hy Documentation

Release 0.12.1

Paul Tagliamonte

January 24, 2017

Contents

Try Hy https://try-hy.appspot.com

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

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

List hylang-discuss

IRC `#hy` on Freenode

Build status

Hy is a wonderful dialect of Lisp that's embedded in Python.

Since Hy transforms its Lisp code into the Python Abstract Syntax Tree, you have the whole beautiful world of Python at your fingertips, in Lisp form!

CHAPTER 1

Documentation Index

Contents:

1.1 Quickstart

(Thanks to Karen Rustad for Cuddles!)

HOW TO GET HY REAL FAST:

1. Create a Virtual Python Environment.
2. Activate your Virtual Python Environment.

3. Install hy from GitHub with `$ pip install git+https://github.com/hylang/hy.git`.
4. Start a REPL with `hy`.
5. Type stuff in the REPL:

```
=> (print "Hy!")
Hy!
=> (defn salutationsnm [name] (print (+ "Hy " name "!")))
=> (salutationsnm "YourName")
Hy YourName!

etc
```

6. Hit CTRL-D when you're done.

OMG! That's amazing! I want to write a Hy program.

7. Open up an elite programming editor and type:

```
#! /usr/bin/env hy
(print "I was going to code in Python syntax, but then I got Hy.")
```

8. Save as `awesome.hy`.
9. Make it executable:

```
chmod +x awesome.hy
```

10. And run your first Hy program:

```
./awesome.hy
```

11. Take a deep breath so as to not hyperventilate.
12. Smile villainously and sneak off to your hydeaway and do unspeakable things.

1.2 Tutorial

Welcome to the Hy tutorial!

In a nutshell, Hy is a Lisp dialect, but one that converts its structure into Python ... literally a conversion into Python's abstract syntax tree! (Or to put it in more crude terms, Hy is lisp-stick on a Python!)

This is pretty cool because it means Hy is several things:

- A Lisp that feels very Pythonic
- For Lispers, a great way to use Lisp's crazy powers but in the wide world of Python's libraries (why yes, you now can write a Django application in Lisp!)
- For Pythonistas, a great way to start exploring Lisp, from the comfort of Python!
- For everyone: a pleasant language that has a lot of neat ideas!

1.2.1 Basic intro to Lisp for Pythonistas

Okay, maybe you've never used Lisp before, but you've used Python!

A "hello world" program in Hy is actually super simple. Let's try it:

```
(print "hello world")
```

See? Easy! As you may have guessed, this is the same as the Python version of:

```
print "hello world"
```

To add up some super simple math, we could do:

```
(+ 1 3)
```

Which would return 4 and would be the equivalent of:

```
1 + 3
```

What you'll notice is that the first item in the list is the function being called and the rest of the arguments are the arguments being passed in. In fact, in Hy (as with most Lisps) we can pass in multiple arguments to the plus operator:

```
(+ 1 3 55)
```

Which would return 59.

Maybe you've heard of Lisp before but don't know much about it. Lisp isn't as hard as you might think, and Hy inherits from Python, so Hy is a great way to start learning Lisp. The main thing that's obvious about Lisp is that there's a lot of parentheses. This might seem confusing at first, but it isn't so hard. Let's look at some simple math that's wrapped in a bunch of parentheses that we could enter into the Hy interpreter:

```
(setv result (- (/ (+ 1 3 88) 2) 8))
```

This would return 38. But why? Well, we could look at the equivalent expression in python:

```
result = ((1 + 3 + 88) / 2) - 8
```

If you were to try to figure out how the above were to work in python, you'd of course figure out the results by solving each inner parenthesis. That's the same basic idea in Hy. Let's try this exercise first in Python:

```
result = ((1 + 3 + 88) / 2) - 8
# simplified to...
result = (92 / 2) - 8
# simplified to...
result = 46 - 8
# simplified to...
result = 38
```

Now let's try the same thing in Hy:

```
(setv result (- (/ (+ 1 3 88) 2) 8))
; simplified to...
(setv result (- (/ 92 2) 8))
; simplified to...
(setv result (- 46 8))
; simplified to...
(setv result 38)
```

As you probably guessed, this last expression with `setv` means to assign the variable "result" to 38.

See? Not too hard!

This is the basic premise of Lisp. Lisp stands for "list processing"; this means that the structure of the program is actually lists of lists. (If you're familiar with Python lists, imagine the entire same structure as above but with square brackets instead, any you'll be able to see the structure above as both a program and a data structure.) This is easier to understand with more examples, so let's write a simple Python program, test it, and then show the equivalent Hy program:

```
def simple_conversation():
    print "Hello!  I'd like to get to know you.  Tell me about yourself!"
    name = raw_input("What is your name? ")
    age = raw_input("What is your age? ")
    print "Hello " + name + "!  I see you are " + age + " years old."

simple_conversation()
```

If we ran this program, it might go like:

```
Hello!  I'd like to get to know you.  Tell me about yourself!
What is your name? Gary
What is your age? 38
Hello Gary!  I see you are 38 years old.
```

Now let's look at the equivalent Hy program:

```
(defn simple-conversation []
   (print "Hello!  I'd like to get to know you.  Tell me about yourself!")
   (setv name (raw-input "What is your name? "))
   (setv age (raw-input "What is your age? "))
   (print (+ "Hello " name "!  I see you are "
             age " years old.")))

(simple-conversation)
```

If you look at the above program, as long as you remember that the first element in each list of the program is the function (or macro... we'll get to those later) being called and that the rest are the arguments, it's pretty easy to figure out what this all means. (As you probably also guessed, `defn` is the Hy method of defining methods.)

Still, lots of people find this confusing at first because there's so many parentheses, but there are plenty of things that can help make this easier: keep indentation nice and use an editor with parenthesis matching (this will help you figure out what each parenthesis pairs up with) and things will start to feel comfortable.

There are some advantages to having a code structure that's actually a very simple data structure as the core of Lisp is based on. For one thing, it means that your programs are easy to parse and that the entire actual structure of the program is very clearly exposed to you. (There's an extra step in Hy where the structure you see is converted to Python's own representations ... in "purer" Lisps such as Common Lisp or Emacs Lisp, the data structure you see in the code and the data structure that is executed is much more literally close.)

Another implication of this is macros: if a program's structure is a simple data structure, that means you can write code that can write code very easily, meaning that implementing entirely new language features can be very fast. Previous to Hy, this wasn't very possible for Python programmers ... now you too can make use of macros' incredible power (just be careful to not aim them footward)!

1.2.2 Hy is a Lisp-flavored Python

Hy converts to Python's own abstract syntax tree, so you'll soon start to find that all the familiar power of python is at your fingertips.

You have full access to Python's data types and standard library in Hy. Let's experiment with this in the hy interpreter:

```
=> [1 2 3]
[1, 2, 3]
=> {"dog" "bark"
... "cat" "meow"}
...
{'dog': 'bark', 'cat': 'meow'}
```

```
=> (, 1 2 3)
(1, 2, 3)
=> #{3 1 2}
{1, 2, 3}
=> 1/2
Fraction(1, 2)
```

Notice the last two lines: Hy has a fraction literal like Clojure.

If you are familiar with other Lisps, you may be interested that Hy supports the Common Lisp method of quoting:

```
=> '(1 2 3)
(1L 2L 3L)
```

You also have access to all the built-in types' nice methods:

```
=> (.strip " fooooo   ")
"fooooo"
```

What's this? Yes indeed, this is precisely the same as:

```
" fooooo   ".strip()
```

That's right—Lisp with dot notation! If we have this string assigned as a variable, we can also do the following:

```
(setv this-string " fooooo   ")
(this-string.strip)
```

What about conditionals?:

```
(if (try-some-thing)
  (print "this is if true")
  (print "this is if false"))
```

As you can tell above, the first argument to `if` is a truth test, the second argument is the body if true, and the third argument (optional!) is if false (ie. `else`).

If you need to do more complex conditionals, you'll find that you don't have `elif` available in Hy. Instead, you should use something called `cond`. In Python, you might do something like:

```
somevar = 33
if somevar > 50:
    print "That variable is too big!"
elif somevar < 10:
    print "That variable is too small!"
else:
    print "That variable is jusssssst right!"
```

In Hy, you would do:

```
(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 jusssssst right!")])
```

What you'll notice is that `cond` switches off between a statement that is executed and checked conditionally for true or falseness, and then a bit of code to execute if it turns out to be true. You'll also notice that the `else` is implemented

at the end simply by checking for `True` – that's because `True` will always be true, so if we get this far, we'll always run that one!

You might notice above that if you have code like:

```
(if some-condition
  (body-if-true)
  (body-if-false))
```

But wait! What if you want to execute more than one statement in the body of one of these?

You can do the following:

```
(if (try-some-thing)
  (do
    (print "this is if true")
    (print "and why not, let's keep talking about how true it is!"))
  (print "this one's still simply just false"))
```

You can see that we used `do` to wrap multiple statements. If you're familiar with other Lisps, this is the equivalent of `progn` elsewhere.

Comments start with semicolons:

```
(print "this will run")
; (print "but this will not")
(+ 1 2 3)  ; we'll execute the addition, but not this comment!
```

Hashbang (`#!`) syntax is supported:

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

Looping is not hard but has a kind of special structure. In Python, we might do:

```
for i in range(10):
    print "'i' is now at " + str(i)
```

The equivalent in Hy would be:

```
(for [i (range 10)]
  (print (+ "'i' is now at " (str i))))
```

You can also import and make use of various Python libraries. For example:

```
(import os)

(if (os.path.isdir "/tmp/somedir")
  (os.mkdir "/tmp/somedir/anotherdir")
  (print "Hey, that path isn't there!"))
```

Python's context managers (`with` statements) are used like this:

```
(with [f (open "/tmp/data.in")]
  (print (.read f)))
```

which is equivalent to:

```
with open("/tmp/data.in") as f:
    print f.read()
```

And yes, we do have List comprehensions! In Python you might do:

```
odds_squared = [
  pow(num, 2)
  for num in range(100)
  if num % 2 == 1]
```

In Hy, you could do these like:

```
(setv odds-squared
  (list-comp
    (pow num 2)
    (num (range 100))
    (= (% num 2) 1)))
```

```
; And, an example stolen shamelessly from a Clojure page:
; Let's list all the blocks of a Chessboard:

(list-comp
  (, x y)
  (x (range 8)
   y "ABCDEFGH"))

; [(0, 'A'), (0, 'B'), (0, 'C'), (0, 'D'), (0, 'E'), (0, 'F'), (0, 'G'), (0, 'H'),
;  (1, 'A'), (1, 'B'), (1, 'C'), (1, 'D'), (1, 'E'), (1, 'F'), (1, 'G'), (1, 'H'),
;  (2, 'A'), (2, 'B'), (2, 'C'), (2, 'D'), (2, 'E'), (2, 'F'), (2, 'G'), (2, 'H'),
;  (3, 'A'), (3, 'B'), (3, 'C'), (3, 'D'), (3, 'E'), (3, 'F'), (3, 'G'), (3, 'H'),
;  (4, 'A'), (4, 'B'), (4, 'C'), (4, 'D'), (4, 'E'), (4, 'F'), (4, 'G'), (4, 'H'),
;  (5, 'A'), (5, 'B'), (5, 'C'), (5, 'D'), (5, 'E'), (5, 'F'), (5, 'G'), (5, 'H'),
;  (6, 'A'), (6, 'B'), (6, 'C'), (6, 'D'), (6, 'E'), (6, 'F'), (6, 'G'), (6, 'H'),
;  (7, 'A'), (7, 'B'), (7, 'C'), (7, 'D'), (7, 'E'), (7, 'F'), (7, 'G'), (7, 'H')]
```

Python has support for various fancy argument and keyword arguments. In Python we might see:

```
>>> def optional_arg(pos1, pos2, keyword1=None, keyword2=42):
...   return [pos1, pos2, keyword1, keyword2]
...
>>> optional_arg(1, 2)
[1, 2, None, 42]
>>> optional_arg(1, 2, 3, 4)
[1, 2, 3, 4]
>>> optional_arg(keyword1=1, pos2=2, pos1=3, keyword2=4)
[3, 2, 1, 4]
```

The same thing in Hy:

```
=> (defn optional-arg [pos1 pos2 &optional keyword1 [keyword2 42]]
...  [pos1 pos2 keyword1 keyword2])
=> (optional-arg 1 2)
[1 2 None 42]
=> (optional-arg 1 2 3 4)
[1 2 3 4]
```

If you're running a version of Hy past 0.10.1 (eg, git master), there's also a nice new keyword argument syntax:

```
=> (optional-arg :keyword1 1
...              :pos2 2
...              :pos1 3
...              :keyword2 4)
[3, 2, 1, 4]
```

Otherwise, you can always use *apply*. But what's *apply*?

Are you familiar with passing in **args* and ***kwargs* in Python?:

```
>>> args = [1 2]
>>> kwargs = {"keyword2": 3
...           "keyword1": 4}
>>> optional_arg(*args, **kwargs)
```

We can reproduce this with *apply*:

```
=> (setv args [1 2])
=> (setv kwargs {"keyword2" 3
...              "keyword1" 4})
=> (apply optional-arg args kwargs)
[1, 2, 4, 3]
```

There's also a dictionary-style keyword arguments construction that looks like:

```
(defn another-style [&key {"key1" "val1" "key2" "val2"}]
  [key1 key2])
```

The difference here is that since it's a dictionary, you can't rely on any specific ordering to the arguments.

Hy also supports `*args` and `**kwargs`. In Python:

```
def some_func(foo, bar, *args, **kwargs):
  import pprint
  pprint.pprint((foo, bar, args, kwargs))
```

The Hy equivalent:

```
(defn some-func [foo bar &rest args &kwargs kwargs]
  (import pprint)
  (pprint.pprint (, foo bar args kwargs)))
```

Finally, of course we need classes! In Python, we might have a class like:

```
class FooBar(object):
    """
    Yet Another Example Class
    """
    def __init__(self, x):
        self.x = x

    def get_x(self):
        """
        Return our copy of x
        """
        return self.x
```

In Hy:

```
(defclass FooBar [object]
  "Yet Another Example Class"

  (defn --init-- [self x]
    (setv self.x x))

  (defn get-x [self]
    "Return our copy of x"
    self.x))
```

You can also do class-level attributes. In Python:

```
class Customer(models.Model):
    name = models.CharField(max_length=255)
    address = models.TextField()
    notes = models.TextField()
```

In Hy:

```
(defclass Customer [models.Model]
  [name (models.CharField :max-length 255})
   address (models.TextField)
   notes (models.TextField)])
```

1.2.3 Macros

One really powerful feature of Hy are macros. They are small functions that are used to generate code (or data). When program written in Hy is started, the macros are executed and their output is placed in the program source. After this, the program starts executing normally. Very simple example:

```
=> (defmacro hello [person]
...   `(print "Hello there," ~person))
=> (hello "Tuukka")
Hello there, Tuukka
```

The thing to notice here is that hello macro doesn't output anything on screen. Instead it creates piece of code that is then executed and prints on screen. This macro writes a piece of program that looks like this (provided that we used "Tuukka" as parameter):

```
(print "Hello there," Tuukka)
```

We can also manipulate code with macros:

```
=> (defmacro rev [code]
...   (let [op (last code) params (list (butlast code))]
...   `(~op ~@params)))
=> (rev (1 2 3 +))
6
```

The code that was generated with this macro just switched around some of the elements, so by the time program started executing, it actually reads:

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

Sometimes it's nice to have a very short name for a macro that doesn't take much space or use extra parentheses. Reader macros can be pretty useful in these situations (and since Hy operates well with unicode, we aren't running out of characters that soon):

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

Macros are useful when one wishes to extend Hy or write their own language on top of that. Many features of Hy are macros, like `when`, `cond` and `->`.

What if you want to use a macro that's defined in a different module? The special form `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 translate from Hy to Python. Instead, use `require`, which imports the module and makes macros available at compile-time. `require` uses the same syntax as `import`.

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

1.2.4 Hy <-> Python interop

By importing Hy, you can use Hy directly from Python!

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

```
(defn greet [name] (print "hello from hy," name))
```

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

```
import hy
import greetings

greetings.greet("Foo")
```

You can also declare a function in python (or even a class!) and use it in Hy!

If you save the following in `greetings.py` in Python:

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

You can use it in Hy:

```
(import greetings)
(.greet greetings "foo")
```

To use keyword arguments, you can use in `greetings.py`:

```
def greet(name, title="Sir"):
    print("Greetings, %s %s" % (title,name))
```

```
(import greetings)
(.greet greetings "Foo")
(.greet greetings "Foo" "Darth")
(apply (. greetings greet) ["Foo"] {:title "Lord"})
```

Which would output:

```
Greetings, Sir Foo

Greetings, Darth Foo

Greetings, Lord Foo
```

1.2.5 Protips!

Hy also features something known as the "threading macro", a really neat feature of Clojure's. The "threading macro" (written as `->`) is used to avoid deep nesting of expressions.

The threading macro inserts each expression into the next expression's first argument place.

Let's take the classic:

```
(loop (print (eval (read))))
```

Rather than write it like that, we can write it as follows:

```
(-> (read) (eval) (print) (loop))
```

Now, using python-sh, we can show how the threading macro (because of python-sh's setup) can be used like a pipe:

```
=> (import [sh [cat grep wc]])
=> (-> (cat "/usr/share/dict/words") (grep "-E" "^hy") (wc "-l"))
210
```

Which, of course, expands out to:

```
(wc (grep (cat "/usr/share/dict/words") "-E" "^hy") "-l")
```

Much more readable, no? Use the threading macro!

1.3 Hy Style Guide

> "You know, Minister, I disagree with Dumbledore on many counts…but you cannot deny he's got style…" — Phineas Nigellus Black, *Harry Potter and the Order of the Phoenix*

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

1.3.1 Prelude

The Tao of Hy

```
Ummon asked the head monk, "What sutra are you lecturing on?"
"The Nirvana Sutra."
"The Nirvana Sutra has the Four Virtues, hasn't it?"
"It has."
Ummon asked, picking up a cup, "How many virtues has this?"
"None at all," said the monk.
"But ancient people said it had, didn't they?" said Ummon.
"What do you think of what they said?"
Ummon struck the cup and asked, "You understand?"
"No," said the monk.
"Then," said Ummon, "You'd better go on with your lectures on the sutra."
-- the (koan) macro
```

The following illustrates a brief list of design decisions that went into the making of Hy.

- Look like a Lisp; DTRT with it (e.g. dashes turn to underscores, earmuffs turn to all-caps).
- We're still Python. Most of the internals translate 1:1 to Python internals.
- Use Unicode everywhere.
- Fix the bad decisions in Python 2 when we can (see `true_division`).
- When in doubt, defer to Python.
- If you're still unsure, defer to Clojure.

- If you're even more unsure, defer to Common Lisp.
- Keep in mind we're not Clojure. We're not Common Lisp. We're Homoiconic Python, with extra bits that make sense.

1.3.2 Layout & Indentation

- Avoid trailing spaces. They suck!
- Indentation shall be 2 spaces (no hard tabs), except when matching the indentation of the previous line.

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

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

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

;; Hysterically ridiculous
(defn fib [n]
    (if (<= n 2)
            n ;; yes, I love randomly hitting the space key
      (+ (fib (- n 1)) (fib (- n 2)))))
```

- Parentheses must *never* be left alone, sad and lonesome on their own line.

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

;; Hysterically ridiculous
(defn fib [n]
  (if (<= n 2)
    n
    (+ (fib (- n 1)) (fib (- n 2)))
  )
)  ; GAH, BURN IT WITH FIRE
```

- Vertically align `let` blocks.

```
(let [foo (bar)
      qux (baz)]
   (foo qux))
```

- Inline comments shall be two spaces from the end of the code; they must always have a space between the comment character and the start of the comment. Also, try to not comment the obvious.

```
;; Good
(setv ind (dec x))  ; indexing starts from 0

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

;; Bad
(setv ind (dec x));typing words for fun
```

1.3.3 Coding Style

- As a convention, try not to use `def` for anything other than global variables; use `setv` inside functions, loops, etc.

```
;; Good (and preferred)
(def *limit* 400000)

(defn fibs [a b]
  (while True
    (yield a)
    (setv (, a b) (, b (+ a b)))))

;; Bad (and not preferred)
(defn fibs [a b]
  (while True
    (yield a)
    (def (, a b) (, b (+ a b)))))
```

- Do not use s-expression syntax where vector syntax is intended. For instance, the fact that the former of these two examples works is just because the compiler isn't overly strict. In reality, the correct syntax in places such as this is the latter.

```
;; Bad (and evil)
(defn foo (x) (print x))
(foo 1)

;; Good (and preferred)
(defn foo [x] (print x))
(foo 1)
```

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

```
;; Preferred
(def *names*
  (with [f (open "names.txt")]
    (-> (.read f) (.strip) (.replace "\"" "") (.split ",") (sorted))))

;; Not so good
(def *names*
  (with [f (open "names.txt")]
  (sorted (.split "," (.replace "\"" "" (.strip (.read f)))))))

;; Probably not a good idea
(defn square? [x]
  (->> 2 (pow (int (sqrt x))) (= x)))
```

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

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

;; Not so good
(with [fd (open "/etc/passwd")]
  (print (fd.readlines)))
```

1.3.4 Conclusion

> "Fashions fade, style is eternal" —Yves Saint Laurent

This guide is just a set of community guidelines, and obviously, community guidelines do not make sense without an active community. Contributions are welcome. Join us at #hy in freenode, blog about it, tweet about it, and most importantly, have fun with Hy.

1.3.5 Thanks

- This guide is heavily inspired from @paultag 's blog post Hy Survival Guide
- The Clojure Style Guide

1.4 Documentation Index

Contents:

1.4.1 Command Line Interface

hy

Command Line Options

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

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

-i <command>
Execute the Hy code in *command*, then stay in REPL.

-m <module>
Execute the Hy code in *module*, including `defmain` if defined.

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

New in version 0.11.0.

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

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

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

--show-tracebacks
Print extended tracebacks for Hy exceptions.

New in version 0.9.12.

-v
Print the Hy version number and exit.

hyc

Command Line Options

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

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

(hy-hy "Afroman")
```

Then run:

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

hy2py

New in version 0.10.1.

Command Line Options

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

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

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

1.4.2 Hy (the language)

Warning: This is incomplete; please consider contributing to the documentation effort.

Theory of Hy

Hy maintains, over everything else, 100% compatibility in both directions with Python itself. All Hy code follows a few simple rules. Memorize this, as it's going to come in handy.

These rules help ensure that Hy code is idiomatic and interfaceable in both languages.

- Symbols in earmufs will be translated to the upper-cased version of that string. For example, `foo` will become `FOO`.
- UTF-8 entities will be encoded using punycode and prefixed with `hy_`. For instance, `FLOWER` will become `hy_w7h`, `HEART` will become `hy_g6h`, and `iHEARTu` will become `hy_iu_t0x`.
- Symbols that contain dashes will have them replaced with underscores. For example, `render-template` will become `render_template`. This means that symbols with dashes will shadow their underscore equivalents, and vice versa.

Notes on Syntax

integers

New in version 0.11.1.

In addition to regular numbers, standard notation from Python 3 for non-base 10 integers is used. `0x` for Hex, `0o` for Octal, `0b` for Binary.

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

Built-Ins

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

.

New in version 0.10.0.

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

For instance,

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

Compiles down to:

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

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

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

->

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

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

->>

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

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

apply

`apply` is used to apply an optional list of arguments and an optional dictionary of kwargs to a function. The symbol mangling transformations will be applied to all keys in the dictionary of kwargs, provided the dictionary and its keys are defined in-place.

Usage: `(apply fn-name [args] [kwargs])`

Examples:

```
(defn thunk []
  "hy there")

(apply thunk)
;=> "hy there"

(defn total-purchase [price amount &optional [fees 1.05] [vat 1.1]]
  (* price amount fees vat))

(apply total-purchase [10 15])
;=> 173.25

(apply total-purchase [10 15] {"vat" 1.05})
;=> 165.375

(apply total-purchase [] {"price" 10 "amount" 15 "vat" 1.05})
;=> 165.375

(apply total-purchase [] {:price 10 :amount 15 :vat 1.05})
;=> 165.375
```

and

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

```
=> (and True False)
False

=> (and True True)
True

=> (and True 1)
1

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

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

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

as->

New in version 0.12.0.

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

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

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

=> (-> 0 inc inc)
2

;; create data for our cuttlefish database

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

;; retrieve name of first entry
```

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

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

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

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

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

assert

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

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

(assert False)
; AssertionError

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

assoc

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

Examples of usage:

```
=>(let [collection {}]
... (assoc collection "Dog" "Bark")
... (print collection))
{u'Dog': u'Bark'}

=>(let [collection {}]
... (assoc collection "Dog" "Bark" "Cat" "Meow")
... (print collection))
{u'Cat': u'Meow', u'Dog': u'Bark'}

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

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

break

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

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

cond

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

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

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

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

```
=> (defn check-value [value]
...   (cond [(< value 5) (print "value is smaller than 5")]
...         [(= value 5) (print "value is equal to 5")]
...         [(> value 5) (print "value is greater than 5")]
...         [True (print "value is something that it should not be")]))

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

continue

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

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

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

dict-comp

`dict-comp` is used to create dictionaries. It takes three or four parameters. The first two parameters are for controlling the return value (key-value pair) while the third is used to select items from a sequence. The fourth and optional parameter can be used to filter out some of the items in the sequence based on a conditional expression.

```
=> (dict-comp x (* x 2) [x (range 10)] (odd? x))
{1: 2, 3: 6, 9: 18, 5: 10, 7: 14}
```

do

`do` is used to evaluate each of its arguments and return the last one. Return values from every other than the last argument are discarded. It can be used in `lambda` or `list-comp` to perform more complex logic as shown in one of the following examples.

Some example usage:

```
=> (if True
...   (do (print "Side effects rock!")
...       (print "Yeah, really!")))
Side effects rock!
Yeah, really!

;; assuming that (side-effect) is a function that we want to call for each
;; and every value in the list, but whose return value we do not care about
=> (list-comp (do (side-effect x)
...                 (if (< x 5) (* 2 x)
...                     (* 4 x)))
...             (x (range 10)))
[0, 2, 4, 6, 8, 20, 24, 28, 32, 36]
```

`do` can accept any number of arguments, from 1 to n.

def / setv

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

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

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

They can be used to assign multiple variables at once:

```
=> (setv a 1 b 2)
(1L, 2L)
=> a
1L
=> b
2L
=>
```

defclass

New classes are declared with `defclass`. It can takes two optional parameters: a vector defining a possible super classes and another vector containing attributes of the new class as two item vectors.

```
(defclass class-name [super-class-1 super-class-2]
  [attribute value]

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

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

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

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

defn

`defn` macro is used to define functions. It takes three parameters: the *name* of the function to define, a vector of *parameters*, and the *body* of the function:

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

Parameters may have the following keywords in front of them:

&optional Parameter is optional. The parameter can be given as a two item list, where the first element is parameter name and the second is the default value. The parameter can be also given as a single item, in which case the default value is `None`.

```
=> (defn total-value [value &optional [value-added-tax 10]]
...   (+ (/ (* value value-added-tax) 100) value))

=> (total-value 100)
110.0
```

```
=> (total-value 100 1)
101.0
```

&key Parameter is a dict of keyword arguments. The keys of the dict specify the parameter names and the values give the default values of the parameters.

```
=> (defn key-parameters [&key {"a" 1 "b" 2}]
... (print "a is" a "and b is" b))
=> (key-parameters :a 1 :b 2)
a is 1 and b is 2
=> (key-parameters :b 1 :a 2)
a is 2 and b is 1
```

The following declarations are equivalent:

```
(defn key-parameters [&key {"a" 1 "b" 2}])

(defn key-parameters [&optional [a 1] [b 2]])
```

&kwargs Parameter will contain 0 or more keyword arguments.

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

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

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

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

&rest Parameter will contain 0 or more positional arguments. No other positional arguments may be specified after this one.

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

```
=> (defn zig-zag-sum [&rest numbers]
     (let [odd-numbers (list-comp x [x numbers] (odd? x))
           even-numbers (list-comp x [x numbers] (even? x))]
       (- (sum odd-numbers) (sum even-numbers))))

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

&kwonly New in version 0.12.0.

Parameters that can only be called as keywords. Mandatory keyword-only arguments are declared with the argument's name; optional keyword-only arguments are declared as a two-element list containing the argument name followed by the default value (as with *&optional* above).

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

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

Availability: Python 3.

defmain

New in version 0.10.1.

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

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

is the equivalent of:

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

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

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

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

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

```
(import argparse)

(defmain [&rest _]
  (setv parser (argparse.ArgumentParser))
  (.add-argument parser "STRING"
    :help "string to replicate")
```

```
  (.add-argument parser "-n" :type int :default 3
    :help "number of copies")
  (setv args (parser.parse_args))

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

  0)
```

defmacro

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

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

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

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

defmacro/g!

New in version 0.9.12.

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

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

See also:

Section *Using gensym for Safer Macros*

defmacro!

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

For example,

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

```
=> (triple-3 (expensive-get-number))  ; easier with defmacro!
spam
42
```

defreader

New in version 0.9.12.

`defreader` defines a reader macro, enabling you to restructure or modify syntax.

```
=> (defreader ^ [expr] (print expr))
=> #^(1 2 3 4)
(1 2 3 4)
=> #^"Hello"
"Hello"
```

See also:

Section *Reader Macros*

del

New in version 0.9.12.

`del` removes an object from the current namespace.

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

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

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

doto

New in version 0.10.1.

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

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

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

eval

`eval` evaluates a quoted expression and returns the value. The optional second and third arguments specify the dictionary of globals to use and the module name. The globals dictionary defaults to `(local)` and the module name defaults to the name of the current module.

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

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

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

eval-and-compile

eval-when-compile

first / car

`first` and `car` are macros for accessing the first element of a collection:

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

for

`for` is used to call a function for each element in a list or vector. The results of each call are discarded and the `for` expression returns `None` instead. The example code iterates over *collection* and for each *element* in *collection* calls the `side-effect` function with *element* as its argument:

```
;; assuming that (side-effect) is a function that takes a single parameter
(for [element collection] (side-effect element))

;; for can have an optional else block
(for [element collection] (side-effect element)
     (else (side-effect-2)))
```

The optional `else` block is only executed if the `for` loop terminates normally. If the execution is halted with `break`, the `else` block does not execute.

```
=> (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
```

genexpr

`genexpr` is used to create generator expressions. It takes two or three parameters. The first parameter is the expression controlling the return value, while the second is used to select items from a list. The third and optional parameter can be used to filter out some of the items in the list based on a conditional expression. `genexpr` is similar to `list-comp`, except it returns an iterable that evaluates values one by one instead of evaluating them immediately.

```
=> (def collection (range 10))
=> (def filtered (genexpr x [x collection] (even? x)))
=> (list filtered)
[0, 2, 4, 6, 8]
```

gensym

New in version 0.9.12.

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

```
=> (gensym)
u':G_1235'

=> (gensym "x")
u':x_1236'
```

See also:

Section *Using gensym for Safer Macros*

get

`get` is used to access single elements in lists and dictionaries. `get` takes two parameters: the *data structure* and the *index* or *key* of the item. It will then return the corresponding value from the dictionary or the list. Example usage:

```
=> (let [animals {"dog" "bark" "cat" "meow"}
...      numbers ["zero" "one" "two" "three"]]
...   (print (get animals "dog"))
...   (print (get numbers 2)))
bark
two
```

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.

global

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

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

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

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

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

if / if* / if-not

New in version 0.10.0: if-not

`if / if* / if-not` respect Python *truthiness*, that is, a *test* fails if it evaluates to a "zero" (including values of `len` zero, `None`, and `False`), and passes otherwise, but values with a `__bool__` method (`__nonzero__` in Python 2) can overrides this.

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

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

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

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

Example usage:

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

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

(if-not (money-left? account)
```

```
  (print "let's go and work")
  (print "let's go shopping"))
```

lif and lif-not

New in version 0.10.0.

New in version 0.11.0: lif-not

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

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

import

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

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

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

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

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

```
                  isfile :as file?]]
        [sys :as systest])

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

lambda / fn

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

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

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

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

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

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

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

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

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

last

New in version 0.11.0.

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

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

let

`let` is used to create lexically scoped variables. They are created at the beginning of the `let` form and cease to exist after the form. The following example showcases this behaviour:

```
=> (let [x 5] (print x)
...   (let [x 6] (print x))
...   (print x))
5
6
5
```

The `let` macro takes two parameters: a vector defining *variables* and the *body* which gets executed. *variables* is a vector of variable and value pairs.

Note that the variable assignments are executed one by one, from left to right. The following example takes advantage of this:

```
=> (let [x 5
         y (+ x 1)] (print x y))
5 6
```

list-comp

`list-comp` performs list comprehensions. It takes two or three parameters. The first parameter is the expression controlling the return value, while the second is used to select items from a list. The third and optional parameter can be used to filter out some of the items in the list based on a conditional expression. Some examples:

```
=> (def collection (range 10))
=> (list-comp x [x collection])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

=> (list-comp (* x 2) [x collection])
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

=> (list-comp (* x 2) [x collection] (< x 5))
[0, 2, 4, 6, 8]
```

nonlocal

New in version 0.11.1.

PYTHON 3.0 AND UP ONLY!

`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 `let` or `fn` scopes:

```
(let [x 0]
  (for [y (range 10)]
    (let [z (inc y)]
      (nonlocal x)  ; allow the setv to "jump scope" to resolve x
      (setv x (+ x y))))
  x)

(defn some-function []
  (let [x 0]
    (register-some-callback
```

```
      (fn [stuff]
        (nonlocal x)
        (setv x stuff)))))
```

In the first example, without the call to `(nonlocal x)`, this code would result in an UnboundLocalError being raised during the call to `setv`.

In the second example, 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.

not

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

```
=> (not True)
False

=> (not False)
True

=> (not None)
True
```

or

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

```
=> (or True False)
True

=> (and False False)
False

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

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

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

print

`print` is used to output on screen. Example usage:

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

Note: `print` always returns `None`.

quasiquote

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

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

quote

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

```
=> (setv x '(print "Hello World"))
; variable x is set to expression & not evaluated
=> x
(u'print' u'Hello World')
=> (eval x)
Hello World
```

require

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

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

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

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

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

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

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

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

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

```
  `(list (map (fn [_] ~expr) (range ~n))))

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

And then, in your main program, you write:

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

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

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

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

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

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

rest / cdr

`rest` and `cdr` return the collection passed as an argument without the first element:

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

set-comp

`set-comp` is used to create sets. It takes two or three parameters. The first parameter is for controlling the return value, while the second is used to select items from a sequence. The third and optional parameter can be used to filter out some of the items in the sequence based on a conditional expression.

```
=> (setv data [1 2 3 4 5 2 3 4 5 3 4 5])
=> (set-comp x [x data] (odd? x))
{1, 3, 5}
```

cut

`cut` can be used to take a subset of a list and create a new list from it. The form takes at least one parameter specifying the list to cut. Two optional parameters can be used to give the start and end position of the subset. If they are not

supplied, the default value of `None` will be used instead. The third optional parameter is used to control step between the elements.

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

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

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

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

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

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

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

raise

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

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

(raise IOError)
; raise an IOError

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

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

try

The `try` form is used to start a `try` / `except` block. The form is used as follows:

```
(try
    (error-prone-function)
    (except [e ZeroDivisionError] (print "Division by zero"))
    (else (print "no errors"))
    (finally (print "all done")))
```

`try` must contain at least one `except` block, and may optionally include an `else` or `finally` block. If an error is raised with a matching except block during the execution of `error-prone-function`, that `except` block will be executed. If no errors are raised, the `else` block is executed. The `finally` block will be executed last regardless of whether or not an error was raised.

unless

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

```
(unless conditional statement)

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

unquote

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

```
(def name "Cuddles")
(quasiquote (= name (unquote name)))
;=> (u'=' u'name' u'Cuddles')

`(= name ~name)
;=> (u'=' u'name' u'Cuddles')
```

unquote-splice

`unquote-splice` forces the evaluation of a symbol within a quasiquoted form, much like `unquote`. `unquote-splice` can only be used when the symbol being unquoted contains an iterable value, as it "splices" that iterable into the quasiquoted form. `unquote-splice` is aliased to the `~@` symbol.

```
(def nums [1 2 3 4])
(quasiquote (+ (unquote-splice nums)))
;=> (u'+' 1L 2L 3L 4L)

`(+ ~@nums)
;=> (u'+' 1L 2L 3L 4L)
```

when

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

```
(when conditional statement)

(if conditional (do statement))
```

while

`while` is used to execute one or more blocks as long as a condition is met. The following example will output "Hello world!" to the screen indefinitely:

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

with

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

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

(with [(expr)] block)

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

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

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

with-decorator

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

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

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

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

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

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

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

#@ New in version 0.12.0.

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

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

with-gensyms

New in version 0.9.12.

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

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

expands to:

```
(let [a (gensym)
      b (gensym)
      c (gensym)]
  ...)
```

See also:

Section *Using gensym for Safer Macros*

xor

New in version 0.12.0.

`xor` is used in logical expressions to perform exclusive or. It takes two parameters. It returns `True` if only of the parameters is `True`. In all other cases `False` is returned. Example usage:

```
=> (xor True False)
True

=> (xor True True)
False

=> (xor [] [0])
True
```

yield

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

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

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

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

=> (list-comp value [value (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-comp x [x (take 15 (random-numbers 1 50))])
[7, 41, 6, 22, 32, 17, 5, 38, 18, 38, 17, 14, 23, 23, 19]
```

yield-from

New in version 0.9.13.

PYTHON 3.3 AND UP ONLY!

`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.

1.4.3 Hy Core

Core Functions

butlast

Usage: `(butlast coll)`

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

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

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

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

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

coll?

New in version 0.10.0.

Usage: `(coll?  x)`

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

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

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

```
True

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

comp

Usage: `(comp f g)`

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

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

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

complement

New in version 0.12.0.

Usage: `(complement f)`

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

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

cons

New in version 0.10.0.

Usage: `(cons a b)`

Returns a fresh *cons cell* with car *a* and cdr *b*.

```
=> (setv a (cons 'hd 'tl))

=> (= 'hd (car a))
True

=> (= 'tl (cdr a))
True
```

cons?

New in version 0.10.0.

Usage: `(cons?  foo)`

Checks whether *foo* is a *cons cell*.

```
=> (setv a (cons 'hd 'tl))

=> (cons? a)
True

=> (cons? None)
False

=> (cons? [1 2 3])
False
```

constantly

New in version 0.12.0.

Usage `(constantly 42)`

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

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

dec

Usage: `(dec x)`

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

```
=> (dec 3)
2

=> (dec 0)
-1

=> (dec 12.3)
11.3
```

disassemble

New in version 0.10.0.

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

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

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

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

empty?

Usage: `(empty? coll)`

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

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

=> (empty? "")
True

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

every?

New in version 0.10.0.

Usage: `(every? pred coll)`

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

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

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

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

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

float?

Usage: `(float? x)`

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

```
=> (float? 3.2)
True
```

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

fraction

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

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

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

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

even?

Usage: `(even? x)`

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

```
=> (even? 2)
True

=> (even? 13)
False

=> (even? 0)
True
```

identity

Usage: `(identity x)`

Returns the argument supplied to the function.

```
=> (identity 4)
4

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

inc

Usage: `(inc x)`

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

```
=> (inc 3)
4

=> (inc 0)
1
```

```
=> (inc 12.3)
13.3
```

instance?

Usage: `(instance? class x)`

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

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

=> (instance? int 7)
True

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

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

integer?

Usage: `(integer? x)`

Returns *True* if *x* is an integer. For Python 2, this is either `int` or `long`. For Python 3, this is `int`.

```
=> (integer? 3)
True

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

interleave

New in version 0.10.1.

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

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

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

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

interpose

New in version 0.10.1.

Usage: `(interpose item seq)`

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

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

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

iterable?

Usage: `(iterable? x)`

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

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

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

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

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

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

iterator?

Usage: `(iterator? x)`

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

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

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

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

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

juxt

New in version 0.12.0.

Usage: `(juxt f &rest fs)`

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

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

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

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

keyword

New in version 0.10.1.

Usage: `(keyword "foo")`

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

```
=> (keyword "foo")
u'\ufdd0:foo'

=> (keyword 1)
u'\ufdd0:1'
```

keyword?

New in version 0.10.1.

Usage: `(keyword? foo)`

Check whether *foo* is a *keyword*.

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

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

list*

Usage: `(list* head &rest tail)`

Generates a chain of nested cons cells (a dotted list) containing the arguments. If the argument list only has one element, return it.

```
=> (list* 1 2 3 4)
(1 2 3 . 4)

=> (list* 1 2 3 [4])
[1, 2, 3, 4]

=> (list* 1)
1

=> (cons? (list* 1 2 3 4))
True
```

macroexpand

New in version 0.10.0.

Usage: `(macroexpand form)`

Returns the full macro expansion of *form*.

```
=> (macroexpand '(-> (a b) (x y)))
(u'x' (u'a' u'b') u'y')

=> (macroexpand '(-> (a b) (-> (c d) (e f))))
(u'e' (u'c' (u'a' u'b') u'd') u'f')
```

macroexpand-1

New in version 0.10.0.

Usage: `(macroexpand-1 form)`

Returns the single step macro expansion of *form*.

```
=> (macroexpand-1 '(-> (a b) (-> (c d) (e f))))
(u'_>' (u'a' u'b') (u'c' u'd') (u'e' u'f'))
```

merge-with

New in version 0.10.1.

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

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

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

name

New in version 0.10.1.

Usage: `(name :keyword)`

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

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

neg?

Usage: `(neg? x)`

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

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

=> (neg? 3)
False

=> (neg? 0)
False
```

none?

Usage: `(none? x)`

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

```
=> (none? None)
True

=> (none? 0)
False

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

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

nth

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

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

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

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

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

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

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

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

numeric?

Usage: `(numeric? x)`

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

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

=> (numeric? 3.2)
True

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

odd?

Usage: `(odd? x)`

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

```
=> (odd? 13)
True

=> (odd? 2)
False

=> (odd? 0)
False
```

partition

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

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

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

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

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

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

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

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

pos?

Usage: `(pos? x)`

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

```
=> (pos? 3)
True

=> (pos? -2)
False

=> (pos? 0)
False
```

second

Usage: `(second coll)`

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

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

some

New in version 0.10.0.

Usage: `(some pred coll)`

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

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

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

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

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

=> (none? (some even? []))
True
```

string?

Usage: `(string? x)`

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

```
=> (string? "foo")
True

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

symbol?

Usage: `(symbol? x)`

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

```
=> (symbol? 'foo)
True

=> (symbol? '[a b c])
False
```

zero?

Usage: `(zero? x)`

Returns `True` if *x* is zero.

```
=> (zero? 3)
False

=> (zero? -2)
False

=> (zero? 0)
True
```

Sequence Functions

Sequence functions can either create or operate on a potentially infinite sequence without requiring the sequence be fully realized in a list or similar container. They do this by returning a Python iterator.

We can use the canonical infinite Fibonacci number generator as an example of how to use some of these functions.

```
(defn fib []
  (setv a 0)
  (setv b 1)
  (while True
    (yield a)
    (setv (, a b) (, b (+ a b)))))
```

Note the `(while True ...)` loop. If we run this in the REPL,

```
=> (fib)
<generator object fib at 0x101e642d0>
```

Calling the function only returns an iterator, but does no work until we consume it. Trying something like this is not recommend as the infinite loop will run until it consumes all available RAM, or in this case until I killed it.

```
=> (list (fib))
[1]    91474 killed     hy
```

To get the first 10 Fibonacci numbers, use *take*. Note that *take* also returns a generator, so I create a list from it.

```
=> (list (take 10 (fib)))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

To get the Fibonacci number at index 9, (starting from 0):

```
=> (nth (fib) 9)
34
```

cycle

Usage: `(cycle coll)`

Returns an infinite iterator of the members of coll.

```
=> (list (take 7 (cycle [1 2 3])))
[1, 2, 3, 1, 2, 3, 1]

=> (list (take 2 (cycle [1 2 3])))
[1, 2]
```

distinct

Usage: `(distinct coll)`

Returns an iterator containing only the unique members in *coll*.

```
=> (list (distinct [ 1 2 3 4 3 5 2 ]))
[1, 2, 3, 4, 5]

=> (list (distinct []))
[]

=> (list (distinct (iter [ 1 2 3 4 3 5 2 ])))
[1, 2, 3, 4, 5]
```

drop

Usage: `(drop n coll)`

Returns an iterator, skipping the first *n* members of *coll*. Raises `ValueError` if *n* is negative.

```
=> (list (drop 2 [1 2 3 4 5]))
[3, 4, 5]

=> (list (drop 4 [1 2 3 4 5]))
```

```
[5]

=> (list (drop 0 [1 2 3 4 5]))
[1, 2, 3, 4, 5]

=> (list (drop 6 [1 2 3 4 5]))
[]
```

drop-last

Usage: `(drop-last n coll)`

Returns an iterator of all but the last *n* items in *coll*. Raises `ValueError` if *n* is negative.

```
=> (list (drop-last 5 (range 10 20)))
[10, 11, 12, 13, 14]

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

=> (list (drop-last 100 (range 100)))
[]

=> (list (take 5 (drop-last 100 (count 10))))
[10, 11, 12, 13, 14]
```

drop-while

Usage: `(drop-while pred coll)`

Returns an iterator, skipping members of *coll* until *pred* is `False`.

```
=> (list (drop-while even? [2 4 7 8 9]))
[7, 8, 9]

=> (list (drop-while numeric? [1 2 3 None "a"]))
[None, u'a']

=> (list (drop-while pos? [2 4 7 8 9]))
[]
```

filter

Usage: `(filter pred coll)`

Returns an iterator for all items in *coll* that pass the predicate *pred*.

See also *remove*.

```
=> (list (filter pos? [1 2 3 -4 5 -7]))
[1, 2, 3, 5]

=> (list (filter even? [1 2 3 -4 5 -7]))
[2, -4]
```

flatten

New in version 0.9.12.

Usage: `(flatten coll)`

Returns a single list of all the items in *coll*, by flattening all contained lists and/or tuples.

```
=> (flatten [1 2 [3 4] 5])
[1, 2, 3, 4, 5]

=> (flatten ["foo" (, 1 2) [1 [2 3] 4] "bar"])
['foo', 1, 2, 1, 2, 3, 4, 'bar']
```

iterate

Usage: `(iterate fn x)`

Returns an iterator of *x*, *fn(x)*, *fn(fn(x))*, etc.

```
=> (list (take 5 (iterate inc 5)))
[5, 6, 7, 8, 9]

=> (list (take 5 (iterate (fn [x] (* x x)) 5)))
[5, 25, 625, 390625, 152587890625]
```

read

Usage: `(read &optional [from-file eof])`

Reads the next Hy expression from *from-file* (defaulting to `sys.stdin`), and can take a single byte as EOF (defaults to an empty string). Raises `EOFError` if *from-file* ends before a complete expression can be parsed.

```
=> (read)
(+ 2 2)
('+' 2 2)
=> (eval (read))
(+ 2 2)
4

=> (import io)
=> (def buffer (io.StringIO "(+ 2 2)\n(- 2 1)"))
=> (eval (apply read [] {"from_file" buffer}))
4
=> (eval (apply read [] {"from_file" buffer}))
1

=> ; assuming "example.hy" contains:
=> ;   (print "hello")
=> ;   (print "hyfriends!")
=> (with [f (open "example.hy")]
...   (try
...     (while True
...            (let [exp (read f)]
...              (do
...                (print "OHY" exp)
...                (eval exp))))
```

```
...     (except [e EOFError]
...             (print "EOF!"))))
OHY ('print' 'hello')
hello
OHY ('print' 'hyfriends!')
hyfriends!
EOF!
```

read-str

Usage: `(read-str "string")`

This is essentially a wrapper around *read* which reads expressions from a string:

```
=> (read-str "(print 1)")
(u'print' 1L)
=> (eval (read-str "(print 1)"))
1
=>
```

remove

Usage: `(remove pred coll)`

Returns an iterator from *coll* with elements that pass the predicate, *pred*, removed.

See also *filter*.

```
=> (list (remove odd? [1 2 3 4 5 6 7]))
[2, 4, 6]

=> (list (remove pos? [1 2 3 4 5 6 7]))
[]

=> (list (remove neg? [1 2 3 4 5 6 7]))
[1, 2, 3, 4, 5, 6, 7]
```

repeat

Usage: `(repeat x)`

Returns an iterator (infinite) of `x`.

```
=> (list (take 6 (repeat "s")))
[u's', u's', u's', u's', u's', u's']
```

repeatedly

Usage: `(repeatedly fn)`

Returns an iterator by calling *fn* repeatedly.

```
=> (import [random [randint]])

=> (list (take 5 (repeatedly (fn [] (randint 0 10)))))
[6, 2, 0, 6, 7]
```

take

Usage: `(take n coll)`

Returns an iterator containing the first *n* members of *coll*. Raises `ValueError` if *n* is negative.

```
=> (list (take 3 [1 2 3 4 5]))
[1, 2, 3]

=> (list (take 4 (repeat "s")))
[u's', u's', u's', u's']

=> (list (take 0 (repeat "s")))
[]
```

take-nth

Usage: `(take-nth n coll)`

Returns an iterator containing every *n*-th member of *coll*.

```
=> (list (take-nth 2 [1 2 3 4 5 6 7]))
[1, 3, 5, 7]

=> (list (take-nth 3 [1 2 3 4 5 6 7]))
[1, 4, 7]

=> (list (take-nth 4 [1 2 3 4 5 6 7]))
[1, 5]

=> (list (take-nth 10 [1 2 3 4 5 6 7]))
[1]
```

take-while

Usage: `(take-while pred coll)`

Returns an iterator from *coll* as long as *pred* returns `True`.

```
=> (list (take-while pos? [ 1 2 3 -4 5]))
[1, 2, 3]

=> (list (take-while neg? [ -4 -3 1 2 5]))
[-4, -3]

=> (list (take-while neg? [ 1 2 3 -4 5]))
[]
```

Other Built-Ins

hy.core.reserved

Usage: `(hy.core.reserved.names)`

This module can be used to get a list (actually, a `frozenset`) of the names of Hy's built-in functions, macros, and special forms. The output also includes all Python reserved words. All names are in unmangled form (e.g., `list-comp` rather than `list_comp`).

```
=> (import hy)
=> (in "defclass" (hy.core.reserved.names))
True
```

Included itertools

count cycle repeat accumulate chain compress drop-while remove group-by islice *map take-while tee zip-longest product permutations combinations multicombinations

All of Python's itertools are available. Some of their names have been changed:

- `starmap` has been changed to `*map`
- `combinations_with_replacement` has been changed to `multicombinations`
- `groupby` has been changed to `group-by`
- `takewhile` has been changed to `take-while`
- `dropwhile` has been changed to `drop-while`
- `filterfalse` has been changed to `remove`

1.4.4 Reader Macros

Reader macros gives Lisp the power to modify and alter syntax on the fly. You don't want Polish notation? A reader macro can easily do just that. Want Clojure's way of having a regex? Reader macros can also do this easily.

Syntax

```
=> (defreader ^ [expr] (print expr))
=> #^(1 2 3 4)
(1 2 3 4)
=> #^"Hello"
"Hello"
=> #^1+2+3+4+3+2
1+2+3+4+3+2
```

Hy has no literal for tuples. Lets say you dislike *(, ...)* and want something else. This is a problem reader macros are able to solve in a neat way.

```
=> (defreader t [expr] `(, ~@expr))
=> #t(1 2 3)
(1, 2, 3)
```

You could even do it like Clojure and have a literal for regular expressions!

```
=> (import re)
=> (defreader r [expr] `(re.compile ~expr))
=> #r".*"
<_sre.SRE_Pattern object at 0xcv7713ph15#>
```

Implementation

`defreader` takes a single character as symbol name for the reader macro; anything longer will return an error. Implementation-wise, `defreader` expands into a lambda covered with a decorator. This decorator saves the lambda in a dictionary with its module name and symbol.

```
=> (defreader ^ [expr] (print expr))
;=> (with_decorator (hy.macros.reader ^) (fn [expr] (print expr)))
```

`#` expands into `(dispatch_reader_macro ...)` where the symbol and expression is passed to the correct function.

```
=> #^()
;=> (dispatch_reader_macro ^ ())
=> #^"Hello"
"Hello"
```

Warning: Because of a limitation in Hy's lexer and parser, reader macros can't redefine defined syntax such as `()[]{}`. This will most likely be addressed in the future.

1.4.5 Internal Hy Documentation

Note: These bits are mostly useful for folks who hack on Hy itself, but can also be used for those delving deeper in macro programming.

Hy Models

Introduction to Hy Models

Hy models are a very thin layer on top of regular Python objects, representing Hy source code as data. Models only add source position information, and a handful of methods to support clean manipulation of Hy source code, for instance in macros. To achieve that goal, Hy models are mixins of a base Python class and *HyObject*.

HyObject `hy.models.HyObject` is the base class of Hy models. It only implements one method, `replace`, which replaces the source position of the current object with the one passed as argument. This allows us to keep track of the original position of expressions that get modified by macros, be that in the compiler or in pure hy macros.

`HyObject` is not intended to be used directly to instantiate Hy models, but only as a mixin for other classes.

Compound Models

Parenthesized and bracketed lists are parsed as compound models by the Hy parser.

HyList `hy.models.list.HyList` is the base class of "iterable" Hy models. Its basic use is to represent bracketed `[]` lists, which, when used as a top-level expression, translate to Python list literals in the compilation phase.

Adding a HyList to another iterable object reuses the class of the left-hand-side object, a useful behavior when you want to concatenate Hy objects in a macro, for instance.

HyExpression `hy.models.expression.HyExpression` inherits *HyList* for parenthesized `()` expressions. The compilation result of those expressions depends on the first element of the list: the compiler dispatches expressions between compiler special-forms, user-defined macros, and regular Python function calls.

HyDict `hy.models.dict.HyDict` inherits *HyList* for curly-bracketed `{}` expressions, which compile down to a Python dictionary literal.

The decision of using a list instead of a dict as the base class for `HyDict` allows easier manipulation of dicts in macros, with the added benefit of allowing compound expressions as dict keys (as, for instance, the *HyExpression* Python class isn't hashable).

Atomic Models

In the input stream, double-quoted strings, respecting the Python notation for strings, are parsed as a single token, which is directly parsed as a *HyString*.

An uninterrupted string of characters, excluding spaces, brackets, quotes, double-quotes and comments, is parsed as an identifier.

Identifiers are resolved to atomic models during the parsing phase in the following order:

- *HyInteger*
- *HyFloat*
- *HyComplex* (if the atom isn't a bare `j`)
- *HyKeyword* (if the atom starts with `:`)
- *HySymbol*

HyString `hy.models.string.HyString` is the base class of string-equivalent Hy models. It also represents double-quoted string literals, `""`, which compile down to unicode string literals in Python. `HyStrings` inherit unicode objects in Python 2, and string objects in Python 3 (and are therefore not encoding-dependent).

`HyString` based models are immutable.

Hy literal strings can span multiple lines, and are considered by the parser as a single unit, respecting the Python escapes for unicode strings.

Numeric Models `hy.models.integer.HyInteger` represents integer literals (using the `long` type on Python 2, and `int` on Python 3).

`hy.models.float.HyFloat` represents floating-point literals.

`hy.models.complex.HyComplex` represents complex literals.

Numeric models are parsed using the corresponding Python routine, and valid numeric python literals will be turned into their Hy counterpart.

HySymbol `hy.models.symbol.HySymbol` is the model used to represent symbols in the Hy language. It inherits *HyString*.

`HySymbol` objects are mangled in the parsing phase, to help Python interoperability:

- Symbols surrounded by asterisks (`*`) are turned into uppercase;
- Dashes (`-`) are turned into underscores (_);
- One trailing question mark (`?`) is turned into a leading `is_`.

Caveat: as the mangling is done during the parsing phase, it is possible to programmatically generate HySymbols that can't be generated with Hy source code. Such a mechanism is used by *gensym* to generate "uninterned" symbols.

HyKeyword `hy.models.keyword.HyKeyword` represents keywords in Hy. Keywords are symbols starting with a `:`. The class inherits *HyString*.

To distinguish *HyKeywords* from *HySymbols*, without the possibility of (involuntary) clashes, the private-use unicode character `"\uFDD0"` is prepended to the keyword literal before storage.

Cons Cells

`hy.models.cons.HyCons` is a representation of Python-friendly cons cells. Cons cells are especially useful to mimic features of "usual" LISP variants such as Scheme or Common Lisp.

A cons cell is a 2-item object, containing a `car` (head) and a `cdr` (tail). In some Lisp variants, the cons cell is the fundamental building block, and S-expressions are actually represented as linked lists of cons cells. This is not the case in Hy, as the usual expressions are made of Python lists wrapped in a `HyExpression`. However, the `HyCons` mimics the behavior of "usual" Lisp variants thusly:

- `(cons something None)` is `(HyExpression [something])`
- `(cons something some-list)` is `((type some-list) (+ [something] some-list))` (if `some-list` inherits from `list`).
- `(get (cons a b) 0)` is `a`
- `(cut (cons a b) 1)` is `b`

Hy supports a dotted-list syntax, where `'(a . b)` means `(cons 'a 'b)` and `'(a b . c)` means `(cons 'a (cons 'b 'c))`. If the compiler encounters a cons cell at the top level, it raises a compilation error.

`HyCons` wraps the passed arguments (car and cdr) in Hy types, to ease the manipulation of cons cells in a macro context.

Hy Internal Theory

Overview

The Hy internals work by acting as a front-end to Python bytecode, so that Hy itself compiles down to Python Bytecode, allowing an unmodified Python runtime to run Hy code, without even noticing it.

The way we do this is by translating Hy into an internal Python AST datastructure, and building that AST down into Python bytecode using modules from the Python standard library, so that we don't have to duplicate all the work of the Python internals for every single Python release.

Hy works in four stages. The following sections will cover each step of Hy from source to runtime.

Steps 1 and 2: Tokenizing and Parsing

The first stage of compiling Hy is to lex the source into tokens that we can deal with. We use a project called rply, which is a really nice (and fast) parser, written in a subset of Python called rpython.

The lexing code is all defined in `hy.lex.lexer`. This code is mostly just defining the Hy grammar, and all the actual hard parts are taken care of by rply – we just define "callbacks" for rply in `hy.lex.parser`, which takes the tokens generated, and returns the Hy models.

You can think of the Hy models as the "AST" for Hy, it's what Macros operate on (directly), and it's what the compiler uses when it compiles Hy down.

See also:

Section *Hy Models* for more information on Hy models and what they mean.

Step 3: Hy Compilation to Python AST

This is where most of the magic in Hy happens. This is where we take Hy AST (the models), and compile them into Python AST. A couple of funky things happen here to work past a few problems in AST, and working in the compiler is some of the most important work we do have.

The compiler is a bit complex, so don't feel bad if you don't grok it on the first shot, it may take a bit of time to get right.

The main entry-point to the Compiler is `HyASTCompiler.compile`. This method is invoked, and the only real "public" method on the class (that is to say, we don't really promise the API beyond that method).

In fact, even internally, we don't recurse directly hardly ever, we almost always force the Hy tree through `compile`, and will often do this with sub-elements of an expression that we have. It's up to the Type-based dispatcher to properly dispatch sub-elements.

All methods that preform a compilation are marked with the `@builds()` decorator. You can either pass the class of the Hy model that it compiles, or you can use a string for expressions. I'll clear this up in a second.

First Stage Type-Dispatch Let's start in the `compile` method. The first thing we do is check the Type of the thing we're building. We look up to see if we have a method that can build the `type()` that we have, and dispatch to the method that can handle it. If we don't have any methods that can build that type, we raise an internal `Exception`.

For instance, if we have a `HyString`, we have an almost 1-to-1 mapping of Hy AST to Python AST. The `compile_string` method takes the `HyString`, and returns an `ast.Str()` that's populated with the correct line-numbers and content.

Macro-Expand If we get a `HyExpression`, we'll attempt to see if this is a known Macro, and push to have it expanded by invoking `hy.macros.macroexpand`, then push the result back into `HyASTCompiler.compile`.

Second Stage Expression-Dispatch The only special case is the `HyExpression`, since we need to create different AST depending on the special form in question. For instance, when we hit an `(if True True False)`, we need to generate a `ast.If`, and properly compile the sub-nodes. This is where the `@builds()` with a String as an argument comes in.

For the `compile_expression` (which is defined with an `@builds(HyExpression)`) will dispatch based on the string of the first argument. If, for some reason, the first argument is not a string, it will properly handle that case as well (most likely by raising an `Exception`).

If the String isn't known to Hy, it will default to create an `ast.Call`, which will try to do a runtime call (in Python, something like `foo()`).

Issues Hit with Python AST Python AST is great; it's what's enabled us to write such a powerful project on top of Python without having to fight Python too hard. Like anything, we've had our fair share of issues, and here's a short list of the common ones you might run into.

Python differentiates between Statements and Expressions.

This might not sound like a big deal – in fact, to most Python programmers, this will shortly become a "Well, yeah" moment.

In Python, doing something like:

`print for x in range(10): pass`, because `print` prints expressions, and `for` isn't an expression, it's a control flow statement. Things like `1 + 1` are Expressions, as is `lambda x: 1 + x`, but other language features, such as `if`, `for`, or `while` are statements.

Since they have no "value" to Python, this makes working in Hy hard, since doing something like `(print (if True True False))` is not just common, it's expected.

As a result, we auto-mangle things using a `Result` object, where we offer up any `ast.stmt` that need to get run, and a single `ast.expr` that can be used to get the value of whatever was just run. Hy does this by forcing assignment to things while running.

As example, the Hy:

```
(print (if True True False))
```

Will turn into:

```
if True:
    _mangled_name_here = True
else:
    _mangled_name_here = False

print _mangled_name_here
```

OK, that was a bit of a lie, since we actually turn that statement into:

```
print True if True else False
```

By forcing things into an `ast.expr` if we can, but the general idea holds.

Step 4: Python Bytecode Output and Runtime

After we have a Python AST tree that's complete, we can try and compile it to Python bytecode by pushing it through `eval`. From here on out, we're no longer in control, and Python is taking care of everything. This is why things like Python tracebacks, pdb and django apps work.

Hy Macros

Using gensym for Safer Macros

When writing macros, one must be careful to avoid capturing external variables or using variable names that might conflict with user code.

We will use an example macro `nif` (see http://letoverlambda.com/index.cl/guest/chap3.html#sec_5 for a more complete description.) `nif` is an example, something like a numeric `if`, where based on the expression, one of the 3 forms is called depending on if the expression is positive, zero or negative.

A first pass might be something like:

```
(defmacro nif [expr pos-form zero-form neg-form]
  `(let [obscure-name ~expr]
     (cond [(pos? obscure-name) ~pos-form]
           [(zero? obscure-name) ~zero-form]
           [(neg? obscure-name) ~neg-form])))
```

where `obscure-name` is an attempt to pick some variable name as not to conflict with other code. But of course, while well-intentioned, this is no guarantee.

The method *gensym* is designed to generate a new, unique symbol for just such an occasion. A much better version of `nif` would be:

```
(defmacro nif [expr pos-form zero-form neg-form]
  (let [g (gensym)]
    `(let [~g ~expr]
       (cond [(pos? ~g) ~pos-form]
             [(zero? ~g) ~zero-form]
             [(neg? ~g) ~neg-form]))))
```

This is an easy case, since there is only one symbol. But if there is a need for several gensym's there is a second macro *with-gensyms* that basically expands to a series of `let` statements:

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

expands to:

```
(let [a (gensym)
      b (gensym)
      c (gensym)]
  ...)
```

so our re-written `nif` would look like:

```
(defmacro nif [expr pos-form zero-form neg-form]
  (with-gensyms [g]
    `(let [~g ~expr]
       (cond [(pos? ~g) ~pos-form]
             [(zero? ~g) ~zero-form]
             [(neg? ~g) ~neg-form]))))
```

Finally, though we can make a new macro that does all this for us. *defmacro/g!* will take all symbols that begin with `g!` and automatically call `gensym` with the remainder of the symbol. So `g!a` would become `(gensym "a")`.

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

```
(defmacro/g! nif [expr pos-form zero-form neg-form]
  `(let [~g!res ~expr]
     (cond [(pos? ~g!res) ~pos-form]
           [(zero? ~g!res) ~zero-form]
           [(neg? ~g!res) ~neg-form])))
```

Checking Macro Arguments and Raising Exceptions

Hy Compiler Built-Ins

1.5 Extra Modules Index

These modules are considered no less stable than Hy's built-in functions and macros, but they need to be loaded with `(import ...)` or `(require ...)`.

Contents:

1.5.1 Anaphoric Macros

New in version 0.9.12.

The anaphoric macros module makes functional programming in Hy very concise and easy to read.

> An anaphoric macro is a type of programming macro that deliberately captures some form supplied to the macro which may be referred to by an anaphor (an expression referring to another).
>
> —Wikipedia (https://en.wikipedia.org/wiki/Anaphoric_macro)

To use these macros you need to require the `hy.extra.anaphoric` module like so:

```
(require [hy.extra.anaphoric [*]])
```

ap-if

Usage: `(ap-if (foo) (print it))`

Evaluates the first form for truthiness, and bind it to `it` in both the true and false branches.

ap-each

Usage: `(ap-each [1 2 3 4 5] (print it))`

Evaluate the form for each element in the list for side-effects.

ap-each-while

Usage: `(ap-each-while list pred body)`

Evaluate the form for each element where the predicate form returns `True`.

```
=> (ap-each-while [1 2 3 4 5 6] (< it 4) (print it))
1
2
3
```

ap-map

Usage: `(ap-map form list)`

The anaphoric form of map works just like regular map except that instead of a function object it takes a Hy form. The special name `it` is bound to the current object from the list in the iteration.

```
=> (list (ap-map (* it 2) [1 2 3]))
[2, 4, 6]
```

ap-map-when

Usage: `(ap-map-when predfn rep list)`

Evaluate a mapping over the list using a predicate function to determin when to apply the form.

```
=> (list (ap-map-when odd? (* it 2) [1 2 3 4]))
[2, 2, 6, 4]

=> (list (ap-map-when even? (* it 2) [1 2 3 4]))
[1, 4, 3, 8]
```

ap-filter

Usage: `(ap-filter form list)`

As with `ap-map` we take a special form instead of a function to filter the elements of the list. The special name `it` is bound to the current element in the iteration.

```
=> (list (ap-filter (> (* it 2) 6) [1 2 3 4 5]))
[4, 5]
```

ap-reject

Usage: `(ap-reject form list)`

This function does the opposite of `ap-filter`, it rejects the elements passing the predicate . The special name `it` is bound to the current element in the iteration.

```
=> (list (ap-reject (> (* it 2) 6) [1 2 3 4 5]))
[1, 2, 3]
```

ap-dotimes

Usage `(ap-dotimes n body)`

This function evaluates the body *n* times, with the special variable `it` bound from *0* to *1-n*. It is useful for side-effects.

```
 => (setv n [])
 => (ap-dotimes 3 (.append n it))
 => n
[0, 1, 2]
```

ap-first

Usage `(ap-first predfn list)`

This function returns the first element that passes the predicate or `None`, with the special variable `it` bound to the current element in iteration.

```
=>(ap-first (> it 5) (range 10))
6
```

ap-last

Usage `(ap-last predfn list)`

This function returns the last element that passes the predicate or `None`, with the special variable `it` bound to the current element in iteration.

```
=>(ap-last (> it 5) (range 10))
9
```

ap-reduce

Usage `(ap-reduce form list &optional initial-value)`

This function returns the result of applying form to the first 2 elements in the body and applying the result and the 3rd element etc. until the list is exhausted. Optionally an initial value can be supplied so the function will be applied to initial value and the first element instead. This exposes the element being iterated as `it` and the current accumulated value as `acc`.

```
=>(ap-reduce (+ it acc) (range 10))
45
```

ap-pipe

Usage `(ap-pipe value form1 form2 ...)`

Applies several forms in series to a value from left to right. The special variable `it` in each form is replaced by the result of the previous form.

```
=> (ap-pipe 3 (+ it 1) (/ 5 it))
1.25
=> (ap-pipe [4 5 6 7] (list (rest it)) (len it))
3
```

ap-compose

Usage `(ap-compose form1 form2 ...)`

Returns a function which applies several forms in series from left to right. The special variable `it` in each form is replaced by the result of the previous form.

```
=> (def op (ap-compose (+ it 1) (* it 3)))
=> (op 2)
9
```

xi

Usage `(xi body ...)`

Returns a function with parameters implicitly determined by the presence in the body of xi parameters. An xi symbol designates the ith parameter (1-based, e.g. x1, x2, x3, etc.), or all remaining parameters for xi itself. This is not a replacement for lambda. The xi forms cannot be nested.

This is similar to Clojure's anonymous function literals (`#()`).

```
=> ((xi identity [x1 x5 [x2 x3] xi x4]) 1 2 3 4 5 6 7 8)
[1, 5, [2, 3,] (6, 7, 8), 4]
=> (def add-10 (xi + 10 x1))
=> (add-10 6)
16
```

1.6 Contributor Modules Index

These modules are experimental additions to Hy. Once deemed mature, they will be moved to the `hy.extra` namespace or loaded by default.

Contents:

1.6.1 loop/recur

New in version 0.10.0.

The `loop` / `recur` macro gives programmers a simple way to use tail-call optimization (TCO) in their Hy code.

> A tail call is a subroutine call that happens inside another procedure as its final action; it may produce a return value which is then immediately returned by the calling procedure. If any call that a subroutine performs, such that it might eventually lead to this same subroutine being called again down the call chain, is in tail position, such a subroutine is said to be tail-recursive, which is a special case of recursion. Tail calls are significant because they can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is not needed any more, and it can be replaced by the frame of the tail call. The program can then jump to the called subroutine. Producing such code instead of a standard call sequence is called tail call elimination, or tail call optimization. Tail call elimination allows procedure calls in tail position to be implemented as efficiently as goto statements, thus allowing efficient structured programming.
>
> —Wikipedia (https://en.wikipedia.org/wiki/Tail_call)

Macros

loop

`loop` establishes a recursion point. With `loop`, `recur` rebinds the variables set in the recursion point and sends code execution back to that recursion point. If `recur` is used in a non-tail position, an exception is raised.

Usage: `(loop bindings &rest body)`

Example:

```
(require [hy.contrib.loop [loop]])

(defn factorial [n]
  (loop [[i n] [acc 1]]
    (if (zero? i)
      acc
      (recur (dec i) (* acc i)))))

(factorial 1000)
```

1.6.2 defmulti

defn

New in version 0.10.0.

`defn` lets you arity-overload a function by the given number of args and/or kwargs. This version of `defn` works with regular syntax and with the arity overloaded one. Inspired by Clojures take on `defn`.

```
=> (require [hy.contrib.multi [defn]])
=> (defn fun
...  ([a] "a")
...  ([a b] "a b")
...  ([a b c] "a b c"))

=> (fun 1)
"a"
=> (fun 1 2)
"a b"
=> (fun 1 2 3)
"a b c"

=> (defn add [a b]
...  (+ a b))
=> (add 1 2)
3
```

defmulti

New in version 0.12.0.

`defmulti`, `defmethod` and `default-method` lets you define multimethods where a dispatching function is used to select between different implementations of the function. Inspired by Clojure's multimethod and based on the code by Adam Bard.

```
=> (require [hy.contrib.multi [defmulti defmethod default-method]])
=> (defmulti area [shape]
...  "calculate area of a shape"
...  (:type shape))

=> (defmethod area "square" [square]
...  (* (:width square)
...     (:height square)))

=> (defmethod area "circle" [circle]
...  (* (** (:radius circle) 2)
```

```
...     3.14))

=> (default-method area [shape]
...  0)

=> (area {:type "circle" :radius 0.5})
0.785

=> (area {:type "square" :width 2 :height 2})
4

=> (area {:type "non-euclid rhomboid"})
0
```

`defmulti` is used to define the initial multimethod with name, signature and code that selects between different implementations. In the example, multimethod expects a single input that is type of dictionary and contains at least key :type. The value that corresponds to this key is returned and is used to selected between different implementations.

`defmethod` defines a possible implementation for multimethod. It works otherwise in the same way as `defn`, but has an extra parameters for specifying multimethod and which calls are routed to this specific implementation. In the example, shapes with "square" as :type are routed to first function and shapes with "circle" as :type are routed to second function.

`default-method` specifies default implementation for multimethod that is called when no other implementation matches.

Interfaces of multimethod and different implementation don't have to be exactly identical, as long as they're compatible enough. In practice this means that multimethod should accept the broadest range of parameters and different implementations can narrow them down.

```
=> (require [hy.contrib.multi [defmulti defmethod]])
=> (defmulti fun [&rest args]
...  (len args))

=> (defmethod fun 1 [a]
...  a)

=> (defmethod fun 2 [a b]
...  (+ a b))

=> (fun 1)
1

=> (fun 1 2)
3
```

1.6.3 Profile

New in version 0.10.0.

The `profile` macros make it easier to find bottlenecks.

Macros

profile/calls

`profile/calls` allows you to create a call graph visualization. **Note:** You must have Graphviz installed for this to work.

Usage: *(profile/calls (body))*

Example:

```
(require [hy.contrib.profile [profile/calls]])
(profile/calls (print "hey there"))
```

profile/cpu

`profile/cpu` allows you to profile a bit of code.

Usage: *(profile/cpu (body))*

Example:

```
(require [hy.contrib.profile [profile/cpu]])
(profile/cpu (print "hey there"))
```

```
hey there
<pstats.Stats instance at 0x14ff320>
         2 function calls in 0.000 seconds

 Random listing order was used

 ncalls  tottime  percall  cumtime  percall filename:lineno(function)        1    0.000    0.000    0
      1    0.000    0.000    0.000    0.000 {print}
```

1.6.4 Lazy sequences

New in version 0.12.0.

The sequences module contains a few macros for declaring sequences that are evaluated only as much as the client code requires. Unlike generators, they allow accessing the same element multiple times. They cache calculated values, and the implementation allows for recursive definition of sequences without resulting in recursive computation.

To use these macros, you need to require them and import some other names like so:

```
(require [hy.contrib.sequences [defseq seq]])
(import [hy.contrib.sequences [Sequence end-sequence]])
```

The simplest sequence can be defined as `(seq [n] n)`. This defines a sequence that starts as `[0 1 2 3 ...]` and continues forever. In order to define a finite sequence, you need to call `end-sequence` to signal the end of the sequence:

```
(seq [n]
     "sequence of 5 integers"
     (cond [(< n 5) n]
           [True (end-sequence)]))
```

This creates the following sequence: `[0 1 2 3 4]`. For such a sequence, `len` returns the amount of items in the sequence and negative indexing is supported. Because both of these require evaluating the whole sequence, calling one on an infinite sequence would take forever (or at least until available memory has been exhausted).

Sequences can be defined recursively. For example, the Fibonacci sequence could be defined as:

```
(defseq fibonacci [n]
  "infinite sequence of fibonacci numbers"
  (cond [(= n 0) 0]
        [(= n 1) 1]
        [True (+ (get fibonacci (- n 1))
                 (get fibonacci (- n 2)))]))
```

This results in the sequence `[0 1 1 2 3 5 8 13 21 34 ...]`.

seq

Usage: `(seq [n] (* n n))`

Creates a sequence defined in terms of `n`.

defseq

Usage: `(defseq numbers [n] n)`

Creates a sequence defined in terms of `n` and assigns it to a given name.

end-sequence

Usage: `(seq [n] (if (< n 5) n (end-sequence)))`

Signals the end of a sequence when an iterator reaches the given point of the sequence. Internally, this is done by raising `IndexError`, catching that in the iterator, and raising `StopIteration`.

1.6.5 walk

New in version 0.11.0.

Functions

walk

Usage: *(walk inner outer form)*

`walk` traverses `form`, an arbitrary data structure. Applies `inner` to each element of form, building up a data structure of the same type. Applies `outer` to the result.

Example:

```
=> (import [hy.contrib.walk [walk]])
=> (setv a '(a b c d e f))
=> (walk ord identity a)
(97 98 99 100 101 102)
=> (walk ord first a)
97
```

postwalk

Usage: *(postwalk f form)*

Performs depth-first, post-order traversal of `form`. Calls `f` on each sub-form, uses `f` 's return value in place of the original.

```
=> (import [hy.contrib.walk [postwalk]])
=> (def trail '([1 2 3] [4 [5 6 [7]]]))
=> (defn walking [x]
     (print "Walking:" x)
     x )
=> (postwalk walking trail)
Walking: 1
Walking: 2
Walking: 3
Walking: (1 2 3)
Walking: 4
Walking: 5
Walking: 6
Walking: 7
Walking: (7)
Walking: (5 6 [7])
Walking: (4 [5 6 [7]])
Walking: ([1 2 3] [4 [5 6 [7]]])
([1 2 3] [4 [5 6 [7]]])
```

prewalk

Usage: *(prewalk f form)*

Performs depth-first, pre-order traversal of `form`. Calls `f` on each sub-form, uses `f` 's return value in place of the original.

```
=> (import [hy.contrib.walk [prewalk]])
=> (def trail '([1 2 3] [4 [5 6 [7]]]))
=> (defn walking [x]
     (print "Walking:" x)
     x )
=> (prewalk walking trail)
Walking: ([1 2 3] [4 [5 6 [7]]])
Walking: [1 2 3]
Walking: 1
Walking: 2
Walking: 3
Walking: [4 [5 6 [7]]]
Walking: 4
Walking: [5 6 [7]]
Walking: 5
Walking: 6
Walking: [7]
Walking: 7
([1 2 3] [4 [5 6 [7]]])
```

1.7 Hacking on Hy

1.7.1 Join our Hyve!

Please come hack on Hy!

Please come hang out with us on `#hy` on `irc.freenode.net`!

Please talk about it on Twitter with the `#hy` hashtag!

Please blog about it!

Please don't spraypaint it on your neighbor's fence (without asking nicely)!

1.7.2 Hack!

Do this:

1. Create a virtual environment:

```
$ virtualenv venv
```

   and activate it:

```
$ . venv/bin/activate
```

   or use virtualenvwrapper to create and manage your virtual environment:

```
$ mkvirtualenv hy
$ workon hy
```

2. Get the source code:

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

   or use your fork:

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

3. Install for hacking:

```
$ cd hy/
$ pip install -e .
```

4. Install other develop-y requirements:

```
$ pip install -r requirements-dev.txt
```

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

1.7.3 Test!

Tests are located in `tests/`. We use nose.

To run the tests:

```
$ nosetests
```

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
```

1.7.4 Document!

Documentation is located in `docs/`. We use Sphinx.

To build the docs in HTML:

```
$ cd docs
$ make html
```

Write docs—docs are good! Even this doc!

1.7.5 Contributing

1.7.6 Contributor Guidelines

Contributions are welcome & greatly appreciated, every little bit helps in making Hy more awesome.

Pull requests are great! We love them; here is a quick guide:

- Fork the repo and create a topic branch for a feature/fix. Avoid making changes directly on the master branch. If you would like to contribute but don't know how to begin, the good-first-bug label of the issue tracker is the place to go. (If you're new to Git: Start Here)
- Before contributing make sure you check the docs. There are two versions of docs available:
  - latest, for use with the bleeding-edge GitHub version.
  - stable, for use with the PyPI version.
- All incoming features should be accompanied with tests.
- If you are contributing a major change to the Hy language (e.g. changing the behavior of or removing functions or macros), or you're unsure of the proposed change, please open an issue in the issue tracker before submitting the PR. This will allow others to give feedback on your idea, and it will avoid constant changes or wasted work. For other PRs (such as documentation fixes or code cleanup), you can directly open the PR without first opening a corresponding issue.
- Before you submit a PR, please run the tests and check your code against the style guide. You can do both of these things at once:

```
    $ make d
```

- Make commits into logical units, so that it is easier to track & navigate later. Before submitting a PR, try squashing the commits into changesets that are easy to come back to later. Also, make sure you don't leave spurious whitespace in the changesets; this avoids creation of whitespace fix commits later.
- As far as commit messages go, try to adhere to the following:
  - Try sticking to the 50 character limit for the first line of Git commit messages.
  - For more detail/explanations, follow this up with a blank line and continue describing the commit in detail.
- Finally, add yourself to the AUTHORS file (as a separate commit): you deserve it :)

- All incoming changes need to be acked by 2 different members of Hylang's core team. Additional review is clearly welcome, but we need a minimum of 2 signoffs for any change.
- If a core member is sending in a PR, please find 2 core members that doesn't include the PR submitter. The idea here is that one can work with the PR author, and a second acks the entire change set.
- For documentation & other trivial changes, we're good to merge after one ACK. We've got low coverage, so it'd be great to keep that barrier low.

1.7.7 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/.

1.7.8 Core Team

The core development team of Hy consists of following developers:

- Julien Danjou
- Morten Linderud
- J Kenneth King
- Gergely Nagy
- Tuukka Turto
- Karen Rustad
- Abhishek L
- Christopher Allan Webber
- Konrad Hinsen
- Will Kahn-Greene
- Paul Tagliamonte
- Nicolas Dandrimont
- Berker Peksag

- Clinton N. Dreisbach
- han semaj
- Zack M. Davis
- Kodi Arfer

Symbols

C

F