Activation stack exploration and limited dynamic binding in Python.
Assignment 4 for CSC 310, Spring, 2009
Dr. Dale E. Parson, http://faculty.kutztown.edu/parson
Due date is 11:59 PM on April 18

cp -pr ~parson/ProcLang/dynamic_scope ~/ProcLang/dynamic_scope

cd ~/ProcLang/dynamic_scope

ls -l

-rwxr-xr-x 1 Owner None  56 Apr  9 08:49  __init__.py
-rwxr-xr-x 1 Owner None 2916 Apr  9 20:38 dynamicScope.py
-rwxr-xr-x 1 Owner None  737 Apr  9 10:05 makefile
-rwxr-xr-x 1 Owner None 1855 Apr  6 14:11 makelib
-rwxr-xr-x 1 Owner None 4722 Apr  9 08:48 mergesort.py
-rw-r--r-- 1 Owner None 3306 Apr  9 08:59 mergesort.ref
-rwxr-xr-x 1 Owner None 3371 Apr  9 20:49 testdyna1.py
-rw-r--r-- 1 Owner None 1083 Apr  9 20:39 testdyna1.ref
-rwxr-xr-x 1 Owner None 3381 Apr  9 20:55 testdyna2.py
-rw-r--r-- 1 Owner None   0 Apr  9 10:04 testdyna2.ref

Student work goes into dynamicScope.py, testdyna2.py, and (after testdyna2.py is working) testdyna2.ref.

We will be using Python’s inspect module, documented in PY Chapter 13, to inspect the run-time activation stack and to implement a limited form of dynamic scope. Files mergesort.py and mergesort.ref show how to use inspect.stack() and inspect.getargvalues(frame) to inspect a call stack for parameter names and variable name-value bindings. Files testdyna1.py and testdyna1.ref show my tests. In addition to implementing function dynamicScope.getDynamic in file dynamicScope.py, students will write and debug tests within testdyna2.py. Both dynamicScope.py and testdyna2.py have extensive comments that document the requirements of this project. Running “gmake test” executes both my testdyna1.py and your testdyna2.py tests, and “gmake turnitin” by the end of April 18 turns it in.

The functions we need from module inspect are inspect.stack() and inspect.getargvalues(frame). We will go over how these work in class. If you miss the discussion, consult the PY book and also file mergesort.py and the DEBUGStack function in file dynamicScope.py.
dynamicScope.py

```python
#!/opt/csw/bin/python

# Package dynamic_scope, module dynamicScope.py for getting and setting
dynamically scoped variables.
# CSC 310, Spring, 2009, Dr. Dale Parson.
#
# STUDENT NAME:
#

""
Module dynamicScope.py contains methods for getting dynamically
scoped variables on the call stack. Python does not allow re-binding
variables that are neither global nor local to the re-binding function,
but it does allow mutation of objects fetched using dynamic scope.
""

import sys
import inspect

def getDynamic(varname):
    ""
    Locate varname on the call stack, and return its value to the value
    parameter. If varname does not appear on the call stack, then raise
    a NameError with the message (str(varname) + " is undefined.").
    ""
    # STUDENTS MUST IMPLEMENT AND TEST THIS FUNCTION and testdyna2.py.
    # STUDENTS: USE inspect.stack() TO RETRIEVE A LIST OF FRAME RECORDS,
    # AND IGNORE ENTRY [0], WHICH IS THE ACTIVATION FRAME FOR
    getDynamic
    # ITSELF. Invoke inspect.getargvalues(frame) for each frame except
    # the initial frame, and inspect the dictionary of locals for the
    # variable name. If it is in the locals, RETURN an ordered 2-tuple:
    #   (FUNCNAME, VALUE)
    # where FUNCNAME is the name of the function binding the variable,
    # and VALUE is the value associated with varname.
    # Make sure to test the return value in your test driver.
    # If varname is not a variable on the stack, raise a NameError
    # with the message: (str(varname) + " is undefined.").
    pass # REPLACE WITH YOUR CODE

def DEBUGStack(heading):
    ""
    Print a stack trace of function name, args, *varargs, **kwargs,
    and variable : value bindings to sys.err.
    ""
    # STUDENT: This one is already done. You can use it as a debugging aid.
```
```python
sys.stderr.write("DEBUG callstack for " + str(heading) + "\n")
framerecs = inspect.stack()[1:]  # Ignore my own frame.
for frec in framerecs:
    f = frec[0]  # pull out the next activation frame reference
    fname = frec[3]
sys.stderr.write("\tFUNCTION " + fname + "\n")
    for arg in args:
        sys.stderr.write("\t\tARG " + str(arg) + "\n")
    if (varargs):
        sys.stderr.write("\t\tVARARG " + str(varargs) + "\n")
    if (varkw):
        sys.stderr.write("\t\tVARKW " + str(varkw) + "\n")
    for k in locals.keys():
        sys.stderr.write("\t\tLOCAL " + str(k) + " \rightarrow "
                       + str(locals[k]) + "\n")

mergesort.py
1  #!/opt/csw/bin/python
2  # mergesort.py -- A test driver of merge sort & binary search.
3  # Migrated from ~parson/DataStructures/mergesort/mergesort.cxx, Fall, 2008
4  # from CSC237, Fall, 2008, Dr. Dale Parson.
5  # lexical_nesting_mergesort/mergesort.py uses lexical nesting
6  # of helper functions within the mergesort function.
7  # Otherwise it is identical to mergesort/mergesort.
8  #
9  # CHANGE APRIL 9, 2009: This version of mergesort uses some functions
10 # from Python’s inspect module to show how to examine the call stack
11 # frames for arguments and local variables. Inspect supports examining
12 # the call stack, but it is missing support for examining nested
13 # lexical environments at run time.
14
15 import sys  # see argv, stderr, stdout, exit
16 import dynamicScope # inspection module being tested
17
18 intlist = []
19 usage = "usage: mergesort ZERO_OR_MORE_INTEGERS"
20
21 # THE FOLLOWING EXIT CODES CONSTITUTE THE RETURN VALUES FROM
22 # MAIN:
23 SUCCESS   =  0  # successful program execution
24 PARMERR   =  1  # invalid command line values
25
26 def main():
27    ""
```
Function: main

main is a test driver that exercises implementations of
a merge sort and binary search.

Parameters:

argv[1] .. argv[argc-1] is normally an array of integers
to be sorted. Any invalid, non-BASE10 numeric string gets
an error message via cerr. The program treats it as 0,
but exits with PARMERR exit status on an invalid number in argv.

Return value: SUCCESS (0) or PARMERR on parameter error

global intlist  # global declaration needed to modify global var
exit_status = SUCCESS
# argv[0] is the name of the program being run.
for i in range(1, len(sys.argv)):
    try:
        value = int(sys.argv[i])
    except ValueError:
        sys.stderr.write(sys.argv[0] + ", Invalid numeric argument: "
                        + sys.argv[i] + "\n")
        exit_status = PARMERR
        value = 0
    intlist.append(value)
print \"MAIN FOR MERGESORT READ \" + str(len(intlist)) + \" INTS\" 
printarray(\"initial\", intlist)
mergesort(intlist)
printarray(\"final\", intlist)
return(exit_status)

def printarray(prompt, ilist):
    sys.stdout.write(prompt + \"\t\"
    for value in ilist:
        sys.stdout.write(str(value) + \"\t\"
    print \"n\"

# This version of mergesort uses nested lexical scope for its helper functions.
# NOTE THE CALL TO dynamicScope.DEBUGStack().
def mergesort(merger):
    # This is the mergesort’s splitphase for the assignment.
def splitphase():
    dynamicScope.DEBUGStack(\"splitphase\")
    qid = 0
    while (len(merger) > 0):  # while there are still runs to split
if (len(merger) < runlen):
    rl = len(merger)
else:
    rl = runlen
while (rl > 0):
    splitter[qid].append(merger[0])
    del(merger[0])
    rl = rl - 1
qid = qid ^ 1 ; # switch to other splitter

# This is the mergesort’s mergephase for the assignment.
# THIS LEXICALLY NESTED FUNCTION HAPPENS TO HAVE NO PARAMETERS.
# IT USES PARAMETERS AND VARIABLES BOUND BY THE ENCLOSING ENVIRONMENT.
def mergephase():
    while (splitter[0] and splitter[1]):
        srl = [runlen]  # remaining run lengths for each queue
        if (len(splitter[1]) < runlen):
            srl.append(len(splitter[1]))
        else:
            srl.append(runlen)
        while (srl[0] and srl[1]):  # both splitter queues have run contents
            if (splitter[0][0] <= splitter[1][0]):
                merger.append(splitter[0][0])
                del(splitter[0][0])
                srl[0] -= 1
            else:
                merger.append(splitter[1][0])
                del(splitter[1][0])
                srl[1] -= 1
        if (srl[0] > 0):
            qtodrain = 0
        else:
            qtodrain = 1
        while (srl[qtodrain] > 0):
            merger.append(splitter[qtodrain][0])
            del(splitter[qtodrain][0])
            srl[qtodrain] -= 1
        # splitter[0] may have a leftover run or partial run
        while (len(splitter[0]) > 0):
            merger.append(splitter[0][0])
            del(splitter[0][0])
        # NESTED DEFINITIONS ARE COMPLETE, NOW COMES THE OUTER FUNCTION BODY.
        runlen = 1
splitter = [[], []]  # a list of two empty lists
while (runlen < len(merger)):
    splitphase()
    mergephase()
    runlen *= 2

result = main()
if (result != 0):
sys.exit(result)

mergesort.ref
1  DEBUG callstack for splitphase
2      FUNCTION splitphase
3          LOCAL merger -> [11, -11, 0, 99, -100]
4          LOCAL splitter -> [[], []]
5          LOCAL runlen -> 1
6      FUNCTION mergesort
7          ARG merger
8          LOCAL merger -> [11, -11, 0, 99, -100]
9          LOCAL splitter -> [[], []]
10         LOCAL runlen -> 1
11         LOCAL mergephase -> <function mergephase at 0x00C21C70>
12         LOCAL splitphase -> <function splitphase at 0x00C21C30>
13      FUNCTION main
14          LOCAL i -> 5
15          LOCAL exit_status -> 0
16          LOCAL value -> -100
17    FUNCTION ?
18          LOCAL mergesort -> <function mergesort at 0x00C21BF0>
19          LOCAL printarray -> <function printarray at 0x00C21BB0>
20          LOCAL SUCCESS -> 0
21          LOCAL __builtins__ -> <module '__builtin__' (built-in)>
22          LOCAL dynamicScope -> <module 'dynamicScope' from 'C:\cygwin\home\Owner\ProcLang\dynamic_scope\dynamicScope.py'>
23          LOCAL __file__ -> ./mergesort.py
24          LOCAL sys -> <module 'sys' (built-in)>
25          LOCAL usage -> usage: mergesort ZERO_OR_MORE_INTEGERS
26          LOCAL intlist -> [11, -11, 0, 99, -100]
27          LOCAL __name__ -> __main__
28          LOCAL main -> <function main at 0x00B98930>
29          LOCAL __doc__ -> None
30          LOCAL PARMERR -> 1
31  DEBUG callstack for splitphase
32      FUNCTION splitphase
33          LOCAL merger -> [-11, 11, 0, 99, -100]
34          LOCAL splitter -> [[], []]
LOCAL runlen -> 2
FUNCTION mergesort
ARG merger
LOCAL merger -> [-11, 11, 0, 99, -100]
LOCAL splitter -> [[]] [[]]
LOCAL runlen -> 2
LOCAL mergephase -> <function mergephase at 0x00C21C70>
LOCAL splitphase -> <function splitphase at 0x00C21C30>
FUNCTION main
LOCAL i -> 5
LOCAL exit_status -> 0
LOCAL value -> -100
FUNCTION ?
LOCAL mergesort -> <function mergesort at 0x00C21BF0>
LOCAL printarray -> <function printarray at 0x00C21BB0>
LOCAL SUCCESS -> 0
LOCAL __builtins__ -> <module '__builtin__' (built-in)>
LOCAL dynamicScope -> <module 'dynamicScope' from 'C:\cygwin\home\Owner\ProcLang\dynamic_scope\dynamicScope.py'>
LOCAL __name__ -> __main__
LOCAL main -> <function main at 0x00B98930>
LOCAL __doc__ -> None
LOCAL PARMERR -> 1
DEBUG callstack for splitphase
FUNCTION splitphase
LOCAL merger -> [-11, 0, 11, 99, -100]
LOCAL splitter -> [[]] [[]]
LOCAL runlen -> 4
FUNCTION mergesort
ARG merger
LOCAL merger -> [-11, 0, 11, 99, -100]
LOCAL splitter -> [[]] [[]]
LOCAL runlen -> 4
LOCAL mergephase -> <function mergephase at 0x00C21C70>
LOCAL splitphase -> <function splitphase at 0x00C21C30>
FUNCTION main
LOCAL i -> 5
LOCAL exit_status -> 0
LOCAL value -> -100
FUNCTION ?
LOCAL mergesort -> <function mergesort at 0x00C21BF0>
LOCAL printarray -> <function printarray at 0x00C21BB0>
LOCAL SUCCESS -> 0
LOCAL __builtins__ -> <module '__builtin__' (built-in)>
LOCAL dynamicScope -> <module 'dynamicScope' from 'C:\cygwin\home\Owner\ProcLang\dynamic_scope\dynamicScope.py'>
LOCAL __file__ -> ./mergesort.py
LOCAL sys -> <module 'sys' (built-in)>
LOCAL usage -> usage:  mergesort ZERO_OR_MORE_INTEGERS
LOCAL intlist -> [-11, 0, 11, 99, -100]
LOCAL __name__ -> __main__
LOCAL main -> <function main at 0x00B98930>
LOCAL __doc__ -> None
LOCAL PARMERR -> 1

testdyna1.py
1  #!/opt/csw/bin/python
2  #       testdyna1.py -- A test driver of of module dynamic_scope/dynamicScope.
3  #       Dr. D. Parson, CSC310, Spring, 2009
4  #
5  #       This test driver is complete. See testdyna2.py for the STUDENT
6  #       assigned test driver.
7  
8  import dynamicScope # inspection module being tested
9  
10 var_a = 50
11 var_b = “fifty”
12 var_c = { var_a : var_b }
13  
14 def caller1(var_a):
15    # A Python assignment to any non-local variable that is
16    # not declared ‘global’ creates a new local variable.
17    # Binding a non-global variable using assignment within
18    # a function always creates a local. Neither lexically
19    # non-local variables nor dynamically non-local variables
20    # in this assignment can be re-bound in Python. On the
21    # other hand, if they are mutable objects, they can be mutated.
22    var_a = [“list”, “of”, “sixty”]
23  print “caller1 before tweak_stack call”
24  print “tvar_a -> “, dynamicScope.getDynamic(‘var_a’)
25  print “tvar_b -> “, dynamicScope.getDynamic(‘var_b’)
26  print “tvar_c -> “, dynamicScope.getDynamic(‘var_c’)
27  tweak_stack()
28  print “caller1 after tweak_stack call”
29  print “tvar_a -> “, dynamicScope.getDynamic(‘var_a’)
30  print “tvar_b -> “, dynamicScope.getDynamic(‘var_b’)
31  print “tvar_c -> “, dynamicScope.getDynamic(‘var_c’)
32
def caller2(var_a):
    global var_b
    var_b = ('a', 'tuple')
    var_a = set([var_a]) # a set is mutable
    var_c = {1 : 11}
    print "caller2 before tweak_stack call"
    print "\tvar_a -> ", dynamicScope.getDynamic('var_a')
    print "\tvar_b -> ", dynamicScope.getDynamic('var_b')
    print "\tvar_c -> ", dynamicScope.getDynamic('var_c')
    tweak_stack()
    print "caller2 after tweak_stack call"
    print "\tvar_a -> ", dynamicScope.getDynamic('var_a')
    print "\tvar_b -> ", dynamicScope.getDynamic('var_b')
    print "\tvar_c -> ", dynamicScope.getDynamic('var_c')

def tweak_stack():
    # Mutate any mutable dynamic variable.
    fa, vala = dynamicScope.getDynamic('var_a')
    fb, valb = dynamicScope.getDynamic('var_b')
    fc, valc = dynamicScope.getDynamic('var_c')
    if (type(vala) == list):
        vala.append("appended tweak a")
    elif (type(vala) == set):
        vala.add("added tweak a")
    elif (type(vala) == dict):
        vala['vala'] = 'vala tweak'
    if (type(valb) == list):
        vala.append("appended tweak b")
    elif (type(valb) == set):
        vala.add("added tweak b")
    elif (type(valb) == dict):
        valb['valb'] = 'valb tweak'
    if (type(valc) == list):
        vala.append("appended tweak b")
    elif (type(valc) == set):
        vala.add("added tweak b")
    elif (type(valc) == dict):
        valc['valc'] = 'valc tweak'

print "module testdyna1 before calling caller1"
print "\tvar_a -> ", dynamicScope.getDynamic('var_a')
print "\tvar_b -> ", dynamicScope.getDynamic('var_b')
print "\tvar_c -> ", dynamicScope.getDynamic('var_c')
caller1('param to caller1')
print "module testdyna1 after calling caller1"
print "\tvar_a -> ", dynamicScope.getDynamic('var_a')
print "\tvar_b -> ", dynamicScope.getDynamic('var_b')
print "\tvar_c -> ", dynamicScope.getDynamic('var_c')
caller2('param to caller2')
print "module testdyna1 after calling caller2"
print "\tvar_a -> ", dynamicScope.getDynamic('var_a')
print "\tvar_b -> ", dynamicScope.getDynamic('var_b')
print "\tvar_c -> ", dynamicScope.getDynamic('var_c')
testdyna1.ref
module testdyna1 before calling caller1
var_a -> ('?', 50)
var_b -> ('?', 'fifty')
var_c -> ('?', {50: 'fifty'})
caller1 before tweak_stack call
var_a -> ('caller1', ['list', 'of', 'sixty'])
var_b -> ('?', 'fifty')
var_c -> ('?', {50: 'fifty'})
caller1 after tweak_stack call
var_a -> ('caller1', ['list', 'of', 'sixty', 'appended tweak a'])
var_b -> ('?', 'fifty')
var_c -> ('?', {'valc': 'valc tweak', 50: 'fifty'})
module testdyna1 after calling caller1
var_a -> ('?', 50)
var_b -> ('?', 'fifty')
var_c -> ('?', {'valc': 'valc tweak', 50: 'fifty'})
caller2 before tweak_stack call
var_a -> ('caller2', set(['param to caller2']))
var_b -> ('?', ('a', 'tuple'))
var_c -> ('caller2', {1: 11})
caller2 after tweak_stack call
var_a -> ('caller2', set(['param to caller2', 'added tweak a']))
var_b -> ('?', ('a', 'tuple'))
var_c -> ('caller2', {'valc': 'valc tweak', 1: 11})
module testdyna1 after calling caller2
var_a -> ('?', 50)
var_b -> ('?', ('a', 'tuple'))
var_c -> ('?', {'valc': 'valc tweak', 50: 'fifty'})
#!/opt/csw/bin/python
#
testdyna2.py -- A test driver of of module dynamic_scope/dynamicScope.
# Dr. D. Parson, CSC310, Spring, 2009
#
# This test driver is incomplete. Students must complete it.
#
import dynamicScope # inspection module being tested
#
# STUDENTS must write the following tests and invoke print similar to 
to testdyna1.py to show results.
#
# 1. Define variables ‘var_a’ and ‘var_b’ and ‘var_c’ at module scope.
# #
# 2. Write a function ‘outer1’ that takes a parameter called ‘param’
# and that also defines a lexically nested function called ‘inner’
# that takes no parameters.
# #
# Within ‘outer1’ create a local variable ‘var_a’ and give it a mutable
# value.
# #
# Within ‘inner’ create a local variable ‘var_a’ and give it a different
# mutable value.
# #
# Write the code for ‘inner’ to use getDynamic() and mutation of its
# value to manipulate dynamic variable bindings, including ‘var_a’ and
# ‘var_b’ and ‘var_c’, on the call stack.
# #
# Write the code for ‘outer1’ to use getDynamic() and mutation of its value
# to manipulate dynamic variable bindings, including ‘var_a’ and
# ‘var_b’ and ‘var_c’, on the call stack.
# Make sure that ‘outer1’ invokes ‘inner’ at least one time.
# #
# In the final statement before ‘outer1’ returns, have it call
# function ‘outer2’ (below), a peer function of ‘outer1’ at module scope,
# that takes a zero-parameter function as an argument. (NOTE:
# A second-class function is a function passed as an argument.)
# Pass the ‘inner’ function (not a string ‘inner’) as the argument to
# outer2.
# #
# Return function ‘inner’ (not a string) as the return value from ‘outer1’.
# (NOTE: A first-class function is a function that can be returned from
# another function, and also bound to a variable in the caller’s
# environment. It is also a closure if it retains bindings to lexically
# bound symbols in its lexical environment.
# #
Define function ‘outer2’ that takes a function parameter called ‘secondClassFunction’ as outlined above.

Have ‘outer2’ declare “global var_a” as its first statement.

Write the code for ‘outer2’ to use getDynamic() and mutation of its value to manipulate dynamic variable bindings, including ‘var_a’ and ‘var_b’ and ‘var_c’, on the call stack.

Make sure that ‘outer2’ invokes ‘secondClassFunction’ at least one time.

Define function ‘wrapper’ that takes no parameters and invokes “outer1(‘three’)” and then returns without doing anything else.

5a. Print dumps of ‘var_a’ and ‘var_b’ and ‘var_c’ before and between calls to the following functions, similar to testdyna1.py.

5b. Invoke “functionObject = outer1(3)”

5c. Invoke “functionObject()”

5d. Invoke “outer2(functionObject)”

5e. Invoke “outer2(wrapper)”

After running “gmake clean test” verify that the contents of file testdyna2.out are correct and then copy this file to testdyna2.ref.

Then re-run “gmake test” to make that the results do not change.

You can invoke the dynamicScope.DEBUGStack() function to assist with debugging. Its output should not appear in any .out or .ref file.

In other words, remove calls to it or comment them out when you are done debugging.

Make sure to test the return value for getDynamic() as specified in dynamicScope.py.