```python
#!/usr/bin/env python
# test-tc.py - A Test Driver For The 'tconfpy' Configuration File Parser
# Copyright (c) 2003-2005 TundraWare Inc.  All Rights Reserved.

PROGNAME = "tconfpy Test Driver"
RCSID = "$Id: test-tc.py,v 1.145 2005/01/20 09:51:33 tundra Exp $"
VERSION = RCSID.split()[2]

# Copyright Information

CPRT         = chr(169)
DATE         = "2003-2005"
OWNER        = "TundraWare Inc."
RIGHTS       = "All Rights Reserved"
COPYRIGHT    = "Copyright %s %s %s,  %s." % (CPRT, DATE, OWNER, RIGHTS)
PROGINFO     = PROGNAME + " " + VERSION
BANNER       = "%s\n%s\n%s\n" % (PROGINFO, COPYRIGHT, RCSID)

#----------------------------------------------------------------------------#
#                     Imports And Other Odds-And-Ends                        #
#----------------------------------------------------------------------------#

from tconfpy import *
import sys



# Translation table for booleans into equivalent writeable state

TF = {True:"RW", False:"RO"}


#----------------------------------------------------------------------------#
#                     Build An Initial Symbol Table                          #
#----------------------------------------------------------------------------#


# One of the options of this test driver is to initialize a test
# symbol table.  To make it easy to change the contents of this test
# symbol table, we use the following scheme:


# The data below is used to populate an initial symbol table.  The
# init code takes care of actually reading this table and populating
# the symbol table for you.  So, you just have to maintain this table
# properly and the program will do all the actual initialization work
# for you.


# Each symbol has an entry in the form of a Python list with the
# following items/order:
#
#      Name, Value, Writeable, Type, Default, LegalVals, Min, Max
#
# **** NOTE: Where 'None' appears, the init code just uses the
#             ***default*** established by the VarDescriptor base class.


syms = [ ["int1",    1,         True,  TYPE_INT,     0,    [1, 2, 23],                        None, None ],
         ["int2",    1,         True,  TYPE_INT,     0,    [],                                1,    30   ],
         ["float1",  1.0,       None,  TYPE_FLOAT,   0.5,  [3.14, 2.73],                      None, None ],
         ["float2",  1.0,       None,  TYPE_FLOAT,   0.5,  [],                                -1.2, 0.5  ],
         ["str1"  ,  "stringy", None,  None,         None, [r'^box$', r'^Bax', r'a+bc'], 3,    8    ],
         ["cmplx1",  4+5j,      None,  TYPE_COMPLEX, 0-0j, [],                                None, None ],
         ["cmplx2",  4+5j,      None,  TYPE_COMPLEX, 0-0j, [1-1j, 1+1j],                      None, None ],
         ["bool1",   True,      None,  TYPE_BOOL,    None, None,                              None, None ],
         ["ro1",     "ReadVar", False, None,         None, None,                              None, None ],
       ]
```

```


# We use the exact same scheme for building variable templates

templs = [ ["templb",  True,        True,  TYPE_BOOL,     False, [],                          None, None ],
           ["templc",  4+5j,        None,  TYPE_COMPLEX,  0-0j,  [0-0j, 1-1j, 1+1j],          None, None ],
           ["templf",  1.0,         None,  TYPE_FLOAT,    0.5,   [0.5, 3.14, 2.73],           None, None ],
           ["templi",  1,           None,  TYPE_INT,      0,     [0, 1, 2, 23],               None, None ],
           ["templs",  "stringy",   None,  None,          None,  [r'^box$', r'^Bax', r'a+bc'], 3,    8    ]
         ]



#####
# Deriving Your Own Types
#####

# Here's a way to inherit the base symbol definition from the module
# and then derive a new object type for your own convenience.  This is
# a way to set your own defaults, for example.
#
# Note the use of 'super()' here to run each constructor in the
# inheritance tree only once.


class mycmplx(VarDescriptor):
    def __init__(self):
        super(mycmplx, self).__init__()
        self.Value = 2-2j
        self.Default = 39-4j
        self.Type = TYPE_COMPLEX


# We'll use this below to manually enter additional variables into the
# initial symbol table.


#----------------------------------------------------------------------------#
# Nothing Else Should Need Changing Below Here
#----------------------------------------------------------------------------#


#----------------------------------------------------------------------------#
#                       Utility And Support Functions                        #
#----------------------------------------------------------------------------#


#####
# Build A Symbol Or Template Table Using The Data Table Format Defined above
#####

def BuildSymTbl(syms, istemplate=False):

    if istemplate:
        symtbl = Template()
    else:
        symtbl = SymbolTable()

    for sym in syms:

        # Create and init the descriptor
        des = VarDescriptor()

        pos = 1
        for attr in ("Value", "Writeable", "Type", "Default", "LegalVals", "Min", "Max"):

            val = sym[pos]
            if val != None:
                exec("des." + attr + "=val")

            pos += 1

        symtbl.Symbols[sym[0]] = des
```

```python

    return symtbl

# End of 'BuildSymTbl()'


#####
# Format and return contents of a symbol table entry
#####


def dumpsymbol(val, d):

    retval = str(val) + " " * (18 - len(val))

    for v, p in ((d.Value, 20),
                 (TF[d.Writeable], 2),
                 (str(d.Type).split()[1][1:-2], 7),
                 (d.Default, 8),
                 (d.LegalVals, 8),
                 (d.Min, 8),
                 (d.Max, 8)):

        retval += str(v) + (p - len(str(v)) + 2) * ' '

    return retval

# End of 'dumpsymbol()'

#####
# Routine to dump returned values
#####


def dumpreturn(name, data, isdict=False):

    print name
    print len(name) * '=' + '\n'


    items = data
    if isdict:
        items = data.keys()
        items.sort()

    for val in items:
        if isdict:
            print dumpsymbol(val, data[val])
        else:
            print val

    print '\n'


# End of 'dumpreturn()'


#----------------------------------------------------------------------------#

#####
# Beginning of test driver
#####

files = 1

# Make sure we got legit arguments

if len(sys.argv) < 2:
    print BANNER
    print "Usage: test-tc.py [symtbl] [inmem] [nonewvars] [templates] [temponly] [limitns] [litvars] [nopredefs] [debug]  file file ..."
    sys.exit(1)

# Process all the requested configuration files,
# dumping the what tconfpy returns for each one.

ST       = False
```

```
INMEM    = False
ALLOWVAR = True
TEMPLATE = False
TEMPONLY = False
LIMITNS  = False
LITVARS  = False
PREDEFS  = True
DEBUG    = False



for fn in sys.argv[files:]:

    # Handle inline options

    if fn == "symtbl":
        ST = True

    elif fn == "inmem":
        INMEM = True

    elif fn == "nonewvars":
        ALLOWVAR = False

    elif fn == "templates":
        TEMPLATE = True

    elif fn == "temponly":
        TEMPONLY = True

    elif fn == "limitns":
        LIMITNS = True

    elif fn == "litvars":
        LITVARS = True

    elif fn == "nopredefs":
        PREDEFS = False

    elif fn == "debug":
        DEBUG = True


    # Everything else presumed to be a configuration file
    else:

        # Preserve the file name for later use
        name = fn

        # The default is no pre-defined symbols
        st = SymbolTable()

        if ST:
            st = BuildSymTbl(syms)

            # Another way to add pre-defined symbols is to explicitly
            # load them into the symbol table.  We'll the derived
            # complex type we created earlier for this.

            mc1 = mycmplx()
            mc2 = mycmplx()

            mc2.Value     = 8-3.14159j
            mc2.Writeable = False

            # And stuff them into the symbol table

            st.Symbols["MyComplex1"] = mc1
            st.Symbols["MyComplex2"] = mc2

        if LIMITNS:

            # Limit the number of namespaces the user can use
            des = VarDescriptor()
            des.Value = ""              # This is the initial namespace
            des.LegalVals = [r"^NS.*", ] # List of permitted namespace regexes
            des.Min = 3
```

```
            des.Max = 8
            st.Symbols["NAMESPACE"] = des


        # Support for templating

        # Default is no templates
        tl = Template()

        if TEMPLATE:
            tl = BuildSymTbl(templs, istemplate=True)

        # Support for in-memory parsing

        if INMEM:
            fn = []

            try:
                x=open(name)
                fn = x.readlines()
                x.close()

            except:
                print "Cannot Open File: '%s'!" % name

        # Call the parser and process the results

        retval = ParseConfig(fn,
                             CallingProgram="FeeFiFoFum ",
                             InitialSymTable=st,
                             AllowNewVars=ALLOWVAR,
                             Templates=tl,
                             TemplatesOnly=TEMPONLY,
                             LiteralVars=LITVARS,
                             ReturnPredefs=PREDEFS,
                             Debug=DEBUG
                            )

        print "Total Lines Processed: %s " % retval.TotalLines

        if retval.ErrMsgs:
            print "Errors Were Found In '%s'!" % name
        else:
            print "No Errors Found In '%s'!" % name

        print

        # Format and display the results

        dumpreturn("SYMBOL TABLE", retval.Symbols, isdict=True)
        dumpreturn("ERRORS", retval.ErrMsgs)
        dumpreturn("WARNINGS", retval.WarnMsgs)

        if DEBUG:
            dumpreturn("DEBUG", retval.DebugMsgs)

        dumpreturn("LITERAL LINES", retval.LiteralLines)
```