Programming with lists in LISP and Python – JL Popyack v.1.3, April 2011

	LISP	Python
Atom	unquoted literal `fox` number `38` string `"fox in socks"`	number `38` string `"fox in socks"`
List	delimited by parentheses `(fox goose corn)`	delimited by brackets `["fox","goose","corn"]`
Empty List	`()` `NIL`	`[]`
Nested List	List elements can be atoms or lists `((fox corn)(farmer goose))`	List elements can be atoms or lists `[["fox","corn","farmer"]["goose"]]`
Assignment	`(setf x` `'((fox corn farmer)(goose))`	`x =` `[["fox","corn","farmer"]["goose"]]`
First Element	`(car x)` `(first x)`	`x[0]`
Rest of List	`(cdr x)` `(rest x)`	`x[1:]`
Composition	`(caddadr L)` is the same as `(car (cdr (cdr (car (cdr L)))))`	
Joining Lists	`(cons x L)` inserts x as the new *first element* of a list. The *rest of the list* is `L`. A new list is created; L is not changed. `(append L x)` adds x to the end of a list L and returns it as a new list; L is not changed.	`L.insert(0,x)` inserts x as the new *first element* of a list. `L` is changed. `L.append(x)` adds x to the end of a list `L`. `L` is changed..
Combining Lists		`L1 + L2` combines two lists.

Copying and Referencing Lists	`(setf L2 L1)`
	`L1` and `L2` refer to the same list.
	`(setf L2 (cdr L1))`
	`L2` refers to the same list as the rest of `L1`.
		`L1 = [1, 2]; L2 = L1`
		`L1` and `L2` refer to the same list.
		`L1 = [3, 4]; L2 = L1[:]`
		`L2` is a copy of `L1` ; *not* the same list.

True/False	`T` `NIL` or `()`	`true` `false`
Apply Function	First element of unquoted list is function name; remainder are arguments `(sqrt 9)`	Standard function notation `sqrt(9)`

Define Function	`(defun addOne (n)`
	`(+ 1 n)`
	`)`
		Indentation is meaningful and required; colon used as delimiter
		`def addOne(n):`
		`return 1+n`

Comments	Ignore everything following '	Ignore everything following #
Arithmetic	+ - * / `(- (* b b) (* 4 a c))`	+ - * / % Standard arithmetic notation b*b – 4*a*c

Comparison	`eq` compares whether two items point at the same object
	`equal` compares structures
	= compares numbers or atoms
	`(setf x '(a (b c) 1 2 3))`
	`(setf y (car (cdr x)))`
	`(setf z (cdr x))`
	`(setf w (car z))`
	`(eq y w)` `'returns T`
	`(eq y '(b c))` `'returns NIL`
	`(equal y '(b c))` `'returns T`
		Usual comparison operator ==
		Compares structures.

List Membership	`(member x L)` determines whether x is a member of list L and returns a pointer to x (interpreted as "true") or `NIL`. Uses `eq` for comparison. `(member L M :test #equal)` Uses `equal` for comparison.	`x in L` determines whether x is a member of list L and returns `true` or `false`. Compares structures.
Applying function to list	(mapcar f L) returns a list containing the results of applying `f` to each member of `L` . `(mapcar addOne '(3,-1,5,4.5))` returns `(4 0 6 5.5)` using `addOne` above. `(mapcar '+, '(1 2) '(3,4))` returns `(4 6)`	map(f,L) returns a list containing the results of applying `f` to each member of `L` . `map(addOne,[3,-1,5,4.5])` returns `[4, 0, 6, 5.5]` using `addOne` above. `map(operator.add, [1,2], [3,4])` returns `[4, 6]`

Evaluating expressions created dynamically		`min(3,1,4,1,5,9)` returns 1
		`max(3,1,4,1,5,9)` returns 9
		`f = min` `# not "min"`
		`apply(f,[3,1,4,1,5,9])` returns 1
		`f = "max"` `# eval accepts a string`
		`eval( f+"(3,1,4,1,5,9)" )` returns 9