## 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 8
   2.3 Basic operations 9
   2.4 Functions, classes, and modules 10
   2.5 Macros 12
   2.6 Hyrule 13
   2.7 Next steps 13

3 Syntax 15
   3.1 An introduction to models 15
   3.2 Non-form syntactic elements 16
      3.2.1 Shebang 16
      3.2.2 Whitespace 16
      3.2.3 Comments 16
      3.2.4 Discard prefix 16
   3.3 Identifiers 17
      3.3.1 Numeric literals 17
      3.3.2 Keywords 17
      3.3.3 Symbols 18
      3.3.4 Mangling 18
   3.4 String literals 19
      3.4.1 Bracket strings 19
   3.5 Sequential forms 20
      3.5.1 Expressions 20
      3.5.2 List, tuple, and set literals 20
      3.5.3 Dictionary literals 21
      3.5.4 Format strings 21
   3.6 Additional sugar 22
   3.7 Reader macros 22

4 Semantics 23
   4.1 Implicit names 23
   4.2 Order of evaluation 23
   4.3 When bytecode is regenerated 24
5 Macros
5.1 Using gensym for Safer Macros ........................................ 25

6 The Hy REPL
6.1 Output functions .................................................................. 27
6.2 Special variables .................................................................. 27
6.3 Startup files ........................................................................ 27

7 Environment Variables .......................................................... 29

8 Command Line Interface
8.1 hy ................................................................................. 31
8.1.1 Command Line Options ................................................... 31
8.2 hy2py ............................................................................. 32
8.3 hyc .................................................................................. 32

9 Hy <-> Python interop
9.1 Using Python from Hy .......................................................... 33
9.2 Using Hy from Python .......................................................... 33
9.2.1 Compiled files ................................................................ 34
9.2.2 Launching a Hy REPL from Python ................................. 34
9.2.3 Evaluating strings of Hy code from Python ....................... 35
9.3 Libraries that expect Python .................................................. 35

10 Model Patterns
10.1 A motivating example ........................................................... 37
10.2 Usage .............................................................................. 38

11 Cheatsheet
11.1 Hy ................................................................................. 41
11.2 Core .............................................................................. 42

12 API
12.1 Core Macros ..................................................................... 43
12.1.1 Placeholder macros ........................................................ 65
12.2 Hy .................................................................................. 66
12.3 Reader Macros ................................................................. 71
12.4 Python Operators .............................................................. 74
12.5 Reserved ........................................................................ 78

13 Hacking on Hy
13.1 Join our Hyve! .................................................................. 79
13.2 Hack! ............................................................................... 79
13.3 Test! ............................................................................... 80
13.4 Document! ...................................................................... 80
13.5 Contributor Guidelines ....................................................... 81
13.5.1 Issues ........................................................................ 81
13.5.2 Pull requests ............................................................... 81
13.6 Contributor Code of Conduct ............................................ 83
13.7 Core Team ..................................................................... 83

Hy Module Index ........................................................................ 85

Index ......................................................................................... 87
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 release of Hy, just use the command `pip3 install --user hy`. Then you can start an interactive read-eval-print loop (REPL) with the command `hy`, or run a Hy program with `hy myprogram.hy`. 
Hy (or “Hylang” for long; named after the insect order Hymenoptera, since Paul Tagliamonte was studying swarm behavior when he created the language) 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 punctuation 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.

```lisp
(defmacro do-while [condition #* body]
  `(do
     (while ~condition
      ~(body))))

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

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

```python
with open("foo") as o:
  f1 = o.read()
with open("bar") as o:
  f2 = o.read()
print(len(f1) + len(f2))
```
In Hy, `with` returns the value of its last body form, so you can use it like an ordinary function call:

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

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

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

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

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

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

Here’s a more complex example:

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

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

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

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

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

The basic unit of Lisp syntax, which is similar to a C or Python expression, is the form. 92, *, and (* 92 2) are all forms. A Lisp program consists of a sequence of forms nested within forms. Forms are typically separated from each other by whitespace, but some forms, such as string literals ("Hy, world!"), can contain whitespace themselves. An expression is a form enclosed in parentheses; its first child form, called the head, determines what the expression does, and should generally be a function 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.

```
(setv password "susan") ; My daughter's name
```

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

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

## 2.2 Literals

Hy has literal syntax for all of the same data types that Python does. Here’s an example of Hy code for each type and the Python equivalent.
<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: 4}</td>
<td>dict</td>
</tr>
</tbody>
</table>

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`:

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

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

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

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

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

Conditional logic can be built with `if`:

```
(if (= 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.
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:

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

For branching on more than one case, try cond:

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

The macro (when CONDITION THEN-1 THEN-2 ...) is shorthand for (if CONDITION (do THEN-1 THEN-2 ...) None).

Hy’s basic loops are while and for:

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

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

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

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

### 2.4 Functions, classes, and modules

Define named functions with defn:

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

Define anonymous functions with fn:
``` Scheme
(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:

``` Scheme
(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`:

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

Note that unlike Python, Hy doesn’t always evaluate function arguments (or the items in a literal list, or the items in a literal dictionary, etc.) in the order they appear in the code. But you can always force a particular evaluation order with `do`, or with other macros that provide an implicit `do`, like `when` or `fn`.

Define classes with `defclass`:

``` Scheme
(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:

``` Scheme
(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`:

``` Scheme
(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 `quote` and `unquote` to define a new control construct `do-while`.

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`.

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

Hy also supports reader macros, which are similar to ordinary macros, but operate on raw source text rather than pre-parsed Hy forms. They can choose how much of the source code to consume after the point they are called, and return any code. Thus, reader macros can add entirely new syntax to Hy. For example, you could add a literal notation for Python’s `decimal.Decimal` class like so:
=> (import  decimal  [Decimal]  fractions  [Fraction])
=> (defreader d

...  (.slurp-space &reader)
...  `(Decimal ~(.read-ident &reader)))
=> (print (repr #d .1))
Decimal('0.1')
=> (print (Fraction #d .1))
1/10
=> ;; Contrast with the normal floating-point .1:
=> (print (Fraction .1))
3602879701896397/36028797018963968

require can pull in a reader macro defined in a different module with syntax like (require mymodule :readers [d]).

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.

Bear in mind that Hy is still unstable, and with each release along the way to Hy 1.0, there are new breaking changes. Refer to the NEWS file for how to update your code when you upgrade Hy, and be sure you’re reading the version of this manual (shown at the top of each page) that matches the version of Hy you’re running.
This chapter describes how Hy source code is understood at the level of text, as well as the abstract syntax objects that the reader (a.k.a. the parser) turns text into, as when invoked with `hy.read`. The basic units of syntax at the textual level are called **forms**, and the basic objects representing forms are called **models**.

Following Python, Hy is in general case-sensitive. For example, `foo` and `FOO` are different symbols, and the Python-level variables they refer to are also different.

### 3.1 An introduction to models

Reading a Hy program produces a nested structure of model objects. Models can be very similar to the kind of value they represent (such as `Integer`, which is a subclass of `int` or they can be somewhat different (such as `Set`, which is ordered, unlike actual `set`s). All models inherit from `Object`, which stores textual position information, so tracebacks can point to the right place in the code. The compiler takes whatever models are left over after parsing and macro expansion and translates them into Python `ast` nodes (e.g., `Integer` becomes `ast.Constant`), which can then be evaluated or rendered as Python code. Macros (that is, regular macros, as opposed to reader macros) operate on the model level, taking some models as arguments and returning more models for compilation or further macro expansion; they’re free to do quite different things with a given model than the compiler does, if it pleases them to, like using an `Integer` to construct a `Symbol`.

In general, a model doesn’t count as equal to the value it represents. For example, `=(hy.models.String "foo")"foo"` returns `False`. But you can promote a value to its corresponding model with `hy.as-model`, or you can demote a model with the usual Python constructors like `str` or `int`, or you can evaluate a model as Hy code with `hy.eval`.

Models can be created with the constructors, with the `quote` or `quasiquote` macros, or with `hy.as-model`. Explicit creation is often not necessary, because the compiler will autopromote (via `hy.as-model`) any object it’s trying to evaluate.

Note that when you want plain old data structures and don’t intend to produce runnable Hy source code, you’ll usually be better off using Python’s basic data structures (`tuple`, `list`, `dict`, etc.) than models. Yes, “homoiconicity” is a fun word, but a Hy `List` won’t provide any advantage over a Python `list` when you’re managing a list of email addresses or something.

The default representation of models (via `hy.repr`) uses quoting for readability, so `(hy.models.Integer 5)` is represented as ‘5. Python representations (via `repr()`) use the constructors, and by default are pretty-printed; you can disable this globally by setting `hy.modelsPRETTY` to `False`, or temporarily with the context manager `hy.models.pretty`. You can also color these Python representations with `colorama` by setting `hy.models.COLORED` to `True`.

```python
class hy.models.Object
    An abstract base class for Hy models, which represent forms.
```

15
class hy.models.Lazy(gen)

The output of `hy.read-many`. It represents a sequence of forms, and can be treated as an iterator. Reading each form lazily, only after evaluating the previous form, is necessary to handle reader macros correctly; see `hy.read-many`.

### 3.2 Non-form syntactic elements

#### 3.2.1 Shebang

If a Hy program begins with `#!`, Hy assumes the first line is a shebang line and ignores it. It’s up to your OS to do something more interesting with it.

Shebangs aren’t real Hy syntax, so `hy.read-many` only allows them if its option `skip_shebang` is enabled.

#### 3.2.2 Whitespace

Hy has lax whitespace rules less similar to Python’s than to those of most other programming languages. Whitespace can separate forms (e.g., `a b` is two forms whereas `ab` is one) and it can occur inside some forms (like string literals), but it’s otherwise ignored by the reader, producing no models.

The reader only grants this special treatment to the ASCII whitespace characters, namely U+0009 (horizontal tab), U+000A (line feed), U+000B (vertical tab), U+000C (form feed), U+000D (carriage return), and U+0020 (space). Non-ASCII whitespace characters, such as U+2009 (THIN SPACE), are treated as any other character. So yes, you can have exotic whitespace characters in variable names, although this is only especially useful for obfuscated code contests.

#### 3.2.3 Comments

Comments begin with a semicolon (`;`) and continue through the end of the line.

There are no multi-line comments in the style of C’s `/* ... */`, but you can use the discard prefix or string literals for similar purposes.

#### 3.2.4 Discard prefix

Like Clojure, Hy supports the Extensible Data Notation discard prefix `#_`, which is kind of like a structure-aware comment. When the reader encounters `#_`, it reads and then discards the following form. Thus `#_` is similar to `;` except that reader macros still get executed, and normal parsing resumes after the next form ends rather than at the start of the next line: `[dilly #_ and krunk]` is equivalent to `[dilly krunk]`, whereas `[dilly ; and krunk]` is equivalent to just `[dilly]. Comments indicated by `;` can be nested within forms discarded by `#_`, but `#_` has no special meaning within a comment indicated by `;`. 
3.3 Identifiers

Identifiers are a broad class of syntax in Hy, comprising not only variable names, but any nonempty sequence of characters that aren’t ASCII whitespace nor one of the following: ()[]{}"'. The reader will attempt to read each identifier as a numeric literal, then attempt to read it as a keyword if that fails, then fall back on reading it as a symbol if that fails.

3.3.1 Numeric literals

All of Python’s syntax for numeric literals is supported in Hy, resulting in an Integer, Float, or Complex. Hy also provides a few extensions:

• Commas (,) can be used like underscores (_) to separate digits without changing the result. Thus, 10_000_000_000 may also be written 10,000,000,000.

• Integers can begin with leading zeroes, even without a radix prefix like 0x. Leading zeroes don’t automatically cause the literal to be interpreted in octal like they do in C. For octal, use the prefix 0o, as in Python.

• NaN, Inf, and -Inf are understood as literals. Each produces a Float.

• Hy allows complex literals as understood by the constructor for complex, such as 5+4j. (This is also legal Python, but Hy reads it as a single Complex, and doesn’t otherwise support infix addition or subtraction, whereas Python parses it as an addition expression.)

```python
class hy.models.Integer(number, *args, **kwargs)
    Represents a literal integer (int).

class hy.models.Float(num, *args, **kwargs)
    Represents a literal floating-point real number (float).

class hy.models.Complex(real, imag=0, *args, **kwargs)
    Represents a literal floating-point complex number (complex).
```

3.3.2 Keywords

An identifier starting with a colon (:), such as :foo, is a Keyword.

Literal keywords are most often used for their special treatment in expressions that aren’t macro calls: they set keyword arguments, rather than being passed in as values. For example, (f :foo 3) calls the function f with the parameter foo set to 3. The keyword is also mangled at compile-time. To prevent a literal keyword from being treated specially in an expression, you can quote the keyword, or you can use it itself as a keyword argument, as in (f :foo :bar).

Otherwise, keywords are simple model objects that evaluate to themselves. Users of other Lisps should note that it’s often a better idea to use a string than a keyword, because the rest of Python uses strings for cases in which other Lisps would use keywords. In particular, strings are typically more appropriate than keywords as the keys of a dictionary. Notice that (dict :a 1 :b 2) is equivalent to {"a" 1 "b" 2}, which is different from {a 1 b 2} (see Dictionary literals).

The empty keyword : is syntactically legal, but you can’t compile a function call with an empty keyword argument.

```python
class hy.models.Keyword(value, from_parser=False)
    Represents a keyword, such as :foo.

    Variables
    name – The string content of the keyword, not including the leading :. No mangling is performed.
```
__bool__ ()

The empty keyword is false. All others are true.

__call__ (data, default=<object object>)

Get the element of data named (hy.mangle self.name). Thus, (:foo bar) is equivalent to (get bar "foo") (which is different from (get bar :foo); dictionary keys are typically strings, not hy.models.Keyword objects).

The optional second parameter is a default value; if provided, any KeyError from get will be caught, and the default returned instead.

### 3.3.3 Symbols

Symbols are the catch-all category of identifiers. In most contexts, symbols are compiled to Python variable names, after being mangled. You can create symbol objects with the quote operator or by calling the Symbol constructor (thus, Symbol plays a role similar to the intern function in other Lisps). Some example symbols are hello, **+, 3fiddy, $40, justwrong, and .

As a special case, the symbol ... compiles to the Ellipsis object, as in Python.

class hy.models.Symbol(s, from_parser=False)

Represents a symbol.

Symbol objects behave like strings under operations like get, len(), and bool; in particular, (bool (hy.models.Symbol "False")) is true. Use hy.eval to evaluate a symbol.

### 3.3.4 Mangling

Since the rules for Hy symbols and keywords 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. Underscores are typically the ASCII underscore _, but they may also be any Unicode character that normalizes (according to NFKC) to _. Leading underscores have special significance in Python, and Python normalizes all Unicode before this test, so we’ll process the remainder of the name and then add the leading underscores back onto the final mangled name.

2. Convert ASCII 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 ASCII ?, 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 or it contains a character that’s illegal in that position.
   - 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 and is_--has_dashes becomes hyx_is_XhyphenHminusX_has_dashes.

5. Take any leading underscores removed in the first step, transliterate them to ASCII, and add them back to the mangled name. Thus, (hy.mangle '_tasty?') is "_is_tasty" instead of "is__tasty" and (hy.mangle '__--has-dashes?) is "__hyx_is_XhyphenHminusX_has_dashes".
6. Finally, normalize any leftover non-ASCII characters. The result may still not be ASCII (e.g., is already Python-
legal and normalized, so it passes through the whole mangling procedure unchanged), but it is now guaranteed
that any names are equal as strings if and only if they refer to the same Python identifier.

You can invoke the mangler yourself with the function `hy.mangle`, and try to undo this (perhaps not quite successfully)
with `hy.unmangle`.

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 `hy.unmangle`, 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 under
mangling, so you can’t use e.g. `foo-bar` and `foo_bar` as separate variables.

3.4 String literals

Hy allows double-quoted strings (e.g., "hello"), but not single-quoted strings like Python. The single-quote charac-
ter ' is reserved for preventing the evaluation of a form, (e.g., '(+ 1 1))), as in most Lisps (see Additional sugar).
Python’s so-called triple-quoted strings (e.g., '''hello''' and """hello"""") aren’t supported. However, in Hy, un-
like Python, any string literal can contain newlines; furthermore, Hy has bracket strings. For consistency with Python’s
triple-quoted strings, all literal newlines in literal strings are read as in "\n" (U+000A, line feed) regardless of the
newline style in the actual code.

String literals support a variety of backslash escapes. Unrecognized escape sequences are a syntax error. To create a
“raw string” that interprets all backslashes literally, prefix the string with r, as in r"\not".

Like Python, Hy treats all string literals as sequences of Unicode characters by default. The result is the model type
`String`. You may prefix a string literal with b to treat it as a sequence of bytes, producing `Bytes` instead.

Unlike Python, Hy only recognizes string prefixes (r, b, and f) in lowercase, and doesn’t allow the no-op prefix u.

F-strings are a string-like compound construct documented further below.

```python
class hy.models.String(s=None, brackets=None)
    Represents a literal string (str).
    Variables
    brackets – The custom delimiter used by the bracket string that parsed to this object, or None
    if it wasn’t a bracket string. The outer square brackets and # aren’t included, so the brackets
    attribute of the literal #[[hello]] is the empty string.

class hy.models.Bytes
    Represents a literal bytestring (bytes).
```

3.4.1 Bracket strings

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 an f-string.) For example:

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

```
=> (print #*[==[1 + 1 = 2]==])
1 + 1 = 2
```
Bracket strings are always raw Unicode strings, and don’t allow the r or b prefixes.
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.

3.5 Sequential forms

Sequential forms (*Sequence*) are nested forms comprising any number of other forms, in a defined order.

```python
class hy.models.Sequence(iterable=(), /)
```

An abstract base class for sequence-like forms. Sequence models can be operated on like tuples: you can iterate over them, index into them, and append them with +, but you can’t add, remove, or replace elements. Appending a sequence to another iterable object reuses the class of the left-hand-side object, which is useful when e.g. you want to concatenate models in a macro.

When you’re recursively descending through a tree of models, testing a model with `isinstance x hy.models.Sequence` is useful for deciding whether to iterate over x. You can also use the Hyrule function `coll?` for this purpose.

3.5.1 Expressions

Expressions (*Expression*) are denoted by parentheses: ( ... ). The compiler evaluates expressions by checking the first element. If it’s a symbol, and the symbol is the name of a currently defined macro, the macro is called. Otherwise, the expression is compiled into a Python-level call, with the first element being the calling object. The remaining forms are understood as arguments. Use `unpack-iterable` or `unpack-mapping` to break up data structures into individual arguments at runtime.

The empty expression () is legal at the reader level, but has no inherent meaning. Trying to compile it is an error. For the empty tuple, use #().

```python
class hy.models.Expression(iterable=(), /)
```

Represents a parenthesized Hy expression.

3.5.2 List, tuple, and set literals

Literal lists (*List*), tuples (*Tuple*), and sets (*Set*) are denoted respectively by [ ... ], #( ... ), and #{ ... }.

```python
class hy.models.List(iterable=(), /)
```

Represents a literal list.

Many macros use this model type specially, for something other than defining a list. For example, `defn` expects its function parameters as a square-bracket-delimited list, and `for` expects a list of iteration clauses.

```python
class hy.models.Tuple(iterable=(), /)
```

Represents a literal tuple.

```python
class hy.models.Set(iterable=(), /)
```

Represents a literal set. Unlike actual sets, the model retains duplicates and the order of elements.
### 3.5.3 Dictionary literals

Literal dictionaries (`dict`, `Dict`) are denoted by `{ ... }`. Even-numbered child forms (counting the first as 0) become the keys whereas odd-numbered child forms become the values. For example, `{"a" 1 "b" 2}` produces a dictionary mapping "a" to 1 and "b" to 2. Trying to compile a `Dict` with an odd number of child models is an error.

As in Python, calling `dict` with keyword arguments is often more convenient than using a literal dictionary.

```python
class hy.models.Dict(iterable=(), /)
    Represents a literal `dict`. keys, values, and items methods are provided, each returning a list, although this model type does none of the normalization of a real `dict`. In the case of an odd number of child models, keys returns the last child whereas values and items ignores it.
```

### 3.5.4 Format strings

A format string (or “f-string”, or “formatted string literal”) is a string literal with embedded code, possibly accompanied by formatting commands. The result is an `FString`, 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 debugging `=`, conversion specifier, 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}")
axxxx
```

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

Also unlike Python, comments and backslashes are allowed in replacement fields. The same reader is used for the form to be evaluated as for elsewhere in the language. Thus e.g. `f"{"a"}"` is legal, and equivalent to "a".

```python
class hy.models.FString(s=None, brackets=None)
    
    Variables
    brackets – As in `hy.models.String`.
```

```python
class hy.models.FComponent(s=None, conversion=None)
    An analog of ast.FormattedValue. The first node in the contained sequence is the value being formatted. The rest of the sequence contains the nodes in the format spec (if any).
```
3.6 Additional sugar

Syntactic sugar is available to construct two-item expressions with certain macros. When the sugary characters are encountered by the reader, a new expression is created with the corresponding macro as the first element and the next parsed form as the second. No parentheses are required. Thus, since ' is short for quote, 'FORM is read as (quote FORM). This is all resolved at the reader level, so the model that gets produced is the same whether you take your code with sugar or without.

<table>
<thead>
<tr>
<th>Macro</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>quasiquote</td>
<td>`FORM</td>
</tr>
<tr>
<td>quote</td>
<td>'FORM</td>
</tr>
<tr>
<td>unpack-iterable</td>
<td>#* FORM</td>
</tr>
<tr>
<td>unpack-mapping</td>
<td>#** FORM</td>
</tr>
<tr>
<td>unquote</td>
<td>~FORM</td>
</tr>
<tr>
<td>unquote-splice</td>
<td>~@FORM</td>
</tr>
</tbody>
</table>

3.7 Reader macros

A hash (#) followed by a symbol invokes the reader macro named by the symbol. (Trying to call an undefined reader macro is a syntax error.) Parsing of the remaining source code is under control of the reader macro until it returns.
This chapter describes features of Hy semantics that differ from Python’s and aren’t better categorized elsewhere, such as in the chapter *Macros*.

### 4.1 Implicit names

Every compilation unit (basically, module) implicitly begins with `(import hy)`. You can see it in the output of `hy2py`. The purpose of this is to ensure Hy can retrieve any names it needs to compile your code. For example, the code `(print '(+ 1 1))` requires constructing a `hy.models.Expression`. Thus you should be wary of assigning to the name `hy`, even locally, because then the wrong thing can happen if the generated code tries to access `hy` expecting to get the module. As a bonus, you can say things like `(print (hy.repr #(1 2)))` without explicitly importing `hy` first.

If you restrict yourself to a subset of Hy, it’s possible to write a Hy program, translate it to Python with `hy2py`, remove the `import hy`, and get a working Python program that doesn’t depend on Hy itself. Unfortunately, Python is too dynamic for the Hy compiler to be able to tell in advance when this will work, which is why the automatic import is unconditional.

Hy needs to create temporary variables to accomplish some of its tricks. For example, in order to represent `(print (with ...))` in Python, the result of the `with` form needs to be set to a temporary variable. These names begin with `_hy_`, so it’s wise to avoid this prefix in your own variable names. Such temporary variables are scoped in the same way surrounding ordinary variables are, and they aren’t explicitly cleaned up, so theoretically, they can waste memory and lead to `object.__del__()` being called later than you expect. When in doubt, check the `hy2py` output.

### 4.2 Order of evaluation

Like many programming languages, but unlike Python, Hy doesn’t guarantee in all cases the order in which function arguments are evaluated. More generally, the evaluation order of the child models of a `hy.models.Sequence` is unspecified. For example, `(f (g) (h))` might evaluate (part of) (h) before (g), particularly if f is a function whereas h is a macro that produces Python-level statements. So if you need to be sure that g is called first, call it before f.
4.3 When bytecode is regenerated

The first time Hy is asked to execute a file, it will produce a bytecode file (unless `PYTHONDONTWRITEBYTECODE` is set). Subsequently, if the source file hasn’t changed, Hy will load the bytecode instead of recompiling the source. Python behaves similarly, but the difference between recompilation and loading bytecode is more consequential in Hy because of how Hy lets you run and generate code at compile-time with things like macros, reader macros, and `eval-and-compile`. You may be surprised by behavior like the following:

```sh
$ echo '(defmacro m [] 1)' >a.hy
$ echo '(require a) (print (a.m))' >b.hy
$ hy b.hy
1
$ echo '(defmacro m [] 2)' >a.hy
$ hy b.hy
1
```

Why didn’t the second run of `b.hy` print 2? Because `b.hy` was unchanged, so it didn’t get recompiled, so its bytecode still had the old expansion of the macro `m`. 
5.1 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 \texttt{nif} (see \url{http://letoverlambda.com/index.cl/guest/chap3.html#sec_5} for a more complete description.) \texttt{nif} is an example, something like a numeric \texttt{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:

\begin{verbatim}
defmacro nif [expr pos-form zero-form neg-form]  `(do  (setv obscure-name ~expr)  (cond (> obscure-name 0) ~pos-form  (= obscure-name 0) ~zero-form  (< obscure-name 0) ~neg-form)))
\end{verbatim}

where \texttt{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 \texttt{gensym} is designed to generate a new, unique symbol for just such an occasion. A much better version of \texttt{nif} would be:

\begin{verbatim}
defmacro nif [expr pos-form zero-form neg-form]  `(do  (setv g (hy.gensym))  `(do  (setv ~g ~expr)  (cond (> ~g 0) ~pos-form  (= ~g 0) ~zero-form  (< ~g 0) ~neg-form)))
\end{verbatim}

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 \texttt{with-gensyms} that basically expands to a \texttt{setv} form:

\begin{verbatim}
(with-gensyms [a b c]  ...)
\end{verbatim}

expands to:

\begin{verbatim}
(do  (setv a (hy.gensym)
\end{verbatim}
so our re-written nif would look like:

```
(defmacro nif [expr pos-form zero-form neg-form]
  (with-gensyms [g]
    `(do
      (setv ~g expr)
      (cond (> ~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:

```
(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)))
```
Hy’s read-eval-print loop (REPL) is implemented in the class `hy.cmdline.HyREPL`. The REPL can be started interactively from the command line or programmatically with the instance method `hy.cmdline.HyREPL.run()`.

Two environment variables useful for the REPL are `HY_HISTORY`, which specifies where the REPL input history is saved, and `HYSTARTUP`, which specifies a file to run when the REPL starts.

```python
class hy.cmdline.HyREPL(spy=False, output_fn=None, locals=None, filename='<stdin>:
    A subclass of code.InteractiveConsole for Hy.
    run()
    Start running the REPL. Return 0 when done.
```

### 6.1 Output functions

By default, the return value of each REPL input is printed with `hy.repr`. To change this, you can set the REPL output function with e.g. the command-line argument `--repl-output-fn`. Use `repr()` to get Python representations like Python’s own REPL.

Regardless of the output function, no output is produced when the value is `None`, as in Python.

### 6.2 Special variables

The REPL maintains a few special convenience variables. *1 holds the result of the most recent input, like _ in the Python REPL. *2 holds the result of the input before that, and *3 holds the result of the input before that. Finally, *e holds the most recent uncaught exception.

### 6.3 Startup files

Any macros or Python objects defined in the REPL startup file will be brought into the REPL’s namespace. Two variables are special in the startup file:

- **repl-spy**
  - If true, print equivalent Python code before executing each piece of Hy code.

- **repl-output-fn**
  - The output function, as a unary callable object.

Hy startup files can do a number of other things like set banner messages or change the prompts. The following example shows a number of possibilities:
;; Wrapping in an `eval-and-compile` ensures these 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>"))

(import
  re
  json
  pathlib [Path]
  hy.pypos *
  hyrule [pp pformat])

(require
  hyrule [unless])

(setv
  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"
)

(defn slurp [path]
  (setv path (Path path))
  (when (path.exists)
    (path.read-text)))

(defmacro greet [person]
  `(print ~person))
CHAPTER
SEVEN

ENVIRONMENT VARIABLES

Hy treats the following environment variables specially. Boolean environment variables are interpreted as false when set to the empty string and true when set to anything else.

HYSTARTUP
(Default: nothing) Path to a file containing Hy source code to execute when starting the REPL.

HY_COLORED_ASTOBJECTS
(Default: false) Whether to use ANSI color when printing the Python `repr()`s of Hy models.

HY_COLORED_ERRORS
(Default: false) Whether to use ANSI color when printing certain error messages.

HY_DEBUG
(Default: false) Does something mysterious that’s probably similar to HY_FILTER_INTERNAL_ERRORS.

HY_FILTER_INTERNAL_ERRORS
(Default: true) Whether to hide some parts of tracebacks that point to internal Hy code and won’t be helpful to the typical Hy user.

HY_HISTORY
(Default: ~/.hy-history) Path to which REPL input history will be saved.

HY_MESSAGE_WHEN_COMPILING
(Default: false) Whether to print “Compiling FILENAME” to standard error before compiling each file of Hy source code. This is helpful for debugging whether files are being loaded from bytecode or re-compiled.
CHAPTER EIGHT

COMMAND LINE INTERFACE

8.1 hy

8.1.1 Command Line Options

- <command>
  Execute the Hy code in command.
  
  $ hy -c "(print (+ 2 2))"
  4

-<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.

--spy
  Print equivalent Python code before executing in REPL. For example:
  
  => (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.

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

-v
  Print the Hy version number and exit.
8.2 hy2py

hy2py is a program to convert Hy source code into Python source code. Use `hy2py --help` for usage instructions. It can take its input from standard input or from a filename provided as a command-line argument. The result is written to standard output.

**Warning:** hy2py can execute arbitrary code. Don’t give it untrusted input.

8.3 hyc

hyc is a program to compile files of Hy code into Python bytecode. Use `hyc --help` for usage instructions. The generated bytecode files are named and placed according to the usual scheme of your Python executable, as indicated by `importlib.util.cache_from_source()`.

**Warning:** hyc can execute arbitrary code. Don’t give it untrusted input.
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.

### 9.1 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:

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

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

### 9.2 Using Hy from Python

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

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

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

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

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

```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.

### 9.2.1 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.**

### 9.2.2 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
```
9.2.3 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` function to turn the expression into a Hy model:

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

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

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

9.3 Libraries that expect Python

There are various means by which Hy may interact poorly with a Python library because the library doesn’t account for the possibility of Hy. For example, when you run the command-line program `hy`, `sys.executable` will be set to this program rather than the original Python binary. This is helpful more often than not, but will lead to trouble if e.g. the library tries to call `sys.executable` with the `-c` option. In this case, you can try setting `sys.executable` back to `hy.sys-executable`, which is a saved copy of the original value. More generally, you can use `hy2py`, or you can put a simple Python wrapper script like `import hy, my_hy_program` in front of your code; importing `hy` first is necessary here to install the hooks that allow Python to load your Hy module.
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.

### 10.1 A motivating example

The kind of problem that model patterns are suited for is the following. Suppose you want to validate and extract the components of a form like:

```
(setv form '(try
  (foo1)
  (foo2)
  (except [EType1]
    (foo3))
  (except [e EType2]
    (foo4)
    (foo5))
  (except []
    (foo6))
  (finally
    (foo7)
    (foo8)))
```

You could do this with loops and indexing, but it would take a lot of code and be error-prone. Model patterns concisely express the general form of an expression to be matched, like what a regular expression does for text. Here’s a pattern for a `try` form of the above kind:

```
(import
  funcparserlib.parser [maybe many]
  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:
which is conveniently utilized with an assignment such as `(setv [body except-clauses else-part finally-part] result)`. Notice that `else-part` will be set to `None` because there is no `else` clause in the original form.

## 10.2 Usage

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

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

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

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

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

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

(print (hy.repr (pairs a 1 b 2 c 3)))
; => [#("a" 1) #("b" 2) #("c" 3)]
```
A failed parse will raise `funcparserlib.parser.NoParseError`. 
## 11.1 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>
## 11.2 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>^ . 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 dfor 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 yield yield-from</code></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>@ % + - * ** / //</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>
12.1 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`.

`(annotate (value, type))`

`annotate` and its shorthand form `#^` are used to denote annotations, including type hints, in three different contexts:

- Standalone variable annotations ([PEP 526](https://www.python.org/dev/peps/pep-0526/))
- Variable annotations in a `setv` call
- Function-parameter annotations ([PEP 3107](https://www.python.org/dev/peps/pep-03107/))

The difference between `annotate` and `#^` is that `annotate` requires parentheses and takes the name to be annotated first (like Python), whereas `#^` doesn’t require parentheses (it only applies to the next two forms) and takes the type second:

```
(setv (annotate x int) 1)
(setv #^int x 1)
```

The order difference is not merely visual: `#^` actually evaluates the type first.

Here are examples with `#^` for all the places you can use annotations:

```
; Annotate the variable `x` as an `int` (equivalent to `x: int`).
#^int x
; You can annotate with expressions (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).
```

(continues on next page)
(defn #^int add1 [#^int x] (+ x 1))
(fn #^int [#^int y] (+ y 2))

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

An issue with type annotations is that, as of this writing, we know of no Python type-checker that can work with `ast` objects or bytecode files. They all need Python source text. So you'll have to translate your Hy with `hy2py` in order to actually check the types.

. 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.

Examples

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

Compiles down to:
```
foo.bar("qux").baz[1 + 2].frob
```

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

(fn (args))
As `defn`, but no name for the new function is required (or allowed), and the newly created function object is returned. Decorators aren't allowed, either. However, the function body is understood identically to that of `defn`, without any of the restrictions of Python’s `lambda`. See `fn/a` for the asynchronous equivalent.

(fn/a (name, #* args))
As `fn`, but the created function object will be a coroutine.

(defn (name, #* args))
`defn` compiles to a function definition (or possibly to an assignment of a lambda expression). It always returns `None`. It requires two arguments: a name (given as a symbol; see `fn` for anonymous functions) and a “lambda list”, or list of parameters (also given as symbols). Any further arguments constitute the body of the function:

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

An empty body is implicitly `(return None)`. If there at least two body forms, and the first of them is a string literal, this string becomes the docstring of the function. The final body form is implicitly returned; thus, `(defn f [] 5)` is equivalent to `(defn f [] (return 5))`.

`defn` accepts two additional, optional arguments: a bracketed list of decorators and an annotation (see `^`) for the return value. These are placed before the function name (in that order, if both are present):

```
(defn [decorator1 decorator2] ^annotation name [params] ...)
```

To define asynchronous functions, see `defn/a` and `fn/a`.

`defn` lambda lists support all the same features as Python parameter lists and hence are complex in their full generality. The simplest case is a (possibly empty) list of symbols, indicating that all parameters are required,
and can be set by position, as in \((f \ value)\), or by name, as in \((f :argument \ value)\). To set a default value for a parameter, replace the parameter with the bracketed list \([pname \ value]\), where \(pname\) is the parameter name as a symbol and \(value\) is an arbitrary form. Beware that, per Python, \(value\) is evaluated when the function is defined, not when it's called, and if the resulting object is mutated, all calls will see the changes.

Further special lambda-list syntax includes:

**/**

If the symbol \(/\) is given in place of a parameter, it means that all the preceding parameters can only be set positionally.

**/*

If the symbol \(*\) is given in place of a parameter, it means that all the following parameters can only be set by name.

**#* args

If the parameter list contains \(** arg**s\) or (unpack-iterable \(args\)), then \(args\) is set to a tuple containing all otherwise unmatched positional arguments. The name \(args\) is merely cherished Python tradition; you can use any symbol.

**#** kwargs

\(** kwargs\) (a.k.a. (unpack-mapping \(kwargs\)) is like \(** arg**s\), but collects unmatched keyword arguments into a dictionary.

Each of these special constructs is allowed only once, and has the same restrictions as in Python; e.g., \(** arg**s\) must precede \(**** kwargs\) if both are present. Here's an example with a complex lambda list:

```
(defn f [a / b [c 3] * d e **kwargs]
[a b c d e kwargs])
(print (hy.repr (f 1 2 :d 4 :e 5 :f 6)))
; => [1 2 3 4 5 {"f" 6}]
```

\(\texttt{defn/a}\) (name, lambda-list, \(*\ body\))

As \(\texttt{defn}\), but defines a coroutine like Python's \texttt{async def}.

\(\texttt{defmacro}\) (name, lambda-list, \(*\ body\))

\(\texttt{defmacro}\) is used to define macros. The general format is \(\texttt{(defmacro name [parameters] expr)}\).

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 
  ... (unquote (get code 1))
  ... (unquote (get code 0))
  ... (unquote (get code 2)))))

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

\(\textbf{Note:}\) because all values are passed to macros unevaluated, \(\texttt{defmacro}\) cannot use keyword arguments, or \(kwargs\). All arguments are passed in positionally. Parameters can still be given default values however:
(defmacro a-macro [a [b 1]])
... `([-a ~b])
=> (a-macro 2)
[2 1]
=> (a-macro 2 3)
[2 3]
=> (a-macro :b 3)
[:b 3]

(if (test, then, else))
if compiles to an if expression (or compound if statement). The form test is evaluated and categorized as true or false according to bool. If the result is true, then is evaluated and returned. Otherwise, else is evaluated and returned.

(if (has-money-left account)
  (print "Let's go shopping!")
  (print "Back to work.")
)

See also:
• do, to execute several forms as part of any of if’s three arguments.
• when, for shorthand for (if condition (do ...) None).
• cond, for shorthand for nested if forms.

(await 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))
hello
world

(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 $x \leq y < z$. 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.

```hy
(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.

```hy
(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**

```hy
=> (+ 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`.

```hy
=> (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`.

```hy
=> (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.

```hy
=> (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**

```hy
(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 compiles to an import statement, which makes objects in a different module available in the current module. Hy’s syntax for the various kinds of import looks like this:

;; Import each of these modules
;; Python: import sys, os.path
(import sys os.path)

;; Import several names from a single module
;; Python: from os.path import exists, isdir as is_dir, isfile
(import os.path [exists isdir :as dir? isfile])

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

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

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

_all__ can be set to control what’s imported by import *, as in Python, but beware that all names in __all__
must be mangled. The macro export is a handy way to set __all__ in a Hy program.

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

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

(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

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

(defmacro m [x]
  (add x 2))
(print (m 3)); prints 5
(print (add 3 6)); raises NameError: name 'add' is not defined

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

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

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

Here’s a more complex example:

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

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

The general form of lfor is:

(lfor CLAUSES VALUE)

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

- Iteration clauses, which look like LVALUE ITERABLE. The LVALUE is usually just a symbol, but could be more complicated, like [x y].
- :async LVALUE ITERABLE, which is an asynchronous form of iteration clause.
- :do FORM, which simply evaluates the FORM. If you use (continue) or (break) here, they will apply to the innermost iteration clause before the :do.
- :setv LVALUE RVALUE, which is equivalent to :do (setv LVALUE RVALUE).
- :if CONDITION, which is equivalent to :do (when (not CONDITION) (continue)).

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

(dfor(#* args))
dfor creates a dictionary comprehension. Its syntax is the same as that of lfor except that it takes two trailing arguments. The first is a form producing the key of each dictionary element, and the second produces the value. Thus:

=> (dfor x (range 5) x (* x 10))
{0 0 1 10 2 20 3 30 4 40}
**hy, Release 0.25.0**

**gfor(**

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

**Examples**

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

**sfor(**

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

**setv(**

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

**Examples**

```
=> (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
```

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

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

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

```
=> (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(**

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**

```
(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

(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:

(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.

If lfor, sfor, dfor, or gfor (but not for) is in the body of a let, assignments in iteration clauses and :setv clauses will create a new variable in the comprehenion form’s own scope, without touching any outer let-bound variable of the same name.

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

(let [x 5
... y (+ x 1)]
... (print x y))
5 6

(let [x 1
... x (fn [] x)]
... (x))
1

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:
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 (match (+ 1 1)
... 1 "one"
... 2 "two"
... 3 "three")
"two"
```

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
=> (import  dataclasses [dataclass])
=> (defclass [dataclass] 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). The class name may also be preceded by a list of decorators, as in defn.

Examples

```hy
=> (defclass class-name [super-class-1 super-class-2]
...  "docstring"
... ...
... (setv attribute1 value1)
... (setv attribute2 value2)
... ...
... (defn method [self] (print "hello!")))
```

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

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

del removes an object from the current namespace.

Examples

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

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

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

(nonlocal object))

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.

12.1. Core Macros
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

```clojure
;; 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

```clojure
=> (setv x '(print "Hello World"))
=> x ; variable x is set to unevaluated expression
=> (hy.eval x)
Hello World
```
```
(require (#* args))
require is used to import macros and reader macros from one or more given modules. It allows parameters in all the same formats as `import`. `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
Reader macros are required using :readers [...]. The :macros kwarg can be optionally added for readability:

```hy
g=> (require mymodule :readers *)
g=> (require mymodule :readers [!])
g=> (require mymodule [foo] :readers [!])
g=> (require mymodule :readers [!] [foo])
g=> (require mymodule :macros [foo] :readers [!])
```

Do note however, that requiring :readers, but not specifying any regular macros, will not bring that module's macros in under their absolute paths:

```hy
g=> (require mymodule :readers [!])
g=> (mymodule.foo)
Traceback (most recent call last):
  File "stdin-cd49eaaabebc174c87ebe6bf15f2f8a28660feba", line 1, in <module>
    (mymodule.foo)
NameError: name 'mymodule' is not defined
```

Unlike requiring regular macros, reader macros cannot be renamed with :as, and are not made available under their absolute paths to their source module:

```hy
g=> (require mymodule :readers [!])
HySyntaxError: ...

=> (require mymodule :readers [! :as &])
HySyntaxError: ...

=> (require mymodule)
=> mymodule.! x
NameError: name 'mymodule' is not defined
```

To define which macros are collected by (require mymodule *), set the variable _hy_export_macros (analogous to Python's __all__) to a list of mangled macro names, which is accomplished most conveniently with export. The default behavior is to collect all macros other than those whose mangled names begin with an ASCII underscore (_).

When requiring reader macros, (require mymodule :readers *) will collect all reader macros both defined and required within mymodule.
Macros that call macros

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

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

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

And then, in your main program, you write:

```
(require mymodule [foo])
(print (mymodule.foo 3))
```

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

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

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

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

```
=> (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.
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))
(f 4)
14
```

```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)
[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]
```

**(raise)** *(exception None]*)

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

**Examples**

```hy
(raise)
; re-raise the last exception
(raise IOError)
; raise an IOError
(raise (IOError "foobar"))
; raise an IOError("foobar")
```

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

12.1. Core Macros
try compiles to a try statement, which can catch exceptions and run cleanup actions. It begins with any number of body forms. Then follows any number of except or except* (PEP 654) forms, which are expressions that begin with the symbol in question, followed by a list of exception types, followed by more body forms. Finally there are an optional else form and an optional finally form, which again are expressions that begin with the symbol in question and then comprise body forms. As in Python, at least one of except, except*, or finally is required; else is only allowed if at least one except or except* is provided; except* requires Python 3.11; and except and except* may not both be used in the same try.

Here’s an example of several of the allowed kinds of child forms:

```hy
(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")))
```

Exception lists can be in any of several formats:

- [] to catch any subtype of Exception, like Python’s except:
  ```hy```

- [ETYPE] to catch only the single type ETYPE, like Python’s `except ETYPE:
  ```hy```

- [[ETYPE1 ETYPE2 ...]] to catch any of the named types, like Python’s except ETYPE1, ETYPE2, ...

- [VAR ETYPE] to catch ETYPE and bind it to VAR, like Python’s except ETYPE as VAR:

- [VAR [ETYPE1 ETYPE2 ...]] to catch any of the named types and bind it to VAR, like Python’s except ETYPE1, ETYPE2, ... as VAR:

The return value of try is the last form evaluated among the main body, except forms, except* forms, and else.

(unpack-iterable())

(unpack-mapping())

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

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

```hy```

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

unpack-iterable is usually written with the shorthand #*, and unpack-mapping with #**.
Unpacking is allowed in a variety of contexts, and you can unpack more than once in one expression (PEP 3132, PEP 448).

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

(\textit{unquote(symbol)})

Within a quasiquoted form, \textit{unquote} forces evaluation of a symbol. \textit{unquote} is aliased to the tilde (~) symbol.

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

(\textit{unquote-splice(symbol)})

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

```
=> (setv nums [1 2 3 4])
=> (quasiquote (+ (unquote-splice nums)))
'= (+ 1 2 3 4)
=> '+ ~@nums
'= (+ 1 2 3 4)
=> '(+ ~@(when (< (get nums 0) 0) nums))
'[1 2]
```

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

(\textit{while}\texttt{(condition, ## body)})

\textit{while} compiles to a \texttt{while} statement. It is used to execute a set of forms as long as a condition is met. The first argument to \textit{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 \textit{while} loop can be an \texttt{else} clause, which is executed after the loop terminates, unless it exited abnormally (e.g., with \texttt{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 archetypical 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:

=> (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:

=> (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:
(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:

::

  => (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.

(yield(object))

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

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

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

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

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

(yield-from(object))

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.cond(#* args))

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

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

is equivalent to
(if condition1
  result1
  (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.defreader(key, #* body))

Define a new reader macro.

Reader macros are expanded at read time and allow you to modify the behavior of the Hy reader. Access to the currently instantiated HyReader is available in the body as &reader. See HyReader and its base class Reader for details regarding the available processing methods.

Reader macro names can be any symbol that does not start with a ^ and are callable by prefixing the name with a #. i.e. (defreader upper ...) is called with #upper.

Examples

The following is a primitive example of a reader macro that adds Python’s colon : slice sugar into Hy:

```hy
=> (defreader slice
...   (defn parse-node []
...     (let [node (when (!= ":" (.peekc &reader))
...       (.parse-one-form &reader))]
...       (if (= node '...
...         'Ellipse node))))
...     (with [[&reader.end-identifier ":"]]
...       (let [nodes []]
...         (&reader.slurp-space)
...         (nodes.append (parse-node))
...         (while (&reader.peek-and-getc ":")
...           (nodes.append (parse-node))
...         )
...         `(slice ~@nodes))))

=> (setv an-index 42)
=> #slice a:(+ 1 2):"column"
(slice 42 3 column)
```

See the reader macros docs for more detailed information on how reader macros work and are defined.

macro(hy.core.macros.delmacro(#* names))

Delete a macro(s) from the current module

```hy
=> (require a-module [some-macro])
=> (some-macro)
1
=> (delmacro some-macro)
```
=> (some-macro)
Traceback (most recent call last):
  File "<string>", line 1, in <module>
      (some-macro)
NameError: name 'some_macro' is not defined

=> (delmacro some-macro)
Traceback (most recent call last):
  File "<string>", line 1, in <module>
      (delmacro some-macro)
NameError: macro 'some-macro' is not defined

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.export(#* args))

A convenience macro for defining __all__ and _hy_export_macros, which control which Python objects and macros (respectively) are collected by * imports in import and require (respectively). export allows you to provide the names as symbols instead of strings, and it calls hy.mangle for you on each name.

The syntax is (export objects macros), where objects refers to Python objects and macros to macros. Keyword arguments are allowed. For example,

```
(export
  :objects [my-fun MyClass]
  :macros [my-macro])
```

exports the function my-fun, the class MyClass, and the macro my-macro.

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

Shorthand for (if test (do ...) None). See if. For a logically negated version, see Hyrule's unless.

```
(when panic
  (log.write panic)
  (print "Process returned:" panic.msg)
  (return panic))
```

12.1.1 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
- 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., \texttt{else} in \texttt{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 \texttt{else} inside \texttt{while} may not work.

12.2 Hy

The \texttt{hy} module is auto imported into every Hy module and provides convenient access to the following methods

\begin{verbatim}
(hy.read(stream, filename, reader))
\end{verbatim}

Like \texttt{hy.read-many}, but only one form is read, and shebangs are forbidden. The model corresponding to this specific form is returned, or, if there are no forms left in the stream, \texttt{EOFError} is raised. \texttt{stream.pos} is left where it was immediately after the form.

\begin{verbatim}
(hy.read-many(stream [filename <string>] reader [skip-shebang False]))
\end{verbatim}

Parse all the Hy source code in \texttt{stream}, which should be a textual file-like object or a string. \texttt{filename}, if provided, is used in error messages. If no \texttt{reader} is provided, a new \texttt{hy.reader.hy_reader.HyReader} object is created. If \texttt{skip_shebang} is true and a shebang line is present, it’s detected and discarded first.

Return a value of type \texttt{hy.models.Lazy}. If you want to evaluate this, be careful to allow evaluating each model before reading the next, as in \texttt{(hy.eval (hy.read-many o))}. By contrast, forcing all the code to be read before evaluating any of it, as in \texttt{(hy.eval `(do [&@((hy.read-many o))])})), will yield the wrong result if one form defines a reader macro that’s later used in the same stream to produce new forms.

\begin{itemize}
  \item \textbf{Warning:} Thanks to reader macros, reading can execute arbitrary code. Don’t read untrusted input.
\end{itemize}

\begin{verbatim}
(hy.eval(hytree, locals, module, ast-callback, compiler, filename, source, [import-stdlib True]))
\end{verbatim}

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 \texttt{fix_missing_locations} and related functions in the Python \texttt{ast} library.

\section*{Examples}

\begin{verbatim}
=> (hy.eval `(print "Hello World"))
"Hello World"
\end{verbatim}

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

\begin{verbatim}
=> (hy.eval (hy.read-str "(+ 1 1)"))
2
\end{verbatim}

\section*{Parameters}

\begin{itemize}
  \item \texttt{hytree} (\texttt{Object}) – The Hy AST object to evaluate.
  \item \texttt{locals} (\texttt{Optional[dict]}) – Local environment in which to evaluate the Hy tree. Defaults to the calling frame.
\end{itemize}
• **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

```python
(hy.repr(obj))
```

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

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

Like `repr` in Python, `hy.repr` can round-trip many kinds of values. Round-tripping implies that given an object *x*, `(hy.eval (hy.read (hy.repr x)))` returns *x*, or at least a value that’s equal to *x*. A notable exception to round-tripping is that if a `hy.models.Object` contains a non-model, the latter will be promoted to a model in the output:

```python
(setv
  x (hy.models.List [5])
  output (hy.repr x)
  y (hy.eval (hy.read output)))
(print output) ; '5
(print (type (get x 0))) ; <class 'int'>
(print (type (get y 0))) ; <class 'hy.models.Integer'>
```

```python
(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.

If the argument is already both legal as a Python identifier and normalized according to Unicode normalization form KC (NFKC), it will be returned unchanged. Thus, mangle is idempotent.

Examples

=> (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_XhyphenHminusHgreaterHthan_signX"

(continues on next page)
(hy.unmangle(s))

Stringify the argument and try to convert it to a pretty unmangled form. See Hy’s mangling rules.

Unmangling may not round-trip, because different Hy symbol names can mangle to the same Python identifier. In particular, Python itself already considers distinct strings that have the same normalized form (according to NFKC), such as hello and , to be the same identifier.

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.

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

Examples

```lisp
=> (hy.disassemble '(print "Hello World!")
Module(
  body=[
    Expr(value=Call(func=Name(id='print'), args=[Str(s='Hello World!')],\n         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.

Examples

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

(hy-macroexpand-1(form))

Return the single step macro expansion of form.

Examples

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

(hy-gensym([g 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.

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 its possible to return non-Hy model objects or even create an expression with non-Hy model elements:

```
=> (defmacro hello []
...  "world!"
=> (defmacro print-inc [a]
```

(continues on next page)
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 compositor form and the canonical model tree representation matters, is when comparing some spliced model tree with another known tree:

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

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 compositor form and the canonical model tree representation matters, is when comparing some spliced model tree with another known tree:

```hy
=> (= `(print (+ 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)) `(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)`
```

**hy.errors.COLORED**

This variable is initially False. If it’s set to a true value, Hy will color its error messages with colorama.

### 12.3 Reader Macros

Like regular macros, reader macros should return a Hy form that will then be passed to the compiler for execution. Reader macros access the Hy reader using the &reader name. It gives access to all of the text- and form-parsing logic that Hy uses to parse itself. See *HyReader* and its base class *Reader* for details regarding the available processing methods.

```python
class hy.reader.hy_reader.HyReader
A modular reader for Hy source.

fill_pos(model, start)
Attach line/col information to a model.

Parameters
- **model** (hy.models.Object) – model to set line/col info for.
- **start** (tuple[int, int]) – (line, column) tuple indicating the start location to assign to model.

parse(stream, filename=None)
Yields all hy.models.Object’s in source

Parameters
- **source** – Hy source to be parsed.
- **filename** (str | None) – Filename to use for error messages. If None then previously set filename is used.
```
parse_forms_until(closer)
Yields hy.models.Object’s until character closer is seen.
Useful for reading a sequence such as s-exprs or lists.

parse_one_form()
Read from the stream until a form is parsed.
Guaranteed to return a model (i.e., skips over comments).

Returns
hy.models.Object

read_default(key)
Default reader handler when nothing in the table matches.
Try to read an identifier/symbol. If there’s a double-quote immediately following, then parse it as a string
with the given prefix (e.g., r”… “). Otherwise, parse it as a symbol-like.

class hy.reader.reader.Reader
A reader base class for reading input character-by-character. Only for use as a base class; cannot be instantiated
directly.

See class HyReader for an example of creating a reader class.

ends_ident
Set of characters that indicate the end of an identifier

Type
set[str]

reader_table
A dictionary mapping a reader macro key to its dispatch func

Type
dict[str, Callable]

pos
Read-only (line, column) tuple indicating the current cursor position of the source being read.

Type
tuple[int, int]

chars(eof_ok=False)
Iterator for the character stream.
Consumes characters as they are produced.

Parameters
eof_ok (bool) – Whether or not it’s okay to hit the end of the file while consuming the iterator. Defaults to False

Yields
str – The next character in source.

Raises
PrematureEndOfInput – if eof_ok is False and the iterator hits the end of source

dispatch(tag)
Call the handler for the tag.

Parameters
tag (str) – Reader macro dispatch key.
Returns
Model returned by the reader macro defined for tag.

Return type
`hy.models.Object | None`

`end_identifier(character)`
Temporarily add a new character to the `ends_ident` set.

`getc()`
Get one character from the stream, consuming it.

This function does the bookkeeping for position data, so it’s important that any character consumption go through this function.

Returns
The character under the cursor at `pos`.

Return type
`str`

`getn(n)`
Returns `n` characters.

`peek_and_getc(target)`
Peek one character and check if it’s equal to `target`.

Only consumes the peeked character if it is equal to `target`

Returns
Whether or not the next character in the stream is equal to `target`.

Return type
`bool`

`peekc()`
Peek at a character from the stream without consuming it.

Returns
character at `pos`

Return type
`str`

`peeking(eof_ok=False)`
Iterate over character stream without consuming any characters.

Useful for looking multiple characters ahead.

Parameters
`eof_ok (bool)` – Whether or not it is okay to hit the end of the file while peeking. Defaults to `False`

Yields
`str` – The next character in `source`.

Raises
`PrematureEndOfInput` – if `eof_ok` is `False` and the iterator hits the end of `source`

`read_ident(just.peeking=False)`
Read characters until we hit something in `ends_ident`. 
Parameters

`just_peeking` – Whether or not to consume characters while peeking. Defaults to `False`.

Returns

The identifier read.

Return type

`str`

`saving_chars()`

Save all the characters read while in this block.

Useful for ‘=` mode in f-strings.

Returns

`list[str]`

`slurp_space()`

Returns and consumes 0 or more whitespace characters.

### 12.4 Python Operators

`(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`

`(hy.pyops.%(x, y))`

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

- `(x y) → x % y`

`(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`

`(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`

`(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`
(hy.pyops.+(#* args))

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

- \((+) \rightarrow 0\)
- \((+ x) \rightarrow +x\)
- \((+ x y) \rightarrow x + y\)
- \((+ a1 a2 \ldots an) \rightarrow a1 + a2 + \ldots + an\)

(hy.pyops.-{(a1, #* a-rest)})

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

- \((- x) \rightarrow -x\)
- \((- x y) \rightarrow x - y\)
- \((- a1 a2 \ldots an) \rightarrow a1 - a2 - \ldots - an\)

(hy.pyops./{(a1, #* a-rest)})

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

- \((/ x) \rightarrow 1 / x\)
- \((/ x y) \rightarrow x / y\)
- \((/ a1 a2 \ldots an) \rightarrow a1 / a2 / \ldots / an\)

(hy.pyops./(a1, a2, #* a-rest))

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

- \((// x y) \rightarrow x // y\)
- \((// a1 a2 \ldots an) \rightarrow a1 // a2 // \ldots // an\)

(hy.pyops.<{(a1, #* a-rest)})

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

- \((< x) \rightarrow True\)
- \((< x y) \rightarrow x < y\)
- \((< a1 a2 \ldots an) \rightarrow a1 < a2 < \ldots < an\)

(hy.pyops.<<(a1, a2, #* a-rest)))

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

- \((<< x y) \rightarrow x << y\)
- \((<< a1 a2 \ldots an) \rightarrow a1 << a2 << \ldots << an\)

(hy.pyops.<=(a1, #* a-rest))

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

- \((<= x) \rightarrow True\)
- \((<= x y) \rightarrow x <= y\)
- \((<= a1 a2 \ldots an) \rightarrow a1 <= a2 <= \ldots <= an\)

(hy.pyops.==(a1, #* a-rest))

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

- \((= x) \rightarrow True\)
- \((= x y) \rightarrow x == y\)
The greater-than operator. Its effect can be defined by the equivalent Python:

- \( (> x) \rightarrow \text{True} \)
- \( (> x y) \rightarrow x > y \)
- \( (> a_1 a_2 \ldots a_n) \rightarrow a_1 > a_2 > \ldots > a_n \)

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

- \( (\geq x) \rightarrow \text{True} \)
- \( (\geq x y) \rightarrow x \geq y \)
- \( (\geq a_1 a_2 \ldots a_n) \rightarrow a_1 \geq a_2 \geq \ldots \geq a_n \)

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

- \( (\gg x y) \rightarrow x \gg y \)
- \( (\gg a_1 a_2 \ldots a_n) \rightarrow a_1 \gg a_2 \gg \ldots \gg a_n \)

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

- \( (@ x y) \rightarrow x @ y \)
- \( (@ a_1 a_2 \ldots a_n) \rightarrow a_1 @ a_2 @ \ldots @ a_n \)

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

- \( (^ x y) \rightarrow x ^ y \)

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

- \( (\text{and}) \rightarrow \text{True} \)
- \( (\text{and} x) \rightarrow x \)
- \( (\text{and} x y) \rightarrow x \text{ and } y \)
- \( (\text{and} a_1 a_2 \ldots a_n) \rightarrow a_1 \text{ and } a_2 \text{ and } \ldots \text{ and } a_n \)

Helper for shadow comparison operators

Access item in \( coll \) indexed by \( key1 \), with optional \( keys \) nested-access.

\( \text{get} \) is used to access single elements in collections. \( \text{get} \) takes at least two parameters: the \textit{data structure} and the \textit{index} or \textit{key} of the item. It will then return the corresponding value from the collection. If multiple \textit{index} or \textit{key} values are provided, they are used to access successive elements in a nested structure.

\textbf{Note:} \( \text{get} \) raises a \texttt{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

```hy
=> (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
```

(hy.pyops.in(al, a2, #* a-rest))

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

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

(hy.pyops.is(al, #* a-rest))

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

- `(is x) → True`
- `(is x y) → x is y`
- `(is a1 a2 ... an) → a1 is a2 is ... is an`

(hy.pyops.not-in(al, a2, #* a-rest))

The negated membership test operator. Its effect can be defined by the equivalent Python:

- `(not-in x y) → x not in y`
- `(not-in a1 a2 ... an) → a1 not in a2 not in ... not in an`

(hy.pyops.not?(al, a2, #* a-rest))

The negated identicality test operator. Its effect can be defined by the equivalent Python:

- `(is-not x y) → x is not y`
- `(is-not a1 a2 ... an) → a1 is not a2 is not ... is not an`

(hy.pyops.or(#* args))

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

- `(or) → None`
- `(or x) → x`
- `(or x y) → x or y`
- `(or a1 a2 ... an) → a1 or a2 or ... or an`
(hy.pyops.reduce())

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.

(hy.pyops.|(#* args))

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

- (|) → 0
- (| x) → x
- (| x y) → x | y
- (| a1 a2 ... an) → a1 | a2 | ... | an

(hy.pyops.~(x))

The bitwise NOT operator. Its effect can be defined by the equivalent Python:

- (~ x) → ~x

### 12.5 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

```latex
\texttt{=> (import hy.extra.reserved)}
\texttt{=> (in "defclass" (hy.extra.reserved.names))}
\texttt{True}
```
13.1 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)!

13.2 Hack!

Do this:

1. Create a virtual environment:

$ virtualenv venv

and activate it:

$ . venv/bin/activate

or use virtualenvwrapper to create and manage your virtual environment:

$ mkvirtualenv hy
$ workon hy

2. Get the source code:

$ git clone https://github.com/hylang/hy.git

or use your fork:

$ git clone git@github.com:<YOUR_USERNAME>/hy.git

3. Install for hacking:

$ cd hy/
$ pip install -e .
4. Install other develop-y requirements:

   $ pip install -r requirements-dev.txt

5. 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 contains commits with large diffs but negligible semantic changes.

7. Do awesome things; make someone shriek in delight/disgust at what you have wrought.

### 13.3 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

### 13.4 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!
13.5 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.

13.5.1 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.

13.5.2 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

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.
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.

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

### 13.6 Contributor Code of Conduct

As contributors and maintainers of this project, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, or religion.

Examples of unacceptable behavior by participants include the use of sexual language or imagery, derogatory comments or personal attacks, trolling, public or private harassment, insults, or other unprofessional conduct.

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct. Project maintainers who do not follow the Code of Conduct may be removed from the project team.

This code of conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by opening an issue or contacting one or more of the project maintainers.

This Code of Conduct is adapted from the Contributor Covenant, version 1.1.0, available at [http://contributor-covenant.org/version/1/1/0/](http://contributor-covenant.org/version/1/1/0/).

### 13.7 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, 63
  hy.pyops, 74
  hy.reserved, 78
Symbols

!=() (in module hy.pyops), 74
*() (in module hy.pyops), 74
**() (in module hy.pyops), 74
+() (in module hy.pyops), 74
. (built-in variable), 44
//() (in module hy.pyops), 75
==() (in module hy.pyops), 75
@() (in module hy.pyops), 76
%=() (in module hy.pyops), 74
&() (in module hy.pyops), 74
__bool__() (hy.models.Keyword method), 17
__call__() (hy.models.Keyword method), 18
-() (in module hy.pyops), 75
--repl-output-fn
   command line option, 31
--spy
   command line option, 31
-c
   command line option, 31
-i
   command line option, 31
-m
   command line option, 31
-v
   command line option, 31
~() (in module hy.pyops), 78
|() (in module hy.pyops), 78
>() (in module hy.pyops), 76
>=() (in module hy.pyops), 76
>>() (in module hy.pyops), 76
<() (in module hy.pyops), 75
<=() (in module hy.pyops), 75
<<() (in module hy.pyops), 75
[x y]() (in module hy.pyops), 76

A

and() (in module hy.pyops), 76
annotate()
   built-in function, 43
as-model() (in module hy), 70
assert()
   built-in function, 48
await()
   built-in function, 46

B

break()
   built-in function, 46
built-in function
   annotate(), 43
assert(), 48
await(), 46
break(), 46
chainc(), 46
continue(), 47
cut(), 59
defclass(), 54
defmacro(), 45
defn(), 44
defn/a(), 45
del(), 55
dfor(), 51
do(), 47
eval-and-compile(), 50
eval-when-compile(), 50
fn(), 44
fn/a(), 44
for(), 47
get(), 49
gfor(), 51
global(), 48
if(), 46
import(), 49
let(), 53
lfor(), 51
match(), 53
nonlocal(), 55
py(), 55
pys(), 56
quasiquote(), 56
quote(), 56
raise(), 59
require(), 56
return(), 58
setv(), 52
setx(), 52
sfor(), 52
try(), 59
unpack-iterable(), 60
unpack-mapping(), 60
unquote(), 61
unquote-splice(), 61
while(), 61
with(), 62
with/a(), 63
yield(), 63
yield-from(), 63
Bytes (class in hy.models), 19

C
chainc()  
    built-in function, 46
chars() (hy.reader.reader.Reader method), 72
command line option
    --repl-output-fn, 31
    --spy, 31
    -c, 31
    -i, 31
    -m, 31
    -v, 31
comp-op() (in module hy.pyops), 76
Complex (class in hy.models), 17
cond() (in module hy.core.macros), 63
continue()
    built-in function, 47
cut()
    built-in function, 59

D
defclass()
    built-in function, 54
defmacro()
    built-in function, 45
defn()
    built-in function, 44
defn/a()
    built-in function, 45
defreader() (in module hy.core.macros), 64
del()
    built-in function, 55
delmacro() (in module hy.core.macros), 64
dfor()
    built-in function, 51
Dict (class in hy.models), 21
disassemble() (in module hy), 69
dispatch() (hy.reader.reader.Reader method), 72
do()  
    built-in function, 47
doc() (in module hy.core.macros), 65

E
end_identifier() (hy.reader.reader.Reader method), 73
ends_ident (hy.reader.reader.Reader attribute), 72
environment variable
    HY_COLORED_AST_OBJECTS, 29
    HY_COLORED_ERRORS, 29
    HY_DEBUG, 29
    HY_FILTER_INTERNAL_ERRORS, 29
    HY_HISTORY, 29
    HY_MESSAGE_WHEN_COMPILING, 29
    HYSTARTUP, 29
    PYTHONDONOTWRITEBYTECODE, 24
eval() (in module hy), 66
eval-and-compile()  
    built-in function, 50
eval-when-compile()  
    built-in function, 50
export() (in module hy.core.macros), 65
Expression (class in hy.models), 20

F
FComponent (class in hy.models), 21
fill_pos() (hy.reader.hy_reader.HyReader method), 71
Float (class in hy.models), 17
fn()
    built-in function, 44
fn/a()
    built-in function, 44
for()
    built-in function, 47
FString (class in hy.models), 21

G
gensym() (in module hy), 70
get()
    built-in function, 49
get() (in module hy.pyops), 76
getc() (hy.reader.reader.Reader method), 73
getn() (hy.reader.reader.Reader method), 73
gfor()
    built-in function, 51
global()
    built-in function, 48

H
hy.core.macros
    module, 63
hy.errors.COLORED (in module hy.core.macros), 71
Index

hy.pyops
  module, 74
hy.reserved
  module, 78
HyReader (class in hy.reader.hy_reader), 71
HyREPL (class in hy.cmdline), 27

I
if()
  built-in function, 46
import()
  built-in function, 49
in() (in module hy.pyops), 77
Integer (class in hy.models), 17
is() (in module hy.pyops), 77

K
Keyword (class in hy.models), 17

L
Lazy (class in hy.models), 15
let()
  built-in function, 53
lfor()
  built-in function, 51
List (class in hy.models), 20

M
macroexpand() (in module hy), 69
macroexpand-1() (in module hy), 70
macros() (in module hy.reserved), 78
mangle() (in module hy), 68
match()
  built-in function, 53
module
  hy.core.macros, 63
  hy.pyops, 74
  hy.reserved, 78

N
names() (in module hy.reserved), 78
nonlocal()
  built-in function, 55
not?() (in module hy.pyops), 77
not-in() (in module hy.pyops), 77

O
Object (class in hy.models), 15
or() (in module hy.pyops), 77

P
parse() (hy.reader.hy_reader.HyReader method), 71
parse_forms_until() (hy.reader.hy_reader.HyReader method), 71
parse_one_form() (hy.reader.hy_reader.HyReader method), 72
peek_and_getc() (hy.reader.reader.Reader method), 73
peekc() (hy.reader.reader.Reader method), 73
peeking() (hy.reader.reader.Reader method), 73
pos (hy.reader.reader.Reader attribute), 72
py()
  built-in function, 55
pys()
  built-in function, 56
Python Enhancement Proposals
  PEP 3107, 43
  PEP 3132, 61
  PEP 448, 61
  PEP 526, 43
  PEP 572, 52
  PEP 654, 60
PYTHONDONTWRITEBYTECODE, 24

Q
quasiquote()
  built-in function, 56
quote()
  built-in function, 56

R
raise()
  built-in function, 59
read() (in module hy), 66
read_default() (hy.reader.hy_reader.HyReader method), 72
read_ident() (hy.reader.reader.Reader method), 73
read_many() (in module hy), 66
Reader (class in hy.reader.reader), 72
reader_table (hy.reader.reader.Reader attribute), 72
reduce() (in module hy.pyops), 77
repr() (in module hy), 67
repr-register() (in module hy.pyops), 77
require()
  built-in function, 56
return()
  built-in function, 58
run() (hy.cmdline.HyREPL method), 27

S
saving_chars() (hy.reader.reader.Reader method), 74
Sequence (class in hy.models), 20
Set (class in hy.models), 20
setv()
  built-in function, 52
setx()
  built-in function, 52
sfor()  
   built-in function, 52
slurp_space() (hy.reader.reader.Reader method), 74
String (class in hy.models), 19
Symbol (class in hy.models), 18

T
try()  
   built-in function, 59
Tuple (class in hy.models), 20

U
unmangle() (in module hy), 69
unpack-iterable()  
   built-in function, 60
unpack-mapping()  
   built-in function, 60
unquote()  
   built-in function, 61
unquote-splice()  
   built-in function, 61

W
when() (in module hy.core.macros), 65
while()  
   built-in function, 61
with()  
   built-in function, 62
with/a()  
   built-in function, 63

Y
yield()  
   built-in function, 63
yield-from()  
   built-in function, 63