(= HY (+ PYTHON LISP))

Eléonore Mayola, PhD - @EleonoreMayola
Junior data scientist @MastodonC
Organizer at @PyLadiesLondon

Cuddles is Hy's mascot
by **Karen Rustad Tölva**

About Lisp

- **1958**: second oldest high-level programming language still in use today
- Created as a practical mathematical notation for computer programs
- Became the most widely used language in Artificial Intelligence research
- Characterised by parenthesised lists also called **S-expressions**:
`(function argument1 argument2)`
- Today Lisp dialects are used for general purpose; Common Lisp, Scheme, Clojure, and Hy
- Lisps are valued for their clear, elegant syntax and ability to extend the language (macros)

Intro to Hy

Hy is a Lisp dialect that is embedded in Python. Within Hy you have access to all of Python data structures and the standard library.
It's as easy as `(print "Hello, world!")`.
Try Hy in your browser at try-hy.appspot.com/

Calculations go from
```
(3.5 + 5.1 + 4.6) / 3
```
to
```
(/ (+ 3.5 5.1 4.6) 3)
```

An **if/else** statement looks like:
```
=> (if (= 3 (+ 1 2))
... (print "This is true")
... (print "This is false"))
This is true
```

And a **for loop** becomes:
```
=> (for [i (range 6)]
... (print (+ "i equals " (str i))))
i equals 0
i equals 1
i equals 2
i equals 3
i equals 4
i equals 5
```

*** See more code examples at bit.ly/try-hy**
*** Get started with Hy by following instructions at docs.hylang.org/en/latest/quickstart.html**

A Hy program

My example Hy program that analyzes text files. You can find it at bit.ly/text-analysis-hy.
My imports : `(import os re pprint [collections [Counter]])`
The function performing the analysis is:

```
(defn analyze-texts [dirpath]
  (setv text-files (list-text-files dirpath))
  (list (map (fn [f]
               (setv filename f)
               (-> f
                   read-text
                   clean-text
                   remove-stopwords
                   (summarise-text filename)))
             text-files)))
```

"**defn**" the function definition, inspired from Clojure

"**setv**" sets a variable by binding a symbol to a value, a function...

"**fn**" defines an anonymous function

"**map**" returns an iterable that applies the anonymous function to each file in the text-files list.

"**->**" the thread first, inspired from Clojure. Enables function chaining without several levels of nesting.

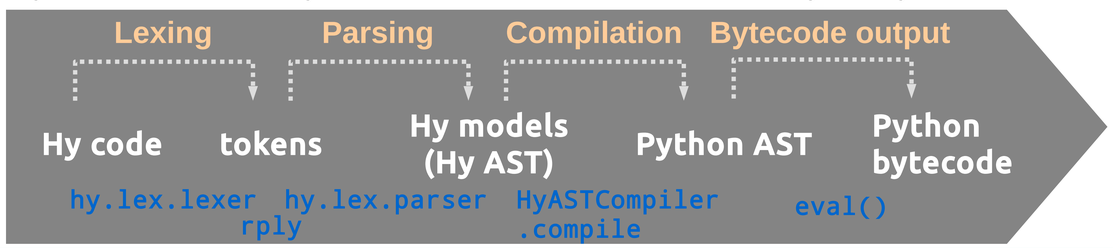
You can define a "main" function to reproduce the "if __name__ == '__main__'" behavior.
By using `(defmain [&rest args] …)`, you can run a main from the command line with its arguments.

Hy under the hood

Basic steps of compilation:
Lexing → lexical analysis: breaks up the code into tokens
Parsing → syntax analysis: convert a sequence of tokens into a parse tree
Code generation → translate the parse tree into bytecode

AST (Abstract Syntax Tree): data structure used by compilers to represent the structure of the source code. It's the result of the parsing (syntax analysis) step.

Hy first translates to a Python AST which is then built down into Python bytecode.



Hy Models: a layer on top of Python objects representing Hy source code as data.
They define Hy objects that can add info to help the manipulation of the Hy source code.

Hy Macros

Lisp's macros enable you to extend the syntax of the language.
You can define a new macro by using `defmacro`. But a lot of the functions we've seen so far are actually macros themselves.

Let's look at a simple macro, the `when` macro:
```
(defmacro when [test &rest body]
  "Execute `body` when `test` is true"
  `(if ~test (do ~@body)))
```

Let's define a macro called `when-int`:
```
(defmacro when-int [value &rest body]
  "Execute `body` when `value` is an
integer"
  `(if (integer? ~value) (do ~@body)))
```

A **quote** ` defines a symbol, to get an unevaluated data structure.
A tilde ~ enables **unquoting** a form within a quoted structure.
A ~@ character does **unquote splicing**: unquote and unnest.

My favourite Hy macro so far is `defmain`:
```
(defmacro defmain [args &rest body]
  "Write a function named \"main\" and do the
if __main__ dance"
  (let [retval (gensym)
        mainfn `(fn [~@args]
                  ~@body)]
    `(when (= --name-- "__main__")
       (import sys)
       (setv ~retval (apply ~mainfn
sys.argv))
       (if (integer? ~retval)
         (sys.exit ~retval)))))
```

When writing macros some variable names can create conflicts. The method `gensym` and the macro `with-gensym` generate one or several new and unique symbols: `=> (gensym "a")  u':a_1235'`

References:

My repository for this poster
github.com/Eleonore9/hy-python-lisp

- **Why Lisp?** gigamonkeys.com/book/introduction-why-lisp.html
- **More on (Common) Lisp**: gigamonkeys.com/book/syntax-and-semantics.html
- **Hy's docs**: docs.hylang.org/en/latest/
- **More docs**: github.com/hylang/hy/blob/master/docs/language/api.rst
- **Hy's source code**: github.com/hylang/hy
- **Podcast.__init__ episode 23**: pythonpodcast.com/hylang-developers.html
- **Talks and posts on Hy:** gist.github.com/Eleonore9/6ae886f4ac3a70cbcb28852bc8f6a25
- **Compilers**: en.wikipedia.org/wiki/Compiler
- **Abstract Syntax Tree**: en.wikipedia.org/wiki/Abstract_syntax_tree
- **Lisp's macros**: gigamonkeys.com/book/macros-standard-control-constructs.html
- **Writing macros (in Clojure)**: braveclojure.com/writing-macros/

Tools:
- **Emacs Hy-mode**: github.com/hylang/hy-mode
- **Vim-hy**: github.com/hylang/vim-hy
- **Paredit or smartparens** (keep parens in pairs)
- **Code analyser**: github.com/hylang/hydiomatic
- **Hy debugger**: github.com/hylang/hdb