Hy is a lisp dialect built on top of python. This is achieved by converting hy code to python's abstract syntax tree (ast). This allows hy to call native python code or python to call native hy code as well

This tutorial works for hy   0.9.12

```
;; this gives an gentle introduction to hy for a quick trial head to
;; http://try-hy.appspot.com
;;
; Semicolon comments, like other LISPS

;; s-expression basics
; lisp programs are made of symbolic expressions or sexps which
; resemble
(some-function args)
; now the quintessential hello world
(print "hello world")

;; simple data types
; All simple data types are exactly similar to their python counterparts
; which
42 ; => 42
3.14 ; => 3.14
True ; => True
4+10j ; => (4+10j) a complex number

; lets start with some really simple arithmetic
(+ 4 1) ;=> 5
; the operator is applied to all arguments, like other lisps
(+ 4 1 2 3) ;=> 10
(- 2 1) ;=> 1
(* 4 2) ;=> 8
(/ 4 1) ;=> 4
(% 4 2) ;=> 0 the modulo operator
; power is represented by ** operator like python
(** 3 2) ;=> 9
; nesting forms will do the expected thing
(+ 2 (* 4 2)) ;=> 10
; also logical operators and or not and equal to etc. do as expected
(= 5 4) ;=> False
(not (= 5 4)) ;=> True

;; variables
; variables are set using setv, variable names can use utf-8 except
; for ()[]{}",'`;#|
(setv a 42)
(setv   3.14159)
(def *foo* 42)
;; other container data types
; strings, lists, tuples & dicts
; these are exactly same as python's container types
"hello world" ;=> "hello world"
; string operations work similar to python
(+ "hello " "world") ;=> "hello world"
; lists are created using [], indexing starts at 0
(setv mylist [1 2 3 4])
```

```
; tuples are immutable data structures
(setv mytuple (, 1 2))
; dictionaries are key value pairs
(setv dict1 {"key1" 42 "key2" 21})
; :name can be used to define keywords in hy which can be used for keys
(setv dict2 {:key1 41 :key2 20})
; use `get' to get the element at an index/key
(get mylist 1) ;=> 2
(get dict1 "key1") ;=> 42
; Alternatively if keywords were used they can directly be called
(:key1 dict2) ;=> 41

;; functions and other program constructs
; functions are defined using defn, the last sexp is returned by default
(defn greet [name]
  "A simple greeting" ; an optional docstring
  (print "hello " name))

(greet "bilbo") ;=> "hello bilbo"

; functions can take optional arguments as well as keyword arguments
(defn foolists [arg1 &optional [arg2 2]]
  [arg1 arg2])

(foolists 3) ;=> [3 2]
(foolists 10 3) ;=> [10 3]

; anonymous functions are created using `fn' or `lambda' constructs
; which are similiar to `defn'
(map (fn [x] (* x x)) [1 2 3 4]) ;=> [1 4 9 16]

;; Sequence operations
; hy has some builtin utils for sequence operations etc.
; retrieve the first element using `first' or `car'
(setv mylist [1 2 3 4])
(setv mydict {"a" 1 "b" 2})
(first mylist) ;=> 1

; slice lists using slice
(slice mylist 1 3) ;=> [2 3]

; get elements from a list or dict using `get'
(get mylist 1) ;=> 2
(get mydict "b") ;=> 2
; list indexing starts from 0 same as python
; assoc can set elements at keys/indexes
(assoc mylist 2 10) ; makes mylist [1 2 10 4]
(assoc mydict "c" 3) ; makes mydict {"a" 1 "b" 2 "c" 3}
; there are a whole lot of other core functions which makes working with
; sequences fun

;; Python interop
;; import works just like in python
(import datetime)
```

```
(import [functools [partial reduce]]) ; imports fun1 and fun2 from module1
(import [matplotlib.pyplot :as plt]) ; doing an import foo as bar
; all builtin python methods etc. are accessible from hy
; a.foo(arg) is called as (.foo a arg)
(.split (.strip "hello world  ")) ;=> ["hello" "world"]

;; Conditionals
; (if condition (body-if-true) (body-if-false)
(if (= passcode "moria")
  (print "welcome")
  (print "Speak friend, and Enter!"))

; nest multiple if else if clauses with cond
(cond
 [(= someval 42)
  (print "Life, universe and everything else!")]
 [(> someval 42)
  (print "val too large")]
 [(< someval 42)
  (print "val too small")])

; group statements with do, these are executed sequentially
; forms like defn have an implicit do
(do
 (setv someval 10)
 (print "someval is set to " someval)) ;=> 10

; create lexical bindings with `let', all variables defined thusly
; have local scope
(let [[nemesis {"superman" "lex luther"
                "sherlock" "moriarty"
                "seinfeld" "newman"}]]
  (for [(, h v) (.items nemesis)]
    (print (.format "{0}'s nemesis was {1}" h v))))

;; classes
; classes are defined in the following way
(defclass Wizard [object]
  [[--init-- (fn [self spell]
             (setv self.spell spell) ; init the spell attr
             None)]
   [get-spell (fn [self]
              self.spell)]])

;; do checkout hylang.org
```

### Further Reading

This tutorial is just a very basic introduction to hy/lisp/python.

Hy docs are here: http://hy.readthedocs.org

Hy's Github repo: http://github.com/hylang/hy

On freenode irc #hy, twitter hashtag #hylang