form: structure is indicated by parentheses rather than whitespace, making it convenient for command-line use.

As in other Lisps, the value of a simplistic syntax is that it facilitates Lisp's signature feature: metaprogramming through macros, which are functions that manipulate code objects at compile time to produce new code objects, which are then executed as if they had been part of the original code. In fact, Hy allows arbitrary computation at compile-time. For example, here's a simple macro that implements a C-style do-while loop, which executes its body for as long as the condition is true, but at least once.

```
(defmacro do-while [condition &rest body]
  `(do
    ~body
    (while ~condition
      ~body)))

(setv x 0)
(do-while x
  (print "This line is executed once."))
```

Hy also removes Python's restrictions on mixing expressions and statements, allowing for more direct and functional code. For example, Python doesn't allow with blocks, which close a resource once you're done using it, to return values. They can only execute a set of statements:

```
with open("foo") as o:
    f1 = o.read()
with open("bar") as o:
    f2 = o.read()
print(len(f1) + len(f2))
```

In Hy, *with* returns the value of its last body form, so you can use it like an ordinary function call:

```
(print (+
  (len (with [o (open "foo")] (.read o))
  (len (with [o (open "bar")] (.read o))))))
```

To be even more concise, you can put a `with` form in a *generator expression*:

```
(print (sum (gfor
  filename ["foo" "bar"]
  (len (with [o (open filename)] (.read o))))))
```

Finally, Hy offers several generalizations to Python's binary operators. Operators can be given more than two arguments (e.g., `(+ 1 2 3)`), including augmented assignment operators (e.g., `(+= x 1 2 3)`). They are also provided as ordinary first-class functions of the same name, allowing them to be passed to higher-order functions: `(sum xs)` could be written `(reduce + xs)`.

The Hy compiler works by reading Hy source code into Hy model objects and compiling the Hy model objects into Python abstract syntax tree (`ast`) objects. Python AST objects can then be compiled and run by Python itself, byte-compiled for faster execution later, or rendered into Python source code. You can even *mix Python and Hy code in the same project, or even the same file,* which can be a good way to get your feet wet in Hy.

1.2 Hy versus other Lisps

At run-time, Hy is essentially Python code. Thus, while Hy's design owes a lot to Clojure, it is more tightly coupled to Python than Clojure is to Java; a better analogy is CoffeeScript's relationship to JavaScript. Python's built-in functions and data structures are directly available:

```
(print (int "deadbeef" :base 16))  ; 3735928559
(print (len [1 10 100]))           ; 3
```

The same goes for third-party Python libraries from PyPI and elsewhere. Here's a tiny CherryPy web application in Hy:

```
(import cherrypy)

(defclass HelloWorld []
  (#@ cherrypy.expose (defn index [self]
    "Hello World!")))

(cherrypy.quickstart (HelloWorld))
```

You can even run Hy on PyPy for a particularly speedy Lisp.

Like all Lisps, Hy is homoiconic. Its syntax is represented not with cons cells or with Python's basic data structures, but with simple subclasses of Python's basic data structures called *models*. Using models in place of plain `list`s, `set`s, and so on has two purposes: models can keep track of their line and column numbers for the benefit of error messages, and models can represent syntactic features that the corresponding primitive type can't, such as the order in which elements appear in a set literal. However, models can be concatenated and indexed just like plain lists, and you can return ordinary Python types from a macro or give them to `eval` and Hy will automatically promote them to models.

Hy takes much of its semantics from Python. For example, Hy is a Lisp-1 because Python functions use the same namespace as objects that aren't functions. In general, any Python code should be possible to literally translate to Hy. At the same time, Hy goes to some lengths to allow you to do typical Lisp things that aren't straightforward in Python. For example, Hy provides the aforementioned mixing of statements and expressions, *name mangling* that transparently converts symbols with names like `valid?` to Python-legal identifiers, and a *let* macro to provide block-level scoping in place of Python's usual function-level scoping.

Overall, Hy, like Common Lisp, is intended to be an unopinionated big-tent language that lets you do what you want. If you're interested in a more small-and-beautiful approach to Lisp, in the style of Scheme, check out Hissp, another Lisp embedded in Python that was created by a Hy developer.

CHAPTER 2

Tutorial

This chapter provides a quick introduction to Hy. It assumes a basic background in programming, but no specific prior knowledge of Python or Lisp.

2.1 Lisp-stick on a Python

Let's start with the classic:

```
(print "Hy, world!")
```

This program calls the `print()` function, which, like all of Python's built-in functions, is available in Hy.

All of Python's binary and unary operators are available, too, although == is spelled = in deference to Lisp tradition. Here's how we'd use the addition operator +:

```
(+ 1 3)
```

This code returns `4`. It's equivalent to `1 + 3` in Python and many other languages. Languages in the Lisp family, including Hy, use a prefix syntax: `+`, just like `print` or `sqrt`, appears before all of its arguments. The call is delimited by parentheses, but the opening parenthesis appears before the operator being called instead of after it, so instead of `sqrt(2)`, we write `(sqrt 2)`. Multiple arguments, such as the two integers in `(+ 1 3)`, are separated by whitespace. Many operators, including `+`, allow more than two arguments: `(+ 1 2 3)` is equivalent to `1 + 2 + 3`.

Here's a more complex example:

```
(- (* (+ 1 3 88) 2) 8)
```

This code returns `176`. Why? We can see the infix equivalent with the command `echo "(- (* (+ 1 3 88) 2) 8)" | hy2py`, which returns the Python code corresponding to the given Hy code, or by passing the `--spy` option to Hy when starting the REPL, which shows the Python equivalent of each input line before the result. The infix equivalent in this case is:

```
((1 + 3 + 88) * 2) - 8
```

To evaluate this infix expression, you'd of course evaluate the innermost parenthesized expression first and work your way outwards. The same goes for Lisp. Here's what we'd get by evaluating the above Hy code one step at a time:

```
(- (* (+ 1 3 88) 2) 8)
(- (* 92 2) 8)
(- 184 8)
176
```

The basic unit of Lisp syntax, which is similar to a C or Python expression, is the **form**. `92`, `*`, and `(* 92 2)` are all forms. A Lisp program consists of a sequence of forms nested within forms. Forms are typically separated from each other by whitespace, but some forms, such as string literals (`"Hy, world!"`), can contain whitespace themselves. An **expression** is a form enclosed in parentheses; its first child form, called the **head**, determines what the expression does, and should generally be a function, macro, or special operator. Functions are the most ordinary sort of head, whereas macros (described in more detail below) are functions executed at compile-time instead and return code to be executed at run-time. Special operators are one of *a fixed set of names* that are hard-coded into the compiler, and used to implement everything else.

Comments start with a `;` character and continue till the end of the line. A comment is functionally equivalent to whitespace.

```
(print (** 2 64))   ; Max 64-bit unsigned integer value
```

Although # isn't a comment character in Hy, a Hy program can begin with a shebang line, which Hy itself will ignore:

```
#!/usr/bin/env hy
(print "Make me executable, and run me!")
```

2.2 Literals

Hy has *literal syntax* for all of the same data types that Python does. Here's an example of Hy code for each type and the Python equivalent.

Hy	Python	Type
`1`	`1`	`int`
`1.2`	`1.2`	`float`
`4j`	`4j`	`complex`
`True`	`True`	`bool`
`None`	`None`	`NoneType`
`"hy"`	`'hy'`	`str`
`b"hy"`	`b'hy'`	`bytes`
`(, 1 2 3)`	`(1, 2, 3)`	`tuple`
`[1 2 3]`	`[1, 2, 3]`	`list`
`#{1 2 3}`	`{1, 2, 3}`	`set`
`{1 2 3 4}`	`{1: 2, 3: 4}`	`dict`

In addition, Hy has a Clojure-style literal syntax for `fractions.Fraction`: `1/3` is equivalent to `fractions.Fraction(1, 3)`.

The Hy REPL prints output in Python syntax by default:

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

But if you start Hy like this (a shell alias might be helpful):

```
$ hy --repl-output-fn=hy.contrib.hy-repr.hy-repr
```

the interactive mode will use *hy-repr* instead of Python's native `repr` function to print out values, so you'll see values in Hy syntax:

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

2.3 Basic operations

Set variables with *setv*:

```
(setv zone-plane 8)
```

Access the elements of a list, dictionary, or other data structure with *get*:

```
(setv fruit ["apple" "banana" "cantaloupe"])
(print (get fruit 0))  ; => apple
(setv (get fruit 1) "durian")
(print (get fruit 1))  ; => durian
```

Access a range of elements in an ordered structure with *cut*:

```
(print (cut "abcdef" 1 4))  ; => bcd
```

Conditional logic can be built with *if / if* / if-not*:

```
(if (= 1 1)
  (print "Math works. The universe is safe.")
  (print "Math has failed. The universe is doomed."))
```

As in this example, `if` is called like `(if CONDITION THEN ELSE)`. It executes and returns the form `THEN` if `CONDITION` is true (according to `bool`) and `ELSE` otherwise. If `ELSE` is omitted, `None` is used in its place.

What if you want to use more than form in place of the `THEN` or `ELSE` clauses, or in place of `CONDITION`, for that matter? Use the special operator *do* (known more traditionally in Lisp as `progn`), which combines several forms into one, returning the last:

```
(if (do (print "Let's check.") (= 1 1))
  (do
    (print "Math works.")
    (print "The universe is safe."))
  (do
    (print "Math has failed.")
    (print "The universe is doomed.")))
```

For branching on more than one case, try *cond*:

```
(setv somevar 33)
(cond
 [(> somevar 50)
  (print "That variable is too big!")]
 [(< somevar 10)
  (print "That variable is too small!")]
 [True
  (print "That variable is jussssst right!")])
```

The macro `(when CONDITION THEN-1 THEN-2 ...)` is shorthand for `(if CONDITION (do THEN-1 THEN-2 ...))`. `unless` works the same as `when`, but inverts the condition with `not`.

Hy's basic loops are *while* and *for*:

```
(setv x 3)
(while (> x 0)
  (print x)
  (setv x (- x 1)))  ; => 3 2 1

(for [x [1 2 3]]
  (print x))         ; => 1 2 3
```

A more functional way to iterate is provided by the comprehension forms such as *lfor*. Whereas `for` always returns `None`, `lfor` returns a list with one element per iteration.

```
(print (lfor x [1 2 3] (* x 2)))  ; => [2, 4, 6]
```

2.4 Functions, classes, and modules

Define named functions with *defn*:

```
(defn fib [n]
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))))
(print (fib 8))  ; => 21
```

Define anonymous functions with *fn*:

```
(print (list (filter (fn [x] (% x 2)) (range 10))))
  ; => [1, 3, 5, 7, 9]
```

Special symbols in the parameter list of `defn` or `fn` allow you to indicate optional arguments, provide default values, and collect unlisted arguments:

```
(defn test [a b &optional c [d "x"] &rest e]
  [a b c d e])
(print (test 1 2))             ; => [1, 2, None, 'x', ()]
(print (test 1 2 3 4 5 6 7))   ; => [1, 2, 3, 4, (5, 6, 7)]
```

Set a function parameter by name with a `:keyword`:

```
(test 1 2 :d "y")              ; => [1, 2, None, 'y', ()]
```

Define classes with *defclass*:

```
(defclass FooBar []
  (defn __init__ [self x]
    (setv self.x x))
  (defn get-x [self]
    self.x))
```

Here we create a new instance `fb` of `FooBar` and access its attributes by various means:

```
(setv fb (FooBar 15))
(print fb.x)         ; => 15
(print (. fb x))     ; => 15
(print (.get-x fb))  ; => 15
(print (fb.get-x))   ; => 15
```

Note that syntax like `fb.x` and `fb.get-x` only works when the object being invoked (`fb`, in this case) is a simple variable name. To get an attribute or call a method of an arbitrary form `FORM`, you must use the syntax `(. FORM x)` or `(.get-x FORM)`.

Access an external module, whether written in Python or Hy, with *import*:

```
(import math)
(print (math.sqrt 2))  ; => 1.4142135623730951
```

Python can import a Hy module like any other module so long as Hy itself has been imported first, which, of course, must have already happened if you're running a Hy program.

2.5 Macros

Macros are the basic metaprogramming tool of Lisp. A macro is a function that is called at compile time (i.e., when a Hy program is being translated to Python `ast` objects) and returns code, which becomes part of the final program. Here's a simple example:

```
(print "Executing")
(defmacro m []
  (print "Now for a slow computation")
  (setv x (% (** 10 10 7) 3))
  (print "Done computing")
  x)
(print "Value:" (m))
(print "Done executing")
```

If you run this program twice in a row, you'll see this:

```
$ hy example.hy
Now for a slow computation
Done computing
Executing
Value: 1
Done executing
$ hy example.hy
Executing
Value: 1
Done executing
```

The slow computation is performed while compiling the program on its first invocation. Only after the whole program is compiled does normal execution begin from the top, printing "Executing". When the program is called a second time, it is run from the previously compiled bytecode, which is equivalent to simply:

```
(print "Executing")
(print "Value:" 1)
(print "Done executing")
```

Our macro `m` has an especially simple return value, an integer, which at compile-time is converted to an integer literal. In general, macros can return arbitrary Hy forms to be executed as code. There are several special operators and macros that make it easy to construct forms programmatically, such as *quote* (`'`), *quasiquote* (`` ` ``), *unquote* (`~`), and *defmacro!*. The previous chapter has *a simple example* of using `` ` `` and `~` to define a new control construct `do-while`.

Sometimes it's nice to be able to call a one-parameter macro without parentheses. Tag macros allow this. The name of a tag macro is often just one character long, but since Hy allows most Unicode characters in the name of a macro (or ordinary variable), you won't out of characters soon.

```
=> (deftag  [code]
...   (setv op (last code) params (list (butlast code)))
...   `(~op ~@params))
=> #(1 2 3 +)
6
```

What if you want to use a macro that's defined in a different module? `import` won't help, because it merely translates to a Python `import` statement that's executed at run-time, and macros are expanded at compile-time, that is, during the translation from Hy to Python. Instead, use *require*, which imports the module and makes macros available at compile-time. `require` uses the same syntax as `import`.

```
=> (require tutorial.macros)
=> (tutorial.macros.rev (1 2 3 +))
6
```

2.6 Next steps

You now know enough to be dangerous with Hy. You may now smile villainously and sneak off to your Hydeaway to do unspeakable things.

Refer to Python's documention for the details of Python semantics, and the rest of this manual for Hy-specific features. Like Hy itself, the manual is incomplete, but *contributions* are always welcome.

CHAPTER 3

Hy Style Guide

The Hy style guide intends to be a set of ground rules for the Hyve (yes, the Hy community prides itself in appending Hy to everything) to write idiomatic Hy code. Hy derives a lot from Clojure & Common Lisp, while always maintaining Python interoperability.

3.1 Layout & Indentation

The #1 complaint about Lisp?

> *It's too weird looking with all those parentheses! How do you even* **read** *that?*

And, they're right! Lisp was originally much too hard to read. Then they figured out layout and indentation. And it was glorious.

3.1.1 The Three Laws

Here's the secret: *Real Lispers don't count the brackets.* They fade into the background. When reading Lisp, disregard the trailing closing brackets—those are for the computer, not the human. As in Python, read the code structure by indentation.

Lisp code is made of trees—Abstract Syntax Trees—not strings. S-expressions are very direct textual representation of AST. That's the level of *homoiconicity*—the level Lisp macros operate on. It's not like the C-preprocessor or Python's interpolated eval-string tricks that see code as just letters. That's not how to think of Lisp code; think tree structure, not delimiters.

1. Closing brackets must NEVER be left alone, sad and lonesome on their own line.

```
;; PREFERRED
(defn fib [n]
  (if (<= n 2)
      n
      (+ (fib (- n 1))
```

(continues on next page)

(continued from previous page)

```
         (fib (- n 2)))))  ; Lots of Irritating Superfluous Parentheses
                                ; L.I.S.P. ;))

;; How the experienced Lisper sees it. Indented trees. Like Python.
(defn fib [n
  (if (<= n 2
      n
      (+ (fib (- n 1
         (fib (- n 2

;; BAD
;; We're trying to ignore them and you want to give them their own line?
;; Hysterically ridiculous.
(defn fib [
    n
]  ; My eyes!
  (if (<= n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))
  )
)  ; GAH, BURN IT WITH FIRE!
```

2. New lines must ALWAYS be indented past their parent opening bracket.

```
;; PREFERRED
(foo (, arg1
        arg2))

;; BAD. And evil.
;; Same bracket structure as above, but not enough indent.
(foo (, arg1
  arg2))

;; PREFERRED. Same indent as above, but now it matches the brackets.
(fn [arg]
  arg)

;; Remember, when reading Lisp, you ignore the trailing brackets.
;; Look at what happens if we remove them.
;; Can you tell where they should go by the indentation?

(foo (, arg1
        arg2

(foo (, arg1
  arg2

(fn [arg
  arg

;; See how the structure of those last two became indistinguishable?

;; Reconstruction of the bad example by indent.
;; Not what we started with, is it?
(foo (, arg1)
  arg2)
```

(continues on next page)

(continued from previous page)

```
;; Beware of brackets with reader syntax.
;; You still have to indent past them.

;; BAD
`#{(foo)
 ~@[(bar)
  1 2]}

;; Above, no trail.
`#{(foo
 ~@[(bar
  1 2

;; Reconstruction. Is. Wrong.
`#{(foo)}
 ~@[(bar)]
  1 2

;; PREFERRED
`#{(foo)
   ~@[(bar)
       1
       2]}

;; OK
;; A string is an atom, not a HySequence.
(foo "abc
  xyz")

;; Still readable without trailing brackets.
(foo "abc
  xyz"  ; Double-quote isn't a closing bracket. Don't ignore it.
```

3. New lines must NEVER be indented past the previous element's opening bracket.

```
;; BAD
((get-fn q)
  x
  y)

;; The above with trailing brackets removed. See the problem?
((get-fn q
  x
  y

;; By indentation, this is where the brackets should go.
((get-fn q
  x
  y))

;; OK
((get-fn q) x
            y)

;; The above without trailing brackets. Still OK (for humans).
((get-fn q) x  ; The ) on this line isn't trailing!
            y
```

(continues on next page)

(continued from previous page)

```
;; PREFERRED, since the ) should end the line.
((get-fn q)
 x
 y)
```

3.1.2 Limits

Follow PEP 8 rules for line limits, viz.

- 72 columns max for text (docstrings and comments).
- 79 columns max for other code, OR
- 99 for other code if primarily maintained by a team that can agree to 99.

3.1.3 Whitespace

AVOID trailing spaces. They suck!

AVOID tabs in code. Indent with spaces only.

PREFER the `\t` escape sequence to literal tab characters in one-line string literals.

- Literal tabs are OK inside multiline strings if you also add a warning comment.
- But `\t` is still PREFERRED in multiline strings.
- The comment should PREFERABLY appear just before the string.
- But a blanket warning at the top of a function, class, or file is OK.

3.1.4 Alignment

Line up arguments to function calls when splitting over multiple lines.

- The first argument PREFERABLY stays on the first line with the function name,
- but may instead start on the next line indented one space past its parent bracket.

```
;; PREFERRED. All args aligned with first arg.
(foofunction arg1
             (barfunction bararg1
                          bararg2
                          bararg3)  ; Aligned with bararg1.
             arg3)

;; BAD
(foofunction arg1
             (barfunction bararg1
               bararg2  ; Wrong. Looks like a macro body.
                    bararg3)  ; Why?!
             arg3)

;; PREFERRED. Args can all go on one line if it fits.
(foofunction arg1
```

(continues on next page)

(continued from previous page)

```
              (barfunction bararg1 bararg2 bararg3)
             arg3)

;; OK. Args not on first line, but still aligned.
(foofunction
  arg1  ; Indented one column past parent (
  (barfunction
    bararg1  ; Indent again.
    bararg2  ; Aligned with bararg1.
    bararg3)
  arg3)  ; Aligned with arg1.
```

3.1.5 Hold it Open

If you need to separate a bracket trail use a `#_ /` comment to hold it open. This avoids violating law #1.

```
;; PREFERRED
[(foo)
 (bar)
 (baz)]

;; OK, especially if the list is long. (Not that three is long.)
;; This is better for version control line diffs.
[  ; Opening brackets can't be "trailing closing brackets" btw.
 (foo)
 (bar)
 (baz)
 #_ /]  ; Nothing to see here. Move along.

;; Examples of commenting out items at the end of a list follow.
;; As with typing things in the REPL, these cases are less important
;; if you're the only one that sees them. But even so, maintaining
;; good style can help prevent errors.

;; BAD and a syntax error. Lost a bracket.
[(foo)
 ;; (bar)
 ;; (baz)]

;; BAD. Broke law #1.
[(foo)
 ;; (bar)
 ;; (baz)
 ]

;; PREFERRED
;; The discard syntax respects code structure,
;; so it's less likely to cause errors.
[(foo)
 #_(bar)
 #_(baz)]

;; OK. Adding a final discarded element makes line comments safer.
[(foo)
 ;; (bar)
```

(continues on next page)

(continued from previous page)

```
 ;; (baz)
 #_ /]
```

3.1.6 Snuggle

Brackets like to snuggle, don't leave them out in the cold!

```
;; PREFERRED
[1 2 3]
(foo (bar 2))

;; BAD
[ 1 2 3 ]
( foo ( bar 2 ) )

;; BAD. And ugly.
[ 1 2 3]
(foo( bar 2) )
```

3.1.7 Grouping

Use whitespace to show implicit groups, but be consistent within a form.

```
;; Older Lisps would typically wrap such groups in even more parentheses.
;; (The Common Lisp LOOP macro was a notable exception.)
;; But Hy takes after Clojure, which has a lighter touch.

;; BAD. Can't tell key from value without counting
{1 9 2 8 3 7 4 6 5 5}

;; PREFERRED. This can fit on one line. Clojure would have used commas
;; here, but those aren't whitespace in Hy. Use extra spaces instead.
{1 9  2 8  3 7  4 6  5 5}

;; OK. And preferred if it couldn't fit on one line.
{1 9
 2 8
 3 7
 4 6
 5 5}  ; Newlines show key-value pairs in dict.

;; BAD
;; This grouping makes no sense.
#{1 2
  3 4}  ; It's a set, so why are there pairs?

;; BAD
;; This grouping also makes no sense. But, it could be OK in a macro or
;; something if this grouping was somehow meaningful there.
[1
 1 2
 1 2 3]  ; wHy do you like random patterns? [sic pun, sorry]
```

(continues on next page)

(continued from previous page)

```
;; Be consistent. Separate all groups the same way in a form.

;; BAD
{1 9  2 8
 3 7  4 6  5 5}  ; Pick one or the other!

;; BAD
{1 9  2 8 3 7  4 6  5 5}  ; You forgot something.

;; Groups of one must also be consistent.

;; PREFERRED
(foo 1 2 3)  ; No need for extra spaces here.

;; OK, but you could have fit this on one line.
(foo 1
     2
     3)

;; OK, but you still could have fit this on one line.
[1
 2]

;; BAD
(foo 1 2  ; This isn't a pair?
     3)  ; Lines or spaces--pick one or the other!

;; PREFERRRED
(foofunction (make-arg)
             (get-arg)
             #tag(do-stuff)  ; Tags belong with what they tag.
             #* args  ; #* goes with what it unpacks.
             :foo spam
             :bar eggs  ; Keyword args are also pairs. Group them.
             #** kwargs)

;; PREFERRED. Spaces divide groups on one line.
(quux :foo spam  :bar eggs  #* with-spam)
{:foo spam  :bar eggs}

;; OK. The colon is still enough to indicate groups.
(quux :foo spam :bar eggs #* with-spam)
{:foo spam :bar eggs}
;; OK.
("foo" spam "bar" eggs}

;; BAD. Can't tell key from value.
(quux :foo :spam :bar :eggs :baz :bacon)
{:foo :spam :bar :eggs :baz :bacon}
{"foo" "spam" "bar" "eggs" "baz" "bacon"}

;; PREFERRED
(quux :foo :spam  :bar :eggs  :baz :bacon)
{:foo :spam  :bar :eggs  :baz :bacon}
{"foo" "spam"  "bar" "eggs"  "baz" "bacon"}

;; OK. Yep, those are pairs too.
```

(continues on next page)

(continued from previous page)

```
(setv x 1
      y 2)

;; PREFERRED. This fits on one line.
(setv x 1  y 2)

;; BAD. Doesn't separate groups.
(print (if (< n 0.0)
           "negative"
           (= n 0.0)
           "zero"
           (> n 0.0)
           "positive"
           "not a number"))

;; BAD. And evil. Broke law #3. Shows groups but args aren't aligned.
(print (if (< n 0.0)
               "negative"
           (= n 0.0)
               "zero"
           (> n 0.0)
               "positive"
           "not a number"))

;; BAD. Shows groups but args aren't aligned.
;; If the then-parts weren't atoms, this would break law #3.
(print (if (< n 0.0)
         "negative"
           (= n 0.0)
         "zero"
           (> n 0.0)
         "positive"
           "not a number"))

;; OK. Redundant (do) forms allow extra indent to show groups
;; without violating law #3.
(print (if (< n 0.0)
           (do
             "negative")
           (= n 0.0)
           (do
             "zero")
           (> n 0.0)
           (do
             "positive")
           "not a number"))
```

Separate toplevel forms (including toplevel comments not about a particular form) with a single blank line, rather than two as in Python.

- This can be omitted for tightly associated forms.

Methods within a defclass need not be separated by blank line.

3.1.8 Special Arguments

Macros and special forms are normally indented one space past the parent bracket, but can also have "special" arguments that are indented like function arguments.

- Macros with an `&rest body` argument contain an implicit `do`.
- The body is never special, but the arguments before it are.

```
;; PREFERRED
(assoc foo  ; foo is special
  "x" 1  ; remaining args are not special. Indent 2 spaces.
  "y" 2)

;; PREFERRED
;; The do form has no special args. Indent like a function call.
(do (foo)
    (bar)
    (baz))

;; OK
;; No special args to distinguish. This is also valid function indent.
(do
  (foo)
  (bar)
  (baz))

;; PREFERRED
(defn fib [n]
  (if (<= n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

;; OK
(defn fib
      [n]  ; name and argslist are special. Indent like function args.
  ;; The defn body is not special. Indent 1 space past parent bracket.
  (if (<= n 2)
      n
    (+ (fib (- n 1))  ; Emacs-style else indent.
       (fib (- n 2)))))
```

3.1.9 Removing Whitespace

Removing whitespace can also make groups clearer.

```
;; lookups

;; OK
(. foo ["bar"])

;; PREFERRED
(. foo["bar"])

;; BAD. Doesn't show groups clearly.
(import foo foo [spam :as sp eggs :as eg] bar bar [bacon])
```

(continues on next page)

(continued from previous page)

```
;; OK. Extra spaces show groups.
(import foo  foo [spam :as sp  eggs :as eg]  bar  bar [bacon])

;; PREFERRED. Removing spaces is even clearer.
(import foo foo[spam :as sp  eggs :as eg] bar bar[bacon])

;; OK. Newlines show groups.
(import foo
        foo [spam :as sp
             eggs :as eg]
        bar
        bar [bacon])

;; PREFERRED, It's more consistent with the preferred one-line version.
(import foo
        foo[spam :as sp
            eggs :as eg]
        bar
        bar[bacon])

;; Avoid whitespace after tags.

;; Note which shows groups better.

;; BAD
(foofunction #tag "foo" #tag (foo) #* (get-args))

;; OK
(foofunction #tag "foo"  #tag (foo)  #* (get-args))

;; PREFERRED
(foofunction #tag"foo" #tag(foo) #*(get-args))

;; PREFERRED
;; Can't group these by removing whitespace. Use extra spaces instead.
(foofunction #x foo  #x bar  #* args)

;; OK
;; Same idea, but this could have fit on one line.
(foofunction #x foo
             #x bar
             #* args)

;; OK, but you don't need to separate function name from first arg.
(foofunction  #x foo  #x bar  #* args)

;; OK. But same idea.
;; No need to separate the first group from the function name.
(foofunction
  #x foo
  #x bar
  #* args)

;; PREFERRED. It's still clear what this is tagging.
;; And you don't have to re-indent.
#_
```

(continues on next page)

(continued from previous page)

```
(def foo []
  stuff)

;; OK, but more work.
#_(def foo []
    stuff)

;; BAD, you messed up the indent and broke law #2.
#_(def foo []
  stuff)

;; BAD, keep the tag grouped with its argument.
#_

(def foo []
  stuff)
```

3.1.10 Close Bracket, Close Line

A *single* closing bracket SHOULD end the line, unless it's in the middle of an implicit group.

- If the forms are small and simple you can maybe leave them on one line.

A *train* of closing brackets MUST end the line.

```
;; One-liners are overrated.
;; Maybe OK if you're just typing into the REPL.
;; But even then, maintaining good style can help prevent errors.

;; BAD. One-liner is too hard to read.
(defn fib [n] (if (<= n 2) n (+ (fib (- n 1)) (fib (- n 2)))))

;; BAD. Getting better, but the first line is still too complex.
(defn fib [n] (if (<= n 2) n (+ (fib (- n 1))
                                (fib (- n 2)))))
;; OK. Barely.
(defn fib [n]
  (if (<= n 2) n (+ (fib (- n 1))  ; This line is pushing it.
                    (fib (- n 2)))))

;; OK
(defn fib [n]  ; Saw a "]", newline.
  (if (<= n 2)  ; OK to break here, since there's only one pair.
      n
    (+ (fib (- n 1))  ; Whitespace separation (Emacs else-indent).
       (fib (- n 2)))))

;; OK
(defn fib [n]  ; Saw a "]", end line. (Margin comments don't count.)
  (if (<= n 2) n  ; Saw a ")", but it's in a pair starting in this line.
      (+ (fib (- n 1))  ; Saw a "))" MUST end line.
         (fib (- n 2)))))

;; OK. Pairs.
(print (if (< n 0.0) "negative"  ; Single ) inside group. No break.
           (= n 0.0) "zero"
```

(continues on next page)

(continued from previous page)

```
           (> n 0.0) "positive"
           :else "not a number"))  ; :else is not magic; True works too.

;; OK. Avoided line breaks at single ) to show pairs.
(print (if (< n 0.0) "negative"
           (= n 0.0) "zero"
           (> n 0.0) (do (do-foo)  ; Single ) inside group. No break.
                         (do-bar)
                         "positive")
           "not a number"))  ; Implicit else is PREFERRED.

;; BAD
(print (if (< n 0.0) "negative"
           (= n 0.0) "zero"
           (and (even? n)
                (> n 0.0)) "even-positive"  ; Bad. "))" must break.
           (> n 0.0) "positive"
           "not a number"))

;; BAD
(print (if (< n 0.0) "negative"
           (= n 0.0) "zero"
           (and (even? n)
                (> n 0.0)) (do (do-foo)  ; Y U no break?
                               (do-bar)
                               "even-positive")
           (> n 0.0) "positive"
           "not a number"))

;; OK. Blank line separates multiline groups.
(print (if (< n 0.0) "negative"

           (= n 0.0) "zero"

           (and (even? n)
                (> n 0.0))
           (do (do-foo)
               (do-bar)
               "even-positive")

           (> n 0.0) "positive"

           "not a number"))

;; BAD. Groups are not separated consistently.
(print (if (< n 0.0) "negative"
           (= n 0.0) "zero"

           (> n 0.0)
           (do (do-foo)
               "positive")

           "not a number"))

;; OK. Single )'s and forms are simple enough.
(with [f (open "names.txt")]
  (-> (.read f) .strip (.replace "\"" "") (.split ",") sorted)))
```

(continues on next page)

(continued from previous page)

```
;; PREFERRED. Even so, this version is much clearer.
(with [f (open "names.txt")]
  (-> (.read f)
      .strip
      (.replace "\"" "")
      (.split ",")
      sorted)))
```

3.1.11 Comments

Prefer docstrings to comments where applicable—in `fn`, `defclass`, at the top of the module, and in any other macros derived from these that can take a docstring (e.g. `defmacro/g!`, `deftag`, `defn`).

Docstrings contents follow the same conventions as Python.

The `(comment)` macro is still subject to the three laws. If you're tempted to violate them, consider discarding a string instead with `#_`.

Semicolon comments always have one space between the semicolon and the start of the comment. Also, try to not comment the obvious.

Comments with more than a single word should start with a capital letter and use punctuation.

Separate sentences with a single space.

```
;; This commentary is not about a particular form.
;; These can span multiple lines. Limit them to column 72, per PEP 8.
;; Separate them from the next form or form comment with a blank line.

;; PREFERRED.
(setv ind (dec x))  ; Indexing starts from 0,
                    ; margin comment continues on new line.

;; OK
;; Style-compliant but just states the obvious.
(setv ind (dec x))  ; Sets index to x-1.

;; BAD
(setv ind (dec x));typing words for fun

;; Comment about the whole foofunction call.
;; These can also span mulitple lines.
(foofunction ;; Form comment about (get-arg1). Not a margin comment!
             (get-arg1)
             ;; Form comment about arg2. The indent matches.
             arg2)
```

Indent form comments at the same level as the form they're commenting about; they must always start with exactly two semicolons `;;`. Form comments appear directly above what they're commenting on, never below.

General toplevel commentary is not indented; these must always start with exactly two semicolons `;;` and be separated from the next form with a blank line. For long commentary, consider using a `#_` applied to a string for this purpose instead.

Margin comments start two spaces from the end of the code; they must always start with a single semicolon `;`. Margin comments may be continued on the next line.

When commenting out entire forms, prefer the `#_` syntax. But if you do need line comments, use the more general double-colon form.

3.2 Coding Style

3.2.1 Pythonic Names

Use Python's naming conventions where still applicable to Hy.

- The first parameter of a method is `self`,
- of a classmethod is `cls`.

3.2.2 Threading Macros

PREFER the threading macro or the threading tail macros when encountering deeply nested s-expressions. However, be judicious when using them. Do use them when clarity and readability improves; do not construct convoluted, hard to understand expressions.

```
;; BAD. Not wrong, but could be much clearer with a threading macro.
(setv NAMES
  (with [f (open "names.txt")]
    (sorted (.split (.replace (.strip (.read f))
                              "\""
                              "")
                    ","))))

;; PREFERRED. This compiles exactly the same way as the above.
(setv NAMES
  (with [f (open "names.txt")]
    (-> (.read f)
        .strip
        (.replace "\"" "")
        (.split ",")
        sorted)))

;; BAD. Probably. The macro makes it less clear in this case.
(defn square? [x]
  (->> 2
       (pow (int (sqrt x)))
       (= x)))

;; OK. Much clearer that the previous example above.
(defn square? [x]
  (-> x
      sqrt
      int
      (pow 2)
      (= x))

;; PREFERRED. Judicious use.
;; You don't have to thread everything if it improves clarity.
(defn square? [x]
  (= x (-> x sqrt int (pow 2))))
```

(continues on next page)

(continued from previous page)

```
;; OK. Still clear enough with no threading macro this time.
(defn square? [x]
  (= x (pow (int (sqrt x))  ; saw a "))", break.
            2))  ; aligned with first arg to pow
```

3.2.3 Method Calls

Clojure-style dot notation is PREFERRED over the direct call of the object's method, though both will continue to be supported.

```
;; PREFERRED
(with [fd (open "/etc/passwd")]
  (print (.readlines fd)))

;; OK
(with [fd (open "/etc/passwd")]
  (print (fd.readlines)))
```

3.2.4 Use More Arguments

PREFER using multiple arguments to multiple forms. But judicious use of redundant forms can clarify intent. AVOID the separating blank line for toplevel forms in this case.

```
;; BAD
(setv x 1)
(setv y 2)
(setv z 3)
(setv foo 9)
(setv bar 10)

;; OK
(setv x 1
      y 2
      z 3
      foo 9
      bar 10)

;; PREFERRED
(setv x 1
      y 2
      z 3)
(setv foo 9
      bar 10)
```

3.2.5 Imports

As in Python, group imports.

- Standard library imports (including Hy's) first.
- Then third-party modules,

- and finally internal modules.

PREFER one import form for each group.

PREFER alphabetical order within groups.

Require macros before any imports and group them the same way.

But sometimes imports are conditional or must be ordered a certain way for programmatic reasons, which is OK.

```
;; PREFERRED
(require hy.extra.anaphoric [%])
(require thirdparty [some-macro])
(require mymacros [my-macro])

(import json re)
(import numpy :as np
        pandas :as pd)
(import mymodule1)
```

3.2.6 Underscores

Prefer hyphens when separating words.

- PREFERRED `foo-bar`
- BAD `foo_bar`

Don't use leading hyphens, except for "operators" or symbols meant to be read as including one, e.g. `-Inf`, `->foo`.

Prefix private names with an underscore, not a dash. to avoid confusion with negated literals like `-Inf`, `-42` or `-4/2`.

- PREFERRED `_x`
- BAD `-x`

Write Python's magic "dunder" names the same as in Python. Like `__init__`, not `--init--` or otherwise, to be consistent with the private names rule above.

Private names should still separate words using dashes instead of underscores, to be consistent with non-private parameter names and such that need the same name sans prefix, like `foo-bar`, not `foo_bar`.

- PREFERRED `_foo-bar`
- BAD `_foo_bar`

```
;; BAD
;; What are you doing?
(_= spam 2)  ; Throwing it away?
(_ 100 7)  ; i18n?

;; PREFERRED
;; Clearly subtraction.
(-= spam 2)
(- 100 7)

;; BAD
;; This looks weird.
(_>> foo bar baz)

;; PREFERRED
```

(continues on next page)

(continued from previous page)

```
;; OH, it's an arrow!
(->> foo bar baz)

;; Negative x?
(setv -x 100)  ; BAD. Unless you really meant that?

;; PREFERRED
;; Oh, it's just a module private.
(setv _x 100)

;; BAD
(class Foo []
  (defn __init-- [self] ...))

;; OK
(class Foo []
  ;; Less weird?
  (defn --init-- [self] ...))

;; PREFERRED
(class Foo []
  (defn __init__ [self] ...))

;; OK, but would be module private. (No import *)
(def ->dict [&rest pairs]
  (dict (partition pairs)))
```

3.3 Thanks

- This guide is heavily inspired from @paultag ‘s blog post Hy Survival Guide
- The Clojure Style Guide
- Parinfer and Parlinter (the three laws)
- The Community Scheme Wiki scheme-style (ending bracket ends the line)
- Riastradh’s Lisp Style Rules (Lisp programmers do not . . . Azathoth forbid, count brackets)

CHAPTER 4

Documentation Index

Contents:

4.1 Command Line Interface

4.1.1 hy

Command Line Options

-c <command>

Execute the Hy code in *command*.

```
$ hy -c "(print (+ 2 2))"
4
```

-i <command>

Execute the Hy code in *command*, then stay in REPL.

-m <module>

Execute the Hy code in *module*, including `defmain` if defined.

The *-m* flag terminates the options list so that all arguments after the *module* name are passed to the module in `sys.argv`.

New in version 0.11.0.

--spy

Print equivalent Python code before executing in REPL. For example:

```
=> (defn salutationsnm [name] (print (+ "Hy " name "!")))
def salutationsnm(name):
    return print(((u'Hy ' + name) + u'!'))
=> (salutationsnm "YourName")
```

(continues on next page)

(continued from previous page)

```
salutationsnm(u'YourName')
Hy YourName!
=>
```

–*spy* only works on REPL mode. .. versionadded:: 0.9.11

--repl-output-fn
Format REPL output using specific function (e.g., hy.contrib.hy-repr.hy-repr)

New in version 0.13.0.

-v
Print the Hy version number and exit.

4.1.2 hyc

Command Line Options

file[, fileN]
Compile Hy code to Python bytecode. For example, save the following code as `hyname.hy`:

```
(defn hy-hy [name]
  (print (+ "Hy " name "!")))

(hy-hy "Afroman")
```

Then run:

```
$ hyc hyname.hy
$ python hyname.pyc
Hy Afroman!
```

4.1.3 hy2py

New in version 0.10.1.

Command Line Options

-s
--with-source
Show the parsed source structure.

-a
--with-ast
Show the generated AST.

-np
--without-python
Do not show the Python code generated from the AST.

4.2 Hy <-> Python interop

Despite being a Lisp, Hy aims to be fully compatible with Python. That means every Python module or package can be imported in Hy code, and vice versa.

Mangling allows variable names to be spelled differently in Hy and Python. For example, Python's `str.format_map` can be written `str.format-map` in Hy, and a Hy function named `valid?` would be called `is_valid` in Python. In Python, you can import Hy's core functions `mangle` and `unmangle` directly from the `hy` package.

4.2.1 Using Python from Hy

You can embed Python code directly into a Hy program with the special operators *py* and *pys*.

Using a Python module from Hy is nice and easy: you just have to *import* it. If you have the following in `greetings.py` in Python:

```
def greet(name):
    print("hello," name)
```

You can use it in Hy:

```
(import greetings)
(.greet greetings "foo") ; prints "hello, foo"
```

You can also import `.pyc` bytecode files, of course.

4.2.2 Using Hy from Python

Suppose you have written some useful utilities in Hy, and you want to use them in regular Python, or to share them with others as a package. Or suppose you work with somebody else, who doesn't like Hy (!), and only uses Python.

In any case, you need to know how to use Hy from Python. Fear not, for it is easy.

If you save the following in `greetings.hy`:

```
(setv this-will-have-underscores "See?")
(defn greet [name] (print "Hello from Hy," name))
```

Then you can use it directly from Python, by importing Hy before importing the module. In Python:

```
import hy
import greetings

greetings.greet("Foo") # prints "Hello from Hy, Foo"
print(greetings.this_will_have_underscores) # prints "See?"
```

If you create a package with Hy code, and you do the `import hy` in `__init__.py`, you can then directly include the package. Of course, Hy still has to be installed.

Compiled files

You can also compile a module with `hyc`, which gives you a `.pyc` file. You can import that file. Hy does not *really* need to be installed ; however, if in your code, you use any symbol from *Hy Core*, a corresponding `import` statement will be generated, and Hy will have to be installed.

Even if you do not use a Hy builtin, but just another function or variable with the name of a Hy builtin, the `import` will be generated. For example, the previous code causes the import of `name` from `hy.core.language`.

Bottom line: in most cases, Hy has to be installed.

Launching a Hy REPL from Python

You can use the function `run_repl()` to launch the Hy REPL from Python:

```
>>> import hy.cmdline
>>> hy.cmdline.run_repl()
hy 0.12.1 using CPython(default) 3.6.0 on Linux
=> (defn foo [] (print "bar"))
=> (test)
bar
```

If you want to print the Python code Hy generates for you, use the `spy` argument:

```
>>> import hy.cmdline
>>> hy.cmdline.run_repl(spy=True)
hy 0.12.1 using CPython(default) 3.6.0 on Linux
=> (defn test [] (print "bar"))
def test():
    return print('bar')
=> (test)
test()
bar
```

Evaluating strings of Hy code from Python

Evaluating a string (or `file` object) containing a Hy expression requires two separate steps. First, use the `read_str` function (or `read` for a `file` object) to turn the expression into a Hy model:

```
>>> import hy
>>> expr = hy.read_str("(- (/ (+ 1 3 88) 2) 8)")
```

Then, use the `eval` function to evaluate it:

```
>>> hy.eval(expr)
38.0
```

4.3 Syntax

4.3.1 identifiers

An identifier consists of a nonempty sequence of Unicode characters that are not whitespace nor any of the following: `( ) [ ] { } ' "`. Hy first tries to parse each identifier into a numeric literal, then into a keyword if that fails, and finally into a symbol if that fails.

4.3.2 numeric literals

In addition to regular numbers, standard notation from Python for non-base 10 integers is used. `0x` for Hex, `0o` for Octal, `0b` for Binary.

```
(print 0x80 0b11101 0o102 30)
```

Underscores and commas can appear anywhere in a numeric literal except the very beginning. They have no effect on the value of the literal, but they're useful for visually separating digits.

```
(print 10,000,000,000 10_000_000_000)
```

Unlike Python, Hy provides literal forms for NaN and infinity: `NaN`, `Inf`, and `-Inf`.

4.3.3 string literals

Hy allows double-quoted strings (e.g., `"hello"`), but not single-quoted strings like Python. The single-quote character `'` is reserved for preventing the evaluation of a form (e.g., `'(+ 1 1)`), as in most Lisps.

Python's so-called triple-quoted strings (e.g., `'''hello'''` and `"""hello"""`) aren't supported. However, in Hy, unlike Python, any string literal can contain newlines. Furthermore, Hy supports an alternative form of string literal called a "bracket string" similar to Lua's long brackets. Bracket strings have customizable delimiters, like the here-documents of other languages. A bracket string begins with `#[FOO[` and ends with `]FOO]`, where `FOO` is any string not containing `[` or `]`, including the empty string. (If `FOO` is exactly `f` or begins with `f-`, the bracket string is interpreted as a *format string*.) For example:

```
=> (print #[["That's very kind of yuo [sic]" Tom wrote back.]])
"That's very kind of yuo [sic]" Tom wrote back.
=> (print #[==[1 + 1 = 2]==])
1 + 1 = 2
```

A bracket string can contain newlines, but if it begins with one, the newline is removed, so you can begin the content of a bracket string on the line following the opening delimiter with no effect on the content. Any leading newlines past the first are preserved.

Plain string literals support a variety of backslash escapes. To create a "raw string" that interprets all backslashes literally, prefix the string with `r`, as in `r"slash\not"`. Bracket strings are always raw strings and don't allow the `r` prefix.

Like Python, Hy treats all string literals as sequences of Unicode characters by default. You may prefix a plain string literal (but not a bracket string) with `b` to treat it as a sequence of bytes.

Unlike Python, Hy only recognizes string prefixes (`r`, etc.) in lowercase.

4.3.4 format strings

A format string (or "f-string", or "formatted string literal") is a string literal with embedded code, possibly accompanied by formatting commands. Hy f-strings work much like Python f-strings except that the embedded code is in Hy rather than Python, and they're supported on all versions of Python.

```
=> (print f"The sum is {(+ 1 1)}.")
The sum is 2.
```

Since `!` and `:` are identifier characters in Hy, Hy decides where the code in a replacement field ends, and any conversion or format specifier begins, by parsing exactly one form. You can use `do` to combine several forms into one, as usual. Whitespace may be necessary to terminate the form:

```
=> (setv foo "a")
=> (print f"{foo:x<5}")
...
NameError: name 'hyx_fooXcolonXxXlessHthan_signX5' is not defined
=> (print f"{foo :x<5}")
axxxx
```

Unlike Python, whitespace is allowed between a conversion and a format specifier.

Also unlike Python, comments and backslashes are allowed in replacement fields. Hy's lexer will still process the whole format string normally, like any other string, before any replacement fields are considered, so you may need to backslash your backslashes, and you can't comment out a closing brace or the string delimiter.

4.3.5 keywords

An identifier headed by a colon, such as `:foo`, is a keyword. If a literal keyword appears in a function call, it's used to indicate a keyword argument rather than passed in as a value. For example, `(f :foo 3)` calls the function `f` with the keyword argument named `foo` set to `3`. Hence, trying to call a function on a literal keyword may fail: `(f :foo)` yields the error `Keyword argument :foo needs a value`. To avoid this, you can quote the keyword, as in `(f ':foo)`, or use it as the value of another keyword argument, as in `(f :arg :foo)`.

Keywords can be called like functions as shorthand for `get`. `(:foo obj)` is equivalent to `(get obj :foo)`. An optional `default` argument is also allowed: `(:foo obj 2)` or `(:foo obj :default 2)` returns `2` if `(get obj :foo)` raises a `KeyError`.

4.3.6 symbols

Symbols are identifiers that are neither legal numeric literals nor legal keywords. In most contexts, symbols are compiled to Python variable names. Some example symbols are `hello`, `+++`, `3fiddy`, `$40`, `justwrong`, and .

Since the rules for Hy symbols are much more permissive than the rules for Python identifiers, Hy uses a mangling algorithm to convert its own names to Python-legal names. The rules are:

- Convert all hyphens (`-`) to underscores (`_`). Thus, `foo-bar` becomes `foo_bar`.
- If the name ends with `?`, remove it and prepend `is_`. Thus, `tasty?` becomes `is_tasty`.
- If the name still isn't Python-legal, make the following changes. A name could be Python-illegal because it contains a character that's never legal in a Python name, it contains a character that's illegal in that position, or it's equal to a Python reserved word.
  - Prepend `hyx_` to the name.
  - Replace each illegal character with `XfooX`, where `foo` is the Unicode character name in lowercase, with spaces replaced by underscores and hyphens replaced by `H`. Replace `X` itself the same way. If the character doesn't have a name, use `U` followed by its code point in lowercase hexadecimal.

  Thus, `green` becomes `hyx_greenXshamrockX` and `if` becomes `hyx_if`.
- Finally, any added `hyx_` or `is_` is added after any leading underscores, because leading underscores have special significance to Python. Thus, `_tasty?` becomes `_is_tasty` instead of `is__tasty`.

Mangling isn't something you should have to think about often, but you may see mangled names in error messages, the output of `hy2py`, etc. A catch to be aware of is that mangling, as well as the inverse "unmangling" operation offered by the `unmangle` function, isn't one-to-one. Two different symbols can mangle to the same string and hence compile to the same Python variable. The chief practical consequence of this is that `-` and `_` are interchangeable in all symbol names, so you shouldn't assign to the one-character name `_` , or else you'll interfere with certain uses of subtraction.

4.3.7 discard prefix

Hy supports the Extensible Data Notation discard prefix, like Clojure. Any form prefixed with #_ is discarded instead of compiled. This completely removes the form so it doesn't evaluate to anything, not even None. It's often more useful than linewise comments for commenting out a form, because it respects code structure even when part of another form is on the same line. For example:

```
=> (print "Hy" "cruel" "World!")
Hy cruel World!
=> (print "Hy" #_"cruel" "World!")
Hy World!
=> (+ 1 1 (print "Math is hard!"))
Math is hard!
Traceback (most recent call last):
   ...
TypeError: unsupported operand type(s) for +: 'int' and 'NoneType'
=> (+ 1 1 #_(print "Math is hard!"))
2
```

4.4 Built-Ins

Hy features a number of special forms that are used to help generate correct Python AST. The following are "special" forms, which may have behavior that's slightly unexpected in some situations.

4.4.1 ^

The ^ symbol is used to denote annotations in three different contexts:

- Standalone variable annotations.
- Variable annotations in a setv call.
- Function argument annotations.

They implement PEP 526 and PEP 3107.

Here is some example syntax of all three usages:

```
; Annotate the variable x as an int (equivalent to `x: int`).
(^int x)
; Can annotate with expressions if needed (equivalent to `y: f(x)`).
(^(f x) y)

; Annotations with an assignment: each annotation (int, str) covers the term that
; immediately follows.
; Equivalent to: x: int = 1; y = 2; z: str = 3
(setv ^int x 1 y 2 ^str z 3)

; Annotate a as an int, c as an int, and b as a str.
; Equivalent to: def func(a: int, b: str = None, c: int = 1): ...
(defn func [^int a &optional ^str b ^int [c 1]] ...)
```

The rules are:

- The value to annotate with is the value that immediately follows the caret.

- There must be no space between the caret and the value to annotate, otherwise it will be interpreted as a bitwise XOR like the Python operator.
- The annotation always comes (and is evaluated) *before* the value being annotated. This is unlike Python, where it comes and is evaluated *after* the value being annotated.

Note that variable annotations are only supported on Python 3.6+.

For annotating items with generic types, the *of* macro will likely be of use.

4.4.2 .

New in version 0.10.0.

`.` is used to perform attribute access on objects. It uses a small DSL to allow quick access to attributes and items in a nested data structure.

For instance,

```
(. foo bar baz [(+ 1 2)] frob)
```

Compiles down to:

```
foo.bar.baz[1 + 2].frob
```

`.` compiles its first argument (in the example, *foo*) as the object on which to do the attribute dereference. It uses bare symbols as attributes to access (in the example, *bar*, *baz*, *frob*), and compiles the contents of lists (in the example, `[(+ 1 2)]`) for indexation. Other arguments raise a compilation error.

Access to unknown attributes raises an `AttributeError`. Access to unknown keys raises an `IndexError` (on lists and tuples) or a `KeyError` (on dictionaries).

4.4.3 ->

`->` (or the *threading macro*) is used to avoid nesting of expressions. The threading macro inserts each expression into the next expression's first argument place. The following code demonstrates this:

```
=> (defn output [a b] (print a b))
=> (-> (+ 4 6) (output 5))
10 5
```

4.4.4 ->>

`->>` (or the *threading tail macro*) is similar to the *threading macro*, but instead of inserting each expression into the next expression's first argument, it appends it as the last argument. The following code demonstrates this:

```
=> (defn output [a b] (print a b))
=> (->> (+ 4 6) (output 5))
5 10
```

4.4.5 and

`and` is used in logical expressions. It takes at least two parameters. If all parameters evaluate to `True`, the last parameter is returned. In any other case, the first false value will be returned. Example usage:

```
=> (and True False)
False

=> (and True True)
True

=> (and True 1)
1

=> (and True [] False True)
[]
```

Note: `and` short-circuits and stops evaluating parameters as soon as the first false is encountered.

```
=> (and False (print "hello"))
False
```

4.4.6 as->

New in version 0.12.0.

Expands to sequence of assignments to the provided name, starting with head. The previous result is thus available in the subsequent form. Returns the final result, and leaves the name bound to it in the local scope. This behaves much like the other threading macros, but requires you to specify the threading point per form via the name instead of always the first or last argument.

```
;; example how -> and as-> relate

=> (as-> 0 it
...       (inc it)
...       (inc it))
2

=> (-> 0 inc inc)
2

;; create data for our cuttlefish database

=> (setv data [{:name "hooded cuttlefish"
...              :classification {:subgenus "Acanthosepion"
...                               :species "Sepia prashadi"}
...              :discovered {:year 1936
...                           :name "Ronald Winckworth"}}
...             {:name "slender cuttlefish"
...              :classification {:subgenus "Doratosepion"
...                               :species "Sepia braggi"}
...              :discovered {:year 1907
...                           :name "Sir Joseph Cooke Verco"}}])

;; retrieve name of first entry
=> (as-> (first data) it
...       (:name it))
'hooded cuttlefish'
```

(continues on next page)

(continued from previous page)

```
;; retrieve species of first entry
=> (as-> (first data) it
...       (:classification it)
...       (:species it))
'Sepia prashadi'

;; find out who discovered slender cuttlefish
=> (as-> (filter (fn [entry] (= (:name entry)
...                            "slender cuttlefish")) data) it
...       (first it)
...       (:discovered it)
...       (:name it))
'Sir Joseph Cooke Verco'

;; more convoluted example to load web page and retrieve data from it
=> (import [urllib.request [urlopen]])
=> (as-> (urlopen "http://docs.hylang.org/en/stable/") it
...       (.read it)
...       (.decode it "utf-8")
...       (drop (.index it "Welcome") it)
...       (take 30 it)
...       (list it)
...       (.join "" it))
'Welcome to Hy's documentation!
```

Note: In these examples, the REPL will report a tuple (e.g. *('Sepia prashadi', 'Sepia prashadi')*) as the result, but only a single value is actually returned.

4.4.7 assert

`assert` is used to verify conditions while the program is running. If the condition is not met, an `AssertionError` is raised. `assert` may take one or two parameters. The first parameter is the condition to check, and it should evaluate to either `True` or `False`. The second parameter, optional, is a label for the assert, and is the string that will be raised with the `AssertionError`. For example:

```
(assert (= variable expected-value))

(assert False)
; AssertionError

(assert (= 1 2) "one should equal two")
; AssertionError: one should equal two
```

4.4.8 assoc

`assoc` is used to associate a key with a value in a dictionary or to set an index of a list to a value. It takes at least three parameters: the *data structure* to be modified, a *key* or *index*, and a *value*. If more than three parameters are used, it will associate in pairs.

Examples of usage:

```
=>(do
... (setv collection {})
... (assoc collection "Dog" "Bark")
... (print collection))
{u'Dog': u'Bark'}

=>(do
... (setv collection {})
... (assoc collection "Dog" "Bark" "Cat" "Meow")
... (print collection))
{u'Cat': u'Meow', u'Dog': u'Bark'}

=>(do
... (setv collection [1 2 3 4])
... (assoc collection 2 None)
... (print collection))
[1, 2, None, 4]
```

Note: `assoc` modifies the datastructure in place and returns `None`.

4.4.9 await

`await` creates an await expression. It takes exactly one argument: the object to wait for.

```
=> (import asyncio)
=> (defn/a main []
...   (print "hello")
...   (await (asyncio.sleep 1))
...   (print "world"))
=> (asyncio.run (main))
hello
world
```

4.4.10 break

`break` is used to break out from a loop. It terminates the loop immediately. The following example has an infinite `while` loop that is terminated as soon as the user enters *k*.

```
(while True (if (= "k" (input "? "))
              (break)
              (print "Try again")))
```

4.4.11 cmp

`cmp` creates a comparison expression. It isn't required for unchained comparisons, which have only one comparison operator, nor for chains of the same operator. For those cases, you can use the comparison operators directly with Hy's usual prefix syntax, as in `(= x 1)` or `(< 1 2 3)`. The use of `cmp` is to construct chains of heterogeneous operators, such as `x <= y < z`. It uses an infix syntax with the general form

```
(cmp ARG OP ARG OP ARG...)
```

Hence, `(cmp x <= y < z)` is equivalent to `(and (<= x y) (< y z))`, including short-circuiting, except that `y` is only evaluated once.

Each `ARG` is an arbitrary form, which does not itself use infix syntax. Use *py* if you want fully Python-style operator syntax. You can also nest `cmp` forms, although this is rarely useful. Each `OP` is a literal comparison operator; other forms that resolve to a comparison operator are not allowed.

At least two `ARG`s and one `OP` are required, and every `OP` must be followed by an `ARG`.

As elsewhere in Hy, the equality operator is spelled =, not == as in Python.

4.4.12 comment

The `comment` macro ignores its body and always expands to `None`. Unlike linewise comments, the body of the `comment` macro must be grammatically valid Hy, so the compiler can tell where the comment ends. Besides the semicolon linewise comments, Hy also has the `#_` discard prefix syntax to discard the next form. This is completely discarded and doesn't expand to anything, not even `None`.

```
=> (print (comment <h1>Surprise!</h1>
...                <p>You'd be surprised what's grammatically valid in Hy.</p>
...                <p>(Keep delimiters in balance, and you're mostly good to go.)</p>)
...        "Hy")
None Hy
=> (print #_(comment <h1>Surprise!</h1>
...                  <p>You'd be surprised what's grammatically valid in Hy.</p>
...                  <p>(Keep delimiters in balance, and you're mostly good to go.)</
↪p>))
...        "Hy")
Hy
```

4.4.13 cond

`cond` can be used to build nested `if` statements. The following example shows the relationship between the macro and its expansion:

```
(cond [condition-1 result-1]
      [condition-2 result-2])

(if condition-1 result-1
  (if condition-2 result-2))
```

If only the condition is given in a branch, then the condition is also used as the result. The expansion of this single argument version is demonstrated below:

```
(cond [condition-1]
      [condition-2])

(if condition-1 condition-1
  (if condition-2 condition-2))
```

As shown below, only the first matching result block is executed.

```
=> (defn check-value [value]
...   (cond [(< value 5) (print "value is smaller than 5")]
...         [(= value 5) (print "value is equal to 5")]
```

(continues on next page)

(continued from previous page)

```
...         [(> value 5) (print "value is greater than 5")]
...         [True (print "value is something that it should not be")]))

=> (check-value 6)
value is greater than 5
```

4.4.14 continue

`continue` returns execution to the start of a loop. In the following example, `(side-effect1)` is called for each iteration. `(side-effect2)`, however, is only called on every other value in the list.

```
;; assuming that (side-effect1) and (side-effect2) are functions and
;; collection is a list of numerical values

(for [x collection]
  (side-effect1 x)
  (if (% x 2)
    (continue))
  (side-effect2 x))
```

4.4.15 do

`do` (called `progn` in some Lisps) takes any number of forms, evaluates them, and returns the value of the last one, or `None` if no forms were provided.

```
=> (+ 1 (do (setv x (+ 1 1)) x))
3
```

4.4.16 doc / #doc

Documentation macro and tag macro. Gets help for macros or tag macros, respectively.

```
=> (doc doc)
Help on function (doc) in module hy.core.macros:

(doc)(symbol)
    macro documentation

    Gets help for a macro function available in this module.
    Use ``require`` to make other macros available.

    Use ``#doc foo`` instead for help with tag macro ``#foo``.
    Use ``(help foo)`` instead for help with runtime objects.

=> (doc comment)
Help on function (comment) in module hy.core.macros:

(comment)(*body)
    Ignores body and always expands to None

=> #doc doc
```

(continues on next page)

(continued from previous page)

```
Help on function #doc in module hy.core.macros:

#doc(symbol)
    tag macro documentation

Gets help for a tag macro function available in this module.
```

4.4.17 dfor

`dfor` creates a dictionary comprehension. Its syntax is the same as that of *lfor* except that the final value form must be a literal list of two elements, the first of which becomes each key and the second of which becomes each value.

```
=> (dfor x (range 5) [x (* x 10)])
{0: 0, 1: 10, 2: 20, 3: 30, 4: 40}
```

4.4.18 setv

`setv` is used to bind a value, object, or function to a symbol. For example:

```
=> (setv names ["Alice" "Bob" "Charlie"])
=> (print names)
[u'Alice', u'Bob', u'Charlie']

=> (setv counter (fn [collection item] (.count collection item)))
=> (counter [1 2 3 4 5 2 3] 2)
2
```

You can provide more than one target–value pair, and the assignments will be made in order:

```
(setv  x 1  y x  x 2)
(print x y)  ; => 2 1
```

You can perform parallel assignments or unpack the source value with square brackets and *unpack-iterable, unpack-mapping*:

```
(setv duo ["tim" "eric"])
(setv [guy1 guy2] duo)
(print guy1 guy2)  ; => tim eric

(setv [letter1 letter2 #* others] "abcdefg")
(print letter1 letter2 others)   ; => a b ['c', 'd', 'e', 'f', 'g']
```

4.4.19 setx

Whereas `setv` creates an assignment statement, `setx` creates an assignment expression (see **PEP 572**). It requires Python 3.8 or later. Only one target–value pair is allowed, and the target must be a bare symbol, but the `setx` form returns the assigned value instead of `None`.

```
=> (when (> (setx x (+ 1 2)) 0)
...   (print x "is greater than 0"))
3 is greater than 0
```

4.4.20 defclass

New classes are declared with `defclass`. It can take optional parameters in the following order: a list defining (a) possible super class(es) and a string (docstring).

```
(defclass class-name [super-class-1 super-class-2]
  "docstring"

  (setv attribute1 value1)
  (setv attribute2 value2)

  (defn method [self] (print "hello!")))
```

Both values and functions can be bound on the new class as shown by the example below:

```
=> (defclass Cat []
...   (setv age None)
...   (setv colour "white")
...
...   (defn speak [self] (print "Meow")))

=> (setv spot (Cat))
=> (setv spot.colour "Black")
'Black'
=> (.speak spot)
Meow
```

4.4.21 defn

`defn` is used to define functions. It requires two arguments: a name (given as a symbol) and a list of parameters (also given as symbols). Any remaining arguments constitute the body of the function.

```
(defn name [params] bodyform1 bodyform2...)
```

If there at least two body forms, and the first of them is a string literal, this string becomes the docstring of the function.

Parameters may be prefixed with the following special symbols. If you use more than one, they can only appear in the given order (so all *&optional* parameters must precede any *&rest* parameter, *&rest* must precede *&kwonly*, and *&kwonly* must precede *&kwargs*). This is the same order that Python requires.

&optional All following parameters are optional. They may be given as two-argument lists, where the first element is the parameter name and the second is the default value. The parameter can also be given as a single item, in which case the default value is `None`.

The following example defines a function with one required positional argument as well as three optional arguments. The first optional argument defaults to `None` and the latter two default to `"("` and `")"`, respectively.

```
=> (defn format-pair [left-val &optional right-val  [open-text "("] [close-text ")
↪"]]
...   (+ open-text (str left-val) ", " (str right-val) close-text))

=> (format-pair 3)
'(3, None)'

=> (format-pair "A" "B")
'(A, B)'
```

(continues on next page)

(continued from previous page)

```
=> (format-pair "A" "B" "<" ">")
'<A, B>'

=> (format-pair "A" :open-text "<" :close-text ">")
'<A, None>'
```

&rest The following parameter will contain a list of 0 or more positional arguments. No other positional parameters may be specified after this one.

The following code example defines a function that can be given 0 to n numerical parameters. It then sums every odd number and subtracts every even number.

```
=> (defn zig-zag-sum [&rest numbers]
     (setv odd-numbers (lfor x numbers :if (odd? x) x)
           even-numbers (lfor x numbers :if (even? x) x))
     (- (sum odd-numbers) (sum even-numbers)))

=> (zig-zag-sum)
0
=> (zig-zag-sum 3 9 4)
8
=> (zig-zag-sum 1 2 3 4 5 6)
-3
```

&kwonly New in version 0.12.0.

All following parmaeters can only be supplied as keywords. Like `&optional`, the parameter may be marked as optional by declaring it as a two-element list containing the parameter name following by the default value.

```
=> (defn compare [a b &kwonly keyfn [reverse False]]
...   (setv result (keyfn a b))
...   (if (not reverse)
...     result
...     (- result)))
=> (compare "lisp" "python"
...         :keyfn (fn [x y]
...                  (reduce - (map (fn [s] (ord (first s))) [x y]))))
-4
=> (compare "lisp" "python"
...         :keyfn (fn [x y]
...                   (reduce - (map (fn [s] (ord (first s))) [x y])))
...         :reverse True)
4
```

```
=> (compare "lisp" "python")
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: compare() missing 1 required keyword-only argument: 'keyfn'
```

&kwargs Like `&rest`, but for keyword arugments. The following parameter will contain 0 or more keyword arguments.

The following code examples defines a function that will print all keyword arguments and their values.

```
=> (defn print-parameters [&kwargs kwargs]
...    (for [(, k v) (.items kwargs)] (print k v)))
```

(continues on next page)

(continued from previous page)

```
=> (print-parameters :parameter-1 1 :parameter-2 2)
parameter_1 1
parameter_2 2

; to avoid the mangling of '-' to '_', use unpacking:
=> (print-parameters #** {"parameter-1" 1 "parameter-2" 2})
parameter-1 1
parameter-2 2
```

The following example uses all of `&optional`, `&rest`, `&kwonly`, and `&kwargs` in order to show their interactions with each other. The function renders an HTML tag. It requires an argument `tag-name`, a string which is the tag name. It has one optional argument, `delim`, which defaults to `""` and is placed between each child. The rest of the arguments, `children`, are the tag's children or content. A single keyword-only argument, `empty`, is included and defaults to `False`. `empty` changes how the tag is rendered if it has no children. Normally, a tag with no children is rendered like `<div></div>`. If `empty` is `True`, then it will render like `<div />`. The rest of the keyword arguments, `props`, render as HTML attributes.

```
=> (defn render-html-tag [tag-name &optional [delim ""] &rest children &kwonly [empty␣
↪False] &kwargs attrs]
...   (setv rendered-attrs (.join " " (lfor (, key val) (.items attrs) (+ (unmangle␣
↪(str key)) "=\"" (str val) "\""))))
...   (if rendered-attrs  ; If we have attributes, prefix them with a space after the␣
↪tag name
...     (setv rendered-attrs (+ " " rendered-attrs)))
...   (setv rendered-children (.join delim children))
...   (if (and (not children) empty)
...     (+ "<" tag-name rendered-attrs " />")
...     (+ "<" tag-name rendered-attrs ">" rendered-children "</" tag-name ">")))

=> (render-html-tag "div")
'<div></div'>

=> (render-html-tag "img" :empty True)
'<img />'

=> (render-html-tag "img" :id "china" :class "big-image" :empty True)
'<img id="china" class="big-image" />'

=> (render-html-tag "p" " --- " (render-html-tag "span" "" :class "fancy" "I'm fancy!
↪") "I'm to the right of fancy" "I'm alone :(")
'<p><span class="fancy">I\'m fancy!</span> --- I\'m to right right of fancy --- I\'m␣
↪alone :(</p>'
```

4.4.22 defn/a

`defn/a` macro is a variant of `defn` that instead defines coroutines. It takes three parameters: the *name* of the function to define, a vector of *parameters*, and the *body* of the function:

```
(defn/a name [params] body)
```

4.4.23 defmain

New in version 0.10.1.

The `defmain` macro defines a main function that is immediately called with `sys.argv` as arguments if and only if this file is being executed as a script. In other words, this:

```
(defmain [&rest args]
  (do-something-with args))
```

is the equivalent of:

```
def main(*args):
    do_something_with(args)
    return 0

if __name__ == "__main__":
    import sys
    retval = main(*sys.argv)

    if isinstance(retval, int):
        sys.exit(retval)
```

Note that as you can see above, if you return an integer from this function, this will be used as the exit status for your script. (Python defaults to exit status 0 otherwise, which means everything's okay!) Since `(sys.exit 0)` is not run explicitly in the case of a non-integer return from `defmain`, it's a good idea to put `(defmain)` as the last piece of code in your file.

If you want fancy command-line arguments, you can use the standard Python module `argparse` in the usual way:

```
(import argparse)

(defmain [&rest _]
  (setv parser (argparse.ArgumentParser))
  (.add-argument parser "STRING"
    :help "string to replicate")
  (.add-argument parser "-n" :type int :default 3
    :help "number of copies")
  (setv args (parser.parse_args))

  (print (* args.STRING args.n))

  0)
```

4.4.24 defmacro

`defmacro` is used to define macros. The general format is `(defmacro name [parameters] expr)`.

The following example defines a macro that can be used to swap order of elements in code, allowing the user to write code in infix notation, where operator is in between the operands.

```
=> (defmacro infix [code]
...   (quasiquote (
...     (unquote (get code 1))
...     (unquote (get code 0))
...     (unquote (get code 2)))))
```

(continues on next page)

(continued from previous page)

```
=> (infix (1 + 1))
2
```

4.4.25 defmacro/g!

New in version 0.9.12.

`defmacro/g!` is a special version of `defmacro` that is used to automatically generate *gensym* for any symbol that starts with `g!`.

For example, `g!a` would become `(gensym "a")`.

See also:

Section *Using gensym for Safer Macros*

4.4.26 defmacro!

`defmacro!` is like `defmacro/g!` plus automatic once-only evaluation for `o!` parameters, which are available as the equivalent `g!` symbol.

For example,

```
=> (defn expensive-get-number [] (print "spam") 14)
=> (defmacro triple-1 [n] `(+ ~n ~n ~n))
=> (triple-1 (expensive-get-number))  ; evals n three times
spam
spam
spam
42
=> (defmacro/g! triple-2 [n] `(do (setv ~g!n ~n) (+ ~g!n ~g!n ~g!n)))
=> (triple-2 (expensive-get-number))  ; avoid repeats with a gensym
spam
42
=> (defmacro! triple-3 [o!n] `(+ ~g!n ~g!n ~g!n))
=> (triple-3 (expensive-get-number))  ; easier with defmacro!
spam
42
```

4.4.27 deftag

New in version 0.13.0.

`deftag` defines a tag macro. A tag macro is a unary macro that has the same semantics as an ordinary macro defined with `defmacro`. It is called with the syntax `#tag FORM`, where `tag` is the name of the macro, and `FORM` is any form. The `tag` is often only one character, but it can be any symbol.

```
=> (deftag  [expr] `[~expr ~expr])
<function <lambda> at 0x7f76d0271158>
=> # 5
[5, 5]
=> (setv x 0)
=> #(+= x 1)
[None, None]
```

(continues on next page)

(continued from previous page)

```
=> x
2
```

In this example, if you used `(defmacro  ...)` instead of `(deftag  ...)`, you would call the macro as `( 5)` or `( (+= x 1))`.

The syntax for calling tag macros is similar to that of reader macros a la Common Lisp's `SET-MACRO-CHARACTER`. In fact, before Hy 0.13.0, tag macros were called "reader macros", and defined with `defreader` rather than `deftag`. True reader macros are not (yet) implemented in Hy.

4.4.28 del

New in version 0.9.12.

`del` removes an object from the current namespace.

```
=> (setv foo 42)
=> (del foo)
=> foo
Traceback (most recent call last):
  File "<console>", line 1, in <module>
NameError: name 'foo' is not defined
```

`del` can also remove objects from mappings, lists, and more.

```
=> (setv test (list (range 10)))
=> test
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
=> (del (cut test 2 4)) ;; remove items from 2 to 4 excluded
=> test
[0, 1, 4, 5, 6, 7, 8, 9]
=> (setv dic {"foo" "bar"})
=> dic
{"foo": "bar"}
=> (del (get dic "foo"))
=> dic
{}
```

4.4.29 doto

New in version 0.10.1.

`doto` is used to simplify a sequence of method calls to an object.

```
=> (doto [] (.append 1) (.append 2) .reverse)
[2, 1]
```

```
=> (setv collection [])
=> (.append collection 1)
=> (.append collection 2)
=> (.reverse collection)
=> collection
[2, 1]
```

4.4.30 eval-and-compile

`eval-and-compile` is a special form that takes any number of forms. The input forms are evaluated as soon as the `eval-and-compile` form is compiled, instead of being deferred until run-time. The input forms are also left in the program so they can be executed at run-time as usual. So, if you compile and immediately execute a program (as calling `hy foo.hy` does when `foo.hy` doesn't have an up-to-date byte-compiled version), `eval-and-compile` forms will be evaluated twice.

One possible use of `eval-and-compile` is to make a function available both at compile-time (so a macro can call it while expanding) and run-time (so it can be called like any other function):

```
(eval-and-compile
  (defn add [x y]
    (+ x y)))

(defmacro m [x]
  (add x 2))

(print (m 3))     ; prints 5
(print (add 3 6)) ; prints 9
```

Had the `defn` not been wrapped in `eval-and-compile`, `m` wouldn't be able to call `add`, because when the compiler was expanding `(m 3)`, `add` wouldn't exist yet.

4.4.31 eval-when-compile

`eval-when-compile` is like `eval-and-compile`, but the code isn't executed at run-time. Hence, `eval-when-compile` doesn't directly contribute any code to the final program, although it can still change Hy's state while compiling (e.g., by defining a function).

```
(eval-when-compile
  (defn add [x y]
    (+ x y)))

(defmacro m [x]
  (add x 2))

(print (m 3))     ; prints 5
(print (add 3 6)) ; raises NameError: name 'add' is not defined
```

4.4.32 first

`first` is a function for accessing the first element of a collection.

```
=> (first (range 10))
0
```

It is implemented as `(next (iter coll) None)`, so it works with any iterable, and if given an empty iterable, it will return `None` instead of raising an exception.

```
=> (first (repeat 10))
10
=> (first [])
None
```

4.4.33 for

for is used to evaluate some forms for each element in an iterable object, such as a list. The return values of the forms are discarded and the for form returns None.

```
=> (for [x [1 2 3]]
...   (print "iterating")
...   (print x))
iterating
1
iterating
2
iterating
3
```

In its square-bracketed first argument, for allows the same types of clauses as *lfor*.

```
=> (for [x [1 2 3]  :if (!= x 2)  y [7 8]]
...   (print x y))
1 7
1 8
3 7
3 8
```

Furthermore, the last argument of for can be an (else ...) form. This form is executed after the last iteration of the for's outermost iteration clause, but only if that outermost loop terminates normally. If it's jumped out of with e.g. break, the else is ignored.

```
=> (for [element [1 2 3]] (if (< element 3)
...                             (print element)
...                             (break))
...    (else (print "loop finished")))
1
2

=> (for [element [1 2 3]] (if (< element 4)
...                             (print element)
...                             (break))
...    (else (print "loop finished")))
1
2
3
loop finished
```

4.4.34 gensym

New in version 0.9.12.

gensym is used to generate a unique symbol that allows macros to be written without accidental variable name clashes.

```
=> (gensym)
HySymbol('_G\uffff1')

=> (gensym "x")
HySymbol('_x\uffff2')
```

See also:

Section *Using gensym for Safer Macros*

4.4.35 get

`get` is used to access single elements in collections. `get` takes at least two parameters: the *data structure* and the *index* or *key* of the item. It will then return the corresponding value from the collection. If multiple *index* or *key* values are provided, they are used to access successive elements in a nested structure. Example usage:

```
=> (do
...   (setv animals {"dog" "bark" "cat" "meow"}
...         numbers (, "zero" "one" "two" "three")
...         nested [0 1 ["a" "b" "c"] 3 4])
...   (print (get animals "dog"))
...   (print (get numbers 2))
...   (print (get nested 2 1)))

bark
two
b
```

Note: `get` raises a KeyError if a dictionary is queried for a non-existing key.

Note: `get` raises an IndexError if a list or a tuple is queried for an index that is out of bounds.

4.4.36 gfor

`gfor` creates a generator expression. Its syntax is the same as that of *lfor*. The difference is that `gfor` returns an iterator, which evaluates and yields values one at a time.

```
=> (setv accum [])
=> (list (take-while
...   (fn [x] (< x 5))
...   (gfor x (count) :do (.append accum x) x)))
[0, 1, 2, 3, 4]
=> accum
[0, 1, 2, 3, 4, 5]
```

4.4.37 global

`global` can be used to mark a symbol as global. This allows the programmer to assign a value to a global symbol. Reading a global symbol does not require the `global` keyword – only assigning it does.

The following example shows how the global symbol `a` is assigned a value in a function and is later on printed in another function. Without the `global` keyword, the second function would have raised a `NameError`.

```
(defn set-a [value]
  (global a)
  (setv a value))
```

(continues on next page)

(continued from previous page)

```
(defn print-a []
  (print a))

(set-a 5)
(print-a)
```

4.4.38 if / if* / if-not

New in version 0.10.0: if-not

`if / if* / if-not` respect Python *truthiness*, that is, a *test* fails if it evaluates to a "zero" (including values of `len` zero, `None`, and `False`), and passes otherwise, but values with a `__bool__` method can override this.

The `if` macro is for conditionally selecting an expression for evaluation. The result of the selected expression becomes the result of the entire `if` form. `if` can select a group of expressions with the help of a `do` block.

`if` takes any number of alternating *test* and *then* expressions, plus an optional *else* expression at the end, which defaults to `None`. `if` checks each *test* in turn, and selects the *then* corresponding to the first passed test. `if` does not evaluate any expressions following its selection, similar to the `if/elif/else` control structure from Python. If no tests pass, `if` selects *else*.

The `if*` special form is restricted to 2 or 3 arguments, but otherwise works exactly like `if` (which expands to nested `if*` forms), so there is generally no reason to use it directly.

`if-not` is similar to `if*` but the second expression will be executed when the condition fails while the third and final expression is executed when the test succeeds – the opposite order of `if*`. The final expression is again optional and defaults to `None`.

Example usage:

```
(print (if (< n 0.0) "negative"
           (= n 0.0) "zero"
           (> n 0.0) "positive"
           "not a number"))

(if* (money-left? account)
  (print "let's go shopping")
  (print "let's go and work"))

(if-not (money-left? account)
  (print "let's go and work")
  (print "let's go shopping"))
```

4.4.39 lif and lif-not

New in version 0.10.0.

New in version 0.11.0: lif-not

For those that prefer a more Lispy `if` clause, we have `lif`. This *only* considers `None` to be false! All other "false-ish" Python values are considered true. Conversely, we have `lif-not` in parallel to `if` and `if-not` which reverses the comparison.

```
=> (lif True "true" "false")
"true"
=> (lif False "true" "false")
"true"
=> (lif 0 "true" "false")
"true"
=> (lif None "true" "false")
"false"
=> (lif-not None "true" "false")
"true"
=> (lif-not False "true" "false")
"false"
```

4.4.40 import

`import` is used to import modules, like in Python. There are several ways that `import` can be used.

```
;; Imports each of these modules
;;
;; Python:
;; import sys
;; import os.path
(import sys os.path)

;; Import from a module
;;
;; Python: from os.path import exists, isdir, isfile
(import [os.path [exists isdir isfile]])

;; Import with an alias
;;
;; Python: import sys as systest
(import [sys :as systest])

;; You can list as many imports as you like of different types.
;;
;; Python:
;; from tests.resources import kwtest, function_with_a_dash
;; from os.path import exists, isdir as is_dir, isfile as is_file
;; import sys as systest
(import [tests.resources [kwtest function-with-a-dash]]
        [os.path [exists
                  isdir :as dir?
                  isfile :as file?]]
        [sys :as systest])

;; Import all module functions into current namespace
;;
;; Python: from sys import *
(import [sys [*]])
```

4.4.41 fn

`fn`, like Python's `lambda`, can be used to define an anonymous function. Unlike Python's `lambda`, the body of the function can comprise several statements. The parameters are similar to `defn`: the first parameter is vector of parameters and the rest is the body of the function. `fn` returns a new function. In the following example, an anonymous function is defined and passed to another function for filtering output.

```
=> (setv people [{:name "Alice" :age 20}
...              {:name "Bob" :age 25}
...              {:name "Charlie" :age 50}
...              {:name "Dave" :age 5}])

=> (defn display-people [people filter]
...   (for [person people] (if (filter person) (print (:name person)))))

=> (display-people people (fn [person] (< (:age person) 25)))
Alice
Dave
```

Just as in normal function definitions, if the first element of the body is a string, it serves as a docstring. This is useful for giving class methods docstrings.

```
=> (setv times-three
...    (fn [x]
...     "Multiplies input by three and returns the result."
...     (* x 3)))
```

This can be confirmed via Python's built-in `help` function:

```
=> (help times-three)
Help on function times_three:

times_three(x)
Multiplies input by three and returns result
(END)
```

4.4.42 fn/a

`fn/a` is a variant of `fn` than defines an anonymous coroutine. The parameters are similar to `defn/a`: the first parameter is vector of parameters and the rest is the body of the function. `fn/a` returns a new coroutine.

4.4.43 last

New in version 0.11.0.

`last` can be used for accessing the last element of a collection:

```
=> (last [2 4 6])
6
```

4.4.44 lfor

The comprehension forms `lfor`, *sfor*, *dfor*, *gfor*, and *for* are used to produce various kinds of loops, including Python-style comprehensions. `lfor` in particular creates a list comprehension. A simple use of `lfor` is:

```
=> (lfor x (range 5) (* 2 x))
[0, 2, 4, 6, 8]
```

x is the name of a new variable, which is bound to each element of (range 5). Each such element in turn is used to evaluate the value form (* 2 x), and the results are accumulated into a list.

Here's a more complex example:

```
=> (lfor
...   x (range 3)
...   y (range 3)
...   :if (!= x y)
...   :setv total (+ x y)
...   [x y total])
[[0, 1, 1], [0, 2, 2], [1, 0, 1], [1, 2, 3], [2, 0, 2], [2, 1, 3]]
```

When there are several iteration clauses (here, the pairs of forms x (range 3) and y (range 3)), the result works like a nested loop or Cartesian product: all combinations are considered in lexicographic order.

The general form of lfor is:

```
(lfor CLAUSES VALUE)
```

where the VALUE is an arbitrary form that is evaluated to produce each element of the result list, and CLAUSES is any number of clauses. There are several types of clauses:

- Iteration clauses, which look like LVALUE ITERABLE. The LVALUE is usually just a symbol, but could be something more complicated, like [x y].
- :async LVALUE ITERABLE, which is an asynchronous form of iteration clause.
- :do FORM, which simply evaluates the FORM. If you use (continue) or (break) here, they will apply to the innermost iteration clause before the :do.
- :setv LVALUE RVALUE, which is equivalent to :do (setv LVALUE RVALUE).
- :if CONDITION, which is equivalent to :do (unless CONDITION (continue)).

For lfor, sfor, gfor, and dfor, variables are scoped as if the comprehension form were its own function, so variables defined by an iteration clause or :setv are not visible outside the form. In fact, these forms are implemented as generator functions whenever they contain Python statements, with the attendant consequences for calling return. By contrast, for shares the caller's scope.

4.4.45 nonlocal

New in version 0.11.1.

nonlocal can be used to mark a symbol as not local to the current scope. The parameters are the names of symbols to mark as nonlocal. This is necessary to modify variables through nested fn scopes:

```
(defn some-function []
  (setv x 0)
  (register-some-callback
    (fn [stuff]
      (nonlocal x)
      (setv x stuff))))
```

Without the call to (nonlocal x), the inner function would redefine x to stuff inside its local scope instead of overwriting the x in the outer function.

See PEP3104 for further information.

4.4.46 not

`not` is used in logical expressions. It takes a single parameter and returns a reversed truth value. If `True` is given as a parameter, `False` will be returned, and vice-versa. Example usage:

```
=> (not True)
False

=> (not False)
True

=> (not None)
True
```

4.4.47 or

`or` is used in logical expressions. It takes at least two parameters. It will return the first non-false parameter. If no such value exists, the last parameter will be returned.

```
=> (or True False)
True

=> (and False False)
False

=> (and False 1 True False)
1
```

Note: `or` short-circuits and stops evaluating parameters as soon as the first true value is encountered.

```
=> (or True (print "hello"))
True
```

4.4.48 of

`of` is an alias for get, but with special semantics designed for handling PEP 484's generic types.

`of` has three forms:

- `(of T)` will simply become `T`.
- `(of T x)` will become `(get T x)`.
- `(of T x y ...)` (where the `...` represents zero or more arguments) will become `(get T (, x y ...))`.

For instance:

```
(of str)  ; => str

(of List int)  ; => List[int]
(of Set int)   ; => Set[int]

(of Dict str str)   ; => Dict[str, str]
```

(continues on next page)

(continued from previous page)

```
(of Tuple str int)  ; => Tuple[str, int]

(of Callable [int str] str)  ; => Callable[[int, str], str]
```

4.4.49 py

`py` parses the given Python code at compile-time and inserts the result into the generated abstract syntax tree. Thus, you can mix Python code into a Hy program. Only a Python expression is allowed, not statements; use *pys* if you want to use Python statements. The value of the expression is returned from the `py` form.

```
(print "A result from Python:" (py "'hello' + 'world'"))
```

The code must be given as a single string literal, but you can still use macros, *eval*, and related tools to construct the `py` form. If having to backslash-escape internal double quotes is getting you down, try a *bracket string*. If you want to evaluate some Python code that's only defined at run-time, try the standard Python function `eval()`.

Python code need not syntactically round-trip if you use `hy2py` on a Hy program that uses `py` or `pys`. For example, comments will be removed.

4.4.50 pys

As *py*, but the code can consist of zero or more statements, including compound statements such as `for` and `def`. `pys` always returns `None`. Also, the code string is dedented with `textwrap.dedent()` before parsing, which allows you to intend the code to match the surrounding Hy code, but significant leading whitespace in embedded string literals will be removed.

```
(pys "myvar = 5")
(print "myvar is" myvar)
```

4.4.51 quasiquote

`quasiquote` allows you to quote a form, but also selectively evaluate expressions. Expressions inside a `quasiquote` can be selectively evaluated using `unquote` (`~`). The evaluated form can also be spliced using `unquote-splice` (`~@`). Quasiquote can be also written using the backquote (`` ` ``) symbol.

```
;; let `qux' be a variable with value (bar baz)
`(foo ~qux)
; equivalent to '(foo (bar baz))
`(foo ~@qux)
; equivalent to '(foo bar baz)
```

4.4.52 quote

`quote` returns the form passed to it without evaluating it. `quote` can alternatively be written using the apostrophe (`'`) symbol.

```
=> (setv x '(print "Hello World"))
=> x  ; variable x is set to unevaluated expression
HyExpression([
```

(continues on next page)

(continued from previous page)

```
  HySymbol('print'),
  HyString('Hello World')])
=> (eval x)
Hello World
```

4.4.53 require

`require` is used to import macros from one or more given modules. It allows parameters in all the same formats as `import`. The `require` form itself produces no code in the final program: its effect is purely at compile-time, for the benefit of macro expansion. Specifically, `require` imports each named module and then makes each requested macro available in the current module.

The following are all equivalent ways to call a macro named `foo` in the module `mymodule`:

```
(require mymodule)
(mymodule.foo 1)

(require [mymodule :as M])
(M.foo 1)

(require [mymodule [foo]])
(foo 1)

(require [mymodule [*]])
(foo 1)

(require [mymodule [foo :as bar]])
(bar 1)
```

Macros that call macros

One aspect of `require` that may be surprising is what happens when one macro's expansion calls another macro. Suppose `mymodule.hy` looks like this:

```
(defmacro repexpr [n expr]
  ; Evaluate the expression n times
  ; and collect the results in a list.
  `(list (map (fn [_] ~expr) (range ~n))))

(defmacro foo [n]
  `(repexpr ~n (input "Gimme some input: ")))
```

And then, in your main program, you write:

```
(require [mymodule [foo]])

(print (mymodule.foo 3))
```

Running this raises `NameError: name 'repexpr' is not defined`, even though writing `(print (foo 3))` in `mymodule` works fine. The trouble is that your main program doesn't have the macro `repexpr` available, since it wasn't imported (and imported under exactly that name, as opposed to a qualified name). You could do `(require [mymodule [*]])` or `(require [mymodule [foo repexpr]])`, but a less error-prone approach is to change the definition of `foo` to require whatever sub-macros it needs:

```
(defmacro foo [n]
  `(do
    (require mymodule)
    (mymodule.repexpr ~n (input "Gimme some input: "))))
```

It's wise to use `(require mymodule)` here rather than `(require [mymodule [repexpr]])` to avoid accidentally shadowing a function named `repexpr` in the main program.

Qualified macro names

Note that in the current implementation, there's a trick in qualified macro names, like `mymodule.foo` and `M.foo` in the above example. These names aren't actually attributes of module objects; they're just identifiers with periods in them. In fact, `mymodule` and `M` aren't defined by these `require` forms, even at compile-time. None of this will hurt you unless try to do introspection of the current module's set of defined macros, which isn't really supported anyway.

4.4.54 rest

`rest` takes the given collection and returns an iterable of all but the first element.

```
=> (list (rest (range 10)))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Given an empty collection, it returns an empty iterable.

```
=> (list (rest []))
[]
```

4.4.55 return

`return` compiles to a `return` statement. It exits the current function, returning its argument if provided with one or `None` if not.

```
=> (defn f [x] (for [n (range 10)] (when (> n x) (return n))))
=> (f 3.9)
4
```

Note that in Hy, `return` is necessary much less often than in Python, since the last form of a function is returned automatically. Hence, an explicit `return` is only necessary to exit a function early.

```
=> (defn f [x] (setv y 10) (+ x y))
=> (f 4)
14
```

To get Python's behavior of returning `None` when execution reaches the end of a function, put `None` there yourself.

```
=> (defn f [x] (setv y 10) (+ x y) None)
=> (print (f 4))
None
```

4.4.56 sfor

`sfor` creates a set comprehension. `(sfor CLAUSES VALUE)` is equivalent to `(set (lfor CLAUSES VALUE))`. See *lfor*.

4.4.57 cut

`cut` can be used to take a subset of a list and create a new list from it. The form takes at least one parameter specifying the list to cut. Two optional parameters can be used to give the start and end position of the subset. If they are not supplied, the default value of `None` will be used instead. The third optional parameter is used to control step between the elements.

`cut` follows the same rules as its Python counterpart. Negative indices are counted starting from the end of the list. Some example usage:

```
=> (setv collection (range 10))

=> (cut collection)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

=> (cut collection 5)
[5, 6, 7, 8, 9]

=> (cut collection 2 8)
[2, 3, 4, 5, 6, 7]

=> (cut collection 2 8 2)
[2, 4, 6]

=> (cut collection -4 -2)
[6, 7]
```

4.4.58 raise

The `raise` form can be used to raise an `Exception` at runtime. Example usage:

```
(raise)
; re-rase the last exception

(raise IOError)
; raise an IOError

(raise (IOError "foobar"))
; raise an IOError("foobar")
```

`raise` can accept a single argument (an `Exception` class or instance) or no arguments to re-raise the last `Exception`.

4.4.59 try

The `try` form is used to catch exceptions (`except`) and run cleanup actions (`finally`).

```
(try
  (error-prone-function)
  (another-error-prone-function)
  (except [ZeroDivisionError]
    (print "Division by zero"))
  (except [[IndexError KeyboardInterrupt]]
    (print "Index error or Ctrl-C"))
  (except [e ValueError]
    (print "ValueError:" (repr e)))
  (except [e [TabError PermissionError ReferenceError]]
    (print "Some sort of error:" (repr e)))
  (else
    (print "No errors"))
  (finally
    (print "All done")))
```

The first argument of `try` is its body, which can contain one or more forms. Then comes any number of `except` clauses, then optionally an `else` clause, then optionally a `finally` clause. If an exception is raised with a matching `except` clause during the execution of the body, that `except` clause will be executed. If no exceptions are raised, the `else` clause is executed. The `finally` clause will be executed last regardless of whether an exception was raised.

The return value of `try` is the last form of the `except` clause that was run, or the last form of `else` if no exception was raised, or the `try` body if there is no `else` clause.

4.4.60 unless

The `unless` macro is a shorthand for writing an `if` statement that checks if the given conditional is `False`. The following shows the expansion of this macro.

```
(unless conditional statement)

(if conditional
  None
  (do statement))
```

4.4.61 unpack-iterable, unpack-mapping

(Also known as the splat operator, star operator, argument expansion, argument explosion, argument gathering, and varargs, among others. . .)

`unpack-iterable` and `unpack-mapping` allow an iterable or mapping object (respectively) to provide positional or keywords arguments (respectively) to a function.

```
=> (defn f [a b c d] [a b c d])
=> (f (unpack-iterable [1 2]) (unpack-mapping {"c" 3 "d" 4}))
[1, 2, 3, 4]
```

`unpack-iterable` is usually written with the shorthand `#*`, and `unpack-mapping` with `#**`.

```
=> (f #* [1 2] #** {"c" 3 "d" 4})
[1, 2, 3, 4]
```

Unpacking is allowed in a variety of contexts, and you can unpack more than once in one expression (**PEP 3132**, **PEP 448**).

```
=> (setv [a #* b c] [1 2 3 4 5])
=> [a b c]
[1, [2, 3, 4], 5]
=> [#* [1 2] #* [3 4]]
[1, 2, 3, 4]
=> {#** {1 2} #** {3 4}}
{1: 2, 3: 4}
=> (f #* [1] #* [2] #** {"c" 3} #** {"d" 4})
[1, 2, 3, 4]
```

4.4.62 unquote

Within a quasiquoted form, `unquote` forces evaluation of a symbol. `unquote` is aliased to the tilde (~) symbol.

```
=> (setv nickname "Cuddles")
=> (quasiquote (= nickname (unquote nickname)))
HyExpression([
  HySymbol('='),
  HySymbol('nickname'),
  'Cuddles'])
=> `(= nickname ~nickname)
HyExpression([
  HySymbol('='),
  HySymbol('nickname'),
  'Cuddles'])
```

4.4.63 unquote-splice

`unquote-splice` forces the evaluation of a symbol within a quasiquoted form, much like `unquote`. `unquote-splice` can be used when the symbol being unquoted contains an iterable value, as it "splices" that iterable into the quasiquoted form. `unquote-splice` can also be used when the value evaluates to a false value such as `None`, `False`, or `0`, in which case the value is treated as an empty list and thus does not splice anything into the form. `unquote-splice` is aliased to the `~@` syntax.

```
=> (setv nums [1 2 3 4])
=> (quasiquote (+ (unquote-splice nums)))
HyExpression([
  HySymbol('+'),
  1,
  2,
  3,
  4])
=> `(+ ~@nums)
HyExpression([
  HySymbol('+'),
  1,
  2,
  3,
  4])
=> `[1 2 ~@(if (neg? (first nums)) nums)]
HyList([
  HyInteger(1),
  HyInteger(2)])
```

Here, the last example evaluates to `('+' 1 2)`, since the condition `(< (nth nums 0) 0)` is `False`, which makes this `if` expression evaluate to `None`, because the `if` expression here does not have an else clause. `unquote-splice` then evaluates this as an empty value, leaving no effects on the list it is enclosed in, therefore resulting in `('+' 1 2)`.

4.4.64 when

`when` is similar to `unless`, except it tests when the given conditional is `True`. It is not possible to have an `else` block in a `when` macro. The following shows the expansion of the macro.

```
(when conditional statement)

(if conditional (do statement))
```

4.4.65 while

`while` compiles to a *`while`* statement. It is used to execute a set of forms as long as a condition is met. The first argument to `while` is the condition, and any remaining forms constitute the body. The following example will output "Hello world!" to the screen indefinitely:

```
(while True (print "Hello world!"))
```

The last form of a `while` loop can be an `else` clause, which is executed after the loop terminates, unless it exited abnormally (e.g., with `break`). So,

```
(setv x 2)
(while x
   (print "In body")
   (-= x 1)
   (else
     (print "In else")))
```

prints

```
In body
In body
In else
```

If you put a `break` or `continue` form in the condition of a `while` loop, it will apply to the very same loop rather than an outer loop, even if execution is yet to ever reach the loop body. (Hy compiles a `while` loop with statements in its condition by rewriting it so that the condition is actually in the body.) So,

```
(for [x [1]]
   (print "In outer loop")
   (while
     (do
       (print "In condition")
       (break)
       (print "This won't print.")
       True)
     (print "This won't print, either."))
   (print "At end of outer loop"))
```

prints

```
In outer loop
In condition
At end of outer loop
```

4.4.66 with

`with` is used to wrap the execution of a block within a context manager. The context manager can then set up the local system and tear it down in a controlled manner. The archetypical example of using `with` is when processing files. `with` can bind context to an argument or ignore it completely, as shown below:

```
(with [arg (expr)] block)

(with [(expr)] block)

(with [arg (expr) (expr)] block)
```

The following example will open the `NEWS` file and print its content to the screen. The file is automatically closed after it has been processed.

```
(with [f (open "NEWS")] (print (.read f)))
```

`with` returns the value of its last form, unless it suppresses an exception (because the context manager's `__exit__` method returned true), in which case it returns `None`. So, the previous example could also be written

```
(print (with [f (open "NEWS")] (.read f)))
```

4.4.67 with/a

`with/a` behaves like `with`, but is used to wrap the execution of a block within an asynchronous context manager. The context manager can then set up the local system and tear it down in a controlled manner asynchronously.

```
(with/a [arg (expr)] block)

(with/a [(expr)] block)

(with/a [arg (expr) (expr)] block)
```

`with/a` returns the value of its last form, unless it suppresses an exception (because the context manager's `__aexit__` method returned true), in which case it returns `None`.

4.4.68 with-decorator

`with-decorator` is used to wrap a function with another. The function performing the decoration should accept a single value: the function being decorated, and return a new function. `with-decorator` takes a minimum of two parameters: the function performing decoration and the function being decorated. More than one decorator function can be applied; they will be applied in order from outermost to innermost, ie. the first decorator will be the outermost one, and so on. Decorators with arguments are called just like a function call.

```
(with-decorator decorator-fun
   (defn some-function [] ...)
```

(continues on next page)

(continued from previous page)

```
(with-decorator decorator1 decorator2 ...
   (defn some-function [] ...)

(with-decorator (decorator arg) ..
   (defn some-function [] ...)
```

In the following example, `inc-decorator` is used to decorate the function `addition` with a function that takes two parameters and calls the decorated function with values that are incremented by 1. When the decorated `addition` is called with values 1 and 1, the end result will be 4 (`1+1 + 1+1`).

```
=> (defn inc-decorator [func]
...   (fn [value-1 value-2] (func (+ value-1 1) (+ value-2 1))))
=> (defn inc2-decorator [func]
...   (fn [value-1 value-2] (func (+ value-1 2) (+ value-2 2))))

=> (with-decorator inc-decorator (defn addition [a b] (+ a b)))
=> (addition 1 1)
4
=> (with-decorator inc2-decorator inc-decorator
...   (defn addition [a b] (+ a b)))
=> (addition 1 1)
8
```

#@

New in version 0.12.0.

The tag macro `#@` can be used as a shorthand for `with-decorator`. With `#@`, the previous example becomes:

```
=> #@(inc-decorator (defn addition [a b] (+ a b)))
=> (addition 1 1)
4
=> #@(inc2-decorator inc-decorator
...    (defn addition [a b] (+ a b)))
=> (addition 1 1)
8
```

4.4.69 with-gensyms

New in version 0.9.12.

`with-gensym` is used to generate a set of *gensym* for use in a macro. The following code:

```
(with-gensyms [a b c]
  ...)
```

expands to:

```
(do
  (setv a (gensym)
        b (gensym)
        c (gensym))
  ...)
```

See also:

Section *Using gensym for Safer Macros*

4.4.70 xor

New in version 0.12.0.

`xor` performs the logical operation of exclusive OR. It takes two arguments. If exactly one argument is true, that argument is returned. If neither is true, the second argument is returned (which will necessarily be false). Otherwise, when both arguments are true, the value `False` is returned.

```
=> [(xor 0 0) (xor 0 1) (xor 1 0) (xor 1 1)]
[0, 1, 1, False]
```

4.4.71 yield

`yield` is used to create a generator object that returns one or more values. The generator is iterable and therefore can be used in loops, list comprehensions and other similar constructs.

The function `random-numbers` shows how generators can be used to generate infinite series without consuming infinite amount of memory.

```
=> (defn multiply [bases coefficients]
...   (for [(, base coefficient) (zip bases coefficients)]
...    (yield (* base coefficient))))

=> (multiply (range 5) (range 5))
<generator object multiply at 0x978d8ec>

=> (list (multiply (range 10) (range 10)))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

=> (import random)
=> (defn random-numbers [low high]
...   (while True (yield (.randint random low high))))
=> (list (take 15 (random-numbers 1 50)))
[7, 41, 6, 22, 32, 17, 5, 38, 18, 38, 17, 14, 23, 23, 19]
```

4.4.72 yield-from

New in version 0.9.13.

`yield-from` is used to call a subgenerator. This is useful if you want your coroutine to be able to delegate its processes to another coroutine, say, if using something fancy like asyncio.

4.5 Hy Core

4.5.1 Core Functions

butlast

Usage: `(butlast coll)`

Returns an iterator of all but the last item in *coll*.

```
=> (list (butlast (range 10)))
[0, 1, 2, 3, 4, 5, 6, 7, 8]

=> (list (butlast [1]))
[]

=> (list (butlast []))
[]

=> (list (take 5 (butlast (count 10))))
[10, 11, 12, 13, 14]
```

coll?

New in version 0.10.0.

Usage: `(coll? x)`

Returns `True` if *x* is iterable and not a string.

```
=> (coll? [1 2 3 4])
True

=> (coll? {"a" 1 "b" 2})
True

=> (coll? "abc")
False
```

comp

Usage: `(comp f g)`

Compose zero or more functions into a new function. The new function will chain the given functions together, so `((comp g f) x)` is equivalent to `(g (f x))`. Called without arguments, `comp` returns `identity`.

```
=> (setv example (comp str +))
=> (example 1 2 3)
"6"

=> (setv simple (comp))
=> (simple "hello")
"hello"
```

complement

New in version 0.12.0.

Usage: `(complement f)`

Returns a new function that returns the same thing as `f`, but logically inverted. So, `((complement f) x)` is equivalent to `(not (f x))`.

```
=> (setv inverse (complement identity))
=> (inverse True)
False
=> (inverse 1)
False
=> (inverse False)
True
```

constantly

New in version 0.12.0.

Usage `(constantly 42)`

Create a new function that always returns the given value, regardless of the arguments given to it.

```
=> (setv answer (constantly 42))
=> (answer)
42
=> (answer 1 2 3)
42
=> (answer 1 :foo 2)
42
```

dec

Usage: `(dec x)`

Returns one less than *x*. Equivalent to `(- x 1)`. Raises `TypeError` if `(not (numeric? x))`.

```
=> (dec 3)
2

=> (dec 0)
-1

=> (dec 12.3)
11.3
```

disassemble

New in version 0.10.0.

Usage: `(disassemble tree &optional [codegen false])`

Dump the Python AST for given Hy *tree* to standard output. If *codegen* is `True`, the function prints Python code instead.

```
=> (disassemble '(print "Hello World!"))
Module(
 body=[
     Expr(value=Call(func=Name(id='print'), args=[Str(s='Hello World!')], keywords=[],
↪ starargs=None, kwargs=None))])
```

(continues on next page)

(continued from previous page)

```
=> (disassemble '(print "Hello World!") True)
print('Hello World!')
```

empty?

Usage: `(empty? coll)`

Returns `True` if *coll* is empty. Equivalent to `(= 0 (len coll))`.

```
=> (empty? [])
True

=> (empty? "")
True

=> (empty? (, 1 2))
False
```

eval

`eval` evaluates a quoted expression and returns the value. The optional second and third arguments specify the dictionary of globals to use and the module name. The globals dictionary defaults to `(local)` and the module name defaults to the name of the current module. An optional fourth keyword parameter, `compiler`, allows one to re-use an existing `HyASTCompiler` object for the compilation step.

```
=> (eval '(print "Hello World"))
"Hello World"
```

If you want to evaluate a string, use `read-str` to convert it to a form first:

```
=> (eval (read-str "(+ 1 1)"))
2
```

every?

New in version 0.10.0.

Usage: `(every? pred coll)`

Returns `True` if `(pred x)` is logical true for every *x* in *coll*, otherwise `False`. Return `True` if *coll* is empty.

```
=> (every? even? [2 4 6])
True

=> (every? even? [1 3 5])
False

=> (every? even? [2 4 5])
False

=> (every? even? [])
True
```

float?

Usage: `(float? x)`

Returns `True` if *x* is a float.

```
=> (float? 3.2)
True

=> (float? -2)
False
```

fraction

Returns a Python object of type `fractions.Fraction`.

```
=> (fraction 1 2)
Fraction(1, 2)
```

Note that Hy has a built-in fraction literal that does the same thing:

```
=> 1/2
Fraction(1, 2)
```

even?

Usage: `(even? x)`

Returns `True` if *x* is even. Raises `TypeError` if `(not (numeric? x))`.

```
=> (even? 2)
True

=> (even? 13)
False

=> (even? 0)
True
```

identity

Usage: `(identity x)`

Returns the argument supplied to the function.

```
=> (identity 4)
4

=> (list (map identity [1 2 3 4]))
[1 2 3 4]
```

inc

Usage: `(inc x)`

Returns one more than *x*. Equivalent to `(+ x 1)`. Raises `TypeError` if `(not (numeric? x))`.

```
=> (inc 3)
4

=> (inc 0)
1

=> (inc 12.3)
13.3
```

instance?

Usage: `(instance? class x)`

Returns `True` if *x* is an instance of *class*.

```
=> (instance? float 1.0)
True

=> (instance? int 7)
True

=> (instance? str (str "foo"))
True

=> (defclass TestClass [object])
=> (setv inst (TestClass))
=> (instance? TestClass inst)
True
```

integer?

Usage: `(integer? x)`

Returns *True* if *x* is an integer (`int`).

```
=> (integer? 3)
True

=> (integer? -2.4)
False
```

interleave

New in version 0.10.1.

Usage: `(interleave seq1 seq2 ...)`

Returns an iterable of the first item in each of the sequences, then the second, etc.

```
=> (list (interleave (range 5) (range 100 105)))
[0, 100, 1, 101, 2, 102, 3, 103, 4, 104]

=> (list (interleave (range 1000000) "abc"))
[0, 'a', 1, 'b', 2, 'c']
```

interpose

New in version 0.10.1.

Usage: `(interpose item seq)`

Returns an iterable of the elements of the sequence separated by the item.

```
=> (list (interpose "!" "abcd"))
['a', '!', 'b', '!', 'c', '!', 'd']

=> (list (interpose -1 (range 5)))
[0, -1, 1, -1, 2, -1, 3, -1, 4]
```

iterable?

Usage: `(iterable? x)`

Returns `True` if *x* is iterable. Iterable objects return a new iterator when `(iter x)` is called. Contrast with *iterator?*.

```
=> ;; works for strings
=> (iterable? (str "abcde"))
True

=> ;; works for lists
=> (iterable? [1 2 3 4 5])
True

=> ;; works for tuples
=> (iterable? (, 1 2 3))
True

=> ;; works for dicts
=> (iterable? {:a 1 :b 2 :c 3})
True

=> ;; works for iterators/generators
=> (iterable? (repeat 3))
True
```

iterator?

Usage: `(iterator? x)`

Returns `True` if *x* is an iterator. Iterators are objects that return themselves as an iterator when `(iter x)` is called. Contrast with *iterable?*.

```
=> ;; doesn't work for a list
=> (iterator? [1 2 3 4 5])
False

=> ;; but we can get an iter from the list
=> (iterator? (iter [1 2 3 4 5]))
True

=> ;; doesn't work for dict
=> (iterator? {:a 1 :b 2 :c 3})
False

=> ;; create an iterator from the dict
=> (iterator? (iter {:a 1 :b 2 :c 3}))
True
```

juxt

New in version 0.12.0.

Usage: `(juxt f &rest fs)`

Return a function that applies each of the supplied functions to a single set of arguments and collects the results into a list.

```
=> ((juxt min max sum) (range 1 101))
[1, 100, 5050]

=> (dict (map (juxt identity ord) "abcdef"))
{'f': 102, 'd': 100, 'b': 98, 'e': 101, 'c': 99, 'a': 97}

=> ((juxt + - * /) 24 3)
[27, 21, 72, 8.0]
```

keyword

New in version 0.10.1.

Usage: `(keyword "foo")`

Create a keyword from the given value. Strings, numbers, and even objects with the __*name*__ magic will work.

```
=> (keyword "foo")
HyKeyword('foo')

=> (keyword 1)
HyKeyword('foo')
```

keyword?

New in version 0.10.1.

Usage: `(keyword? foo)`

Check whether *foo* is a *keyword*.

```
=> (keyword? :foo)
True

=> (setv foo 1)
=> (keyword? foo)
False
```

macroexpand

New in version 0.10.0.

Usage: `(macroexpand form)`

Returns the full macro expansion of *form*.

```
=> (macroexpand '(-> (a b) (x y)))
HyExpression([
  HySymbol('x'),
  HyExpression([
    HySymbol('a'),
    HySymbol('b')]),
  HySymbol('y')])
=> (macroexpand '(-> (a b) (-> (c d) (e f))))
HyExpression([
  HySymbol('e'),
  HyExpression([
    HySymbol('c'),
    HyExpression([
      HySymbol('a'),
      HySymbol('b')]),
    HySymbol('d')]),
  HySymbol('f')])
```

macroexpand-1

New in version 0.10.0.

Usage: `(macroexpand-1 form)`

Returns the single step macro expansion of *form*.

```
=> (macroexpand-1 '(-> (a b) (-> (c d) (e f))))
HyExpression([
  HySymbol('_>'),
  HyExpression([
    HySymbol('a'),
    HySymbol('b')]),
  HyExpression([
    HySymbol('c'),
    HySymbol('d')]),
  HyExpression([
    HySymbol('e'),
    HySymbol('f')])])
```

mangle

Usage: `(mangle x)`

Stringify the input and translate it according to *Hy's mangling rules*.

```
=> (mangle "foo-bar")
'foo_bar'
```

merge-with

New in version 0.10.1.

Usage: `(merge-with f &rest maps)`

Returns a map that consist of the rest of the maps joined onto first. If a key occurs in more than one map, the mapping(s) from the latter (left-to-right) will be combined with the mapping in the result by calling `(f val-in-result val-in-latter)`.

```
=> (merge-with + {"a" 10 "b" 20} {"a" 1 "c" 30})
{u'a': 11L, u'c': 30L, u'b': 20L}
```

name

New in version 0.10.1.

Usage: `(name :keyword)`

Convert the given value to a string. Keyword special character will be stripped. Strings will be used as is. Even objects with the *__name__* magic will work.

```
=> (name :foo)
u'foo'
```

neg?

Usage: `(neg? x)`

Returns `True` if *x* is less than zero. Raises `TypeError` if `(not (numeric? x))`.

```
=> (neg? -2)
True

=> (neg? 3)
False

=> (neg? 0)
False
```

none?

Usage: `(none? x)`

Returns `True` if *x* is `None`.

```
=> (none? None)
True

=> (none? 0)
False

=> (setv x None)
=> (none? x)
True

=> ;; list.append always returns None
=> (none? (.append [1 2 3] 4))
True
```

nth

Usage: `(nth coll n &optional [default None])`

Returns the *n*-th item in a collection, counting from 0. Return the default value, `None`, if out of bounds (unless specified otherwise). Raises `ValueError` if *n* is negative.

```
=> (nth [1 2 4 7] 1)
2

=> (nth [1 2 4 7] 3)
7

=> (none? (nth [1 2 4 7] 5))
True

=> (nth [1 2 4 7] 5 "default")
'default'

=> (nth (take 3 (drop 2 [1 2 3 4 5 6])) 2))
5

=> (nth [1 2 4 7] -1)
Traceback (most recent call last):
  ...
ValueError: Indices for islice() must be None or an integer: 0 <= x <= sys.maxsize.
```

numeric?

Usage: `(numeric? x)`

Returns `True` if *x* is a numeric, as defined in Python's `numbers.Number` class.

```
=> (numeric? -2)
True

=> (numeric? 3.2)
True

=> (numeric? "foo")
False
```

odd?

Usage: `(odd? x)`

Returns `True` if *x* is odd. Raises `TypeError` if `(not (numeric? x))`.

```
=> (odd? 13)
True

=> (odd? 2)
False

=> (odd? 0)
False
```

parse-args

Usage: `(parse-args spec &optional args &kwargs parser-args)`

Return arguments namespace parsed from *args* or `sys.argv` with `argparse.ArgumentParser.parse_args()` according to *spec*.

spec should be a list of arguments which will be passed to repeated calls to `argparse.ArgumentParser.add_argument()`. *parser-args* may be a list of keyword arguments to pass to the `argparse.ArgumentParser` constructor.

```
=> (parse-args [["strings" :nargs "+" :help "Strings"]
                ["-n" "--numbers" :action "append" :type 'int :help "Numbers"]]
               ["a" "b" "-n" "1" "-n" "2"]
               :description "Parse strings and numbers from args")
Namespace(numbers=[1, 2], strings=['a', 'b'])
```

partition

Usage: `(partition coll [n] [step] [fillvalue])`

Chunks *coll* into *n*-tuples (pairs by default).

```
=> (list (partition (range 10)))  ; n=2
[(0, 1), (2, 3), (4, 5), (6, 7), (8, 9)]
```

The *step* defaults to *n*, but can be more to skip elements, or less for a sliding window with overlap.

```
=> (list (partition (range 10) 2 3))
[(0, 1), (3, 4), (6, 7)]
=> (list (partition (range 5) 2 1))
[(0, 1), (1, 2), (2, 3), (3, 4)]
```

The remainder, if any, is not included unless a *fillvalue* is specified.

```
=> (list (partition (range 10) 3))
[(0, 1, 2), (3, 4, 5), (6, 7, 8)]
=> (list (partition (range 10) 3 :fillvalue "x"))
[(0, 1, 2), (3, 4, 5), (6, 7, 8), (9, 'x', 'x')]
```

pos?

Usage: `(pos? x)`

Returns `True` if *x* is greater than zero. Raises `TypeError` if `(not (numeric? x))`.

```
=> (pos? 3)
True

=> (pos? -2)
False

=> (pos? 0)
False
```

second

Usage: `(second coll)`

Returns the second member of *coll*. Equivalent to `(get coll 1)`.

```
=> (second [0 1 2])
1
```

some

New in version 0.10.0.

Usage: `(some pred coll)`

Returns the first logically-true value of `(pred x)` for any `x` in *coll*, otherwise `None`. Return `None` if *coll* is empty.

```
=> (some even? [2 4 6])
True

=> (none? (some even? [1 3 5]))
True

=> (none? (some identity [0 "" []]))
True

=> (some identity [0 "non-empty-string" []])
'non-empty-string'

=> (none? (some even? []))
True
```

list?

Usage: `(list? x)`

Returns `True` if *x* is a list.

```
=> (list? '(inc 41))
True

=> (list? '42)
False
```

string?

Usage: `(string? x)`

Returns `True` if *x* is a string (`str`).

```
=> (string? "foo")
True

=> (string? -2)
False
```

symbol?

Usage: `(symbol? x)`

Returns `True` if *x* is a symbol.

```
=> (symbol? 'foo)
True

=> (symbol? '[a b c])
False
```

tuple?

Usage: `(tuple? x)`

Returns `True` if *x* is a tuple.

```
=> (tuple? (, 42 44))
True

=> (tuple? [42 44])
False
```

zero?

Usage: `(zero? x)`

Returns `True` if *x* is zero.

```
=> (zero? 3)
False

=> (zero? -2)
False
```

(continues on next page)

(continued from previous page)

```
=> (zero? 0)
True
```

4.5.2 Sequence Functions

Sequence functions can either create or operate on a potentially infinite sequence without requiring the sequence be fully realized in a list or similar container. They do this by returning a Python iterator.

We can use the canonical infinite Fibonacci number generator as an example of how to use some of these functions.

```
(defn fib []
  (setv a 0)
  (setv b 1)
  (while True
    (yield a)
    (setv (, a b) (, b (+ a b)))))
```

Note the `(while True ...)` loop. If we run this in the REPL,

```
=> (fib)
<generator object fib at 0x101e642d0>
```

Calling the function only returns an iterator, but does no work until we consume it. Trying something like this is not recommend as the infinite loop will run until it consumes all available RAM, or in this case until I killed it.

```
=> (list (fib))
[1]    91474 killed     hy
```

To get the first 10 Fibonacci numbers, use *take*. Note that *take* also returns a generator, so I create a list from it.

```
=> (list (take 10 (fib)))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

To get the Fibonacci number at index 9, (starting from 0):

```
=> (nth (fib) 9)
34
```

cycle

Usage: `(cycle coll)`

Returns an infinite iterator of the members of coll.

```
=> (list (take 7 (cycle [1 2 3])))
[1, 2, 3, 1, 2, 3, 1]

=> (list (take 2 (cycle [1 2 3])))
[1, 2]
```

distinct

Usage: `(distinct coll)`

Returns an iterator containing only the unique members in *coll*.

```
=> (list (distinct [ 1 2 3 4 3 5 2 ]))
[1, 2, 3, 4, 5]

=> (list (distinct []))
[]

=> (list (distinct (iter [ 1 2 3 4 3 5 2 ])))
[1, 2, 3, 4, 5]
```

drop

Usage: `(drop n coll)`

Returns an iterator, skipping the first *n* members of *coll*. Raises `ValueError` if *n* is negative.

```
=> (list (drop 2 [1 2 3 4 5]))
[3, 4, 5]

=> (list (drop 4 [1 2 3 4 5]))
[5]

=> (list (drop 0 [1 2 3 4 5]))
[1, 2, 3, 4, 5]

=> (list (drop 6 [1 2 3 4 5]))
[]
```

drop-last

Usage: `(drop-last n coll)`

Returns an iterator of all but the last *n* items in *coll*. Raises `ValueError` if *n* is negative.

```
=> (list (drop-last 5 (range 10 20)))
[10, 11, 12, 13, 14]

=> (list (drop-last 0 (range 5)))
[0, 1, 2, 3, 4]

=> (list (drop-last 100 (range 100)))
[]

=> (list (take 5 (drop-last 100 (count 10))))
[10, 11, 12, 13, 14]
```

drop-while

Usage: `(drop-while pred coll)`

Returns an iterator, skipping members of *coll* until *pred* is `False`.

```
=> (list (drop-while even? [2 4 7 8 9]))
[7, 8, 9]

=> (list (drop-while numeric? [1 2 3 None "a"])))
[None, u'a']

=> (list (drop-while pos? [2 4 7 8 9]))
[]
```

filter

Usage: `(filter pred coll)`

Returns an iterator for all items in *coll* that pass the predicate *pred*.

See also *remove*.

```
=> (list (filter pos? [1 2 3 -4 5 -7]))
[1, 2, 3, 5]

=> (list (filter even? [1 2 3 -4 5 -7]))
[2, -4]
```

flatten

New in version 0.9.12.

Usage: `(flatten coll)`

Returns a single list of all the items in *coll*, by flattening all contained lists and/or tuples.

```
=> (flatten [1 2 [3 4] 5])
[1, 2, 3, 4, 5]

=> (flatten ["foo" (, 1 2) [1 [2 3] 4] "bar"])
['foo', 1, 2, 1, 2, 3, 4, 'bar']
```

iterate

Usage: `(iterate fn x)`

Returns an iterator of *x*, *fn(x)*, *fn(fn(x))*, etc.

```
=> (list (take 5 (iterate inc 5)))
[5, 6, 7, 8, 9]

=> (list (take 5 (iterate (fn [x] (* x x)) 5)))
[5, 25, 625, 390625, 152587890625]
```

read

Usage: `(read &optional [from-file eof])`

Reads the next Hy expression from *from-file* (defaulting to `sys.stdin`), and can take a single byte as EOF (defaults to an empty string). Raises `EOFError` if *from-file* ends before a complete expression can be parsed.

```
=> (read)
(+ 2 2)
HyExpression([
  HySymbol('+'),
  HyInteger(2),
  HyInteger(2)])
=> (eval (read))
(+ 2 2)
4
=> (import io)
=> (setv buffer (io.StringIO "(+ 2 2)\n(- 2 1)"))
=> (eval (read :from-file buffer))
4
=> (eval (read :from-file buffer))
1

=> (with [f (open "example.hy" "w")]
...   (.write f "(print 'hello)\n(print \"hyfriends!\")"))
35
=> (with [f (open "example.hy")]
...   (try (while True
...          (setv exp (read f))
...          (print "OHY" exp)
...          (eval exp))
...        (except [e EOFError]
...          (print "EOF!"))))
OHY HyExpression([
  HySymbol('print'),
  HyExpression([
    HySymbol('quote'),
    HySymbol('hello')])])
hello
OHY HyExpression([
  HySymbol('print'),
  HyString('hyfriends!')])
hyfriends!
EOF!
```

read-str

Usage: `(read-str "string")`

This is essentially a wrapper around *read* which reads expressions from a string:

```
=> (read-str "(print 1)")
HyExpression([
  HySymbol('print'),
  HyInteger(1)])
=> (eval (read-str "(print 1)"))
1
```

remove

Usage: `(remove pred coll)`

Returns an iterator from *coll* with elements that pass the predicate, *pred*, removed.

See also *filter*.

```
=> (list (remove odd? [1 2 3 4 5 6 7]))
[2, 4, 6]

=> (list (remove pos? [1 2 3 4 5 6 7]))
[]

=> (list (remove neg? [1 2 3 4 5 6 7]))
[1, 2, 3, 4, 5, 6, 7]
```

repeat

Usage: `(repeat x)`

Returns an iterator (infinite) of `x`.

```
=> (list (take 6 (repeat "s")))
[u's', u's', u's', u's', u's', u's']
```

repeatedly

Usage: `(repeatedly fn)`

Returns an iterator by calling *fn* repeatedly.

```
=> (import [random [randint]])

=> (list (take 5 (repeatedly (fn [] (randint 0 10)))))
[6, 2, 0, 6, 7]
```

take

Usage: `(take n coll)`

Returns an iterator containing the first *n* members of *coll*. Raises `ValueError` if *n* is negative.

```
=> (list (take 3 [1 2 3 4 5]))
[1, 2, 3]

=> (list (take 4 (repeat "s")))
[u's', u's', u's', u's']

=> (list (take 0 (repeat "s")))
[]
```

take-nth

Usage: `(take-nth n coll)`

Returns an iterator containing every *n*-th member of *coll*.

```
=> (list (take-nth 2 [1 2 3 4 5 6 7]))
[1, 3, 5, 7]

=> (list (take-nth 3 [1 2 3 4 5 6 7]))
[1, 4, 7]

=> (list (take-nth 4 [1 2 3 4 5 6 7]))
[1, 5]

=> (list (take-nth 10 [1 2 3 4 5 6 7]))
[1]
```

take-while

Usage: `(take-while pred coll)`

Returns an iterator from *coll* as long as *pred* returns `True`.

```
=> (list (take-while pos? [ 1 2 3 -4 5]))
[1, 2, 3]

=> (list (take-while neg? [ -4 -3 1 2 5]))
[-4, -3]

=> (list (take-while neg? [ 1 2 3 -4 5]))
[]
```

unmangle

Usage: `(unmangle x)`

Stringify the input and return a string that would *mangle* to it. Note that this isn't a one-to-one operation, and nor is `mangle`, so `mangle` and `unmangle` don't always round-trip.

```
=> (unmangle "foo_bar")
'foo-bar'
```

4.5.3 Included itertools

count cycle repeat accumulate chain compress drop-while remove group-by islice *map take-while tee zip-longest product permutations combinations multicombinations

All of Python's itertools are available. Some of their names have been changed:

- `starmap` has been changed to `*map`
- `combinations_with_replacement` has been changed to `multicombinations`
- `groupby` has been changed to `group-by`

- `takewhile` has been changed to `take-while`
- `dropwhile` has been changed to `drop-while`
- `filterfalse` has been changed to `remove`

4.6 Model Patterns

The module `hy.model-patterns` provides a library of parser combinators for parsing complex trees of Hy models. Model patterns exist mostly to help implement the compiler, but they can also be useful for writing macros.

4.6.1 A motivating example

The kind of problem that model patterns are suited for is the following. Suppose you want to validate and extract the components of a form like:

```
(setv form '(try
  (foo1)
  (foo2)
  (except [EType1]
    (foo3))
  (except [e EType2]
    (foo4)
    (foo5))
  (except []
    (foo6))
  (finally
    (foo7)
    (foo8))))
```

You could do this with loops and indexing, but it would take a lot of code and be error-prone. Model patterns concisely express the general form of an expression to be matched, like what a regular expression does for text. Here's a pattern for a `try` form of the above kind:

```
(import [funcparserlib.parser [maybe many]])
(import [hy.model-patterns [*]])
(setv parser (whole [
  (sym "try")
  (many (notpexpr "except" "else" "finally"))
  (many (pexpr
    (sym "except")
    (| (brackets) (brackets FORM) (brackets SYM FORM))
    (many FORM)))
  (maybe (dolike "else"))
  (maybe (dolike "finally"))]))
```

You can run the parser with `(.parse parser form)`. The result is:

```
(,
  ['(foo1) '(foo2)]
  [
    '([EType1] [(foo3)])
    '([e EType2] [(foo4) (foo5)])
    '([] [(foo6)])]
```

(continues on next page)

(continued from previous page)

```
None
'((foo7) (foo8)))
```

which is conveniently utilized with an assignment such as `(setv [body except-clauses else-part finally-part] result)`. Notice that `else-part` will be set to `None` because there is no `else` clause in the original form.

4.6.2 Usage

Model patterns are implemented as funcparserlib parser combinators. We won't reproduce funcparserlib's own documentation, but here are some important built-in parsers:

- `(+ ...)` matches its arguments in sequence.
- `(| ...)` matches any one of its arguments.
- `(>> parser function)` matches `parser`, then feeds the result through `function` to change the value that's produced on a successful parse.
- `(skip parser)` matches `parser`, but doesn't add it to the produced value.
- `(maybe parser)` matches `parser` if possible. Otherwise, it produces the value `None`.
- `(some function)` takes a predicate `function` and matches a form if it satisfies the predicate.

The best reference for Hy's parsers is the docstrings (use `(help hy.model-patterns)`), but again, here are some of the more important ones:

- `FORM` matches anything.
- `SYM` matches any symbol.
- `(sym "foo")` or `(sym ":foo")` matches and discards (per `skip`) the named symbol or keyword.
- `(brackets ...)` matches the arguments in square brackets.
- `(pexpr ...)` matches the arguments in parentheses.

Here's how you could write a simple macro using model patterns:

```
(defmacro pairs [&rest args]
  (import [funcparserlib.parser [many]])
  (import [hy.model-patterns [whole SYM FORM]])
  (setv [args] (->> args (.parse (whole [
    (many (+ SYM FORM))]))))
  `[~@(->> args (map (fn [x]
    (, (name (get x 0)) (get x 1)))))])

(print (pairs  a 1  b 2  c 3))
; => [["a" 1] ["b" 2] ["c" 3]]
```

A failed parse will raise `funcparserlib.parser.NoParseError`.

4.7 Internal Hy Documentation

Note: These bits are mostly useful for folks who hack on Hy itself, but can also be used for those delving deeper in macro programming.

4.7.1 Hy Models

Introduction to Hy Models

Hy models are a very thin layer on top of regular Python objects, representing Hy source code as data. Models only add source position information, and a handful of methods to support clean manipulation of Hy source code, for instance in macros. To achieve that goal, Hy models are mixins of a base Python class and *HyObject*.

HyObject

`hy.models.HyObject` is the base class of Hy models. It only implements one method, `replace`, which replaces the source position of the current object with the one passed as argument. This allows us to keep track of the original position of expressions that get modified by macros, be that in the compiler or in pure hy macros.

`HyObject` is not intended to be used directly to instantiate Hy models, but only as a mixin for other classes.

Compound Models

Parenthesized and bracketed lists are parsed as compound models by the Hy parser.

Hy uses pretty-printing reprs for its compound models by default. If this is causing issues, it can be turned off globally by setting `hy.models.PRETTY` to `False`, or temporarily by using the `hy.models.pretty` context manager.

Hy also attempts to color pretty reprs and errors using `colorama`. These can be turned off globally by setting `hy.models.COLORED` and `hy.errors.COLORED`, respectively, to `False`.

HySequence

`hy.models.HySequence` is the abstract base class of "iterable" Hy models, such as HyExpression and HyList.

Adding a HySequence to another iterable object reuses the class of the left-hand-side object, a useful behavior when you want to concatenate Hy objects in a macro, for instance.

HySequences are (mostly) immutable: you can't add, modify, or remove elements. You can still append to a variable containing a HySequence with += and otherwise construct new HySequences out of old ones.

HyList

`hy.models.HyList` is a *HySequence* for bracketed `[]` lists, which, when used as a top-level expression, translate to Python list literals in the compilation phase.

HyExpression

`hy.models.HyExpression` inherits *HySequence* for parenthesized `()` expressions. The compilation result of those expressions depends on the first element of the list: the compiler dispatches expressions between compiler special-forms, user-defined macros, and regular Python function calls.

HyDict

`hy.models.HyDict` inherits *HySequence* for curly-bracketed `{}` expressions, which compile down to a Python dictionary literal.

Atomic Models

In the input stream, double-quoted strings, respecting the Python notation for strings, are parsed as a single token, which is directly parsed as a *HyString*.

An uninterrupted string of characters, excluding spaces, brackets, quotes, double-quotes and comments, is parsed as an identifier.

Identifiers are resolved to atomic models during the parsing phase in the following order:

- *HyInteger*
- *HyFloat*
- *HyComplex* (if the atom isn't a bare `j`)
- *HyKeyword* (if the atom starts with `:`)
- *HySymbol*

HyString

`hy.models.HyString` represents string literals (including bracket strings), which compile down to unicode string literals (`str`) in Python.

`HyString`s are immutable.

Hy literal strings can span multiple lines, and are considered by the parser as a single unit, respecting the Python escapes for unicode strings.

`HyString`s have an attribute `brackets` that stores the custom delimiter used for a bracket string (e.g., `"=="` for `#[==[hello world]==]` and the empty string for `#[[hello world]]`). `HyString`s that are not produced by bracket strings have their `brackets` set to `None`.

HyBytes

`hy.models.HyBytes` is like `HyString`, but for sequences of bytes. It inherits from `bytes`.

Numeric Models

`hy.models.HyInteger` represents integer literals, using the `int` type.

`hy.models.HyFloat` represents floating-point literals.

`hy.models.HyComplex` represents complex literals.

Numeric models are parsed using the corresponding Python routine, and valid numeric python literals will be turned into their Hy counterpart.

HySymbol

`hy.models.HySymbol` is the model used to represent symbols in the Hy language. Like `HyString`, it inherits from `str` (or `unicode` on Python 2).

Symbols are *mangled* when they are compiled to Python variable names.

HyKeyword

`hy.models.HyKeyword` represents keywords in Hy. Keywords are symbols starting with a `:`. See *keywords*.

4.7.2 Hy Internal Theory

Overview

The Hy internals work by acting as a front-end to Python bytecode, so that Hy itself compiles down to Python Bytecode, allowing an unmodified Python runtime to run Hy code, without even noticing it.

The way we do this is by translating Hy into an internal Python AST datastructure, and building that AST down into Python bytecode using modules from the Python standard library, so that we don't have to duplicate all the work of the Python internals for every single Python release.

Hy works in four stages. The following sections will cover each step of Hy from source to runtime.

Steps 1 and 2: Tokenizing and Parsing

The first stage of compiling Hy is to lex the source into tokens that we can deal with. We use a project called rply, which is a really nice (and fast) parser, written in a subset of Python called rpython.

The lexing code is all defined in `hy.lex.lexer`. This code is mostly just defining the Hy grammar, and all the actual hard parts are taken care of by rply – we just define "callbacks" for rply in `hy.lex.parser`, which takes the tokens generated, and returns the Hy models.

You can think of the Hy models as the "AST" for Hy, it's what Macros operate on (directly), and it's what the compiler uses when it compiles Hy down.

See also:

Section *Hy Models* for more information on Hy models and what they mean.

Step 3: Hy Compilation to Python AST

This is where most of the magic in Hy happens. This is where we take Hy AST (the models), and compile them into Python AST. A couple of funky things happen here to work past a few problems in AST, and working in the compiler is some of the most important work we do have.

The compiler is a bit complex, so don't feel bad if you don't grok it on the first shot, it may take a bit of time to get right.

The main entry-point to the Compiler is `HyASTCompiler.compile`. This method is invoked, and the only real "public" method on the class (that is to say, we don't really promise the API beyond that method).

In fact, even internally, we don't recurse directly hardly ever, we almost always force the Hy tree through `compile`, and will often do this with sub-elements of an expression that we have. It's up to the Type-based dispatcher to properly dispatch sub-elements.

All methods that preform a compilation are marked with the `@builds()` decorator. You can either pass the class of the Hy model that it compiles, or you can use a string for expressions. I'll clear this up in a second.

First Stage Type-Dispatch

Let's start in the `compile` method. The first thing we do is check the Type of the thing we're building. We look up to see if we have a method that can build the `type()` that we have, and dispatch to the method that can handle it. If we don't have any methods that can build that type, we raise an internal `Exception`.

For instance, if we have a `HyString`, we have an almost 1-to-1 mapping of Hy AST to Python AST. The `compile_string` method takes the `HyString`, and returns an `ast.Str()` that's populated with the correct line-numbers and content.

Macro-Expand

If we get a `HyExpression`, we'll attempt to see if this is a known Macro, and push to have it expanded by invoking `hy.macros.macroexpand`, then push the result back into `HyASTCompiler.compile`.

Second Stage Expression-Dispatch

The only special case is the `HyExpression`, since we need to create different AST depending on the special form in question. For instance, when we hit an `(if True True False)`, we need to generate a `ast.If`, and properly compile the sub-nodes. This is where the `@builds()` with a String as an argument comes in.

For the `compile_expression` (which is defined with an `@builds(HyExpression)`) will dispatch based on the string of the first argument. If, for some reason, the first argument is not a string, it will properly handle that case as well (most likely by raising an `Exception`).

If the String isn't known to Hy, it will default to create an `ast.Call`, which will try to do a runtime call (in Python, something like `foo()`).

Issues Hit with Python AST

Python AST is great; it's what's enabled us to write such a powerful project on top of Python without having to fight Python too hard. Like anything, we've had our fair share of issues, and here's a short list of the common ones you might run into.

Python differentiates between Statements and Expressions.

This might not sound like a big deal – in fact, to most Python programmers, this will shortly become a "Well, yeah" moment.

In Python, doing something like:

`print for x in range(10):  pass`, because `print` prints expressions, and `for` isn't an expression, it's a control flow statement. Things like `1 + 1` are Expressions, as is `lambda x:  1 + x`, but other language features, such as `if`, `for`, or `while` are statements.

Since they have no "value" to Python, this makes working in Hy hard, since doing something like `(print (if True True False))` is not just common, it's expected.

As a result, we reconfigure things using a `Result` object, where we offer up any `ast.stmt` that need to get run, and a single `ast.expr` that can be used to get the value of whatever was just run. Hy does this by forcing assignment to things while running.

As example, the Hy:

```
(print (if True True False))
```

Will turn into:

```
if True:
    _temp_name_here = True
else:
    _temp_name_here = False

print(_temp_name_here)
```

OK, that was a bit of a lie, since we actually turn that statement into:

```
print(True if True else False)
```

By forcing things into an `ast.expr` if we can, but the general idea holds.

Step 4: Python Bytecode Output and Runtime

After we have a Python AST tree that's complete, we can try and compile it to Python bytecode by pushing it through `eval`. From here on out, we're no longer in control, and Python is taking care of everything. This is why things like Python tracebacks, pdb and django apps work.

4.7.3 Hy Macros

Using gensym for Safer Macros

When writing macros, one must be careful to avoid capturing external variables or using variable names that might conflict with user code.

We will use an example macro `nif` (see http://letoverlambda.com/index.cl/guest/chap3.html#sec_5 for a more complete description.) `nif` is an example, something like a numeric `if`, where based on the expression, one of the 3 forms is called depending on if the expression is positive, zero or negative.

A first pass might be something like:

```
(defmacro nif [expr pos-form zero-form neg-form]
  `(do
    (setv obscure-name ~expr)
    (cond [(pos? obscure-name) ~pos-form]
          [(zero? obscure-name) ~zero-form]
          [(neg? obscure-name) ~neg-form])))
```

where `obscure-name` is an attempt to pick some variable name as not to conflict with other code. But of course, while well-intentioned, this is no guarantee.

The method *gensym* is designed to generate a new, unique symbol for just such an occasion. A much better version of `nif` would be:

```
(defmacro nif [expr pos-form zero-form neg-form]
  (setv g (gensym))
  `(do
     (setv ~g ~expr)
```

(continues on next page)

(continued from previous page)

```
     (cond [(pos? ~g) ~pos-form]
           [(zero? ~g) ~zero-form]
           [(neg? ~g) ~neg-form])))
```

This is an easy case, since there is only one symbol. But if there is a need for several gensym's there is a second macro *with-gensyms* that basically expands to a `setv` form:

```
(with-gensyms [a b c]
  ...)
```

expands to:

```
(do
  (setv a (gensym)
        b (gensym)
        c (gensym))
  ...)
```

so our re-written `nif` would look like:

```
(defmacro nif [expr pos-form zero-form neg-form]
  (with-gensyms [g]
    `(do
       (setv ~g ~expr)
       (cond [(pos? ~g) ~pos-form]
             [(zero? ~g) ~zero-form]
             [(neg? ~g) ~neg-form]))))
```

Finally, though we can make a new macro that does all this for us. *defmacro/g!* will take all symbols that begin with `g!` and automatically call `gensym` with the remainder of the symbol. So `g!a` would become `(gensym "a")`.

Our final version of `nif`, built with `defmacro/g!` becomes:

```
(defmacro/g! nif [expr pos-form zero-form neg-form]
  `(do
     (setv ~g!res ~expr)
     (cond [(pos? ~g!res) ~pos-form]
           [(zero? ~g!res) ~zero-form]
           [(neg? ~g!res) ~neg-form])))
```

Checking Macro Arguments and Raising Exceptions

4.7.4 Hy Compiler Built-Ins

CHAPTER 5

Extra Modules Index

These modules are considered no less stable than Hy's built-in functions and macros, but they need to be loaded with `(import ...)` or `(require ...)`.

Contents:

5.1 Anaphoric Macros

New in version 0.9.12.

The anaphoric macros module makes functional programming in Hy very concise and easy to read.

> An anaphoric macro is a type of programming macro that deliberately captures some form supplied to the macro which may be referred to by an anaphor (an expression referring to another).
>
> —Wikipedia (https://en.wikipedia.org/wiki/Anaphoric_macro)

To use these macros you need to require the `hy.extra.anaphoric` module like so:

`(require [hy.extra.anaphoric [*]])`

These macros are implemented by replacing any use of the designated anaphoric symbols (`it`, in most cases) with a gensym. Consequently, it's unwise to nest these macros, or to use an affected symbol as something other than a variable name, as in `(print "My favorite Stephen King book is" 'it)`.

5.1.1 ap-if

Usage: `(ap-if test-form then-form else-form)`

As *if*, but the result of the test form is named `it` in the subsequent forms. As with `if`, the else-clause is optional.

```
=> (import os)
=> (ap-if (.get os.environ "PYTHONPATH")
...   (print "Your PYTHONPATH is" it))
```

5.1.2 ap-each

Usage: `(ap-each xs body...)`

Evaluate the body forms for each element `it` of `xs` and return `None`.

```
=> (ap-each [1 2 3] (print it))
1
2
3
```

5.1.3 ap-each-while

Usage: `(ap-each-while xs pred body...)`

As `ap-each`, but the form `pred` is run before the body forms on each iteration, and the loop ends if `pred` is false.

```
=> (ap-each-while [1 2 3 4 5 6] (< it 4) (print it))
1
2
3
```

5.1.4 ap-map

Usage: `(ap-map form xs)`

Create a generator like `map()` that yields each result of `form` evaluated with `it` bound to successive elements of `xs`.

```
=> (list (ap-map (* it 2) [1 2 3]))
[2, 4, 6]
```

5.1.5 ap-map-when

Usage: `(ap-map-when predfn rep xs)`

As `ap-map`, but the predicate function `predfn` (yes, that's a function, not an anaphoric form) is applied to each `it`, and the anaphoric mapping form `rep` is only applied if the predicate is true. Otherwise, `it` is yielded unchanged.

```
=> (list (ap-map-when odd? (* it 2) [1 2 3 4]))
[2, 2, 6, 4]

=> (list (ap-map-when even? (* it 2) [1 2 3 4]))
[1, 4, 3, 8]
```

5.1.6 ap-filter

Usage: `(ap-filter form xs)`

The `filter()` equivalent of `ap-map`.

```
=> (list (ap-filter (> (* it 2) 6) [1 2 3 4 5]))
[4, 5]
```

5.1.7 ap-reject

Usage: `(ap-reject form xs)`

Equivalent to `(ap-filter (not form) xs)`.

```
=> (list (ap-reject (> (* it 2) 6) [1 2 3 4 5]))
[1, 2, 3]
```

5.1.8 ap-dotimes

Usage: `(ap-dotimes n body...)`

Equivalent to `(ap-each (range n) body...)`.

```
 => (setv n [])
 => (ap-dotimes 3 (.append n it))
 => n
[0, 1, 2]
```

5.1.9 ap-first

Usage: `(ap-first form xs)`

Evaluate the predicate `form` for each element `it` of `xs`. When the predicate is true, stop and return `it`. If the predicate is never true, return `None`.

```
=> (ap-first (> it 5) (range 10))
6
```

5.1.10 ap-last

Usage: `(ap-last form list)`

Evaluate the predicate `form` for every element `it` of `xs`. Return the last element for which the predicate is true, or `None` if there is no such element.

```
=> (ap-last (> it 5) (range 10))
9
```

5.1.11 ap-reduce

Usage: `(ap-reduce form xs &optional initial-value)`

This macro is an anaphoric version of `reduce()`. It works as follows:

- Bind `acc` to the first element of `xs`, bind `it` to the second, and evaluate `form`.
- Bind `acc` to the result, bind `it` to the third value of `xs`, and evaluate `form` again.
- Bind `acc` to the result, and continue until `xs` is exhausted.

If `initial-value` is supplied, the process instead begins with `acc` set to `initial-value` and `it` set to the first element of `xs`.

```
=> (ap-reduce (+ it acc) (range 10))
45
```

5.1.12 #%

Usage: `#% expr`

Makes an expression into a function with an implicit `%` parameter list.

A `%i` symbol designates the (1-based) *i* th parameter (such as `%3`). Only the maximum `%i` determines the number of `%i` parameters–the others need not appear in the expression. `%*` and `%**` name the `&rest` and `&kwargs` parameters, respectively.

```
=> (#%[%1 %6 42 [%2 %3] %* %4] 1 2 3 4 555 6 7 8)
[1, 6, 42, [2, 3], (7, 8), 4]
=> (#% %** :foo 2)
{"foo": 2}
```

When used on an s-expression, `#%` is similar to Clojure's anonymous function literals–`#()`.

```
=> (setv add-10 #%(+ 10 %1))
=> (add-10 6)
16
```

`#%` determines the parameter list by the presence of a `%*` or `%**` symbol and by the maximum `%i` symbol found *anywhere* in the expression, so nesting of `#%` forms is not recommended.

5.2 Reserved Names

5.2.1 names

Usage: `(names)`

This function can be used to get a list (actually, a `frozenset`) of the names of Hy's built-in functions, macros, and special forms. The output also includes all Python reserved words. All names are in unmangled form (e.g., `not-in` rather than `not_in`).

```
=> (import hy.extra.reserved)
=> (in "defclass" (hy.extra.reserved.names))
True
```

CHAPTER 6

Contributor Modules Index

These modules are experimental additions to Hy. Once deemed mature, they will be moved to the `hy.extra` namespace or loaded by default.

Contents:

6.1 loop/recur

New in version 0.10.0.

The `loop` / `recur` macro gives programmers a simple way to use tail-call optimization (TCO) in their Hy code.

> A tail call is a subroutine call that happens inside another procedure as its final action; it may produce a return value which is then immediately returned by the calling procedure. If any call that a subroutine performs, such that it might eventually lead to this same subroutine being called again down the call chain, is in tail position, such a subroutine is said to be tail-recursive, which is a special case of recursion. Tail calls are significant because they can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is not needed any more, and it can be replaced by the frame of the tail call. The program can then jump to the called subroutine. Producing such code instead of a standard call sequence is called tail call elimination, or tail call optimization. Tail call elimination allows procedure calls in tail position to be implemented as efficiently as goto statements, thus allowing efficient structured programming.
>
> —Wikipedia (https://en.wikipedia.org/wiki/Tail_call)

6.1.1 Macros

loop

`loop` establishes a recursion point. With `loop`, `recur` rebinds the variables set in the recursion point and sends code execution back to that recursion point. If `recur` is used in a non-tail position, an exception is raised.

Usage: `(loop bindings &rest body)`

Example:

```
(require [hy.contrib.loop [loop]])

(defn factorial [n]
  (loop [[i n] [acc 1]]
    (if (zero? i)
      acc
      (recur (dec i) (* acc i)))))

(factorial 1000)
```

6.2 Profile

New in version 0.10.0.

The `profile` macros make it easier to find bottlenecks.

6.2.1 Macros

profile/calls

`profile/calls` allows you to create a call graph visualization. **Note:** You must have Graphviz installed for this to work.

Usage: *(profile/calls (body))*

Example:

```
(require [hy.contrib.profile [profile/calls]])
(profile/calls (print "hey there"))
```

profile/cpu

`profile/cpu` allows you to profile a bit of code.

Usage: *(profile/cpu (body))*

Example:

```
(require [hy.contrib.profile [profile/cpu]])
(profile/cpu (print "hey there"))
```

```
hey there
<pstats.Stats instance at 0x14ff320>
         2 function calls in 0.000 seconds

 Random listing order was used

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)        1    0.
↪000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     1    0.000    0.000    0.000    0.000 {print}
```

6.3 Lazy sequences

New in version 0.12.0.

The sequences module contains a few macros for declaring sequences that are evaluated only as much as the client code requires. Unlike generators, they allow accessing the same element multiple times. They cache calculated values, and the implementation allows for recursive definition of sequences without resulting in recursive computation.

To use these macros, you need to require them and import some other names like so:

```
(require [hy.contrib.sequences [defseq seq]])
(import [hy.contrib.sequences [Sequence end-sequence]])
```

The simplest sequence can be defined as `(seq [n] n)`. This defines a sequence that starts as `[0 1 2 3 ...]` and continues forever. In order to define a finite sequence, you need to call `end-sequence` to signal the end of the sequence:

```
(seq [n]
     "sequence of 5 integers"
     (cond [(< n 5) n]
           [True (end-sequence)]))
```

This creates the following sequence: `[0 1 2 3 4]`. For such a sequence, `len` returns the amount of items in the sequence and negative indexing is supported. Because both of these require evaluating the whole sequence, calling one on an infinite sequence would take forever (or at least until available memory has been exhausted).

Sequences can be defined recursively. For example, the Fibonacci sequence could be defined as:

```
(defseq fibonacci [n]
  "infinite sequence of fibonacci numbers"
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [True (+ (get fibonacci (- n 1))
                 (get fibonacci (- n 2)))]))
```

This results in the sequence `[0 1 1 2 3 5 8 13 21 34 ...]`.

6.3.1 seq

Usage: `(seq [n] (* n n))`

Creates a sequence defined in terms of `n`.

6.3.2 defseq

Usage: `(defseq numbers [n] n)`

Creates a sequence defined in terms of `n` and assigns it to a given name.

6.3.3 end-sequence

Usage: `(seq [n] (if (< n 5) n (end-sequence)))`

Signals the end of a sequence when an iterator reaches the given point of the sequence. Internally, this is done by raising `IndexError`, catching that in the iterator, and raising `StopIteration`.

6.4 walk

New in version 0.11.0.

6.4.1 Functions

walk

Usage: *(walk inner outer form)*

`walk` traverses `form`, an arbitrary data structure. Applies `inner` to each element of form, building up a data structure of the same type. Applies `outer` to the result.

Example:

```
=> (import [hy.contrib.walk [walk]])
=> (setv a '(a b c d e f))
=> (walk ord identity a)
HyExpression([
  97,
  98,
  99,
  100,
  101,
  102])
=> (walk ord first a)
97
```

postwalk

Usage: *(postwalk f form)*

Performs depth-first, post-order traversal of `form`. Calls `f` on each sub-form, uses `f` 's return value in place of the original.

```
=> (import [hy.contrib.walk [postwalk]])
=> (setv trail '([1 2 3] [4 [5 6 [7]]]))
=> (defn walking [x]
...   (print "Walking:" x :sep "\n")
...   x)
=> (postwalk walking trail)
Walking:
1
Walking:
2
Walking:
3
Walking:
HyExpression([
  HyInteger(1),
  HyInteger(2),
  HyInteger(3)])
Walking:
4
Walking:
```

(continues on next page)

(continued from previous page)

```
5
Walking:
6
Walking:
7
Walking:
HyExpression([
  HyInteger(7)])
Walking:
HyExpression([
  HyInteger(5),
  HyInteger(6),
  HyList([
    HyInteger(7)])])
Walking:
HyExpression([
  HyInteger(4),
  HyList([
    HyInteger(5),
    HyInteger(6),
    HyList([
      HyInteger(7)])])])
Walking:
HyExpression([
  HyList([
    HyInteger(1),
    HyInteger(2),
    HyInteger(3)]),
  HyList([
    HyInteger(4),
    HyList([
      HyInteger(5),
      HyInteger(6),
      HyList([
        HyInteger(7)])])])])
HyExpression([
  HyList([
    HyInteger(1),
    HyInteger(2),
    HyInteger(3)]),
  HyList([
    HyInteger(4),
    HyList([
      HyInteger(5),
      HyInteger(6),
      HyList([
        HyInteger(7)])])])])
```

prewalk

Usage: *(prewalk f form)*

Performs depth-first, pre-order traversal of `form`. Calls `f` on each sub-form, uses `f` 's return value in place of the original.

```
=> (import [hy.contrib.walk [prewalk]])
=> (setv trail '([1 2 3] [4 [5 6 [7]]]))
=> (defn walking [x]
...   (print "Walking:" x :sep "\n")
...   x)
=> (prewalk walking trail)
Walking:
HyExpression([
  HyList([
    HyInteger(1),
    HyInteger(2),
    HyInteger(3)]),
  HyList([
    HyInteger(4),
    HyList([
      HyInteger(5),
      HyInteger(6),
      HyList([
        HyInteger(7)])])])])
Walking:
HyList([
  HyInteger(1),
  HyInteger(2),
  HyInteger(3)])
Walking:
1
Walking:
2
Walking:
3
Walking:
HyList([
  HyInteger(4),
  HyList([
    HyInteger(5),
    HyInteger(6),
    HyList([
      HyInteger(7)])])])
Walking:
4
Walking:
HyList([
  HyInteger(5),
  HyInteger(6),
  HyList([
    HyInteger(7)])])
Walking:
5
Walking:
6
Walking:
HyList([
  HyInteger(7)])
Walking:
7
HyExpression([
  HyList([
```

(continues on next page)

(continued from previous page)

```
    HyInteger(1),
    HyInteger(2),
    HyInteger(3)]),
  HyList([
    HyInteger(4),
    HyList([
      HyInteger(5),
      HyInteger(6),
      HyList([
        HyInteger(7)])])])])])
```

macroexpand-all

Usage: *(macroexpand-all form &optional module-name)*

Recursively performs all possible macroexpansions in form, using the `require` context of `module-name`. *macroexpand-all* assumes the calling module's context if unspecified.

6.4.2 Macros

let

`let` creates lexically-scoped names for local variables. A let-bound name ceases to refer to that local outside the `let` form. Arguments in nested functions and bindings in nested `let` forms can shadow these names.

```
=> (let [x 5]  ; creates a new local bound to name 'x
...   (print x)
...   (let [x 6]  ; new local and name binding that shadows 'x
...     (print x))
...   (print x))  ; 'x refers to the first local again
5
6
5
```

Basic assignments (e.g. `setv`, `+=`) will update the local variable named by a let binding, when they assign to a let-bound name.

But assignments via `import` are always hoisted to normal Python scope, and likewise, `defclass` will assign the class to the Python scope, even if it shares the name of a let binding.

Use `importlib.import_module` and `type` (or whatever metaclass) instead, if you must avoid this hoisting.

The `let` macro takes two parameters: a list defining *variables* and the *body* which gets executed. *variables* is a vector of variable and value pairs.

Like the `let*` of many other Lisps, `let` executes the variable assignments one-by-one, in the order written:

```
.. code-block:: hy
```

=> (let [x 5 … y (+ x 1)] … (print x y)) 5 6

Unlike them, however, each `(let ...)` form uses only one namespace for all its assignments. Thus, `(let [x 1 x (fn [] x)] (x))` returns a function object, not 1 as you might expect.

It is an error to use a let-bound name in a `global` or `nonlocal` form.

6.5 Hy representations

New in version 0.13.0.

`hy.contrib.hy-repr` is a module containing two functions. To import them, say:

```
(import [hy.contrib.hy-repr [hy-repr hy-repr-register]])
```

To make the Hy REPL use it for output, invoke Hy like so:

```
$ hy --repl-output-fn=hy.contrib.hy-repr.hy-repr
```

6.5.1 hy-repr

Usage: `(hy-repr x)`

This function is Hy's equivalent of Python's built-in `repr`. It returns a string representing the input object in Hy syntax.

```
=> (hy-repr [1 2 3])
'[1 2 3]'
=> (repr [1 2 3])
'[1, 2, 3]'
```

Like `repr` in Python, `hy-repr` can round-trip many kinds of values. Round-tripping implies that given an object `x`, `(eval (read-str (hy-repr x)))` returns `x`, or at least a value that's equal to `x`.

6.5.2 hy-repr-register

Usage: `(hy-repr-register the-type fun)`

`hy-repr-register` lets you set the function that `hy-repr` calls to represent a type.

```
=> (defclass C)
=> (hy-repr-register C (fn [x] "cuddles"))
=> (hy-repr [1 (C) 2])
'[1 cuddles 2]'
```

If the type of an object passed to `hy-repr` doesn't have a registered function, `hy-repr` will search the type's method resolution order (its `__mro__` attribute) for the first type that does. If `hy-repr` doesn't find a candidate, it falls back on `repr`.

Registered functions often call `hy-repr` themselves. `hy-repr` will automatically detect self-references, even deeply nested ones, and output `"..."` for them instead of calling the usual registered function. To use a placeholder other than `"..."`, pass a string of your choice to the keyword argument `:placeholder` of `hy-repr-register`.

```
(defclass Container [object]
  (defn __init__ (fn [self value]
    (setv self.value value))))
(hy-repr-register Container :placeholder "HY THERE" (fn [x]
  (+ "(Container " (hy-repr x.value) ")")))
(setv container (Container 5))
(setv container.value container)
(print (hy-repr container))  ; Prints "(Container HY THERE)"
```

CHAPTER 7

Hacking on Hy

7.1 Join our Hyve!

Please come hack on Hy!

Please come hang out with us on `#hy` on `irc.freenode.net`!

Please talk about it on Twitter with the `#hy` hashtag!

Please blog about it!

Please don't spraypaint it on your neighbor's fence (without asking nicely)!

7.2 Hack!

Do this:

1. Create a virtual environment:

   ```
   $ virtualenv venv
   ```

   and activate it:

   ```
   $ . venv/bin/activate
   ```

   or use virtualenvwrapper to create and manage your virtual environment:

   ```
   $ mkvirtualenv hy
   $ workon hy
   ```

2. Get the source code:

   ```
   $ git clone https://github.com/hylang/hy.git
   ```

or use your fork:

```
$ git clone git@github.com:<YOUR_USERNAME>/hy.git
```

3. Install for hacking:

```
$ cd hy/
$ pip install -e .
```

4. Install other develop-y requirements:

```
$ pip install -r requirements-dev.txt
```

5. Do awesome things; make someone shriek in delight/disgust at what you have wrought.

7.3 Test!

Tests are located in `tests/`. We use pytest.

To run the tests:

```
$ pytest
```

Write tests—tests are good!

Also, it is good to run the tests for all the platforms supported and for PEP 8 compliant code. You can do so by running tox:

```
$ tox
```

7.4 Document!

Documentation is located in `docs/`. We use Sphinx.

To build the docs in HTML:

```
$ cd docs
$ make html
```

Write docs—docs are good! Even this doc!

7.5 Contributor Guidelines

Contributions are welcome and greatly appreciated. Every little bit helps in making Hy better. Potential contributions include:

- Reporting and fixing bugs.
- Requesting features.
- Adding features.
- Writing tests for outstanding bugs or untested features. - You can mark tests that Hy can't pass yet as xfail.

- Cleaning up the code.
- Improving the documentation.
- Answering questions on the IRC channel, the mailing list, or Stack Overflow.
- Evangelizing for Hy in your organization, user group, conference, or bus stop.

7.5.1 Issues

In order to report bugs or request features, search the issue tracker to check for a duplicate. (If you're reporting a bug, make sure you can reproduce it with the very latest, bleeding-edge version of Hy from the `master` branch on GitHub. Bugs in stable versions of Hy are fixed on `master` before the fix makes it into a new stable release.) If there aren't any duplicates, then you can make a new issue.

It's totally acceptable to create an issue when you're unsure whether something is a bug or not. We'll help you figure it out.

Use the same issue tracker to report problems with the documentation.

7.5.2 Pull requests

Submit proposed changes to the code or documentation as pull requests (PRs) on GitHub. Git can be intimidating and confusing to the uninitiated. This getting-started guide may be helpful. However, if you're overwhelmed by Git, GitHub, or the rules below, don't sweat it. We want to keep the barrier to contribution low, so we're happy to help you with these finicky things or do them for you if necessary.

Deciding what to do

Issues tagged good-first-bug are expected to be relatively easy to fix, so they may be good targets for your first PR for Hy.

If you're proposing a major change to the Hy language, or you're unsure of the proposed change, create an issue to discuss it before you write any code. This will allow others to give feedback on your idea, and it can avoid wasted work.

File headers

Every Python or Hy file in the source tree that is potentially copyrightable should have the following header (but with `;;` in place of `#` for Hy files):

```
# Copyright [current year] the authors.
# This file is part of Hy, which is free software licensed under the Expat
# license. See the LICENSE.
```

As a rule of thumb, a file can be considered potentially copyrightable if it includes at least 10 lines that contain something other than comments or whitespace. If in doubt, include the header.

Commit formatting

Many PRs are small enough that only one commit is necessary, but bigger ones should be organized into logical units as separate commits. PRs should be free of merge commits and commits that fix or revert other commits in the same PR (`git rebase` is your friend).

Avoid committing spurious whitespace changes.

The first line of a commit message should describe the overall change in 50 characters or less. If you wish to add more information, separate it from the first line with a blank line.

Testing

New features and bug fixes should be tested. If you've caused an xfail test to start passing, remove the xfail mark. If you're testing a bug that has a GitHub issue, include a comment with the URL of the issue.

No PR may be merged if it causes any tests to fail. You can run the test suite and check the style of your code with `make d`. The byte-compiled versions of the test files can be purged using `git clean -dfx tests/`. If you want to run the tests while skipping the slow ones in `test_bin.py`, use `pytest --ignore=tests/test_bin.py`.

NEWS and AUTHORS

If you're making user-visible changes to the code, add one or more items describing it to the NEWS file.

Finally, add yourself to the AUTHORS file (as a separate commit): you deserve it. :)

The PR itself

PRs should ask to merge a new branch that you created for the PR into hylang/hy's `master` branch, and they should have as their origin the most recent commit possible.

If the PR fulfills one or more issues, then the body text of the PR (or the commit message for any of its commits) should say "Fixes #123" or "Closes #123" for each affected issue number. Use this exact (case-insensitive) wording, because when a PR containing such text is merged, GitHub automatically closes the mentioned issues, which is handy. Conversely, avoid this exact language if you want to mention an issue without closing it (because e.g. you've partly but not entirely fixed a bug).

There are two situations in which a PR is allowed to be merged:

1. When it is approved by **two** members of Hy's core team other than the PR's author. Changes to the documentation, or trivial changes to code, need only **one** approving member.
2. When the PR is at least **two weeks** old and **no** member of the Hy core team has expressed disapproval of the PR in its current state. (Exception: a PR to create a new release is not eligible to be merged under this criterion, only the first one.)

Anybody on the Hy core team may perform the merge. Merging should create a merge commit (don't squash unnecessarily, because that would remove separation between logically separate commits, and don't fast-forward, because that would throw away the history of the commits as a separate branch), which should include the PR number in the commit message.

7.6 Contributor Code of Conduct

As contributors and maintainers of this project, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, or religion.

Examples of unacceptable behavior by participants include the use of sexual language or imagery, derogatory comments or personal attacks, trolling, public or private harassment, insults, or other unprofessional conduct.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct. Project maintainers who do not follow the Code of Conduct may be removed from the project team.

This code of conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by opening an issue or contacting one or more of the project maintainers.

This Code of Conduct is adapted from the Contributor Covenant, version 1.1.0, available at http://contributor-covenant.org/version/1/1/0/.

7.7 Core Team

The core development team of Hy consists of following developers:

- Kodi B. Arfer
- Nicolas Dandrimont
- Julien Danjou
- Rob Day
- Simon Gomizelj
- Ryan Gonzalez
- Abhishek Lekshmanan
- Morten Linderud
- Matthew Odendahl
- Paul Tagliamonte
- Brandon T. Willard

Index

Symbols

C

F

P