# CONTENTS

1 Why Hy? ........................................................................................................................................ 3
  1.1 Hy versus Python .................................................................................................................. 3
  1.2 Hy versus other Lisps .......................................................................................................... 4

2 Tutorial ......................................................................................................................................... 7
  2.1 Lisp-stick on a Python .......................................................................................................... 7
  2.2 Literals ................................................................................................................................... 9
  2.3 Basic operations ..................................................................................................................... 9
  2.4 Functions, classes, and modules ............................................................................................ 11
  2.5 Macros .................................................................................................................................. 12
  2.6 Hyrule .................................................................................................................................. 13
  2.7 Next steps ............................................................................................................................. 13

3 Hy Style Guide ........................................................................................................................... 15
  3.1 Layout & Indentation ........................................................................................................... 15
    3.1.1 The Three Laws ............................................................................................................. 15
    3.1.2 Limits ............................................................................................................................ 18
    3.1.3 Whitespace ..................................................................................................................... 18
    3.1.4 Alignment ....................................................................................................................... 18
    3.1.5 Hold it Open ................................................................................................................... 19
    3.1.6 Snuggle .......................................................................................................................... 20
    3.1.7 Grouping ......................................................................................................................... 20
    3.1.8 Special Arguments ......................................................................................................... 23
    3.1.9 Removing Whitespace .................................................................................................... 24
    3.1.10 Close Bracket, Close Line ........................................................................................... 26
    3.1.11 Comments ..................................................................................................................... 28
  3.2 Coding Style .......................................................................................................................... 29
    3.2.1 Pythonic Names ............................................................................................................. 29
    3.2.2 Threading Macros .......................................................................................................... 29
    3.2.3 Method Calls .................................................................................................................. 30
    3.2.4 Use More Arguments .................................................................................................... 30
    3.2.5 Imports .......................................................................................................................... 31
    3.2.6 Underscores ................................................................................................................. 31
  3.3 Thanks ..................................................................................................................................... 32

4 Documentation Index .................................................................................................................... 33
  4.1 Command Line Interface ....................................................................................................... 33
    4.1.1 hy ................................................................................................................................. 33
    4.1.2 hyc ............................................................................................................................... 34
    4.1.3 hy2py ............................................................................................................................ 34
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 alpha of Hy, just use the command `pip3 install --pre --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`.
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 #* body]
  `(do
     ~body
     (while ~condition
       ~body)))
```

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, The with statement returns the value of its last body form, so you can use it like an ordinary function call:
To be even more concise, you can put a `with` form in a `gfor`:

```
(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)`, after importing the function `+` from the module `hy.pyops`.

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 lists, sets, 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 `hy.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.
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:

```hy
(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 `+`:

```hy
(+ 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 \((\text{sqrt} \ 2)\). 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:

\[
(- (\times (\text{+} \ 1 \ 3 \ 88) \text{2}) \ 8)
\]

This code returns 176. Why? We can see the infix equivalent with the command `echo "(- (\times (\text{+} \ 1 \ 3 \ 88) \text{2}) \text{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:

\[
((1 + 3 + 88) \times 2) - 8
\]

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:

\[
\begin{align*}
(- (\times (1 \ 3 \ 88) \text{2}) \text{8}) \\
(- (\times \text{92} \text{2}) \text{8}) \\
(- \text{184} \text{8}) \\
\text{176}
\end{align*}
\]

The basic unit of Lisp syntax, which is similar to a C or Python expression, is the form. 92, \times, and \((\times \text{92} \text{2})\) 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 or macro. 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.

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

\[
(\text{print} \ (** \ 2 \ 64)) \ ; \text{Max 64-bit unsigned integer value}
\]

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
(\text{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.

<table>
<thead>
<tr>
<th>Hy</th>
<th>Python</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>int</td>
</tr>
<tr>
<td>1.2</td>
<td>1.2</td>
<td>float</td>
</tr>
<tr>
<td>4j</td>
<td>4j</td>
<td>complex</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>bool</td>
</tr>
<tr>
<td>None</td>
<td>None</td>
<td>NoneType</td>
</tr>
<tr>
<td>&quot;hy&quot;</td>
<td>'hy'</td>
<td>str</td>
</tr>
<tr>
<td>b&quot;hy&quot;</td>
<td>b'hy'</td>
<td>bytes</td>
</tr>
<tr>
<td>(, 1 2 3)</td>
<td>(1, 2, 3)</td>
<td>tuple</td>
</tr>
<tr>
<td>[1 2 3]</td>
<td>[1, 2, 3]</td>
<td>list</td>
</tr>
<tr>
<td>#{1 2 3}</td>
<td>{1, 2, 3}</td>
<td>set</td>
</tr>
<tr>
<td>{1 2 3 4}</td>
<td>{1: 2, 3}</td>
<td>dict</td>
</tr>
</tbody>
</table>

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 Hy syntax by default, with the function hy.repr:

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

But if you start Hy like this:

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

the REPL will use Python’s native repr function instead, so you’ll see values in Python syntax:

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

2.3 Basic operations

Set variables with setv:

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

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

```hy
(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:

```hy
(print (cut "abcdef" 1 4)); => bcd
```
Conditional logic can be built with `if`:

```hy
(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 macro `do` (known more traditionally in Lisp as `progn`), which combines several forms into one, returning the last:

```hy
(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`:

```hy
(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`:

```hy
(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.
2.4 Functions, classes, and modules

Define named functions with `defn`:

```hy
(defn fib [n]
  (if (< n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2))))

(print (fib 8)) ; => 21
```

Define anonymous functions with `fn`:

```hy
(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:

```hy
(defn test [a b [c None] [d "x"] #* 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`:

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

Define classes with `defclass`:

```hy
(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:

```hy
(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`:

```hy
(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:

```hy
(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:

```hy
(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.

```hy
=> (defmacro "#" [code]
  ... (setv op (get code -1) params (list (cut code -1)))
  ... (~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 Hyrule

Hyrule is Hy’s standard utility library. It provides a variety of functions and macros that are useful for writing Hy programs.

```
=> (import hyrule [inc])
=> (list (map inc [1 2 3]))
[2 3 4]
=> (require hyrule [assoc])
=> (setv d {}) 
=> (assoc d "a" 1 "b" 2)
=> d
{"a" 1  "b" 2}
```

### 2.7 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 documentation 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.
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.

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

(continues on next page)
We're trying to ignore them and you want to give them their own line? Hysterically ridiculous.

```lisp
(defun fib [n]
  ; My eyes!
  (if (<= n 2)
      n
      (+ (fib (- n 1)) (fib (- n 2)))
  )
)
```

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

```lisp
;; 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)

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

;; BAD
hy, Release 1.0a4+56.g15168ad0

```
`#{(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 Sequence.
(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!
```

(continues on next page)
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.
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.

```plaintext
;; 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.
```

(continues on next page)
[(foo)
    #_(bar)
    #_(baz)]

;; OK. Adding a final discarded element makes line comments safer.
[(foo)
    ;; (bar)
    ;; (baz)
    #_ /]

### 3.1.6 Snuggle

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

```hy
;; 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.

```hy
;; 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.
```

(continues on next page)
#{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]

;; 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!

;; PREFERRED
(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.
(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 #* body argument contain an implicit do.
- The body is never special, but the arguments before it are.

```haskell
;; 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
```

(continues on next page)
[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.

```hy
;; 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])

;; 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.
(_
    (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.

```hy
;; 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) ; 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"
        (> 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)))

;; 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 \texttt{fn}, \texttt{defclass}, at the top of the module, and in any other macros derived from these that can take a docstring (e.g. \texttt{defmacro/g!}, \texttt{defn}).

Docstrings contents follow the same conventions as Python.

The \texttt{(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.

\begin{verbatim}
;; 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 multiple lines.
(foofunction ;; Form comment about (get-arg1). Not a margin comment!
             (get-arg1)
             ;; Form comment about arg2. The indent matches.
             arg2)
\end{verbatim}

Indent form comments at the same level as the form they’re commenting about; they must always start with exactly two semicolons \texttt{;;}. 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 \texttt{;;} 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 \texttt{;}. 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))
      "\n"
      "")
      ","))
  )
);

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

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

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

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

;;; OK. Still clear enough with no threading macro this time.
(continues on next page)
(defn square? [x]
  (= x (pow (int (sqrt x)) 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.

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

(import json re)
(import numpy :as np
  pandas :as pd)
(imports 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

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

;;; PREFERRED
;;; Clearly subtraction.
```

(continues on next page)
(-= spam 2)
(- 100 7)

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

;;; PREFERRED
;;; 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 [#* 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)
4.1 Command Line Interface

4.1.1 hy

Command Line Options

- **c** `<command>`
  Execute the Hy code in `command`.

```bash
$ 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:

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

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

--**repl-output-fn**
Format REPL output using specific function (e.g., `repr`)
New in version 0.13.0.
-v
  Print the Hy version number and exit.

4.1.2 hyc

Command Line Options

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

```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 The Hy REPL

Hy’s REPL (read-eval-print loop)\(^1\) functionality is implemented in the hy.cmdline.HyREPL class. The HyREPL extends the Python Standard Library’s code.InteractiveConsole\(^2\) class. For more information about starting the REPL from the command line, see Command Line Interface. A REPL can also be instantiated programatically, by calling hy.cmdline.run_repl - see Launching a Hy REPL from Python.

From a high level, a single cycle of the REPL consists of the following steps:

---
\(^1\) [https://en.wikipedia.org/wiki/Read-eval-print_loop](https://en.wikipedia.org/wiki/Read-eval-print_loop)

\(^2\) [https://docs.python.org/3/library/code.html](https://docs.python.org/3/library/code.html)
4.2.1 REPL Built-ins

Recent Evaluation Results

The results of the three most recent evaluations can be obtained by entering *1 (most recent), *2, and *3. For example:

```plaintext
=> "first"
"first"
=> "second"
"second"
=> "third"
"third"
=> f"{*1},{*2},{*3}"
"third,second,first"
```

**Note:** The result of evaluating *i itself becomes the next most recent result, pushing *1 to *2, *2 to *3, and *3 off the cache.

Most Recent Exception

Once an exception has been thrown in an interactive session, the most recent exception can be obtained by entering *e. For example:

```plaintext
=> *e
Traceback (most recent call last):
  File "stdin-8d630e81640adf6e2670bb457a8234263247e875", line 1, in <module>
    *e
NameError: name 'hyx_XasteriskXe' is not defined
=> (/ 1 0)
Traceback (most recent call last):
  File "stdin-7b3ace8766f1e1cfb3ae7c01a1a61cebed24f482", line 1, in <module>
    (/ 1 0)
ZeroDivisionError: division by zero
=> *e
ZeroDivisionError('division by zero')
=> (type *e)
<class 'ZeroDivisionError'>
```

3 lexing
4 Compiling and Linking under Unix-like systems
4.2.2 Configuration

Location of history

The default location for the history of REPL inputs is ~/.hy-history. This can be changed by setting the environment variable HY_HISTORY to your preferred location. For example, if you are using Bash, it can be set with export HY_HISTORY=/path/to/my/.custom-hy-history.

Initialization Script

Similarly to python’s PYTHONSTARTUP environment variable, when HYSTARTUP is set, Hy will try to execute the file and import/require its defines into the repl namespace. This can be useful to set the repl sys.path and make certain macros and methods available in any Hy repl.

In addition, init scripts can set defaults for repl config values with:

- **repl-spy** (bool) print equivalent Python code before executing.
- **repl-output-fn** (callable) single argument function for printing REPL output.

Init scripts can do a number of other things like set banner messages or change the prompts. The following shows a number of possibilities:

```plaintext
;; Wrapping in an `eval-and-compile` ensures global python packages
;; are available in macros defined in this file as well.
(eval-and-compile
  (import sys os)
  (sys.path.append "/~<path-to-global-libs>"))

;; These modules, macros, and methods are now available in any Hy repl
(import
  re
  json
  pathlib [Path]
  hy.pypos *
  hyrule [pp pformat])

(require
  hyrule [unless])

(setv
  ;; Spy and output-fn will be set automatically for all hy repls
  repl-spy True
  repl-output-fn pformat
  ;; We can even add colors to the prompts. This will set `=>` to green and `...` to red.
  sys.ps1 "\x01\x1b[0;32m\x02=> \x01\x1b[0m\x02"
  sys.ps2 "\x01\x1b[0;31m\x02... \x01\x1b[0m\x02")

;; Functions and Macros will be available in the repl without qualification
(defn slurp [path]
  (setv path (Path path))
  (when (path.exists)
    (path.read-text)))
```

(continues on next page)
4.3 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.

4.3.1 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 The import statement it. If you have the following in `greetings.py` in Python:

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

You can use it in Hy:

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

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

4.3.2 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`:

```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:

```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 API, 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:

```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:

```python
>>> 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:

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

Then, use the hy.eval function to evaluate it:

```python
>>> hy.eval(expr)
38.0
```
4.4 Syntax

4.4.1 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.

4.4.2 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.

4.4.3 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 you [sic]" Tom wrote back."]) "That's very kind of you [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.
4.4.4 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.

```hy
=> (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:

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

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.

Hy’s f-strings are compatible with Python’s “=” debugging syntax, subject to the above limitations on delimiting identifiers. For example:

```hy
=> (setv foo "bar")
=> (print f"{foo = }")
foo = 'bar'
=> (print f"{foo = !s :^7}" )
foo = __bar__
```

4.4.5 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)`. It is important to note that a keyword argument cannot be a Python reserved word. This will raise a `SyntaxError` similar to Python. See `defn` for examples.

Keywords can be called like functions as shorthand for `get`. `(:foo obj)` is equivalent to `(get obj (hy.mangle "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`.

40 Chapter 4. Documentation Index
4.4.6 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 steps are as follows:

1. Remove any leading underscores (_). Leading underscores have special significance in Python, so we will mangle the remainder of the name and then add the leading underscores back in to the final mangled name.

2. Convert hyphens (-) to underscores (_). Thus, **foo-bar** becomes **foo_bar**. If the name at this step starts with a hyphen, this first hyphen is not converted, so that we don’t introduce a new leading underscore into the name. Thus **--has--dashes?** becomes **--has--dashes?** at this step.

3. If the name ends with ?, remove it and prepend is_. Thus, **tasty?** becomes **is_tasty** and **--has--dashes?** becomes **is--has--dashes**.

4. 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 leading hyphens and **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**, **if** becomes **hyx_if**, and **is--has--dashes** becomes **hyx_is_XhyphenHminusX_has--dashes**.

5. Finally, any leading underscores removed in the first step are added back to the mangled name. Thus, **(hy.mangle '_tasty?)** is **"is_tasty"** instead of **"is__tasty"** and **(hy.mangle '___has--dashes?)** is **"___has--dashes"**.

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 (non-initial) _ and __ are interchangeable in all symbol names, so you shouldn’t use, e.g., both **foo-bar** and **foo_bar** as separate variables.

4.4.7 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:

```hy
=> (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'
```
4.5 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.

4.5.1 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:

```hy
(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:

```hy
(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)])
```
which is conveniently utilized with an assignment such as \( \text{(setv [body except-clauses else-part finally-part] result)} \). Notice that \text{else-part} will be set to \text{None} because there is no \text{else} clause in the original form.

### 4.5.2 Usage

Model patterns are implemented as \text{funcparserlib} parser combinators. We won’t reproduce \text{funcparserlib}’s own documentation, but here are some important built-in parsers:

- \( (+ ...) \) matches its arguments in sequence.
- \( (\mid ...) \) matches any one of its arguments.
- \( (\gg \text{parser function}) \) matches \text{parser}, then feeds the result through \text{function} to change the value that’s produced on a successful parse.
- \( (\text{skip parser}) \) matches \text{parser}, but doesn’t add it to the produced value.
- \( (\text{maybe parser}) \) matches \text{parser} if possible. Otherwise, it produces the value \text{None}.
- \( (\text{some function}) \) takes a predicate \text{function} and matches a form if it satisfies the predicate.

The best reference for Hy’s parsers is the docstrings (use \text{(help hy.model-patterns)}), but again, here are some of the more important ones:

- \text{FORM} matches anything.
- \text{SYM} matches any symbol.
- \( (\text{sym "foo"}) \) or \( (\text{sym ":foo"}) \) matches and discards (per \text{skip}) the named symbol or keyword.
- \( (\text{brackets ...}) \) matches the arguments in square brackets.
- \( (\text{pexpr ...}) \) matches the arguments in parentheses.

Here’s how you could write a simple macro using model patterns:

```hy
(defmacro pairs [#* args]
  (import funcparserlib.parser [many])
  (import hy.model-patterns [whole SYM FORM])
  (setv [args] (.parse
    (whole [(many (+ SYM FORM))])
    args))
  `(~@(gfor [a1 a2] args (; (str a1) a2)))
)
```

(\text{print (pairs a 1 b 2 c 3))} ; => \text{[["a" 1] ["b" 2] ["c" 3]]}

A failed parse will raise \text{funcparserlib.parser.NoParseError}.  

4.5. Model Patterns 43
4.6 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.

4.6.1 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 Object Protocol.

Object

hy.models.Object 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.

Object 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.modelsPRETTY 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.

Sequence

hy.models.Sequence is the abstract base class of “iterable” Hy models, such as hy.models.Expression and hy.models.List.

Adding a Sequence 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.

Sequences are (mostly) immutable: you can’t add, modify, or remove elements. You can still append to a variable containing a Sequence with + and otherwise construct new Sequences out of old ones.
List

`hy.models.List` is a `Sequence Protocol` for bracketed `[]` lists, which, when used as a top-level expression, translate to Python list literals in the compilation phase.

Expression

`hy.models.Expression` inherits `Sequence Protocol` for parenthesized `()` expressions. The compilation result of those expressions depends on the first element of the list: the compiler dispatches expressions between macros and and regular Python function calls.

Dict

`hy.models.Dict` inherits `Sequence Protocol` 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 `String`.

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:

- `Integer`
- `Float`
- `Complex` (if the atom isn’t a bare `j`)
- `Keyword` (if the atom starts with `:`)
- `Symbol`

String

`hy.models.String` represents string literals (including bracket strings), which compile down to unicode string literals (`str`) in Python.

Strings 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.

Strings 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])`). Strings that are not produced by bracket strings have their `brackets` set to `None`.
Bytes

`hy.models.Bytes` is like `str`, but for sequences of bytes. It inherits from `bytes`.

Numeric Models

`hy.models.Integer` represents integer literals, using the `int` type.
`hy.models.Float` represents floating-point literals.
`hy.models.Complex` represents complex literals.
Numeric models are parsed using the corresponding Python routine, and valid numeric python literals will be turned into their Hy counterpart.

Symbol

`hy.models.Symbol` is the model used to represent symbols in the Hy language. Like `str`, it inherits from `str` (or `unicode` on Python 2).
Symbols are *mangled* when they are compiled to Python variable names.

Keyword

`hy.models.Keyword` represents keywords in Hy. Keywords are symbols starting with a `:`. See `keywords`.
The `.name` attribute of a `hy.models.Keyword` provides the *(unmangled)* string representation of the keyword without the initial `:`. For example:

```hy
=> (setv x :foo-bar)
=> (print x.name)
foo-bar
```

If needed, you can get the mangled name by calling `mangle`.

4.6.2 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:

```hy
(defmacro nif [expr pos-form zero-form neg-form]
  `(do
     (setv obscure-name ~expr)
     (cond (> obscure-name 0) ~pos-form
        (< obscure-name 0) ~neg-form
        (equalp obscure-name 0) ~zero-form))
```

(continues on next page)
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:

```lisp
(defmacro nif [expr pos-form zero-form neg-form]
  (setv g (hy.gensym))
  `(do
    (setv ~g ~expr)
    (cond (> ~g 0) ~pos-form
          (= ~g 0) ~zero-form
          (< ~g 0) ~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:

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

expands to:

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

so our re-written nif would look like:

```lisp
(defmacro nif [expr pos-form zero-form neg-form]
  (with-gensyms [g]
    `(do
      (setv ~g ~expr)
      (cond (> ~g 0) ~pos-form
            (= ~g 0) ~zero-form
            (< ~g 0) ~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 (hy.gensym "a").

Our final version of nif, built with defmacro/g! becomes:

```lisp
(defmacro/g! nif [expr pos-form zero-form neg-form]
  `(do
    (setv ~g!res ~expr)
    (cond (> ~g!res 0) ~pos-form
          (= ~g!res 0) ~zero-form
          (< ~g!res 0) ~neg-form)))
```
## 5.1 Hy

<table>
<thead>
<tr>
<th>IO</th>
<th>repr repr-register mangle unmangle read read-str</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reader</td>
<td>eval gensym macroexpand macroexpand-1 disassemble as-model</td>
</tr>
</tbody>
</table>
5.2 Core

<table>
<thead>
<tr>
<th>Meta</th>
<th>doc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Macros</td>
<td>when cond</td>
</tr>
<tr>
<td>Special Forms</td>
<td><code>^.</code> annotate fn fn/a defn defn/a defmacro if assert get global import eval-and-compile eval-when-compile await break chainc continue do for lfor dfors gfor sfor setv setx let match defclass del nonlocal py pys quasiquote quote require return cut raise try unpack-iterable/unpack-mapping unquote unquote-splice while with with/a with-decorator yield yield-from</td>
</tr>
<tr>
<td>Arithmetic</td>
<td><code>@ % + - * ** / //</code></td>
</tr>
<tr>
<td>Comparison</td>
<td>cond &lt;&gt; &lt;= &gt;= != is not? in not-in</td>
</tr>
<tr>
<td>Bitwise</td>
<td>`&lt;&lt; &gt;&gt; &amp;</td>
</tr>
<tr>
<td>Logic</td>
<td>not and or</td>
</tr>
</tbody>
</table>
6.1 Core Macros

The following macros are auto imported into all Hy modules as their base names, such that hy.core.macros.foo can be called as just foo.

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.

Syntax sugar for annotate.

Here is some example syntax of all three usages:

Examples

; 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 ^str [b None] ^int [c 1]] ...)

; Function return annotations come before the function name (if it exists)
(defn ^int add1 [^int x] (+ x 1))
(fn ^int [^int y] (+ y 2))

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.

For annotating items with generic types, the \texttt{of} macro will likely be of use.

\textbf{Note:} Since the addition of type annotations, identifiers that start with \texttt{^} are considered invalid as hy would try to read them as types.

\begin{verbatim}
\texttt{(annotate \(value\))}
\end{verbatim}

Expanded form of \texttt{^}. Syntactically equal to \texttt{^} and usable wherever you might use \texttt{^}:

\begin{verbatim}
\begin{verbatim}
(= \(^int \) '(\texttt{annotate int}))
\end{verbatim}
\end{verbatim}
True

\begin{verbatim}
(setv \(\texttt{annotate int}) \texttt{x 1})
\end{verbatim}

\begin{verbatim}
(defn \(\texttt{annotate int}) add1 \[(\texttt{annotate int}) \texttt{x}] (+ \texttt{x 1}))
\end{verbatim}

. 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.

\textbf{Examples}

\begin{verbatim}
(\. \texttt{foo} (\texttt{bar} "qux") \texttt{baz} [(+ 1 2)] \texttt{frob})
\end{verbatim}

Compiles down to:

\begin{verbatim}
\texttt{foo.bar}("qux").baz[1 + 2].frob
\end{verbatim}

. compiles its first argument (in the example, \texttt{foo}) as the object on which to do the attribute dereference. It uses bare symbols as attributes to access (in the example, \texttt{baz}, \texttt{frob}), Expressions as method calls (as in \texttt{bar}), 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 \texttt{AttributeError}. Access to unknown keys raises an \texttt{IndexError} (on lists and tuples) or a \texttt{KeyError} (on dictionaries).

\begin{verbatim}
(\texttt{fn} \texttt{name}, #* \texttt{args}))
\end{verbatim}

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

\begin{verbatim}
=> (setv \texttt{people} \{(\texttt{dict} :name "Alice" :age 20)
\ldots (\texttt{dict} :name "Bob" :age 25)
\ldots (\texttt{dict} :name "Charlie" :age 50)
\ldots (\texttt{dict} :name "Dave" :age 5)]

=> (defn \texttt{display-people} \texttt{[people filter]}
\ldots (for \texttt{[person people]} (\texttt{if} (\texttt{filter person}) (print (:name person))))))
\end{verbatim}
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:

```hy
=> (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:

```hy
=> (help times-three)
Help on function times_three:
times_three(x)
Multiplies input by three and returns result
(END)
```

(\texttt{fn/a})

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

(\texttt{defn})

Define \texttt{name} as a function with \texttt{args} as the signature, annotations, and body.

\texttt{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:

(\texttt{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 positional only arguments must precede \texttt{/}, all positional or keyword arguments must precede a \texttt{* rest parameter} or \texttt{kwonly delimiter} and \texttt{**} must be the last argument). This is the same order that Python requires.

/ \ The preceding parameters can only be supplied as positional arguments.

\textbf{positional or keyword arguments:} All parameters until following \texttt{/} (if its supplied) but before \texttt{*/#/**} can be supplied positionally or by keyword. Optional arguments may be given as two-argument lists, where the first element is the parameter name and the second is the default value. When defining parameters, a positional argument cannot follow a keyword argument.

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

```hy
=> (defn format-pair [left-val [right-val None] [open-text "\"" [close-text \""""]]
   ... (+ open-text (str left-val) "," (str right-val) close-text))
```
The following parameter will contain a list of 0 or more positional arguments. No other positional parameters may be specified after this one. Parameters defined after this but before ** are considered keyword only.

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 [#* numbers]
  (setv odd-numbers (lfor x numbers :if (% x 2) x)
  even-numbers (lfor x numbers :if (= (% x 2) 0) 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
```

- All following parameters can only be supplied as keywords. Like keyword arguments, the parameter may be marked as optional by declaring it as a two-element list containing the parameter name following by the default value. Parameters without a default are considered required:

```
=> (defn compare [a b * 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 (get s 0))) [x y]))))
-4
```

```
=> (compare "lisp" "python"
  ... :keyfn (fn [x y]
  ... (reduce - (map (fn [s] (ord (get s 0))) [x y]))
  ... :reverse True)
4
```

```
=> (compare "lisp" "python")
Traceback (most recent call last):
(continues on next page)"
Like #*, but for keyword arguments. 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:

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

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

; to avoid the mangling of '-' to '_', use unpacking:

```hy
=> (print-parameters **{"parameter-1" 1 "parameter-2" 2})
parameter-1 1
parameter-2 2
```

**Note:** Parameter names cannot be Python reserved words nor can a function be called with keyword arguments that are Python reserved words. This means that the following will raise a `SyntaxError` as they would in Python:

```hy
(defn afunc [a b if])
```

```hy
(dict :a 1 :from 2)
```

This only applies to parameter names and a keyword argument name. The value of the parameter or keyword argument can still be a keyword of a reserved word:

```hy
=> (defn test [a] a)
=> (test :a :from)
:from
```

`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:

**Examples:**
(defmacro (name, lambda-list, #* body))

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.

Examples:

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

=> (infix (1 + 1))
2

The name of the macro can be given as a string literal instead of a symbol. If the name starts with #, the macro can be called on a single argument without parentheses; such a macro is called a tag macro.

=> (defmacro "#x2" [form]
  ...
  (do ~form ~form))

=> (setv foo 1)
=> #x2 (+= foo 1)
=> foo
3

(if test then [else None])

Evaluate a test.

if respects Python truthiness, that is, a test fails if it evaluates to a “zero” (including values of nil, zero, None, and False), and passes otherwise, but values with a __bool__ method can override this.

if takes a test and then expression, plus an optional else expression at the end, which defaults to None. If no tests pass, if selects else.

Examples:

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

(away obj)

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

Examples

=> (import asyncio)
=> (defn/a main []
  ...
  (print "hello")
  ...
  (await (asyncio.sleep 1))
  ...
  (print "world"))
=> (asyncio.run (main))
(break())
break compiles to a break statement, which terminates the enclosing loop. The following example has an infinite while loop that ends when the user enters “k”:

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

In a loop with multiple iteration clauses, such as (for [x xs y ys] ...), break only breaks out of the innermost iteration, not the whole form. To jump out of the whole form, enclose it in a block and use block-ret instead of break, or enclose it in a function and use return.

(chainc(#* args))
chainc 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 chainc is to construct chains of heterogeneous operators, such as x <= y < z. It uses an infix syntax with the general form

(chainc ARG OP ARG OP ARG...)

Hence, (chainc 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 chainc 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 ARGs 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.

(continue())
continue compiles to a continue statement, which returns execution to the start of a loop. In the following example, (.append output x) is executed on each iteration, whereas (.append evens x) is only executed for even numbers.

(setv output [] evens [])
(for [x (range 10)]
    (.append output x)
    (when (% x 2)
        (continue))
    (.append evens x))

In a loop with multiple iteration clauses, such as (for [x xs y ys] ...), continue applies to the innermost iteration, not the whole form. To jump to the next step of an outer iteration, try rewriting your loop as multiple nested loops and interposing a block, as in (for [x xs] (block (for [y ys] ...))). You can then use block-ret in place of continue.

(do(#* body))
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.

Examples
Release 1.0a4+56.g15168ad0

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

(for (#* args))

`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
```

(assert (condition, [label None]))

`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:

**Examples**
(assert (= variable expected-value))

(assert False)
; AssertionError

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

(global (sym))

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.

Examples

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

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

(set-a 5)
(print-a)

(get (coll, key1, #* keys))

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:

Examples

=> (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.

(import (#* forms))

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

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

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

;;; Import with an alias
;;; Python: import sys as systest
(imports 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
(imports 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 *
(imports sys *)

(eval-and-compile(#*body*))

(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
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.

(eval-when-compile(*# body*))

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).

Examples

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

```
(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
```

(lfor(binding, iterable, *# body*))

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] [0 2] [1 0] [1 2] [2 0] [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.
• \texttt{:do FORM}, which simply evaluates the \texttt{FORM}. If you use \texttt{(continue)} or \texttt{(break)} here, they will apply to the innermost iteration clause before the \texttt{:do}.

• \texttt{:setv LVALUE RVALUE}, which is equivalent to \texttt{:do (setv LVALUE RVALUE)}.

• \texttt{:if CONDITION}, which is equivalent to \texttt{:do (if (not CONDITION) (continue))}.

For \texttt{lfor}, \texttt{sfor}, \texttt{gfor}, and \texttt{dfor}, variables defined by an iteration clause or \texttt{:setv} are not visible outside the form. However, variables defined within the body, such as via a \texttt{setx} expression, will be visible outside the form. By contrast, iteration and \texttt{:setv} clauses for \texttt{for} share the caller’s scope and are visible outside the form.

\begin{verbatim}
(dfor(binding, iterable, #* body))
\end{verbatim}

dfor creates a dictionary comprehension. Its syntax is the same as that of \texttt{:hyfunc:'for} 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.

\textbf{Examples}

\begin{verbatim}
=> (dfor x (range 5) [x (* x 10)])
{0 0 1 10 2 20 3 30 4 40}
\end{verbatim}

\begin{verbatim}
(gfor(binding, iterable, #* body))
\end{verbatim}

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

\textbf{Examples}

\begin{verbatim}
=> (import itertools [count take-while])
=> (list (take-while ...
... (* [x] (< x 5)))
... (gfor x (count) :do (.append accum x) x)))
 [0 1 2 3 4]
=> accum
 [0 1 2 3 4 5]
\end{verbatim}

\begin{verbatim}
(sfor(binding, iterable, #* body))
\end{verbatim}

sfor creates a set comprehension. \texttt{(sfor CLAUSES VALUE)} is equivalent to \texttt{(set (lfor CLAUSES VALUE))}. See \texttt{lfor}.

\begin{verbatim}
(setv(#* args))
\end{verbatim}

setv is used to bind a value, object, or function to a symbol.

\textbf{Examples}

\begin{verbatim}
=> (setv names ["Alice" "Bob" "Charlie"])
=> (print names)
[ 'Alice', 'Bob', 'Charlie']

=> (setv counter (fn [collection item] (.count collection item)))
=> (counter [1 2 3 4 5 2 3] 2)
2
\end{verbatim}

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

\begin{verbatim}
=> (setv x 1 y x x 2)
=> (print x y)
2 1
\end{verbatim}
You can perform parallel assignments or unpack the source value with square brackets and `unpack-iterable`:

```lisp
=> (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']
```

(SETX(#{* args}))
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.

Examples

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

(LET(bindings, #* body))
let creates lexically-scoped names for local variables. This form takes a list of binding pairs followed by a body which gets executed. A let-bound name ceases to refer to that local outside the let form, but arguments in nested functions and bindings in nested let forms can shadow these names.

Examples

```lisp
=> (let [x 5 ; creates new local bound names 'x and 'y
... y 6]
... (print x y)
... (let [x 7] ; new local and name binding that shadows 'x
... (print x y))
... (print x y)) ; 'x refers to the first local again
5 6
7 6
5 6
```

Let can also bind names using Python’s extended iterable unpacking syntax to destructure iterables:

```lisp
=> (let [[head #* tail] (, 0 1 2)]
... [head tail])
[0 [1 2]]
```

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, defn or defclass will assign the function or class in the Python scope, even if it shares the name of a let binding. To avoid this hoisting, use importlib.import_module, fn, or type (or whatever metaclass) instead.

Like the let* of many other Lisps, let executes the variable assignments one-by-one, in the order written:

```lisp
=> (let [x 5
... y (+ x 1)]
... (print x y))
5 6
```
Note that let-bound variables continue to exist in the surrounding Python scope. As such, let-bound objects may not be eligible for garbage collection as soon as the let ends. To ensure there are no references to let-bound objects as soon as possible, use `del` at the end of the `let`, or wrap the `let` in a function.

**match(subject, #* cases)**

The `match` form creates a match statement. It requires Python 3.10 or later. The first argument should be the subject, and any remaining arguments should be pairs of patterns and results. The `match` form returns the value of the corresponding result, or `None` if no case matched. For example:

```hy
=> (match (+ 1 1)
    ... 1 "one"
    ... 2 "two"
    ... 3 "three")
"two"
```

You can use `do` to build a complex result form. Patterns, as in Python match statements, are interpreted specially and can’t be arbitrary forms. Use `| ...` for OR patterns, `PATTERN :as NAME` for AS patterns, and syntax like the usual Hy syntax for literal, capture, value, sequence, mapping, and class patterns. Guards are specified with `:if FORM`. Here’s a more complex example:

```hy
=> (match (, 100 200)
    ... [100 300] "Case 1"
    ... [100 200] :if flag "Case 2"
    ... [900 y] f"Case 3, y: {y}"
    ... [100 (| 100 200) :as y] f"Case 4, y: {y}"
    ... _ "Case 5, I match anything!")
```

This will match case 2 if `flag` is true and case 4 otherwise.

`match` can also match against class instances by keyword (or positionally if its `__match_args__` attribute is defined, see pep 636):

```hy
=> (with-decorator
    ... dataclass
    ... (defclass Point [])
    ... (^int x)
    ... (^int y)))
=> (match (Point 1 2)
    ... (Point 1 x) :if (= (% x 2) 0) x
    2)
```

**defclass**(class-name, super-classes, #* body)**

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).

**Examples**

Note that let-bound variables continue to exist in the surrounding Python scope. As such, let-bound objects may not be eligible for garbage collection as soon as the let ends. To ensure there are no references to let-bound objects as soon as possible, use `del` at the end of the `let`, or wrap the `let` in a function.

**match(subject, #* cases)**

The `match` form creates a match statement. It requires Python 3.10 or later. The first argument should be the subject, and any remaining arguments should be pairs of patterns and results. The `match` form returns the value of the corresponding result, or `None` if no case matched. For example:

```hy
=> (match (+ 1 1)
    ... 1 "one"
    ... 2 "two"
    ... 3 "three")
"two"
```

You can use `do` to build a complex result form. Patterns, as in Python match statements, are interpreted specially and can’t be arbitrary forms. Use `| ...` for OR patterns, `PATTERN :as NAME` for AS patterns, and syntax like the usual Hy syntax for literal, capture, value, sequence, mapping, and class patterns. Guards are specified with `:if FORM`. Here’s a more complex example:

```hy
=> (match (, 100 200)
    ... [100 300] "Case 1"
    ... [100 200] :if flag "Case 2"
    ... [900 y] f"Case 3, y: {y}"
    ... [100 (| 100 200) :as y] f"Case 4, y: {y}"
    ... _ "Case 5, I match anything!")
```

This will match case 2 if `flag` is true and case 4 otherwise.

`match` can also match against class instances by keyword (or positionally if its `__match_args__` attribute is defined, see pep 636):

```hy
=> (with-decorator
    ... dataclass
    ... (defclass Point [])
    ... (^int x)
    ... (^int y)))
=> (match (Point 1 2)
    ... (Point 1 x) :if (= (% x 2) 0) x
    2)
```

**defclass**(class-name, super-classes, #* body)**

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

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

**del**(object)

New in version 0.9.12.

del removes an object from the current namespace.

**Examples**

```hy
=> (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.

```hy
=> (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
{}``

**nonlocal**(object)

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:
Examples

(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.

(py (string))

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, hy.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.

(pys (string))

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)

(quasiquote (form))

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.

Examples

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

(quote (form))

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

Examples
(require (#* args))

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:

Examples

(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:

```hy
(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.

---

**Note: 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 you try to do introspection of the current module’s set of defined macros, which isn’t really supported anyway.

---

**return(object)**

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

**Examples**

```hy
=> (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.

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

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

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

---

**cut(coll [start None] [stop None] [step None])**

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 only one is given, it is taken as the `stop` value. The third optional parameter is used to control the step stride 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:

**Examples**

```hy
=> (setv collection (range 10))
=> (cut collection)
```
hy, Release 1.0a4+56.g15168ad0

(continued from previous page)

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

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

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

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

\textbf{raise(\{(exception None)\})}

The \texttt{raise} form can be used to raise an \texttt{Exception} at runtime. Example usage:

**Examples**

\begin{verbatim}
(raise)
 ; re-raise the last exception

(raise IOError)
 ; raise an IOError

(raise (IOError "foobar"))
 ; raise an IOError("foobar")
\end{verbatim}

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

\textbf{try(#* body*)}

The \texttt{try} form is used to catch exceptions (\texttt{except}) and run cleanup actions (\texttt{finally}).

**Examples**

\begin{verbatim}
(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")))
\end{verbatim}

The first argument of \texttt{try} is its body, which can contain one or more forms. Then comes any number of \texttt{except} clauses, then optionally an \texttt{else} clause, then optionally a \texttt{finally} clause. If an exception is raised with a matching \texttt{except} clause during the execution of the body, that \texttt{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.

**unpack-itable/unpack-mapping**
(Also known as the splat operator, star operator, argument expansion, argument explosion, argument gathering, and varargs, among others...)

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

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

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

```scheme
=> (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).

```scheme
=> (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]
```

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

```scheme
=> (setv nickname "Cuddles")
=> (quasiquote (= nickname (unquote nickname)))
'(= nickname "Cuddles")
=> `(= nickname ~nickname)
'(= nickname "Cuddles")
```

**(unquote-splice(symbol))**
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.

```scheme
=> (setv nums [1 2 3 4])
=> (quasiquote (+ (unquote-splice nums)))
'(+ 1 2 3 4)
=> `'(+ ~@nums)
'(+ 1 2 3 4)
```
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).

(while (condition, #* body))

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

(with (#* args))

Wrap execution of body within a context manager given as bracket args. 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 archetypal example of using `with` is when processing files. If only a single expression
is supplied, or the argument is `_`, then no variable is bound to the expression, as shown below.

Examples:

```lisp
=> (with [arg (expr)] block)
=> (with [(expr)] block)
=> (with [arg1 (expr1) _ (expr2) arg3 (expr3)] 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:

```lisp
=> (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:

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

(with/a(##* args))

Wrap execution of `body` within a context manager given as bracket `args`. `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. Examples:

```lisp
:: => (with/a [arg (expr)] block) => (with/a [(expr)] block) => (with/a [_ (expr) arg (expr) _ (expr)]
    block)
```

**Note:** `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`.

(with-decorator(##* args))

`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, i.e. the first decorator will be
the outermost one, and so on. Decorators with arguments are called just like a function call.

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

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

(continues on next page)
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.

Examples

```hy
=> (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 17 14 23 23 19]
```

(yield-from(object))

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.

macro(hy.core.macros.#@(expr))

with-decorator tag macro

macro(hy.core.macros.cond(#* args))

Shorthand for a nested sequence of if forms, like an if-elif-else ladder in Python. Syntax such as

```hy
(cond
  condition1 result1
  condition2 result2)
```

is equivalent to

```hy
(if condition1
  result1
```

(continues on next page)
(if condition2
    result2
    None))

Notice that None is returned when no conditions match; use True as the final condition to change the fallback result. Use do to execute several forms as part of a single condition or result.

With no arguments, cond returns None. With an odd number of arguments, cond raises an error.

macro(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 (help foo) instead for help with runtime objects.

macro(hy.core.macros.when(test, #* body))

 Execute body when test is true

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.

Examples

=> (when conditional statement)
    (if conditional (do statement))

6.1.1 Placeholder macros

There are a few core macros that are unusual in that all they do, when expanded, is crash, regardless of their arguments:

• else
• except
• finally
• unpack-mapping
• unquote
• unquote-splice

The purpose of these macros is merely to reserve their names. Each symbol is interpreted specially by one or more other core macros (e.g., else in while) and thus, in these contexts, any definition of these names as a function or macro would be ignored. If you really want to, you can override these names like any others, but beware that, for example, trying to call your new else inside while may not work.
6.2 Hy

The hy module is auto imported into every Hy module and provides convenient access to the following methods:

- (hy.read-str (input))
  This is essentially a wrapper around hy.read which reads expressions from a string.

**Examples**

```hy
=> (hy.read-str "(print 1)"
' (print 1)

=> (hy.eval (hy.read-str "(print 1)"))
1
```

- (import io)

**Examples**

```hy
=> (import io)
=> (setv buffer (io.StringIO "(+ 2 2)\n(- 2 1)")
=> (hy.eval (hy.read :from-file buffer))
4
=> (hy.eval (hy.read :from-file buffer))
1
```

```hy
=> (with [f (open "example.hy" "w")]
  (.write f "(print 'hello)\n(print "hyfriends!")")
35
=> (with [f (open "example.hy")]
  (try (while True
    ... (setv exp (hy.read f))
    ... (print "OHY" exp)
    ... (hy.eval exp))
    ... (except [e EOFError]
    ... (print "EOF!")))
OHY hy.models.Expression([...
```

(continues on next page)
hy.models.Symbol('print'),
hy.models.Expression([
    hy.models.Symbol('quote'),
    hy.models.Symbol('hello')]
))

hello
OHy hy.models.Expression([
    hy.models.Symbol('print'),
    hy.models.String('hyfriends!')
])
hyfriends!
EOF!

(hy.eval(hytree, locals, module, ast-callback, compiler, filename, source, [import-stdlib True]))

Evaluates a quoted expression and returns the value.

If you're evaluating hand-crafted AST trees, make sure the line numbers are set properly. Try fix_missing_locations and related functions in the Python ast library.

Examples

=> (hy.eval '(print "Hello World")')
"Hello World"

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

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

Parameters

- **hytree** (Object) – The Hy AST object to evaluate.
- **locals** (Optional[dict]) – Local environment in which to evaluate the Hy tree. Defaults to the calling frame.
- **module** (Optional[Union[str, types.ModuleType]]) – Module, or name of the module, to which the Hy tree is assigned and the global values are taken. The module associated with compiler takes priority over this value. When neither module nor compiler is specified, the calling frame's module is used.
- **ast_callback** (Optional[Callable]) – A callback that is passed the Hy compiled tree and resulting expression object, in that order, after compilation but before evaluation.
- **compiler** (Optional[HyASTCompiler]) – An existing Hy compiler to use for compilation. Also serves as the module value when given.
- **filename** (Optional[str]) – The filename corresponding to the source for tree. This will be overridden by the filename field of tree, if any; otherwise, it defaults to "<string>". When compiler is given, its filename field value is always used.
- **source** (Optional[str]) – A string containing the source code for tree. This will be overridden by the source field of tree, if any; otherwise, if None, an attempt will be made to obtain it from the module given by module. When compiler is given, its source field value is always used.

Returns Result of evaluating the Hy compiled tree.
Return type  Any

(hy.repr(obj))
This function is Hy’s equivalent of Python’s built-in repr. It returns a string representing the input object in Hy syntax.

Like repr in Python, hy.repr can round-trip many kinds of values. Round-tripping implies that given an object x, (hy.eval (hy.read-str (hy.repr x))) returns x, or at least a value that’s equal to x.

Examples

=> hy.repr [1 2 3])
"[1 2 3]
=> (repr [1 2 3])
"[1, 2, 3]"

(hy.repr-register(types,f,placeholder))
hy.repr-register lets you set the function that hy.repr calls to represent a type.

Examples

=> (hy.repr-register the-type fun)
=> (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` 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))
'(Container HY THERE)"

(hy.mangle(s))
Stringify the argument and convert it to a valid Python identifier according to Hy’s mangling rules.
Examples

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

=> (hy.mangle 'foo-bar?)
"is_foo_bar"

=> (hy.mangle '*)
"hyx_XasteriskX"

=> (hy.mangle '_foo/a?)
"_hyx_is_fooXsolidusXa"

=> (hy.mangle '-->)
"hyx_XhyphenHminusX_XgreaterHthan_signX"

=> (hy.mangle '<--)
"hyx_XlessHthan_signX__"
```

(hy.unmangle(\(s\)))
Stringify the argument and try to convert it to a pretty unmangled form. This may not round-trip, because different Hy symbol names can mangle to the same Python identifier. See Hy's mangling rules.

Examples

```lisp
=> (hy.unmangle 'foo-bar)
"foo-bar"

=> (hy.unmangle 'is_foo_bar)
"foo-bar?"

=> (hy.unmangle 'hyx_XasteriskX)
"*"

=> (hy.unmangle '_hyx_is_fooXsolidusXa)
"_foo/a?"

=> (hy.unmangle 'hyx_XhyphenHminusX_XgreaterHthan_signX)
"-->" 

=> (hy.unmangle 'hyx_XlessHthan_signX__)
"<--"

=> (hy.unmangle '__dunder_name__')
"__dunder-name__"
```

(hy.disassemble(tree, [codegen False]))
Return the python AST for a quoted Hy `tree` as a string.

If the second argument `codegen` is true, generate python code instead.

New in version 0.10.0.
Dump the Python AST for given Hy tree to standard output. If codegen is True, the function prints Python code instead.

**Examples**

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

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

**hy.macroexpand**(form, [result-ok False])

Return the full macro expansion of form.

New in version 0.10.0.

**Examples**

```
=> (require hyrule [->])
=> (hy.macroexpand '(- (a b) (x y)))
'((x (a b)) y)
```

```
=> (hy.macroexpand-1 '(- (a b) (- (c d) (e f))))
'(e (c (a b) d) f)
```

**hy.macroexpand-1**(form)

Return the single step macro expansion of form.

New in version 0.10.0.

**Examples**

```
=> (require hyrule [->])
=> (hy.macroexpand-1 '(- (a b) (- (c d) (e f))))
'(-> (a b) (c d) (e f))
```

**hy.gensym**(G)

Generate a symbol with a unique name. The argument will be included in the generated symbol, as an aid to debugging. Typically one calls hy.gensym without an argument.

New in version 0.9.12.

**See also:**

Section Using gensym for Safer Macros

The below example uses the return value of f twice but calls it only once, and uses hy.gensym for the temporary variable to avoid collisions with any other variable names.
(defmacro selfadd [x]
(setv g (hy.gensym))
'(do
  (setv ~g ~x)
  (+ ~g ~g)))

(defn f []
  (print "This is only executed once."))
4)
(print (selfadd f)))

(hy.as-model(x))
Recursively promote an object x into its canonical model form.

When creating macros it's possible to return non-Hy model objects or even create an expression with non-Hy model elements:

=> (defmacro hello [] ...
   "world!")

=> (defmacro print-inc [a] ...
   `(print ~(+ a 1)))

=> (print-inc 1)
2 ; in this case the unquote form (+ 1 1) would splice the literal
   ; integer `2` into the print statement, `not` the model representation
   ; ``(hy.model.Integer 2)``

This is perfectly fine, because Hy autoboxes these literal values into their respective model forms at compilation time.

The one case where this distinction between the spliced composite form and the canonical model tree representation matters, is when comparing some spliced model tree with another known tree:

=> (= `(print ~(+ 1 1)) `(print 2))
False ; False because the literal int `2` in the spliced form is not
   ; equal to the ``(hy.model.Integer 2)`` value in the known form.

=> (= (hy.as-model `(print ~(+ 1 1)) `(print 2)))
True ; True because `as-model` has walked the expression and promoted
   ; the literal int `2` to its model for ``(hy.model.Integer 2)``

### 6.3 Python Operators

Python provides various binary and unary operators. These are usually invoked in Hy using core macros of the same name: for example, (+ 1 2) calls the core macro named +, which uses Python's addition operator. An exception to the names being the same is that Python's == is called = in Hy.

By importing from the module hy.pyops (typically with a star import, as in (import hy.pyops *)), you can also use these operators as functions. Functions are first-class objects, so you can say things like (map - xs) to negate all the numbers in the list xs. Since macros shadow functions, forms like (~ 1 2) will still call the macro instead of the function.
The functions in `hy.pyops` have the same semantics as their macro equivalents, with one exception: functions can’t short-circuit, so the functions for the logical operators, such as `and`, unconditionally evaluate all arguments.

```python
(hy.pyops.!= (a1, a2, #* a-rest))
The inequality operator. Its effect can be defined by the equivalent Python:

• `(!= x y) → x != y`
• `(!= a1 a2 ... an) → a1 != a2 != ... != an`
```

```python
(hy.pyops.% (x, y))
The modulus operator. Its effect can be defined by the equivalent Python:

• `(% x y) → x % y`
```

```python
(hy.pyops.& (a1, #* a-rest))
The bitwise AND operator. Its effect can be defined by the equivalent Python:

• `(& x) → x`
• `(& x y) → x & y`
• `(& a1 a2 ... an) → a1 & a2 & ... & an`
```

```python
(hy.pyops.**(a1, a2, #* a-rest))
The exponentiation operator. Its effect can be defined by the equivalent Python:

• `(** x y) → x ** y`
• `(** a1 a2 ... an) → a1 ** a2 ** ... ** an`
```

```python
(hy.pyops.* (#* args))
The multiplication operator. Its effect can be defined by the equivalent Python:

• `(*) → 0`
• `(* x) → x`
• `(* x y) → x * y`
• `(* a1 a2 ... an) → a1 * a2 * ... * an`
```

```python
(hy.pyops.+/ (a1, #* a-rest))
The addition operator. Its effect can be defined by the equivalent Python:

• `(+) → 0`
• `(+ x) → +x`
• `(+ x y) → x + y`
• `(+ a1 a2 ... an) → a1 + a2 + ... + an`
```

```python
(hy.pyops.-/ (a1, #* a-rest))
The subtraction operator. Its effect can be defined by the equivalent Python:

• `(- x) → -x`
• `(- x y) → x - y`
• `(- a1 a2 ... an) → a1 - a2 - ... - an`
```

```python
(hy.pyops./ (a1, #* a-rest))
The division operator. Its effect can be defined by the equivalent Python:

• `(/ x) → 1 / x`
• `(/ x y) → x / y`
```
• (/\ a1 a2 ... an) \rightarrow a1 / a2 / ... / an

\texttt{(\texttt{hy.pyops.} // (al, a2, \#* a-rest))}

The floor division operator. Its effect can be defined by the equivalent Python:

• (// x y) \rightarrow x // y
• (// a1 a2 ... an) \rightarrow a1 // a2 // ... // an

\texttt{(\texttt{hy.pyops.} <(al, \#* a-rest))}

The less-than operator. Its effect can be defined by the equivalent Python:

• (< x) \rightarrow \text{True}
• (< x y) \rightarrow x < y
• (< a1 a2 ... an) \rightarrow a1 < a2 < ... < an

\texttt{(\texttt{hy.pyops.} <<<(al, a2, \#* a-rest))}

The left shift operator. Its effect can be defined by the equivalent Python:

• (<< x y) \rightarrow x << y
• (<< a1 a2 ... an) \rightarrow a1 << a2 << ... << an

\texttt{(\texttt{hy.pyops.} <=(al, \#* a-rest))}

The less-than-or-equal-to operator. Its effect can be defined by the equivalent Python:

• (<= x) \rightarrow \text{True}
• (<= x y) \rightarrow x \leq y
• (<= a1 a2 ... an) \rightarrow a1 \leq a2 \leq ... \leq an

\texttt{(\texttt{hy.pyops.} ==(al, \#* a-rest))}

The equality operator. Its effect can be defined by the equivalent Python:

• (= x) \rightarrow \text{True}
• (= x y) \rightarrow x == y
• (= a1 a2 ... an) \rightarrow a1 == a2 == ... == an

\texttt{(\texttt{hy.pyops.} >>(al, \#* a-rest))}

The greater-than operator. Its effect can be defined by the equivalent Python:

• (> x) \rightarrow \text{True}
• (> x y) \rightarrow x > y
• (> a1 a2 ... an) \rightarrow a1 > a2 > ... > an

\texttt{(\texttt{hy.pyops.} >==(al, \#* a-rest))}

The greater-than-or-equal-to operator. Its effect can be defined by the equivalent Python:

• (>= x) \rightarrow \text{True}
• (>= x y) \rightarrow x >= y
• (>= a1 a2 ... an) \rightarrow a1 >= a2 >= ... >= an

\texttt{(\texttt{hy.pyops.} >>=)(al, a2, \#* a-rest))}

The right shift operator. Its effect can be defined by the equivalent Python:

• (>> x y) \rightarrow x >> y
• (>> a1 a2 ... an) \rightarrow a1 >> a2 >> ... >> an
(hy.pyops.@(a1, #=> (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.

Examples

6.3. Python Operators 83
\begin{itemize}
    \item \texttt{(is x)} \rightarrow \texttt{True}
    \item \texttt{(is x y)} \rightarrow \texttt{x is y}
    \item \texttt{(is a1 a2 ... an)} \rightarrow \texttt{a1 is a2 is ... is an}
\end{itemize}

\texttt{(hy.pyops.not-in(a1, a2, \#* a-rest))}

The negated membership test operator. Its effect can be defined by the equivalent Python:
\begin{itemize}
    \item \texttt{(not-in x y)} \rightarrow \texttt{x not in y}
    \item \texttt{(not-in a1 a2 ... an)} \rightarrow \texttt{a1 not in a2 not in ... not in an}
\end{itemize}

\texttt{(hy.pyops.not?(a1, a2, \#* a-rest))}

The negated identicality test operator. Its effect can be defined by the equivalent Python:
\begin{itemize}
    \item \texttt{(is-not x y)} \rightarrow \texttt{x is not y}
    \item \texttt{(is-not a1 a2 ... an)} \rightarrow \texttt{a1 is not a2 is not ... is not an}
\end{itemize}

\texttt{(hy.pyops.or(#* args))}

The logical disjunction operator. Its effect can be defined by the equivalent Python:
\begin{itemize}
    \item \texttt{(or)} \rightarrow \texttt{None}
    \item \texttt{(or x)} \rightarrow \texttt{x}
    \item \texttt{(or x y)} \rightarrow \texttt{x or y}
    \item \texttt{(or a1 a2 ... an)} \rightarrow \texttt{a1 or a2 or ... or an}
\end{itemize}

\texttt{(hy.pyops.reduce())}

\begin{verbatim}
reduce(function, sequence[, initial]) -> value
Apply a function of two arguments cumulatively to the items of a sequence, from left to right, so as to reduce the
sequence to a single value. For example, reduce(lambda x, y: x+y, [1, 2, 3, 4, 5]) calculates (((1+2)+3)+4)+5).
If initial is present, it is placed before the items of the sequence in the calculation, and serves as a default when
the sequence is empty.
\end{verbatim}

\texttt{(hy.pyops.|(#* args))}

The bitwise OR operator. Its effect can be defined by the equivalent Python:
\begin{itemize}
    \item \texttt{(|)} \rightarrow \texttt{0}
    \item \texttt{(| x)} \rightarrow \texttt{x}
    \item \texttt{(| x y)} \rightarrow \texttt{x | y}
    \item \texttt{(| a1 a2 ... an)} \rightarrow \texttt{a1 | a2 | ... | an}
\end{itemize}

\texttt{(hy.pyops.~(x))}

The bitwise NOT operator. Its effect can be defined by the equivalent Python:
\begin{itemize}
    \item \texttt{(~ x)} \rightarrow \texttt{~x}
\end{itemize}
6.4 Reserved

(hy.reserved.macros())
   Return a frozenset of Hy's core macro names.

(hy.reserved.names())
   Return a frozenset of reserved symbol names.

   The result of the first call is cached.

   The output includes all of Hy's core functions and macros, plus all Python reserved words. All names are in unmangled form (e.g., not-in rather than not_in).

Examples

=> (import hy.extra.reserved)
=> (in "defclass" (hy.extra.reserved.names))
True
7.1 Join our Hyve!

Please come hack on Hy!
Please come hang out with us on the Github Discussions page!
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 development requirements:

   $ pip install -r requirements-dev.txt

5. Optionally, enable the pre-commit hooks defined in .pre-commit-config.yaml:

   $ pre-commit install

   This will ensure your code adheres to the formatting conventions enforced via continuous integration (CI).

6. Optionally, tell git blame to ignore the commits listed in .git-blame-ignore-revs:

   $ git config blame.ignoreRevsFile .git-blame-ignore-revs

   This file is intended to contain commits with large diffs but negligible semantic changes.

7. 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
   $ sphinx-build . _build -b 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 Github Discussions page 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.
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.

Don't commit comments tagged with things like “FIXME”, “TODO”, or “XXX”. Ideas for how the code or documentation should change go in the issues list, not the code or documentation itself.

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.

Code formatting

All Python source code (.py) should be formatted with black and isort. This can be accomplished by running black hy tests and isort hy tests from the root of this repository. Formatting of Python files is checked automatically via GitHub Actions for all pull requests. No PR may be merged if it fails that check.

Testing

Tests can be run by executing pytest in the root of this repository.

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. 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 three days 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. The typical workflow for this is to run the following commands on your own machine, then press the merge button on GitHub.

```
$ git checkout master
$ git pull
$ git checkout $PR_BRANCH
$ git fetch
$ git reset --hard $REMOTE/$PR_BRANCH
$ git rebase master
$ git push -f
```

### 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](http://contributor-covenant.org/version/1/1/0/), available at [http://contributor-covenant.org/version/1/1/0/](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
HY MODULE INDEX

h
hy.core.macros, 73
hy.pyops, 80
hy.reserved, 85
Symbols

!=() (in module hy.pyops), 81
*() (in module hy.pyops), 81
**() (in module hy.pyops), 81
+() (in module hy.pyops), 81
. (built-in variable), 52
/() (in module hy.pyops), 81
//() (in module hy.pyops), 82
=() (in module hy.pyops), 81
@() (in module hy.pyops), 82
#@() (in module hy.core.macros), 73
%(() (in module hy.pyops), 81
&() (in module hy.pyops), 81
-() (in module hy.pyops), 81
--repl-output-fn
   command line option, 33
--spy
   command line option, 33
--with-ast
   command line option, 34
--with-source
   command line option, 34
--without-python
   command line option, 34
-a
   command line option, 34
-c
   command line option, 33
-i
   command line option, 33
-m
   command line option, 33
-np
   command line option, 34
-s
   command line option, 34
-v
   command line option, 33
^ (built-in variable), 51
~() (in module hy.pyops), 84
|() (in module hy.pyops), 84
>() (in module hy.pyops), 82

>=() (in module hy.pyops), 82
>>() (in module hy.pyops), 82
<() (in module hy.pyops), 82
<=() (in module hy.pyops), 82
<<() (in module hy.pyops), 82
[x y]() (in module hy.pyops), 83

A

and() (in module hy.pyops), 83
annotate()
   built-in function, 52
as-model() (in module hy), 80
assert()
   built-in function, 58
await()
   built-in function, 56

B

break()
   built-in function, 57
built-in function
   annotate(), 52
   assert(), 58
   await(), 56
   break(), 57
   chainc(), 57
   continue(), 57
   cut(), 68
   defclass(), 64
   defmacro(), 56
   defn(), 53
   defn/a(), 55
   del(), 65
   dfor(), 62
   do(), 57
   eval-and-compile(), 60
   eval-when-compile(), 61
   fn(), 52
   fn/a(), 53
   for(), 58
   get(), 59
   gfor(), 62
global(), 59
if(), 56
import(), 59
let(), 63
lfor(), 61
match(), 64
nonlocal(), 65
py(), 66
pys(), 66
quasiquote(), 66
quote(), 66
raise(), 69
require(), 67
return(), 68
setv(), 62
setx(), 63
sfor(), 62
try(), 69
unquote(), 70
unquote-splice(), 70
while(), 71
with(), 71
with/a(), 72
with-decorator(), 72
yield(), 73
yield-from(), 73

C
chainc()  built-in function, 57
command line option
   --repl-output-fn, 33
   --spy, 33
   --with-ast, 34
   --with-source, 34
   --without-python, 34
   -a, 34
   -c, 33
   -i, 33
   -m, 33
   -np, 34
   -s, 34
   -v, 33
fileN], 34
file[], 34
comp-op() (in module hy.pyops), 83
cond() (in module hy.core.macros), 73
continue() built-in function, 57
cut() built-in function, 68

D
defclass()
yield-from()
    built-in function, 73