```python
""" Scheme Interpreter in Python adapted from http://norvig.com/lispy.html"""

from __future__ import division
import sys, re

class SchemeError(Exception): pass

################ Symbol, Procedure, Env classes

Symbol = str

class Env(dict):
    "An environment: a dict of {'var':val} pairs, with an outer Env"
    def __init__(self, parms=(), args=(), outer=None):
        self.update(zip(parms,args))
        self.outer = outer
    def find_env(self, var):
        "Returns innermost Env where var appears"
        if var in self:
            return self
        elif self.outer:
            return self.outer.find(var)
        else:
            raise SchemeError("unbound variable " + var)
    def set(self, var, val): self.find_env(var)[var] = val
    def define(self, var, val): self[var] = val
    def lookup(self, var): return self.find_env(var)[var]

def add_globals(env):
    "Add some Scheme standard procedures to an environment"
    import math, operator as op
    env.update(vars(math)) # sin, sqrt, ...
    env.update(
     {'+':op.add, '-':op.sub, '*':op.mul, '/':op.div, 'not':op.not_,
      '>':op.gt, '<':op.lt, '>=':op.ge, '<=':op.le, '=':op.eq,
      'equal?':op.eq, 'eq?':op.is_, 'length':len, 'cons':lambda x,y:[x]+y,
      'car':lambda x:x[0],'cdr':lambda x:x[1:], 'append':op.add,
      'list':lambda *x:list(x), 'list?': lambda x:isa(x,list),
      'null?':lambda x:x==[], 'symbol?':lambda x: isa(x, Symbol),
      'load':lambda x:load(x), 'null':[], 'print':lambda x: sprint(x)})
    return env

global_env = add_globals(Env())

isa = isinstance

################ eval

def eval(x, env=global_env):
    "Evaluate an expression in an environment"
    if isa(x, Symbol):              # variable reference
        return env.lookup(x)
    elif not isa(x, list):          # constant literal
        return x
    elif x[0] == 'quote':           # (quote exp)
        return x[1]
    elif x[0] == 'if':              # (if test conseq alt)
        (_, test, conseq, alt) = x
        return eval((conseq if eval(test, env) else alt), env)
    elif x[0] == 'set!':            # (set! var exp)
        env.set(x[1], eval(x[2], env))
    elif x[0] == 'define':          # (define var exp)
        env.define(x[1], eval(x[2], env))
    elif x[0] == 'lambda':          # (lambda (var*) exp)
        (_, vars, exp) = x
        return lambda *args: eval(exp, Env(vars, args, env))
    elif x[0] == 'begin':           # (begin exp*)
        return [eval(x, env) for x in x[1:]][-1]
    else:                           # (proc exp*)
        exps = [eval(exp, env) for exp in x]
        proc = exps.pop(0)
        return proc(*exps)

################ parse, read, and user interaction

def read(s):
    "Read a Scheme expression from a string."
    return read_from(tokenize(s))

parse = read

def tokenize(s):
    "Convert a string into a list of tokens"
    return s.replace('(',' ( ').replace(')',' ) ').replace('\n', ' ').strip().split()

def read_from(tokens):
    "Read an expression from a sequence of tokens."
    if len(tokens) == 0:
        raise SchemeError('unexpected EOF while reading')
    token = tokens.pop(0)
    if '(' == token:
        L = []
        while tokens[0] != ')':
            L.append(read_from(tokens))
        tokens.pop(0) # pop off ')'
        return L
    elif ')' == token:
        raise SchemeError('unexpected )')
    else:
        return atom(token)

def atom(token):
    "Numbers become numbers; every other token is a symbol"
    try: return int(token)
    except ValueError:
        try: return float(token)
        except ValueError:
            return Symbol(token)
```

```python

def load(filename):
    "Read and eval expressions from file (w/o comments) returns void"
    tokens = tokenize(re.sub(";.*\n", "", open(filename).read()))
    while tokens:
        eval(read_from(tokens))

def sprint(x):
    "print serial form of x if it's not None"
    if x: print to_string(x)

def to_string(exp):
    "Convert a Python object back into a Lisp-readable string"
    return '('+' '.join(map(to_string, exp))+')' if isa(exp, list) else str(exp)

def repl(prompt='pyscm> '):
    "A prompt-read-eval-print loop."
    print "pyscheme, type control-D to exit"
    while True:
        try:
            sprint(eval(parse(raw_input(prompt))))
        except EOFError:
            print "Leaving pscm"
            break
        except SchemeError as e:
            print "SCM ERROR: ", e.args[0]
        except:
          print "ERROR: ", sys.exc_info()[0]

def start():
    print "Loading standard scheme library"
    load("stdlib.ss")
    repl()

# if called as a script
if __name__ == "__main__": start()
```