NAME tconfpy.py

Configuration File Support For Python Applications

SYNOPSIS

It is common to provide an external "configuration file" when writing sophisticated applications. This gives the end-user the ability to easily change program options by editing that file.

`tconfpy` is a Python module for parsing such configuration files. `tconfpy` understands and parses a configuration "language" which has a rich set of string-substitution, variable name, conditional, and validation features.

By using `tconfpy`, you relieve your program of the major responsibility of configuration file parsing and validation, while providing your users a rich set of configuration features.

NOTE: Throughout this document we use the term "configuration file". However, `tconfpy` can parse configurations both in files as well as in-memory lists. Whenever you see the term "file", think "a file or a set of configuration statements stored in a list".

ANOTHER NOTE: Throughout this document we refer to "symbols" and "variables" interchangeably. Strictly speaking, this is not really right. A "symbol" is an entry in a symbol table representing the state of some "variable" that is visible to a person writing a configuration. But it's safe to assume you're smart enough to understand this subtlety and know what is meant by context ;)

JUST ONE LAST NOTE: If you run `tconfpy` directly, it will dump version and copyright information, as well as the value of the current predefined System Variables:

```
python tconfpy.py
```

DOCUMENT ORGANIZATION

This document is divided into 4 major sections:

PROGRAMMING USING THE `tconfpy` API discusses how to call the configuration file parser, the options available when doing this, and what the parser returns. This is the "Programmer's View" of the module and provides in-depth descriptions of the API, data structures, and options available to the programmer.

CONFIGURATION LANGUAGE REFERENCE describes the syntax and semantics of the configuration language recognized by `tconfpy`. This is the "User's View" of the package, but both programmers and people writing configuration files will find this helpful.

ADVANCED TOPICS FOR PROGRAMMERS describes some ways to combine the various `tconfpy` features to do some fairly nifty things.

INSTALLATION explains how to install this package on various platforms. This information can also be found in the `READ-1ST.txt` file distributed with the package.

PROGRAMMING USING THE `tconfpy` API

`tconfpy` is a Python module and thus available for use by any Python program. This section discusses how to invoke the `tconfpy` parser, the options available when doing so, and what the parser returns to the calling program.

One small note is in order here. As a matter of coding style and brevity, the code examples here assume the following Python import syntax:

```
from tconfpy import *
```

If you prefer the more pedestrian:

```
import tconfpy
```

you will have to prepend all references to a `tconfpy` object with `tconfpy.`. So `retval=ParseConfig(...` becomes `retval = tconfpy.ParseConfig(...` and so on.

You will also find the test driver code provided in the `tconfpy` package helpful as you read through the following sections. `test-tc.py` is a utility to help you learn and exercise the `tconfpy` API. Perusing the code therein is helpful as an example of the topics discussed below.

Core Objects

In order to use `tconfpy` effectively, you need to be familiar with the objects used to communicate between your program and the parser. This section provides a brief overview of these objects for reference use later in the document.

The Symbol Table Object

A "symbol table" is a `tconfpy` object defined to hold the symbol table proper, the results of the parse (upon return), and some other data structures used internally by the parser.

The full Symbol Table object definition and initialization looks like this:

```
class SymbolTable(object):
    def __init__(self):

        # These items are populated when the parser returns

        self.Symbols        = {}
        self.DebugMsgs      = []
        self.ErrMsgs        = []
        self.WarnMsgs       = []
        self.LiteralLines   = []
        self.TotalLines     = 0
        self.Visited        = []

        # These items are for internal parser use only
        # Never write an application to depend on their content

        self.Templates      = Template()
        self.ALLOWNEWVAR    = True
        self.TEMPONLY       = False
        self.LITERALVARS    = False
        self.INLITERAL      = False
        self.DEBUG          = False
        self.CondStack      = [["", True],]
```

This object is optionally passed to the API when beginning a parse, if you wish to provide an initial symbol table. In this case, only `Symbols` need be populated.

When the parse is complete, an object of this same type is returned. `Symbols`, `DebugMsgs`, `ErrMsgs`, `WarnMsgs`, `LiteralLines`, `TotalLines`, and `Visited` will be populated with the parse results as appropriate. The remaining elements of the object are meaningless when the parser returns, and applications should never make use of them.

The Variable Descriptor Object

The actual symbol table is kept in the `SymbolTable.Symbols` dictionary. There is one entry for each symbol (variable) encountered in the parse. The symbol's name is used as the dictionary key and an object of type `VarDescriptor` as its entry. This "variable descriptor" is an object that describes the variable's current and default values, whether it can be modified, and any validation constraints:

```
class VarDescriptor(object):
    # Default variable type is a writeable string with no constraints
    def __init__(self):
        self.Value     = ""
        self.Writeable = True
        self.Type      = TYPE_STRING
        self.Default   = ""
        self.LegalVals = []
        self.Min       = None
        self.Max       = None
```

The Template Object

As described later in this document, it is possible to pre-define a variable's type, default value, and validation contraint(s) to be applied only if the variable is actually brought into existence in the configuration file. This is a so-called "template". Templates have their own object definition:

```
class Template(object):
    def __init__(self):
        self.Symbols = {}
```

In effect, this is a subset of the `SymbolTable` object. `Template.Symbols` is populated just like `SymbolTable.Symbols` using symbol names and variable descriptors.

API Overview

The `tconfpy` API consists of a single call. Only the configuration file to be processed is a required parameter, all the others are optional and default as described below:

```
from tconfpy import *

retval = ParseConfig(cfgfile,
                     CallingProgram="tconfpy version-num",
                     InitialSymTable=SymbolTable(),
                     AllowNewVars=True,
```

```
              Templates=Template(),
              TemplatesOnly=False,
              LiteralVars=True,
              ReturnPredefs=True,
              Debug=False
             )
```

where:

cfgfile (Required Parameter - No Default)

Declares where the configuration is found. If this parameter is a **string,** it is treated as the name of a file to parse. If this parameter is a **list,** `tconfpy` presumes this to be an in-memory configuration, and parses the list in sequential order, treating each entry as a configuration line. This allows you to use `tconfpy` for parsing either configuration files or in-memory configurations. If you pass anything other than a string or list here, `tconfpy` will produce an error.

If you do pass the API an in-memory list, `tconfpy` treats each entry as a line in a configuration "file". However, this means that each element of the list must be a **string.** The parser checks this first and only processes entries in the list that are strings. Entries of any other type produce an error and are ignored.

CallingProgram (Default: `tconfpy + Version Number`**)**

Change the prompt introducer string used for Debug, Error, and Warning messages.

By default, each time `tconfpy` produces an Error, Warning, or Debug message, it prepends it with the string `tconfpy` followed by the version number. Since `tconfpy` is designed to be called from applications programs, you may wish to substitute your own program name or other information in these messages. You do so by setting the `CallingProgram` keyword to the desired text.

InitialSymTable (Default: `SymbolTable()`**)**

Used to pass a pre-initialized Symbol Table from the application to the parser. Defaults to an empty symbol table.

AllowNewVars (Default: `True`**)**

Allow the user to create new variables in the configuration file.

Templates (Default: `Template()`**)**

This option is used to pass variable templates to the parser. If present, `tconfpy` expects this option to pass an object of type `Template`. See the section below entitled, **Using Variable Templates** for all the details. By default, an empty template table (no templates) is passed to the parser.

TemplatesOnly (Default: `False`)

If this option is set to `True`, `tconfpy` will not permit a new variable to be created unless a variable template exists for it. By default, `tconfpy` will use a variable template if one is present for a new variable, but it does not require one. If a new variable is created, and no Template exists for it, the variable is just created as a string type with no restrictions on content or length. When this option is set to `True`, then a template **must** exist for each newly created variable.

LiteralVars (Default: `False`)

If set to `True`, this option enables variable substitutions within `.literal` blocks of a configuration file. See the section in the language reference below on `.literal` usage for details.

ReturnPredefs (Default: `True`)

`tconfpy` "prefefines" some variables internally. By default, these are returned in the symbol table along with the variables actually defined in the configuration file. If you want a "pure" symbol table - that is, a table with **only** your variables in it - set this option to `False`.

Debug (Default: `False`)

If set to `True`, `tconfpy` will provide detailed debugging information about each line processed when it returns.

retval An object of type `SymbolTable` used to return parsing results. The results of the parse will be returned in various data structures:

```
SymbolTable.Symbols      => All symbols and their values as a result of the pa
SymbolTable.DebugMsgs    => Any Debug Messages if debug was requested
SymbolTable.ErrMsgs      => Any Error Messages
SymbolTable.WarnMsgs     => Any Warning Messages
SymbolTable.LiteralLines => Any Literal Text found in the configuration file
SymbolTable.TotalLines   => Total number of lines processed
SymbolTable.Visited      => List of configuration files processed
```

You can tell whether a parse was successful by examining `ErrMsgs`. A successful parse will leave this data structure empty (though there may well be Warning Messages present in `WarnMsgs`.)

The Initial Symbol Table API Option

The simplest way to parse a configuration is just to call the parser with the name of a file or an in-memory list containing the configuration statements:

```
retval = ParseConfig("MyConfigFile")

          - OR -

myconfig = [configline1, configline2, ....]
```

```
retval = ParseConfig(myconfig)
```

Assuming your configuration is valid, `ParseConfig()` will return a symbol table populated with all the variables defined in the configuration and their associated values. This symbol table will have **only** the symbols defined in that configuration (plus a few built-in and predefined symbols needed internally by `tconfpy`).

However, the API provides a way for you to pass a "primed" symbol table to the parser that contains predefined symbols/values of your own choosing. Why on earth would you want to do this? There are a number of reasons:

- You may wish to write a configuration file which somehow depends on a predefined variable that only the calling program can know:

  ```
  .if [APPVERSION] == 1.0
       # Set configuration for original application version
  .else
       # Set configuration for newer releases
  .endif
  ```

  In this example, only the calling application can know its own version, so it sets the variable `APPVERSION` in a symbol table which is passed to `ParseConfig()`.

- You may wish to "protect" certain variable names be creating them ahead of time and marking them as "Read Only" (`VarDescriptor.Writeable=False`). This is useful when you want a variable to be available for use within a configuration file, but you do not want users to be able to change its value. In this case, the variable can be referenced in a string substitution or conditional test, but cannot be changed.

- You may want to place limits on what values can be assigned to a particular variable. When a variable is newly defined in a a configuration file, it just defaults to being a string variable without any limits on its length or content (unless you are using Variable Templates). But variables that are created by a program have access to the variable's "descriptor". By setting various attribues of the variable descriptor you can control variable type, content, and range of values. In other words, you can have `tconfpy` "validate" what values the user assigns to particular variables. This substantially simplifies your application because no invalid variable value will ever be returned from the parser.

How To Create An Initial Symbol Table

A `tconfpy` "Symbol Table" is really nothing more than a Python dictionary stored inside a container object. The key for each dictionary entry is the variable's name and the value is a `tconfpy`-specific object called a "variable descriptor". Creating new variables in the symbol table involves nothing more than this:

```
from tconfpy import *

# Create an empty symbol table
MySymTable = SymbolTable()

# Create descriptor for new variable
MyVarDes = VarDescriptor()

# Code to fiddle with descriptor contents goes here
```

```
MyVarDes.Value = "MyVal"

# Now load the variable into the symbol table
MySymTable.Symbols["MyVariableName"] = MyVarDes

# Repeat this process for all variables, then call the parser

retval = ParseConfig("MyConfigFile", InitialSymTable=MySymTable)
```

The heart of this whole business the `VarDescriptor` object. It "describes" the value and properties of a variable. These descriptor objects have the following attributes and defaults:

```
VarDescriptor.Value     = ""
VarDescriptor.Writeable = True
VarDescriptor.Type      = TYPE_STRING
VarDescriptor.Default   = ""
VarDescriptor.LegalVals = []
VarDescriptor.Min       = None
VarDescriptor.Max       = None
```

When `tconfpy` encounters a new variable in a configuration file, it just instantiates one of these descriptor objects with these defaults for that variable. That is, variables newly-defined in a configuration file are entered into the symbol table as string types, with an initial value of "" and with no restriction on content or length.

But, when you create variables under program control to "prime" an initial symbol table, you can modify the content of any of these attributes for each variable. These descriptor attributes are what `tconfpy` uses to validate subsequent attempts to change the variable's value in the configuration file. In other words, modifying a variable's descriptor tells `tconfpy` just what you'll accept as "legal" values for that variable.

Each attribute has a specific role:

VarDescriptor.Value (Default: `Empty String`**)**

Holds the current value for the variable.

VarDescriptor.Writeable (Default: `True`**)**

Sets whether or not the user can change the variable's value. Setting this attribute to `False` makes the variable **Read Only.**

VarDescriptor.Type (Default: `TYPE_STRING`**)**

One of `TYPE_BOOL`, `TYPE_COMPLEX`, `TYPE_FLOAT`, `TYPE_INT`, or `TYPE_STRING. This` defines the type of the variable. Each time `tconfpy` sees a value being assigned to a variable in the configuration file, it checks to see if that variable already exists in the symbol table. If it does, the parser checks the value being assigned and makes sure it matches the type declared for that variable. For example, suppose you did this when defining the variable, `foo`:

```
VarDescriptor.Type = TYPE_INT
```

Now suppose the user puts this in the configuration file:

```
foo = bar
```

This will cause a type mismatch error because `bar` cannot be coerced into an integer type - it is a string.

As a general matter, for existing variables, `tconfpy` attempts to coerce the right-hand-side of an assignment to the type declared for that variable. The least fussy operation here is when the variable is defined as TYPE_STRING because pretty much everything can be coerced into a string. For example, here is how `foo = 3+8j` is treated for different type declarations:

```
VarDescriptor.Type        VarDescriptor.Value
------------------        -------------------

TYPE_BOOL                 Type Error
TYPE_COMPLEX              3+8j         (A complex number)
TYPE_FLOAT                Type Error
TYPE_INT                  Type Error
TYPE_STRING               "3+8j"     (A string)
```

This is why the default type for newly-defined variables in the configuration file is TYPE_STRING: they can accept pretty much **any** value.

NOTE: `tconfpy` always stores a variable's value in it native type in the symbol table entry for that variable - i.e., You'll get the variable's value back from the parser in the proper type. However, when the variable is actually referenced within a configuration file, it is converted to a **string** for purposes of configuration processing.

For instance, when doing conditional comparisons, the parser coerces the variable's value into a string for purposes of the comparsion. Say you have the floating point variable `myfloat` whose value is `6.023` and the configuration file contains a statement like:

```
.if myfloat == 6.023
```

When doing this conditional check, the parser will convert the current value of `myfloat` into a string and compare it to the **string** `"6.023"` on the Right Hand Side.

Similarly, variables are coerced as strings when they are referenced in substitutions:

```
# Assume 'myfloat' has been predefined to be a floating point variable
# Assume 'mybool' has been predefined to be a boolean variable

myfloat = 3.14
mybool  = True

myvar   = [myfloat] is [mybool]

# This sets 'myvar' to the string '3.14 is True'
```

This can be tricky when dealing with Boolean variables. As described later in this document, you can do conditional tests based on the **state** of a Boolean, but if you do this:

```
.if myboolean == whatever...
```

The parser converts `myboolean` to a **string** of either `"True"` or `"False"`, so watch out for this. As a general matter, you're more likely to need the boolean tests that check the state of the variable, but the above construct is handy if you want to use regular string variables to control conditional bodies:

```
 MYCONTROL = True
.if myboolean == MYCONTROL
```

Where, `MYCONTROL` is a regular old string variable - i.e., It has not been defined to be a Boolean by either a Template or Initial Symbol table passed to the parser.

VarDescriptor.Default (Default: `Empty String`**)**

This is a place to store the default value for a given variable. When a variable is newly-defined in a configuration file, `tconfpy` places the first value assigned to that variable into this attribute. For variables already in the symbol table, `tconfpy` does nothing to this attribute. This attribute is not actually used by `tconfpy` for anything. It is provided as a convenience so that the calling program can easily keep track of each variable's default value. This makes it easy to do things like "reset" every variable when restarting a program, for example.

VarDescriptor.LegalVals (Default: `[]`**)**

Sometimes you want to limit a variable to a specific set of values. That's what this attribute is for. `LegalVals` explictly lists every legal value for the variable in question. If the list is empty,then this validation check is skipped.

The exact semantics of LegalVals varies depending on the type of the variable.

```
Variable Type               What LegalVals Does
-------------               -------------------
Boolean                     Nothing - Ignored

Integer, Float, Complex     List of numeric values the
                            user can assign to this variable

                            Examples: [1, 2, 34]
                                      [3.14, 2.73, 6.023e23]
                                      [3.8-4j, 5+8j]

String                      List of Python regular expressions.
                            User must assign a value to this
                            variable that matches at least
                            one of these regular expressions.

                            Example:   [r'a+.*', r'^AnExactString$']
```

The general semantic here is "If Legal Vals is not an empty list, the user must assign a value that matches one of the items in LegalVals."

One special note applies to `LegalVals` for string variables. `tconfpy` always assumes that this list contains Python regular expressions. For validation, it grabs each entry in the list, attempts to compile it as a regex, and checks to see if the value the user wants to set matches. If you define an illegal regular expression here, `tconfpy` will catch it and produce an appropriate error.

You may also want to specify a set of legal strings that are **exact matches** not open-ended regular expressions. For example, suppose you have a variable, `COLOR` and you only want the user to be able to only set it to one of, `Red`, `White`, or `Blue`. In that case, use the Python regular expression metacharacters that indicate "Start Of String" and "End Of String" do do this:

```
des                = VarDescriptor()
des.LegalVals      = [r'^Red$', r'^White$', r'^Blue$']
...

SymTable['COLOR'] = des
```

NOTE: If you want this test to be skipped, then set `LegalVals` to an empty list, []. (This is the default when you first create an instance of `tconfpy.VarDescriptor`.) Do not set it to a Python `None` or anything else. `tconfpy` expects this attribute to be a list in every case.

VarDescriptor.Min and VarDescriptor.Max (Default: `None`)

These set the minimum and maxium legal values for the variables, but the semantics vary by variable type:

```
Variable Type              What Min/Max Do
-------------              ---------------

Boolean, Complex           Nothing - Ignored

Integer, Float             Set Minimum/Maxium allowed values.

String                     Set Minimum/Maximum string length
```

In all cases, if you want either of these tests skipped, set `Min` or `Max` to the Python `None`.

All these various validations are logically "ANDed" together. i.e., A new value for a variable must be allowed AND of the appropriate type AND one of the legal values AND within the min/max range.

`tconfpy` makes no attempt to harmonize these validation conditions with each other. If you specify a value in `LegalVals` that is, say, lower than allowed by `Min` you will always get an error when the user sets the variable to that value: It passed the `LegalVals` validation but failed it for `Min`.

The Initial Symbol Table And Lexical Namespaces

The **CONFIGURATION LANGUAGE REFERENCE** section below discusses lexical namespaces in some detail from the user's point-of-view. However, it is useful for the programmer to understand how they are implemented.

tconfpy is written to use a predefined variable named NAMESPACE as the place where the current namespace is kept. If you do not define this variable in the initial symbol table passed to the parser, tconfpy will create it automatically with an initial value of "".

From a programmer's perspective, there are are few important things to know about namespaces and the NAMESPACE variable:

- You can manually set the initial namespace to something other than "". You do this by creating the NAMESPACE variable in the initial symbol table passed to the parser, and setting the Value attribute of its descriptor to whatever you want as the initial namespace. At startup tconfpy will check this initial value to make sure it conforms to the rules for properly formed names - i.e., It it will check for blank space, a leading $, the presence of square brackets, and so on. If the initial namespace value you provide is illegal, tconfpy will produce an error and reset the initial namespace to "".

- Because lexical namespaces are implemented by treating NAMESPACE as just another variable, all the type and value validations available for string variables can be applied to NAMESPACE. As discussed above, this means you can limit the length and content of what the user assigns to NAMESPACE. In effect, this means you can limit the number and name of namespaces available for use by the user. There is one slight difference here than for other variables. **The root namespace is always legal,** regardless of what other limitations you may impose via the LegalVals, Min, and Max attributes of the NAMESPACE variable descriptor.

- When the call to ParseConfig() completes, the Value attribute of the NAMESPACE variable descriptor will contain the namespace that was in effect when the parse completed. i.e., It will contain the last namespace used.

How The tconfpy Parser Validates The Initial Symbol And Template Tables

When you pass an initial symbol and/or template table to the parser, tconfpy does some basic validation that the table contents properly conform to the VarDescriptor format and generates error messages if it finds problems. However, the program does **not** check your specifications to see if they make sense. For instance if you define an integer with a minimum value of 100 and a maximum value of 50, tconfpy cheerfully accepts these limits even though they are impossible. You'll just be unable to do anything with that variable - any attempt to change its value will cause an error to be recorded. Similarly, if you put a value in LegalVals that is outside the range of Min to Max, tconfpy will accept it quietly.

In the case of templates, tconfpy all makes sure that they are all named "canonically". That is, a template name may not itself contain a namespace. This effectively means that there can be no namespace separator characters (".") in the template name.

The AllowNewVars API Option

By default, tconfpy lets the user define any new variables they wish in a configuration file, merely by placing a line in the file in this form:

```
Varname = Value
```

However, you can disable this capability by calling the parser like this:

```
retval = ParseConfig("myconfigfile", AllowNewVars=False)
```

This means that the configuration file can "reference" any predefined variables, and even change their values (if they are Writeable), but it cannot create **new** variables.

This feature is primarily intended for use when you pass an initial symbol table to the parser and you do not want any other variables defined by the user. Why? There are several possible uses for this option:

- You know every configuration variable name the calling program will use ahead of time. Disabling new variable names keeps the configuration file from getting cluttered with variables that the calling program will ignore anyway, thereby keeping the file more readable.

- You want to insulate your user from silent errors caused by misspellings. Say your program looks for a configuration variable called `MyEmail` but the user enters something like `myemail = foo@bar.com`. `MyEmail` and `myemail` are entirely different variables and only the former is recognized by your calling program. By turning off new variable creation, the user's inadvertent misspelling of the desired variable name will be flagged as an error.

  Note, however, that there is one big drawback to disabling new variable creation. `tconfpy` processes the configuration file on a line-by-line basis. No line "continuation" is supported. For really long variable values and ease of maintenance, it is sometimes helpful to create "intermediate" variables that hold temporary values used to construct a variable actually needed by the calling program. For example:

  ```
  inter1 = Really, really, really, really, long argument #1
  inter2 = Really, really, really, really, long argument #2

  realvar = command [inter1] [inter2]
  ```

  If you disable new variable creation you can't do this anymore unless all the variables `inter1`, `inter2`, and `realvar` are predefined in the initial symbol table passed to the parser.

Using Variable Templates

By default, any time a new variable is encountered in a configuration file, it is created as a string type with no restrictions on its content or length. As described above, you can predefine the variable in the initial symbol table you pass to the parser. This allows you to define that variable's type and to optionally place various restrictions on the values it may take. In other words, you can "declare" the variable ahead of time and `tconfpy` will do so-called "type and value enforcement".

"Variable Templates" are a related kind of idea, with a bit of a twist. They give you a way to "declare" variable type and content restrictions for selected **new variables** discovered in the configuration file. In other words, by using Variable Templates, you can make sure that a new variable also has restrictions placed on its type and/or values.

The obvious question here is, "Why not just do this by predefining every variable of interest in the initial symbol table passed to the parser?" There are several answers to this:

- The `tconfpy` configuration language has very powerful "existential" conditional tests. These test to see if a variable "exists". If you predefine every variable you will ever need, then the kinds of existential tests you can do will be somewhat limited (since every variable **does** already exist).

  With Variable Templates, you can define the type and value constraints of a variable which will be applied, **but only if you actually bring that variable into existence.** This allows constructs like this to work:

```
.if [.PLATFORM] == posix
    posix = True
.endif

.if [.PLATFORM] == nt
    nt = True
.endif

.ifall posix
    ...
.endif

.ifall nt
    ...
.endif

.ifnone posix nt
    ...
.endif
```

In this example, notice that the variables `posix` and `nt` may- or may not be actually created, depending on the value of `.PLATFORM`. The logic later in the example depends upon this. If you were to predefine these two variables (to specify type and/or value restrictions), this type of logical flow would not be possible.

By providing Variable Templates for `posix` and `nt`, you can define their type (likely Boolean in this case) ahead of time **and this will be applied if the variable does come into existence.**

- The other reason for Variable Templates is more subtle, but gives `tconfpy` tremendous versatility beyond just processing configuration files. Variable Templates give you a way to use `tconfpy` to build data validation tools.

  Suppose you have a list of employee records exported in this general format (easy to do with most databases):

  ```
  [Employee#]
  LastName  = ...
  FirstName = ...
  Address   = ...
  City      = ...

  ... and so on
  ```

  By using the empoyee's ID as a lexical namespace, we end up creating new variables for each employee. Say the employee number is `1234`. Then we would get, `1234.LastName`, `1234.FirstName`, and so on.

  Now, here's the subtle part. Notice that the type and content restrictions of these variables is likely to be the **same** for each different employee.

By defining Variable Templates for each of the variables we intend to use over and over again in different namespace contexts, we can **validate** each of them to make sure their content, type, length, and so forth are correct. This makes it possible to use `tconfpy` as the underpinnings of a "data validation" or "cleansing" program.

- Another way to look at this is that Variable Templates give you a way to define type/value restrictions on an entire "class" of variables. Instead of having to explictly predefine variables for every employee in our example above, you just define templates for the variable set that is common to all employees. This is **way** simpler than predefining every possible variable combination ahead of time.

The `Templates` And `TemplatesOnly` API Options

Variable Templates are supported with two API options: `Templates` And `TemplatesOnly`. `Templates` is used to pass a symbol table (separate from the main symbol table) containing the Variable Templates. By default, this option is set to an object of type `Template` containing no templates.

So what exactly is a "Variable Template"? It is the **exact same thing** as a predefined variable you might pass in the initial symbol table. In other words, it is a Python dictionary entry where the key is the variable name and the entry is in `VarDescriptor` format. The big difference here is that normal variables are stored in the symbol table in a `SymbolTable` container object. But templated variables are stored in a `Template` container object. (See **Core Objects** above for the details.)

Templated variables are thus defined (by the calling program) just like you would any other variable - but stored in a different place:

```
from tconfpy import *

# Create an empty template table
MyTemplates = Template()

# Create descriptor for new variable
MyTemplateVarDes = VarDescriptor()

# Code to fiddle with descriptor contents goes here
MyTemplateVarDes.Type  = TYPE_INT

... and so forth

# Now load the variable into the symbol table
MyTemplates.Symbols["MyTemplateName"] = MyTemplateVarDes

# Now pass the Templates to the parser
retval = ParseConfig("myfile", Templates=MyTemplates)
```

NOTE: You may optionally pass either an initial symbol table or a template table or both to the parser when you call it. That is, the initial set of symbols is disjoint from any templates you've defined.

Semantically, the only difference between "regular" and templated variables, is that a templated variable does not come into existence (i.e. Become an entry) in the main symbol table until a variable by that **name** is encountered in the configuration file. Then the variable is created using the template as its entry in the main symbol table.

For example:

```
[1234]
LastName  = Jones
FirstName = William
Address   = 123 Main Street
City      = Anywhere
State     = WI
ZIP       = 00000-0000

[1235]
LastName  = Jones
FirstName = Susan
Address   = 123 Main Street
City      = Anywhere
State     = WI
ZIP       = 00000-0000
```

Suppose you define variable templates for `LastName`, `FirstName`, `Address`, and so on. That is, you define variables by these names, and define whatever type and content restrictions you want in each of their `VarDescriptors`. You then pass these to the parser via the `Templates=` option.

As `tconfpy` parses the file and encounters the new variables `1234.LastName` ... `1235.ZIP`, it uses the following
"rules" when creating new variables:

- See if there is a template variable whose name is the same as the "base" name of the new variable. (The "base" name is just the variable name without the prepended namespace.)

  If there is a template with a matching name, see if the value the user wants to assign to that variable passes all the type/validation rules. If so, load the variable into the symbol table and set its value as requested, **using the `VarDescriptor` object from the template.** (This ensures that future attempts to change the variable's value will also be type/validation checked.)

  If the assignment fails the validation tests, issue an appropriate error and do **not** create the variable in the symbol table.

- If there is no template with a matching name, then just create a new variable as usual - string type with no restrictions, **unless** `TemplatesOnly` is set to `True`. Setting this option to `True` tells the program that you want to allow the creation of **only** those variables for which templates are defined. This is a way to restrict just what new variables can be created in any namespace. `TemplatesOnly` defaults to `False` which means you can create new variables even when no template for them exists.

A few subtleties of templating variables are worth noting here:

- The same template is used over and over again to create a new variable of the same name **in different namespaces.** For example, suppose you've defined a template with the name, `AccountNumber`:

  ```
  [ComputerSupplier]
  AccountNumber = 1234-5

  [Lawyer]
  AccountNumber = 3456-3
  ```

This would create two separate variables in the symbol table, based on the same template: `ComputerSupplier.AccountNumber` and `Lawyer.AccountNumber`.

This works because `tconfpy` checks the so-called "canonical" name of a variable when it is being created (the part of the name without the namespace information) to see if a template exists for it. For this reason, **a template name must never contain a namespace.** If you attempt to create templates with names like `Foo.Templatename`, the parser will reject it and produce an error.

- Notice that for variables actually in the symbol table, `VarDescriptor.Value` holds the current value for the variable. However, this field is meaningless in a template. The template is only used when creating a new variable **to be added the normal symbol table.** The value field of the template's variable descriptor is never used for anything - it is never read nor is it ever set.

- By definition, a templated variable does not actually exist (in the symbol table) until you assign it some value in the configuration file. This means that even if you mark the variable as Read Only in its template, you are able to set it one time - to actually create it. Thereafter, the variable exists in the symbol table with it's `Writeable` attribute set to `False` and future changes to the variable are prevented

In summary, Variable Templates give you a way to place restrictions on variable type and content **in the event that the variable actually comes into existence.** They also give you a way to define such restrictions for an entire class of variables without having to explicitly name each such variable ahead of time. Finally, Variable Templates are an interesting way to use `tconfpy` as the basis for data validation programs.

The `LiteralVars` API Option

`tconfpy` supports the inclusion of literal text anywhere in a configuration file via the `.literal` directive. This directive effectively tells the `tconfpy` parser to pass every line it encounters "literally" until it sees a corresponding `.endlinteral` directive. By default, `tconfpy` does **exactly** this. However, `tconfpy` has very powerful variable substitution mechanisms. You may want to embed variable references in a "literal" block and have them replaced by `tconfpy`.

Here is an example:

```
MyEmail = me@here.com   # This defines variable MyEmail

.literal
    printf("[MyEmail]");  /* A C Statement */
.endliteral
```

By default, `ParseConfig()` will leave everything within the `.literal/.endliteral` block unchanged. In our example, the string:

```
printf("[MyEmail]");  /* A C Statement */
```

would be in the list of literals returned by `ParseConfig()`.

However, we can ask `tconfpy` to do variable replacement **within** literal blocks by setting `LiteralVars=True` in the `ParseConfig()` call:

```
retval = ParseConfig("myconfigfile", LiteralVars=True)
```

In this example, `tconfpy` would return:

```
printf("me@here.com");  /* A C Statement */
```

At first glance this seems only mildly useful, but it is actually very handy. As described later in this document, `tconfpy` has a rich set of conditional operators and string substitution facilities. You can use these features along with literal block processing and variable substitution within those blocks. This effectively lets you use `tconfpy` as a preprocessor for **any** other language or text.

The `ReturnPredefs` API Option

As described below, `tconfpy` internally "predefines" a number of variables. These include variables that describe the current runtime environment as well as variables that substitute for language keywords.

These predefined variables are just stored in the symbol table like any other variable. By default, they are returned with all the "real" variables discovered in the configuration file. If you want **only** the variables actually encountered in the configuration file itself, set `ReturnPredefs=False` in the `ParseConfig()` API call. This will cause `tconfpy` to strip out all the predefined variables before returning the final symbol table.

Note that this option also removes the `NAMESPACE` variable since it is understood to also be outside the configuration file (even though you may have passed an initial version of `NAMESPACE` to the parser).

Note, also, that this option applies only to the variables predefined by `tconfpy` itself. Any variables **you** predefine when passing an initial symbol table will be returned as usual, regardless of the state of this option.

The `Debug` API Option

`tconfpy` has a fairly rich set of debugging features built into its parser. It can provide some detail about each line parsed as well as overall information about the parse. Be default, debugging is turned off. To enable debugging, merely set `Debug=True` in the API call:

```
retval = ParseConfig("myconfigfile", Debug=True)
```

How `tconfpy` Processes Errors

As a general matter, when `tconfpy` encounters an error in the configuration file currently being parsed, it does two things. First, it adds a descriptive error message into the list of errors returned to the calling program (see the next section). Secondly, in many cases, noteably during conditional processing, it sets the parser state so the block in which the error occurred is logically `False`. This does not happen in every case, however. If you are having problems with errors, enable the Debugging features of the package and look at the debug output. It provides detailed information about what caused the error and why.

`tconfpy` Return Values

When `tconfpy` is finished processing the configuration file, it returns an object (of type `SymbolTable`) that contains the entire results of the parse. This includes a symbol table, any relevant error or warning messages, debug information (if you requested this via the API), and any "literal" lines encountred in the configuration.

NOTE: Because the object is of type `SymbolTable`, it contains other data structures used by the parser itself. These are purposely "cleaned out" before the parser returns and should never be used by the calling application for any reason. In other words, use only the data structures documented below when the parser returns control to your calling program.

The return object is an instance of the class `twander.SymbolTable` which has been populated with the results of the parse. In the simplest case, we can parse and extract results like this:

```
from tconfpy import *

retval = ParseConfig("myconfigfile", Debug=True)
```

`retval` now contains the results of the parse:

retval.Symbols

A Python dictionary which lists all the defined symbols and their associated values. A "value" in this case is always an object of type `tconfpy.VarDescriptor` (as described above).

retval.DebugMsgs

A Python list containing detailed debug information for each line parsed as well as some brief summary information about the parse. `retval.DebugMsgs` defaults to an empty list and is only populated if you set `Debug=True` in the API call that initiated the parse (as in the example above).

retval.ErrMsgs

A Python list containing error messages. If this list is empty, you can infer that there were no parsing errors - i.e., The configuration file was OK.

retval.WarnMsgs

A Python list containing warning messages. These describe minor problems not fatal to the parse process, but that you really ought to clean up in the configuration file. It is possible to create a configuration that produces no errors but does produce warnings, for example. It is almost always the case that this configuration is not being handled the way you intended. As a general matter of good practice, apply "belt and suspenders" rules in your applications, and demand that a configration be free of both errors and warnings before proceding.

retval.LiteralLines

As described below, the `tconfpy` configuration language supports a `.literal` directive. This directive allows the user to embed literal text anywhere in the configuration file. This effectively makes `tconfpy` useful as a preprocessor for any other language or text. `retval.LiteralLines` is a Python list containing all literal text discovered during the parse. The lines appear there in the order they were discovered in the configuration file.

retval.TotalLines

Contains a count of the number of the total number of lines processed during the parse.

retval.Visited

Contains a list of all the configuration files and/or in-memory configurations processed.

CONFIGURATION LANGUAGE REFERENCE

`tconfpy` recognizes a full-featured configuration language that includes variable creation and value assignment, a full preprocessor with conditionals, type and value enforcement, and lexical namespaces. This section of the document describes that language and provides examples of how each feature can be used.

`tconfpy` Configuration Language Syntax

`tconfpy` supports a fairly simple and direct configuration language syntax:

- Each line is treated independently. There is no line "continuation".
- The `#` can begin a comment anywhere on a line. This is done blindly. If you need to embed this symbol somewhere within a variable value, use the `[HASH]` variable reference.
- Whitespace is (mostly) insignificant. Leading and trailing whitespace is ignored, as is whitespace around comparison operators. However, there are some places where whitespace matters:

  Variable names may not contain whitespace

  Directives must be followed by whitespace if they take other arguments.

  When assigning a value to a string variable, whitespace within the value on the right-hand-side is preserved. Leading- and trailing whitespace around the right-hand-side of the assignment is ignored.

  Whitespace within both the left- and right-hand-side arguments of a conditional comparison (`.if ... == / != ...`) is significant for purposes of the comparison.

- Case is always significant except when assigning a value to Booleans (described in the section below entitled, **Some Notes On Boolean Variables**).
- Regardless of a variable's type, all variable references return **a string representation of the variable's value!** This is done so that the variable's value can be used for comparison testing and string substitution/concatenation. In other words, variables are stored in their native type in the symbol table that is returned to the calling program. However, they are treated as strings during the parsing of the configuration file whenever they are used in a comparison test or in a substitution.

- Text inside a literal block (see section below on the `.literal` directive) is left untouched. Whitespace, the `#` symbol, and so on are not intepreted in any way and are passed back to the calling program as-is. The one exception to this rule is when variable substitution inside literal blocks is enabled. This is discussed in a later section of this document as well.

- Any line which does not conform to these rules and/or is not in the proper format for one of the operations described below, is considered an error.

Creating Variables And Assigning A Value

The heart of a configuration file is a "variable". Variables are stored in a "Symbol Table" which is returned to the calling program once the configuration file has been processed. The calling program can predefine any variables it wishes before processing a configuration file. You can normally also define your own new variables in the configuration file as desired (unless the programmer has inhibited new variable creation).

Variables are assigned values like this:

```
MyVariable = Some string of text
```

If `MyVariable` is a new variable, `tconfpy` will create it on the spot. If it already exists, `tconfpy` will first check and make sure that `Some string of text` is a legal value for this variable. If not, it will produce an error and refuse to change the current value of `MyVariable`.

Anytime you create a new variable, the first value assigned to it is also considered its "default" value. This may (or may not) be meaningful to the application program.

Variables which are newly-defined in a configuration file are always understood to be **string** variables - i.e., They hold "strings" of text. However, it is possible for the applications programmer to predefine variables with other types and place limitations on what values the variable can take and/or how short or long a string variable may be. (See the previous section, **PROGRAMMING USING THE** `tconfpy` **API** for all the gory details.)

The programmer can also arrange for the configuration file to only have access to variables predefined by the program ahead of time. In that case, if you try to create a new variable, `tconfpy` will produce an appropriate error and the new variable will not be created.

Variable Names

Variables can be named pretty much anything you like, with certain restrictions:

- Variable names may not contain whitespace.

- Variable names may not begin with the `$` character. The one exception to this is when you are referencing the value of an environment variable. References to environment variables begin with `$`:

```
# A reference to an environment variable is legal
x = [$TERM]

# Attempting to create a new variable starting with $ is illegal
$MYVAR = something
```

- Variable names cannot have the `#` character anywhere in them because `tconfpy` sees that character as the beginning a comment.

- Variable names cannot begin with `.` character. `tconfpy` understands a leading period in a variable name to be a "namespace escape". This is discussed in a later section on lexical namespaces.

- Variable names cannot contain the `[` or `]` characters. These are reserved symbols used to indicate a variable **reference.**

- You cannot have a variable whose name is the empty string. This is illegal:

```
= String
```

- The variable named `NAMESPACE` is not available for your own use. `tconfpy` understands this variable to hold the current lexical namespace as described later in this document. If you set it to a new value, it will change the namespace, so be sure this is what you wanted to do.

Getting And Using The Value Of A Variable

You can get the value of any currently defined variable by **referencing** it like this:

```
.... [MyVariable] ...
```

The brackets surrounding any name are what indicate that you want that variable's value.

You can also get the value of any Environment Variable on your system by naming the variable with a leading `$`:

```
... [$USER] ...   # Gets the value of the USER environment variable
```

However you cannot set the value of an environment variable:

```
$USER = me   # This is not permitted
```

This ability to both set and retrieve variable content makes it easy to combine variables through "substitution":

```
MYNAME = Mr. Tconfpy
MYAGE  = 101

Greeting = Hello [MYNAME], you look great for someone [MYAGE]!
```

Several observations are worth noting here:

- The substitution of variables takes place as soon as the parser processes the line `Greeting = ....` That is, variable substitution happens as it is encountered in the configuration file. The only exception to this is if an attempt is made to refer to an undefined/non-existent variable. This generates an error.

- The variable `Greeting` now contains the **string** "Hello Mr. Tconfpy, you look great for someone 101!" This is true even if variable `MYAGE` has been defined by the calling program to be an integer type. To repeat a previously-made point: All variable substitution and comparison operations in a configuration file are done with **strings** regardless of the actual type of the variables involved.

- Variables must be **defined** before they can be **referenced.** `tconfpy` does not support so-called "forward" references.

- Unless a variable as been marked as "Read Only" by the application program, you can continue to change its value as you go. Simply adding another line at the end of our example above will change the value of `Greeting` to something new:

  ```
  Greeting = Generic Greeting Message
  ```

  In other words, the last assignment statement for a given variable "wins". This may seem sort of pointless, but it actually has great utility. You can use the `.include` directive to get, say, a "standard" configuration provided by the system administrator for a particular application. You can then selectively override the variables you want to change in your own configuration file.

Indirect Variable Assignment

The dereferencing of a variable's value can take place on either the right- or **left-hand-side** of an assignment statement. This means so-called "indirect" variable assignments are permitted:

```
CurrentTask    = HouseCleaning
[CurrentTask]  = Dad
```

To understand what this does you need to realize that before `tconfpy` does anything with a statement in a configuration file, it replaces every variable reference with its associated value (or produces an error for references to non-existent variables). So the second statement above is first converted to:

```
HouseCleaning = Dad
```

i.e., The value `Dad` is assigned to a (new) variable called `HouseCleaning`. In other words, putting a variable reference on the left-hand-side of an assignment like this allows you to access another variable which is named "indirectly".

You have to be careful when doing this, though. Consider a similar, but slightly different example:

```
CurrentTask    = House Cleaning   # This is fine
[CurrentTask]  = Dad              # Bad!
```

The reason this no longer works is that the indirect reference causes the second line to parse to:

```
House Cleaning = Dad
```

This is illegal because whitespace is not permitted in variable names. `tconfpy` will produce an error if it sees such a construct. As a general matter, any variable you construct through this indirection method must still conform to all the rules of variable naming: It cannot contain whitespace, begin with `$`, contain `#`, `[`, or `]` and so on.

Another example of how indirection can "bite" you is when the value of the variable begins with a period. As you'll see in the following section on Lexical Namespaces, a variable name beginning with a period is understood to be an "absolute" variable name reference (relative to the root namespace). This can cause unexpected (though correct) behavior when doing indirect variable access:

```
NAMESPACE = NS1
foo   = .bar      # Creates variable NS1.foo with value .bar
[foo] = baz       # Means [NS1.foo] = baz
```

The second assignment statement in this example does not do what you might initially think. Remember, `tconfpy` always does variable dereferencing before anything else, so the second statement becomes:

```
.bar = baz
```

As you'll see in the section on Lexical Namespaces below, this actually means, "Set the variable `bar` in the root namespace to the value `baz`." In other words, if you do indirect variable assignment, and the content of that variable begins with a period, you will be creating/setting a variable in **the root namespace.** Be sure this is what you intended to do.

Get into the habit of reading `[something]` as, "The current value of `something`". See if you understand what the following does (if you don't, try it out with `test-tc.py`):

```
foo   = 1
bar   = 2
[foo] = bar
[bar] = [foo]
```

You can get pretty creative with this since variable references can occur pretty much anywhere in an assignment statement. The only place they cannot appear is **within** another variable reference. That is, you cannot "nest" references:

```
# The Following Is Fine

FOO          = Goodness
BAR          = Me
Oh[FOO][BAR] = Goodness Gracious Me!

# But This Kind Of Nesting Attempt Causes An Error

[FOO[BAR]] = Something Or Other
```

Introducing Lexical Namespaces

So far,the discussion of variables and references has conveniently ignored the presence of another related `tconfpy` feature, "Lexical Namespaces." Namespaces are a way to automatically group related variables together. Suppose you wanted to describe the options on your car in a configuration file. You might do this:

```
MyCar.Brand = Ferrari
MyCar.Model = 250 GTO
MyCar.Color = Red

# And so on ...
```

You'll notice that every variable start with the "thing" that each item has in common - they are features of `MyCar`. We can simplify this considerably by introducing a lexical namespace:

```
[MyCar]

Brand = Ferrari
Model = 250 GTO
Color = Red
```

The first statement looks like a variable reference, but it is not. **A string inside square brackets by itself on a line introduces a namespace.** The first statement in this example sets the namespace to `MyCar`. From that point forward until the namespace is changed again, every variable assignment **and** reference is "relative" to the namespace. What this really means is that `tconfpy` sticks the namspace plus a `.` in front of every variable assigned or referenced. It does this automatically and invisibly, so `Brand` is turned into `MyCar.Brand` and so on. You can actually check this by loading the example above into a test configuration file and running the `test-tc.py` program on it. You will see the "fully qualified" variable names that actually were loaded into the symbol table, each beginning with `MyCar.` and ending with the variable name you specified.

Realize that this is entirely a naming "trick". `tconfpy` has no clue what the namespace **means,** it just combines the current namespace with the variable name to create the actual variable name that will be returned in the symbol table.

You're likely scratching your head wondering why on earth this feature present in `tconfpy`. There are several good reasons for it:

- It reduces typing repetetive information throughout the configuration file. In turn, this reduces the likelyhood of a typographical or spelling error.

- It helps visibly organize the configuration file. A namespace makes it clear which variables are related to each other somehow. This is no big deal in small configurations, but `tconfpy` was written with the idea of supporting configuration files that might contain thousands or even tens of thousands of entries.

- It simplifies the application programmer's job. Say I want to write a program that extracts all the information about your car from the configuration file, but I don't know ahead of time how many things you will describe. All I really have to know is that you are using `MyCar` as the namespace for this information. My program can then just scan the symbol table after the configuration file has been parsed, looking for variables whose name begins with `MyCar.`. So if you want to add other details about your auto like, say, `Age`, `Price`, and so on, you can do so later **and the program does not have to be rewritten.**

- It helps enforce correct configuration files. By default, you can introduce new namespaces into the configuration file any time you like. However, as described in the previous section on the `tconfpy` API, the application programmer can limit you to a predefined set of legal namespaces (via the `LegalVals` attribute of the `NAMESPACE` variable descriptor). By doing this, the programmer is helping you avoid incorrect configuration file entries by limiting just which namespaces you can enter to reference or create variables.

Rules For Using Lexical Namespace

Creating and using lexical namespaces is fairly straightforward, but there are a few restrictions and rules:

- The default initial namespace is the empty string, "". In this one case, `tconfpy` does nothing to variables assigned or referenced. That's why our early examples in the previous section worked. When we assigned a value to a variable and then referenced that variable value, we did so while in the so-called "root" namespace, "". When the namespace is "", nothing is done to the variable names.

Bear in mind that the programmer can change this default namespace to something other than "" before the configuration file is ever processed. If they do this, they would be well advised to let their users know this fact.

- There two ways to change to a new namespace:

```
    [NewNameSpace]            # May optionally have a comment
```

OR

```
    NAMESPACE = NewNamespace  # May optionally have a comment
```

If, at any point, you want to return to the root namespace, you can use one of these two methods:

```
    []
```

OR

```
    NAMESPACE =
```

So, why are there two ways to do the same thing? The first way is the more common, and the more readable way to do it. It appears on a line by itself and makes it clear that the namespace is being changed. However, because variable references cannot be "nested", you can only use strings of text here.

Suppose you want to change the namespace in a way that depends on the value of another variable. For instance:

```
    LOCATION = Timbuktu
    NAMESPACE = [LOCATION]-East
```

In other words, the second form of a namespace change allows you to employ the `tconfpy` string substitution and variable referencing features. Bear in mind that `tconfpy` is case-sensitive so this will not work as you expect:

```
   Namespace = something
```

This just set the value of the variable `Namespace` to `something` and has nothing whatsoever to do with lexical namespaces.

- Whichever method you use to change it, **the new namespace must follow all the same rules used for naming variables.**

For example, both of the following will cause an error:

```
    [$FOO]
```

OR

```
    x = $FOO
    NAMESPACE = [x]
```

- By default, all variable assignments and references are **relative to the currently active namespace:**

```
[MyNameSpace]

foo = 123   # Creates a variable called MyNameSpace.foo
x   = [bar] # Means: MyNameSpace.x = [MyNameSpace.bar]
```

- If you want to set or reference a variable in a namespace different than the current namespace, you must use a so-called "absolute" variable name. You do this by "escaping" the variable name. To escape the name, begin it with a `.` and then use the **full name (including namespace)** of that variable. (This is called the "fully qualified variable name".) For example:

  ```
  [NS1]            # Switch to the namespace NS1
  foo = 14         # Creates NS1.foo

  [NS2]            # Switch to the NS2 namespace
  foo = [.NS1.foo] # Sets NS2.foo = 14
  ```

  There is another clever way to do this without using the escape character. `tconfpy` has no understanding whatsoever of what a lexical namespace actually is. It does nothing more than "glue" the current namespace to any variable names and references in your configuration file. Internally, all variables are named **relative to the root namespace.** This means that you can use the fully qualified variable name without any escape character any time you are in the root namespace:

  ```
  [NS1]            # Switch to the namespace NS1
  foo = 14         # Creates NS1.foo

  []               # Switch to the root namespace
  foo = [NS1.foo]  # Sets foo = 14 - no escape needed
  ```

- Lexical namspaces are implemented by having `NAMESPACE` just be nothing more than (yet) another variable in the symbol table. `tconfpy` just understands that variable to be special - it treats it as the repository of the current lexical namespace. This means you can use the value of NAMESPACE in your own string substitutions:

  ```
  MyVar = [NAMESPACE]-Isn't This Cool?
  ```

  You can even use the current value of NAMESPACE when setting a new namespace:

  ```
  NAMESPACE = [NAMESPACE]-New
  ```

  One final, but very important point is worth noting here. The `NAMESPACE` variable itself is always understood to be **relative to the root namespace.** No matter what the current namespace actually is, `[NAMESPACE]` or `NAMESPACE = ...` always set a variable by that name in the root namespace. Similarly, when we use a variable reference to get the current namespace value (as we did in the example above), `NAMESPACE` is understood to be relative to the root namespace. That's why things like this work:

  ```
  [MyNewSpace]
  x = 100                # MyNewSpace.x = 100
  y = [NAMESPACE]-1      # MyNewSpace.y = MyNewSpace-1

  NAMESPACE = NewSpace   # .NAMESPACE = NewSpace
  ```

Predefined Variables

`tconfpy` predefines a number of variables. The `NAMESPACE` variable we discussed in the previous section is one of them, but there are a number of others of which you should be aware. Note that all predefined variables **are relative to the root namespace.** Except for the `NAMESPACE` variable, they are all Read Only and cannot be modified in your configuration file.

The first group of predefined variables are called "System Variables". As the name implies, they provide information about the system on which you're running. These are primarily useful when doing conditional tests (described later in this document). For example, by doing conditional tests with System Variables you can have one configuration file that works on both Unix and Windows operating systems. The System Variables are:

```
 Variable Name                Contains
 -------------                --------

.MACHINENAME   -  The name of the computer on which you are running.
                  May also include full domain name, depending on system.

.OSDETAILS     -  Detailed information about the operating system in use.

.OSNAME        -  The name of the operating system in use.

.OSRELEASE     -  The version of the operating system in use.

.OSTYPE        -  Generic name of the operating system in use.

.PLATFORM      -  Generic type of the operating system in use.

.PYTHONVERSION -  The version of Python in use.
```

By combining these System Variables as well as the content of selected Environment Variables, you can create complex conditional configurations that "adapt" to the system on which a Python application is running. For example:

```
.if [.MACHINENAME] == foo.bar.com
     BKU = tar

.else

     BKU = [$BACKUPPROGRAM]

.endif
```

The other kind of predefined variables are called "Reserved Variables". `tconfpy` understands a number of symbols as part of its own language. For example, the string `#` tells `tconfpy` to begin a comment until end-of-line. There may be times, however, when **you** need these strings for your own use. In other words, you would like to use one of the strings which comprise the `tconfpy` language for your own purposes and have `tconfpy` ignore them. The Reserved Variables give you a way to do this. The Reserved Variables are:

```
 Variable Name                Contains
 -------------                --------
```

```
DELIML                        [
DELIMR                        ]
DOLLAR                        $
ELSE                          .else
ENDIF                         .endif
ENDLITERAL                    .endliteral
EQUAL                         =
EQUIV                         ==
HASH                          #
IF                            .if
IFALL                         .ifall
IFANY                         .ifall
IFNONE                        .ifnone
INCLUDE                       .include
LITERAL                       .literal
NOTEQUIV                      !=
PERIOD                        .
```

For instance, suppose you wanted to include the # symbol in the value of one of your variables. This will not work, because `tconfpy` interprets it as the beginning of a comment, which is not what you want:

```
MyJersey = Is #23
```

So, we use one of the Reserved Variables to get what we want:

```
MyJersey = Is [HASH]23
```

One word of warning, though. At the end of the day, you still have to create variable names or namespace names that are legal. You can't "sneak" illegal characters into these names using Reserved Variables:

```
foo = [DOLLAR]MyNewNamespace   # No problem
NAMESPACE = [foo]              # No way - namespace cannot start with $
```

Type And Value Enforcement

By default, any variable (or namespace) you create in a configuration file is understood to just hold a string of characters. There are no limits to what that string may contain, how long it is, and so on.

However, `tconfpy` gives the programmer considerable power to enforce variable types and values, if they so choose. (See the section above entitled, **PROGRAMMING USING THE** `tconfpy` **API** for the details.) The programmer can set all kinds of limitations about a variable's type, permissible values, and (in the case of strings) how long or short it may be. The programmer does this by defining these limitations for each variable of interest **prior to calling** `tconfpy` **to parse your configuration file.** In that case, when `tconfpy` actually processes the configuration file, it "enforces" these restrictions any time you attempt to change the value of one of these variables. If you try to assign a value that fails one of these "validation" tests, `tconfpy` will produce an error and leave the variable's value unchanged.

For instance, suppose the programmer has defined variable "Foo" to be a floating point number, and that it must have a value between -10.5 and 100.1. In that case:

```
Foo  = 6.023E23     # Error - Value is out of range
Foo  = MyGoodness   # Error - Value must be a FP number, not a string
```

```
Foo  = -2.387        # Good  - Value is both FP an in range
```

What Specific Validations Are Available?

The programmer has several different restrictions they can place on a variable's value. You do not need to understand how they work, merely what they are so that any error messages you see will make sense.

- The programmer may declare any variable to be **Read Only.** This means you can still use references to that variable to extract its value, but any attempt to change it value within the configuration file will fail and produce an error.

- The programmer may specify the variable's **type** as string (the default), integer, floating point, complex, or boolean.

- The programmer may specify **the set of all legal values** that can be assigned to a variable. For instance, the programmer might specify that the floating point variable `Transcend` can only be set to either 3.14 or 2.73. Similarly, the programmer might specify that the string variable `COLOR` can only ever be set to `Red`, `Green`, or `Blue`. In fact, in the case of string variables, the programmer can actually specify a set of patterns (regular expressions) the value has to match. For instance, they can demand that you only set a particular string variable to strings that begin with `a` and end with `bob`.

- For integer and floating point variables, the programmer can specify a legal **value range** for the variable. If you change the value of such a variable, that value must be within the defined range or you'll get an error.

- For string variables, the programmer can specify a minimum and maxium **length** for the strings you assign to the variable in question.

- The programmer can limit you to **only being able to use existing variables.** (.i.e. The Predefined variables and any variables the programmer has defined ahead of time.) In that case, any attempt to create a new variable in the configuration file will fail and produce an error.

- The programmer can limit you to **only being able to use namespaces they have defined ahead of time.** In that case, if you attempt to enter a namespace not on the list the programmer created ahead of time will fail and produce an error.

- The programmer can enable or prevent **the substitution of variable references in literal blocks** (see below). If they disable this option, something like `[Foo]` is left unchanged within the literal block. i.e., It too, is treated "literally".

Notes On Variable Type/Value Enforcement

There are a few other things you should know about how `tconfpy` enforces restrictions on variables:

- For purposes of processing the configuration file, **variable references are always converted to strings** regardless of the actual type of the variable in question. (Variables are stored in the symbol table in their actual type.)

  For instance, suppose the programmer defines variable `Foo` to be floating point. Then:

```
Foo = 1.23
Bar = Value is [Foo] # Creates a new *string* variable with the
                     # value: "Value is 1.23"
```

In other words, variable values are "coerced" into strings for the purposes of substitution and conditional testing within a configuration file. This is primarily an issue with the conditional comparisons below. For example, the following conditional is `False` because the string representations of the two numbers are different. Assume `f1` and `f2` have been defined as floating point variables by the calling program:

```
f1 = 1.0
f2 = 1.00

.if [f1] == [f2]   # False because "1.0" is not the same string as "1.00"
  ...
```

- You cannot create anything but a string variable within a configuration file. This variable will have no restrictions placed on its values. All validation features **require** the limitations to be specified by the calling program ahead of time.

- Similarly, you cannot change any of the enforcement options from within a configuration file. These features are only available under program control, presumably by the application program that is calling `tconfpy`.

- There is no way to know what the limitations are on a particular variable from within the configuration file. Programmers who use these features should document the variable restrictions they've employed as a part of the documentation for the application in question.

Some Further Notes On Boolean Variables

One last note here concerns Boolean variables. Booleans are actually stored in the symbol table as the Python Boolean values, `True` or `False`. However, `tconfpy` accepts user statements that set the value of the Boolean in a number of formats:

```
Boolean True              Boolean False
------------              -------------

foo = 1                   foo = 0
foo = True                foo = False
foo = Yes                 foo = No
foo = On                  foo = Off
```

This is the one case where `tconfpy` is insensitive to case - `tRUE`, `TRUE`, and `true` are all accepted, for example.

NOTE HOWEVER: If the user wants to do a conditional test on the value of a Boolean they **must** observe case and test for either `True` or `False`:

```
boolvar = No
```

```
.if [boolvar] == False    # This works fine

.if [boolvar] == FALSE    # This does not work - Case is not being observed

.if [boolvar] == Off      # Neither does this - Only True/False can be tested
```

As a practical matter, unless you actually need to do comparisons involving `"True"` and `"False"` strings, it is best to use the Existential Conditionals to test the **state** of a boolean variable. See the section entitled, **Existential Conditionals And Booleans** below, for the details.

The `.include` Directive

At any point in a configuration file, you can "include" another configuration file like this:

```
.include  filename
```

In fact, you can use all the variable substitution and string concatenation features we've already discussed to do this symbolically:

```
Base = MyConfig
Ver  = 1.01

.include [Base]-[Ver].cfg
```

The whitespace after the `.include` directive is mandatory to separate it from the file name. You can have as many `.include` statements in your configuration file as you wish, and they may appear anywhere. The only restriction is that they must appear on a line by themselves (with an optional comment).

Why bother? There are several reasons:

- This makes it easy to break up large, complex configurations into simpler (smaller) pieces. This generally makes things easier to maintain.
- This makes is easy to "factor" common configuration information into separate files which can then be used by different programs as needed.
- The most common use for `.include` is to load a "standard" configuration for your program. Recall that the last assignment of a variable's value "wins". Suppose you want all the standard settings for a program, but you just want to change one or two options. Instead of requiring each user to have the whole set of standard settings in their own configuration file, the system administrator can make them available as a common configuration. You then `.include` that file and override any options you like:

  ```
  # Get the standard options
  .include /usr/local/etc/MyAppStandardConfig.cfg

  # Override the ones you like
  ScreenColor = Blue
  Currency    = Euros
  ```

  This makes maintenance of complex configuration files much simpler. There is only one master copy of the configuration that needs to be edited when system-wide changes are required.

One last thing needs to be noted here. `tconfpy` does not permit so-called "circular" or "recursive" inclusions. If file `a .includes` file `b` and file `b .includes` file `a`, you will have an infinite loop of inclusion, which, uh ..., is a Bad Thing. So, the parser checks each time you attempt to open a new configuration file to see if it's already been processed. If it has, an error is produced, and the `.include` line that would have caused a circular reference is ignored. Thereafter, the program will continue to process the remainder of the configuration as usual.

Conditional Directives

One of the most powerful features of `tconfpy` is its "conditional processing" capabilities. The general idea is to test some condition and **include or exclude configuration information based on the outcome of the test.**

What's the point? You can build large, complex configurations that test things like environment variables, one of the Predefined Variables, or even a variable you've set previously in the configuration file. In other words, resulting configuration is then produced in a way that is appropriate for that particular system, on that particular day, for that particular user, ...

By using conditional directives, you can create a single configuration file that works for every user regardless of operating system, location, and so on.

There are two kinds of conditional directives. "Existential Conditionals" test to see if a configuration or environment variable **exists.** Existential Conditionals pay no attention to the **value** of the variables in question, merely whether or not those variables have been defined.

"Comparison Conditionals" actually **compare** two strings. Typically, one or more variable references appear in the compared strings. In this case, the **value of the variable** is important.

The general structure of any conditional looks like this:

```
ConditionalDirective  Argument(s)

    This is included if the conditional was True

.else    # Optional

    This is included if the conditional was False

.endif   # Required
```

Except for the whitespace after the conditional directive itself, whitespace is not significant. You may indent as you wish.

Conditionals may also be "nested". You can have a conditional within another conditional or `.else` block:

```
ConditionalDirective Argument(s)
      stuff

      ConditionalDirective Argument(s)
          more stuff
      .endif

     interesting stuff
.else
```

```
          yet more stuff

          ConditionalDirective Argument(s)
              other stuff
          .endif

          ending stuff
     .endif
```

There are no explicit limits to how deeply you can nest a configuration. However, you must have an `.endif` that terminates each conditional test. Bear in mind that `tconfpy` pays no attention to your indentation. It associates an `.endif` **with the last conditional it encountered.** That's why it's a really good idea to use some consistent indentation style so **you** can understand the logical structure of the conditions. It's also a good idea to put comments throughout such conditional blocks so it's clear what is going on.

There are a few general rules to keep in mind when dealing with conditionals:

- There must be whitespace between the conditional directive and its arguments (which may- or may not have whitespace in them).

- As with any other kind of `tconfpy` statement, you may place comments anywhere at the end of a conditional directive line or within the conditional blocks.

- Each conditional directive must have a corresponding `.endif`. If you have more conditionals than `.endifs` or vice-versa, `tconfpy` will produce an error message to that effect. It can get complicated to keep track of this, especially with deeply nested conditionals. It is therefore recommended that you always begin and end conditional blocks within the same file. i.e., Don't start a conditional in one file and then `.include` another file that has the terminating `.endif` in it.

- The `.else` clause is optional. However, it can only appear after some preceding conditional directive.

- As in other parts of the `tconfpy` language, variable names and references in conditional directives are always relative to the currently active namespace unless they are escaped with a leading period. Similarly, in this context, Environment Variables, Predefined Variables, and the NAMESPACE Variable are always relative to the root namespace, no matter what namespace is currently active.

Existential Conditional Directives

There are three Existential Conditionals: `.ifall`, `.ifany`, and `.ifnone`. Each has the same syntax:

```
 ExistentialDirective varname ...
    included if test was True

.else # optional
    included if test was False

 .fi
```

In other words, existential conditionals require one or more **variable names.** In each case, the actual content of that variable is ignored. The test merely checks to see if a variable by that name **exists.** Nothing else may appear on an existential conditional line, except, perhaps, a comment.

The three forms of existential conditional tests implement three different kinds of logic:

```
 .ifall  var1 var2 ...

  This is a logical "AND" operation.  ALL of the variables, var1, var2 ...
  must exist for this test to be True.

 .ifany var1 var2 ...

  This is a logical "OR" operation.  It is True of  ANY of the variables,
  var1, var2 ... exist.

 .ifnone  var1 var2 ...

  This is a logical "NOR" operation.  It is True only if NONE of the variables,
  var1, var2 ... exist.
```

Here is an example:

```
FOO = 1
BAR = 2
  z = 0

.ifall FOO BAR
    x = 1
.endif

.ifany FOO foo fOo
    y = 2
.endif

 .ifnone BAR bar Bar SOmething
    z=3
 .endif
```

When `tconfpy` finishes processing this, x=1, y=2, and z=0.

You can also use references to environment variables in an existential conditional test:

```
.ifany $MYPROGOPTIONS
   options = [$MYPROGOPTIONS]

.else
   options = -b20 -c23 -z -r

.endif
```

Finally, you can use variable references here to get the name of a variable to test by "indirection" (as we saw in the previous section on accessing/setting variables indirectly). This should be used sparingly since it can be kind of obscure to understand, but it is possible to do this:

```
foo = MyVarName

.ifany [FOO]
    ...
.endif
```

This will test to see if either the variable `MyVarName` exists.

You can also do indirection through an environment variable. Use this construct with restraint - it can introduce serious obscurity into your configuration file. Still, it has a place. Say the `TERM` environment variable is set to `vt100`:

```
.ifany [$TERM]
    ...
.endif
```

This will test to see if a variable called `vt100` exists in the symbol table. This is a handy way to see if you have a local variable defined appropriate for the currently defined terminal, for instance.

Existential Conditionals And Booleans

As we've just seen, `.ifall/any/none` check to see if the variables named on the Right Hand Side "exist". This is true for all kinds of variables, regardless of their type. For instance:

```
.ifall Boolean1 Boolean2
   ...
.endif

 Foo = Boolean3

.ifall [Foo]
    ...
.endif
```

The first conditional will be `True` only if both variables, `Boolean1` and `Boolean2` exist. Similarly, the second conditional will be `True` only if `Boolean3` exists.

However, when using indirection on a boolean variable (i.e. A variable that has been pre-defined to be a boolean type by the calling program), **the state of the boolean is returned.** For example:

```
.ifall [Boolean1] [Boolean2]
   ...
.endif
```

In this case, the `[Boolean..]` indirection is understood to mean, "Return the logical state of the boolean variable in question". This allows the existential conditionals to act like **logical** operators when their targets are boolean. In the example above, the test will only be True if both booleans are logically True.

You can even mix and match:

```
.ifall Boolean1 [Boolean2]
   ...
.endif
```

This conditional will be `True` only if `Boolean1` **exists** and `Boolean2` is `True`.

It's worth mentioning just **why** the semantics of indirect boolean references are different than for other variable types. If booleans were treated the same as other variables, then `[Boolean]` would return a string representation of the boolean variable's state - i.e., It would return `"True"` or `"False"`. In effect, we would be doing this:

```
.ifall/any/none True False True True False ...
```

This more-or-less makes no sense, since we're checking to see if variables named "True" and "False" exist. What **does** make a lot of sense is to use the state of a boolean variable to drive the conditional logic.

There is one other reason this feature is provided. Earlier versions of `tconfpy` did not have this feature - booleans were treated like any other variable. This meant that doing logical tests on the state of the boolean required a kind of tortuous construct using the Comparsion Conditionals (described in the next section):

```
# Example of AND logic using Comparison Conditional

.if [Bool1][Bool2][Bool3] == TrueTrueTrue
    ...
.endif
```

This is ugly, hard to read, and hard to maintain. By contrast, the method just described allows booleans to be used in their intended manner - to make logical choices. `.ifall` becomes a logical "AND" function. `.ifany` becomes a logical "OR" function, and `.ifnone` becomes a logical "NOR" function when these tests are applied to indirect boolean references.

Both methods of testing boolean state remain supported, so you can use the style most appropriate for your application.

Comparison Conditional Directives

There are two Comparison Conditionals:

```
.if string1 == string2   # True if string1 and string2 are identical
.if string1 != string2   # True if string1 and string2 are different
```

As a general matter, you can put literal strings on both sides of such a test, but the real value of these tests comes when you use variable references within the tested strings. In this case, the value of the variable **does matter.** It is the variable's value that is replaced in the string to test for equality or inequality:

```
MyName = Tconfpy

.if [MyName] == Tconfpy
     MyAge = 100.1

.else
     MyAge = Unknown

.fi
```

These are particularly useful when used in combination with the `tconfpy` Predefinded Variable or environment variables. You can build configurations that "sense" what system is currently running and "adapt" accordingly:

```
AppFiles = MyAppFiles

.if [.OSNAME] == FreeBSD
    files = [$HOME]/[AppFiles]
.endif

.if  [.OSNAME] == Windows
    files = [$USERPROFILE]\[AppFiles]
.endif

.ifnone [files]
    ErrorMessage = I don't know what kind of system I am running!
.endif
```

The `.literal` Directive

By default, `tconfpy` only permits statements it "recognizes" in the configuration file. Anything else is flagged as an unrecognized statement or "syntax error". However, it is possible to "embed" arbitrary text in a configuration file and have `tconfpy` pass it back to the calling program without comment by using the `.literal` directive. It works like this:

```
.literal
     This is literal text that will be passed back.
.endliteral
```

This tells `tconfpy` to ignore everything between `.literal` and `.endliteral` and just pass it back to the calling program (in `retval.Literals` - see previous section on the `tconfpy` API). Literal text is returned in the order it is found in the configuration file.

What good is this? It is a nifty way to embed plain text or even programs written in other languages within a configuration file and pass them back to the calling program. This is especially handy when used in combination with `tconfpy` conditional features:

```
.if [.PLATFORM] == posix
    .literal
       We're Running On A Unix-Like System
    .endliteral

.else
    .literal
       We're Not Running On A Unix-Like System
    .endliteral

.endif
```

In other words, we can use `tconfpy` as a "preprocessor" for other text or computer languages. Obviously, the program has to be written to understand whatever is returned as literal text.

By default, `tconfpy` leaves text within the literal block completely untouched. It simply returns it as it finds it in the literal block. However, the programmer can invoke `tconfpy` with an option (`LiteralVars=True`) that allows **variable substitution within literal blocks.** This allows you to combine the results of your configuration into the literal text that is returned to the calling program. Here is how it works:

```
.ifall $USER
     Greeting = Hello [$USER]. Welcome to [.MACHINENAME]!
.else
     Greeting = Hello Random User. Welcome To Random Machine!
.endif


# Now embed the greeting in a C program
.literal

   #include <stdio.h>

   main()
        {
         printf("[Greeting]");
        }
.endliteral
```

If the calling program sets `LiteralVars=True`, the literal block will return a C program that prints the greeting defined at the top of this example. If they use the default `LiteralVars=False`, the C program would print `[Greeting]`.

In other words, it is possible to have your literal blocks make reference to other configuration variables (and Predefined or Environment Variables). This makes it convenient to combine both configuration information for the program, **and** other, arbitrary textual information that the program may need, all in a single configuration file.

Notice too that the `#` character can be freely included within a literal block. You don't have to use a Reserved Variable reference like `[HASH]` here because **everything** (including whitespace) inside a literal block is left untouched.

If you fail to provide a terminating `.endliteral`, the program will treat everthing as literal until it reaches the end of the configuration file. This will generate an appropriate warning, but will work as you might expect. Everything from the `.literal` directive forward will be treated literally. As a matter of good style, you should always insert an explicit `.endliteral`, even if it is at the end of file.

Placing an `.endliteral` in the configuration file without a preceding `.literal` will also generate a warning message, and the statement will be ignored.

GOTCHAS

`tconfpy` is a "little language". It is purpose-built to do one and only one thing well: process configuration options. Even so, it is complex enough that there are a few things that can "bite" you when writing these configuration files:

- Probably the most common problem is attempting to do this:

```
foo = bar

.if foo == bar
   ...
.endif
```

But this will not work. `tconfpy` is very strict about requiring you to explicitly distinguish between **variable names** and **variable references.**

The example above checks to see if the string `foo` equals the string `bar` - which, of course, it never does.

What you probably want is to compare the value of variable `foo` with some string:

```
foo = bar

.if [foo] == bar
   ...
.endif
```

Now you're comparing the **value** of the variable `foo` with the string `bar`.

This was done for a very good reason. Because you have to explicitly note whether you want the name or value of a variable (instead of having `tconfpy` infer it from context), you can mix both literal text and variable values on either side of a comparison or assignment:

```
foo = bar

foo[foo]foo = bar       # Means: foobarfoo = bar

.if foo[foo] == foobar  # Means: .if foobar == foobar
```

- Namespaces are a handy way to keep configuration options organized, especially in large or complex configurations. However, you need to keep track of the current namespace when doing things:

```
foo = bar
 ....

[NS-NEW]

.if [foo] == something  # Checks value of NS-NEW.foo - will cause error
                        # since no such variable exists
```

- Remember that "last assignment wins" when setting variable values:

```
myvar = 100

... a long configuration file

myvar = 200
```

At the end of all this, `myvar` will be set to 200. This can be especially annoying if you `.include` a configuration file after you've set a value and the included file resets it. As a matter of style, it's best to do all the `.includes` at the top of the master configuration file so you won't get bitten by this one.

- Remember that case matters. `Foo`, `foo`, and `foO` are all different variable names.

- Remember that all variable references are **string replacements** no matter what the type of the variable actually is. `tconfpy` type and value enforcement is used to return the proper value and type to the calling program. But within the actual processing of a configuration file, variable references (i.e., the values of variables) are always treated as **strings.**

- It is possible to load your own, user-defined, type in the variable descriptor object when you predefine a symbol table to be passed to the parser. The problem is that this is more-or-less useless. The parser attempts to coerce data assignments in the configuration into the specified type. But, using only the assignment statements available in this language, you cannot define values in a meaningful way for user-defined types. So, assignment of user-defined variable types will always fail with a type error. Again, `tconfpy` is designed as a small configuration processing language, not as a general purpose programming language. In short, user-defined types are not supported in the variable descriptor processing and will always cause a type error to occur.

ADVANCED TOPICS FOR PROGRAMMERS

Here are some ideas on how you might combine `tconfpy` features to enhance your own applications.

Guaranteeing A Correct Base Configuration

While it is always nice to give users lots of "knobs" to turn, the problem is that the more options you give them, the more they can misconfigure a program. This is especially a problem when you are doing technical support. You'd really like to get them to a "standard" configuration and then work from there to help solve their problem. If your write your program with this in mind, `tconfpy` gives you several ways to easily do this:

- Provide a "standard" system-, or even, enterprise-wide configuration file for your application. This file presumably has all the program options set to "sane" values. All the user has to do is create a configuration file with one line in it:

  ```
  .include /wherever/the/standard/config/file/is
  ```

- Predefine every option variable the program will support. Populate the initial symbol table passed to `ParseConfig()` with these definitions. By properly setting the `Type`, `LegalVals`, and `Min/Max` for each of these variables ahead of time, you can prevent the user from ever entering option values that make no sense or are dangerous.

- Make sure ever program option has a reasonable `Default` value in its variable descriptor. Recall that this attribute is provided for the programmer's convenience. (When a variable descriptor is first instantiated, it defaults to a string type and sets the default attribute to an empty string. However, you can change both type and default value under program control.) If you predefine a variable in the initial symbol table passed to the parser, `tconfpy` will leave this attribute alone. However, variables that are created for the first time in the configuration file will have this attribute set to the first value assigned to the variable. Now provide a "reset" feature in your application. All it has to do is scan through the symbol table and set each option to its default value.

Enforcing Mandatory Configurations

The `tconfpy` type and value validation features give you a handy way to enforce what the legal values for a particular option may be. However, you may want to go further than this. For instance, you may only want to give certain classes of users the ability to change certain options. This is easily done. First, predefine all the options of interest in the symbol table prior to calling the `tconfpy` parser. Next, have your program decide which options the current user is permitted to change. Finally, mark all the options they may not change as "Read Only", by setting the "Writeable" attribute for those options to `False`. Now call the parser.

This general approach allows you to write programs that support a wide range of options which are enabled/disabled on a per-user, per-machine, per-domain, per-ip, per-company... basis.

Iterative Parsing

There may be situations where one "pass" through a configuration file may not be enough. For example, your program may need to read an initial configuration to decide how to further process the remainder of a configuration file. Although it sounds complicated, it is actually pretty easy to do. The idea is to have the program set some variable that selects which part of the configuration file to process, and then call the parser. When the parser returns the symbol table, the program examines the results, makes whatever adjustments to the symbol table it needs to, and passes it back to the parser for another "go". You can keep doing this as often as needed. For instance:

```
# Program calls the parser with PASS set to 1

.if [PASS] == 1
   # Do 1st Pass Stuff
.endif



# Program examines the results of the first pass, does
# what is has to, and sets PASS to 2

.if [PASS] == 2
   # Do 2nd Pass Stuff
.endif

# And so on
```

In fact, you can even make this iterative parsing "goal driven". The program can keep calling the parser, modifing the results, and calling the parser again until some "goal" is met. The goal could be that a particular variable gets defined (like `CONFIGDONE`). The goal might be that a variable is set to a particular value (like, `SYSTEMS=3`).

It might even be tempting to keep parsing iteratively until `tconfpy` no longer returns any errors. This is not recommended, though. A well-formed configuration file should have no errors on any pass. Iterating until `tconfpy` no longer detects errors makes it hard to debug complex configuration files. It is tough to distinguish actual configuration errors from errors would be resolved in a future parsing pass.

INSTALLATION

There are three ways to install `tconfpy` depending on your preferences and type of system. In each of these installation methods you must be logged in with root authority on Unix-like systems or as the Administrator on Win32 systems.

Preparation - Getting And Extracting The Package

For the first two installation methods, you must first download the latest release from:

```
http://www.tundraware.com/Software/tconfpy/
```

Then unpack the contents by issuing the following command:

```
tar -xzvf py-tconfpy-X.XXX.tar.gz   (where X.XXX is the version number)
```

Win32 users who do not have tar installed on their system can find a Windows version of the program at:

```
http://unxutils.sourceforge.net/
```

Install Method #1 - All Systems (Semi-Automated)

Enter the directory created in the unpacking step above. Then issue the following command:

```
python setup.py install
```

This will install the `tconfpy` module and compile it.

You will manually have to copy the 'test-tc.py' program to a directory somewhere in your executable path. Similarly, copy the documentation files to locations appropriate for your system.

Install Method #2 - All Systems (Manual)

Enter the directory created in the unpacking step above. Then, manually copy the tconfpy.py file to a directory somewhere in your PYTHONPATH. The recommended location for Unix-like systems is:

```
.../pythonX.Y/site-packages
```

For Win32 systems, the recommended location is:

```
...\PythonX.Y\lib\site-packages
```

Where X.Y is the Python release number.

You can precompile the `tconfpy` module by starting Python interactively and then issuing the command:

```
import tconfpy
```

Manually copy the 'test-tc.py' program to a directory somewhere in your executable path. Copy the documentation files to locations appropriate for your system.

Install Method #3 - FreeBSD Only (Fully-Automated)

Make sure you are logged in as root, then:

```
cd /usr/ports/devel/py-tconfpy
make install
```

This is a fully-automated install that puts both code and documentation where it belongs. After this command has completed you'll find the license agreement and all the documentation (in the various formats) in:

```
/usr/local/share/doc/py-tconfpy
```

The 'man' pages will have been properly installed so either of these commands will work:

```
man tconfpy
man test-tc
```

Bundling `tconfpy` With Your Own Programs

If you write a program that depends on `tconfpy` you'll need to ensure that the end-users have it installed on their systems. There are two ways to do this:

- Tell them to download and install the package as described above. This is not recommended since you cannot rely on the technical ability of end users to do this correctly.

- Just include 'tconfpy.py' in your program distribution directory. This ensures that the module is available to your program regardless of what the end-user system has installed.

THE `tconfpy` MAILING LIST

TundraWare Inc. maintains a mailing list to help you with your `tconfpy` questions and bug reports. To join the list, send email to **majordomo@tundraware.com** with a single line of text in the body (not the Subject line) of the message:

```
subscribe tconfpy-users your-email-address-goes-here
```

You will be notified when your subscription has been approved. You will also receive detailed information about how to use the list, access archives of previous messages, unsubscribe, and so on.

OTHER

`tconfpy` requires Python 2.3 or later.

BUGS AND MISFEATURES

None known as of this release.

COPYRIGHT AND LICENSING

AUTHOR

```
Tim Daneliuk
tconfpy@tundraware.com
```

DOCUMENT REVISION INFORMATION

$Id: tconfpy.3,v 1.159 2005/01/20 09:32:57 tundra Exp $