Using Python With Synopsys Tools

Version W-2024.09, September 2024

Feedback

Contents

About This Manual

This manual describes how to use the open-source scripting language Python, which has been integrated into Synopsys tools. This manual provides an overview of Python, describes its relationship with Synopsys command shells, and explains how to create scripts and procedures.

The audience for the guide are designers who are experienced with using the Design Compiler^{TM}, Fusion Compiler^{TM}, and IC Compiler II^{TM} tools and who have a basic understanding of programming concepts such as data types, control flow, procedures, and scripting.

This preface includes the following sections:

- New in This Release
- Related Products, Publications, and Trademarks
- Conventions
- Customer Support

New in This Release

Information about new features, enhancements, and changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the Fusion Compiler Release Notes on the SolvNetPlus site.

Related Products, Publications, and Trademarks

For additional information about the Synopsys tool, see the documentation on the Synopsys SolvNetPlus support site at the following address:

https://solvnetplus.synopsys.com

You might also want to see the documentation for the following related Synopsys products:

- Design Compiler
- Fusion Compiler
- IC Compiler II

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
`Purple`	• Within an example, indicates information of special interest. • Within a command-syntax section, indicates a default, such as `include_enclosing =` **`true`** `| false`
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*.
|	Indicates a choice among alternatives, such as `low | medium | high`
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Bold	Indicates a graphical user interface (GUI) element that has an action associated with it.
Edit > Copy	Indicates a path to a menu command, such as opening the **Edit** menu and choosing **Copy**.
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.

Customer Support

Customer support is available through SolvNetPlus.

Accessing SolvNetPlus

The SolvNetPlus site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNetPlus site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNetPlus site, go to the following address:

https://solvnetplus.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

If you need help using the SolvNetPlus site, click REGISTRATION HELP in the top-right menu bar.

Contacting Customer Support

To contact Customer Support, go to https://solvnetplus.synopsys.com.

Statement on Inclusivity and Diversity

Synopsys is committed to creating an inclusive environment where every employee, customer, and partner feels welcomed. We are reviewing and removing exclusionary language from our products and supporting customer-facing collateral. Our effort also includes internal initiatives to remove biased language from our engineering and working environment, including terms that are embedded in our software and IPs. At the same time, we are working to ensure that our web content and software applications are usable to people of varying abilities. You may still find examples of non-inclusive language in our software or documentation as our IPs implement industry-standard specifications that are currently under review to remove exclusionary language.

Feedback

1

Introduction to Python in Synopsys Tools

Synopsys supports Tcl as an extension language. This document describes support for Python as an extension language, which is supported by some products in addition to Tcl. This chapter provides the information you need to run a Synopsys Python tool.

This chapter consists of the following sections:

Introduction

Python is a popular programming language. It was created by Guido van Rossum and released in 1991. Python is an easy-to-learn, powerful, has efficient high-level data structures, and a simple but effective approach to object-oriented programming. Python's elegant syntax and dynamic typing, together with its interpreted nature, make it an ideal language for scripting and rapid application development in many areas on most platforms.

Python is used for:

- Web development (server-side)
- Software development
- Complex mathematical calculations
- System scripting

Why Python?

Python is a well-known and popular language. There is an extensive standard library and there are many open-source packages that can be leveraged. Python is also suitable as an extension language for customizable applications.

Using Python With Synopsys Tools

Python provides the necessary programming constructs—variables, collections, return values, loops, procedures, and so on—for creating scripts with Synopsys commands. This aspect of Python encompasses how variables, expressions, scripts, control flow, and procedures work, as well as the syntax of commands, including Synopsys commands.

The examples in this guide use a mixture of Python and Synopsys commands, so when necessary for clarity, a distinction is made between the Python and Synopsys commands. You can refer to the Synopsys man pages for a description of how these commands are implemented.

If you try to execute the examples in this book, you must do so within a Synopsys command shell because the Python shell does not support Synopsys commands.

2

Python Interpreter and Its Environment

The Python interpreter is linked to the application, similar to how the applications link to Tcl.

Synopsys applications add a module called `snps` into the embedded Python interpreter. The Python language uses the `snps` module to control the tool behavior. That module is imported into the Python global name space. The `snps` module defines the classes and objects that are used to interact with the application built in commands.

Evaluating Python Code From Tcl

The `py_eval` Tcl command is used to evaluate Python code. Run the command from the Tcl command prompt or a Tcl script.

Table 1 Tcl Scripts for Evaluation

Tcl Script	Description
`py_eval (code)`	Evaluates Python code
`-file <path>`	Evaluates Python code in file
`[-verbose]`	Echos commands and displays results during evaluation
`code`	Evaluates Python statements

Changing the Language

Calling Tcl From Python

All application commands, Tcl built-in commands, and user Tcl procedures are accessed by calling methods on the `snps.cmd` object. The method name is the same as the name of the Tcl command or procedure. The `snps.cmd` object is described in more detail below. If you want to evaluate Tcl scripts or files, you use the `cmd.eval` or `cmd.source` method. For example:

```
cmd.eval (script)
```

```
cmd.source (file)
```

Here, `script` specifies the script that you want to evaluate, and `file` specifies the source of the file.

Changing the Language to Python

Applications that support Python include both a Tcl and a Python console. The console language is selected by setting the `sh_language` application variable. When the console language is Python, this is indicated by `-py` at the end of the prompt string.

For example:

```
dcnxt_shell>fc_shell>icc2_shell> set_app_var sh_language python
```

```
dcnxt_shell-py>fc_shell-py>icc2_shell-py> cmd.set_app_var('sh_language',
'tcl')
```

```
dcnxt_shell>fc_shell>icc2_shell>
```

The prompt changes to `dcnxt_shell-py>fc_shell-py>icc2_shell-py>`.

Now, you can use Python syntax (statements and expressions) to control the tool.

Generic Commands

Use `cmd.help ()` for interactive help or to check the help of any specific command. For generic python command examples, see Python Help. Here are a few examples of generic commands:

- `history()` – Shows the history of commands entered interactively.
    - `!<N>` – Reruns the command N from history.
    - `^X^Y` – Reruns the previous command but replaces X with Y.
- `source(fileName, [verbose=True])` – Sources a Python script. This is available in batch too, and is an alias for the `py_eval` command.
- `help(cmd.cmdName) or cmd.cmdName` – Displays Python style help for the specified `cmdName`.

Python Environment

The application startup files continue to be Tcl-based. However, the normal startup scripts can be used to evaluate Python code at tool startup. To import modules that are not shipped with the application, modify the `sys.path` property as you would in any Python program.

3 Running Python Commands

All features of this implementation are available through the Python `snps` module. Any Python code examples here assume that the `from snps import *` was used to import the Python implementation.

All application commands, Tcl built-in commands, and user Tcl procedures are accessed through methods on the `snps.cmd` object. All commands accept zero or more arguments.

Synopsys specific application commands accept the following types of arguments:

- Positional: It must be specified to the command in a specific order
- Dash: It starts with a dash (“-”) and might occur in any order

In Python, positional arguments remain positional, and dash arguments are mapped to a keyword argument.

Python requires that positional arguments must occur before any keyword arguments. The Python versions of the command use the same keyword name as the Tcl command. In Tcl, some dash arguments are Boolean, and they do not allow a value to be specified. Python requires specifying a value when the keyword syntax is used. To set a Boolean keyword argument, use either `1` or `True`.

For example, to find all the soft macros hierarchically in the design matching the pattern `*mem*`, use the following Python code:

```
cells = cmd.get_cells('*mem*', hierarchical=True, filter='is_soft_macro')
```

Some commands use dash argument names that are Python-reserved words. To address this, you can use leading underscores for keyword arguments. For example: `cmd.list_attributes(_class='cell', application=True)cmd.get_defined_attributes(_class='cell').`

Some application commands use repeated dash arguments to specify the information in the command. For example, many timing commands support multiple-through arguments to specify the sequence of pins that a path must follow. Python does not allow repeated keyword arguments. Attempting to do so generates a syntax error. To address this, the implementation leverages the `**kwargs` expansion syntax. An argument name can be specified with a syntax, as shown in the following example:

```
**cmd.arg(argName, argValue [,argName2, argValu2]…)
```

For example, to call the `get_timing_paths` command with multiple through points, use the following command.

```
cmd.get_timing_paths(through=B, **cmd.arg('through', C))
cmd.get_timing_paths(**cmd.arg('through',B, 'through',C))
```

As shown earlier, regular keyword arguments can be freely interspersed with the expansion syntax.

All commands return a value. The type of value can be one of the following:

- `int` or `float`: Python built-in numeric types
- `snps.collection`: An ordered sequence of database objects
- `snps.value`: All other return values

4 Return Values

The `snps.collection` type is an ordered sequence of one or more database objects.

The collections support the Python Iterator Protocol. The elements of a collection can be used in the same way as other Python containers. For example:

```
for c in cmd.get_cells(): cmd.some_command(c)
```

Collections support random index operations, through the Python slice syntax (see Table 2). To know about more possibilities, see Python documentation.

Table 2 Indexing Examples

Syntax	**Description**
`cells[0]`	# Get the first item
`cells[-1]`	# Get the last item
`cells[0:5]`	# Get the first 5 items
`cells[-5:]`	# Get the last 5 items
`cells[::2]`	# Get all items with an even index

The tool builds an internal database of objects and attributes available on those objects. Some examples of the class of database objects are designs, libraries, ports, cells, nets, pins, clocks, and so on. Most commands operate on these objects.

- **Accessing Elements**: Collections represent an ordered sequence of database objects. Collection values support the Python sequence interface. The contents of a collection may be accessed like the standard Python list type. For example (x is a collection value):

`x[0]`	Retrieve the first item. Returns a single item collection.
`x[-1]`	Retrieve the last item. All slice-based indexing is supported.
`len(x)]`	Return the number of objects in the collection

`for y in x:`	Loop over the items. In the loop body, `y` is a single item collection.

- **Building Collections**: Collections can be created by combining one or more collections. Both operator and function-based methods are supported. For example:

`x += y # x.append(y)`	Add the items in the collection `y` into the collection referenced by `x`.
`x + y # x.add(y)`	Return a new collection by combining the objects in `x` and `y`.
`x - y # x.remove(y)`	Return a new collection containing the objects in `x` that are not also in `y`. (`x -= y` is also supported).
`x ^ y # x.remove(y, intersect=True)`	Return a new collection that contains the objects that exists in both `x` and `y`. (`x ^= y` is also supported).

- **Sorting and Filtering**: Collections can be sorted and filtered. For example:

`x.remove_duplicates()`	Returns a new collection with duplicate objects removed.
`x.sort(criteria)`	Returns a new collection sorted by the sort criteria. This is a shorter form for the `cmd.sort_collection(x, criteria)` command.
`x.filter(expr)`	Returns a new collection by evaluating the filter expression. This is a shorter form for the `cmd.filter_collection(x, criteria)` command.

- **Attribute Values**: Collections also provide methods to query and set attribute values on objects. Usually, these are used with single item collections (see, Attribute Iterators below for optimized access to attributes on larger collections):

`x.set(attr, value)`	Calls `cmd.set_attribute(x, attr, value)`
`x.get(attr)`	Calls `cmd.get_attribute(x, attr)`
`x.unset(attr)`	Calls `cmd.remove_attribute(x, attr)`

- **Attribute Iterators**: Collections can be indexed via an integer or slice value. This gives access to the individual items of a collection. Additionally, collections can be indexed by a string or a tuple of strings. Indexing in this way provides access to the attributes of the collection via an iterator. For example:

`for name,areacost in x['full_name', 'areacost']:`	Iterate over the values of the `full_name` and areacost attributes in the collection. In the body of the loop `name` refers to the value of the `full_name` attribute for the current item.
`for obj,(name,areacost) in zip(x, x['full_name', 'areacost']):`	Like the above loop, loop over the collection contents in addition to the attribute values.
`x['full_name', 'areacost'].to_pandas()`	Build a Pandas DataFrame object. The DataFrame contains a column for each attribute specified, with a row for each item in the source collection. See the Pandas User Guide for details about the DataFrame type. If the collection is empty, `None` is returned.
`x['areacost'].to_array()`	If a single attribute name is used, then the NumPy array can be requested. If the collection is empty, `None` is returned.

- **DataFrame Contents**: A Pandas DataFrame is a grouping of multiple NumPy (np) arrays (each column is an array). When attribute values are converted into these arrays, the type of underlying array differs depending on the values in the attribute. Each attribute type converts like this:

`Boolean`	The array is of type `np.dtype('bool')`. If an attribute is unset on an object the value is `False`.
`Int`	The array is of the type `np.dtype('int32')` unless an attribute is not set on an object. In that case, the type is `np.dtype('float64')`. Items with the unset attribute have the value `NaN`.
`Double`	The array is of type `np.dtype('float64')`. If an attribute is unset on an object, the value is `NaN`.
`Float`	The array is of type `np.dtype('float32')`. If an attribute is unset on an object, the value is `NaN`.

`String`	If the attribute values are less than 32 characters and no spaces exist in any attribute value (cannot be interpreted as a Tcl list), then the type is `nd.dtype('<UN')` where N is the length of the longest attribute result. Otherwise the type is `nd.dtype('O')` where O is object storage and each entry has the type `snps.value`. Unset attribute values are empty string.
`Collection`	The array is of type `nd.dtype('O')` and each entry is of type `snps.collection`. Unset attributes use the empty collection.

5

Python Inputs and Outputs

All outputs generated by Python code show up in the application log file. Any output can be captured via a Python execution context.

The `snps.redirect` object provides a Python execution context that is used to redirect application output into a new file or into an in-memory buffer. For example:

```
# redirect to file
with snps.redirect('myFile', compress=True):
     cmd.report_timing()

# Redirect to variable
out = io.StringIO()
with snps.redirect(out):
     cmd.report_timing()
```

The `snps.redirect` constructor can take one of the following forms:

- Redirects to the specified file.

  ```
  snps.redirect(file [, compress=True] [, append=True] [, tee=True])
  ```

- Redirects to a stream – The stream is any object with a write method. Typically, an `io.StringIO` object is used.

  ```
  snps.redirect(stream [, tee=true])
  ```

Feedback

6

Logging and Messaging

The output and messages generated by application commands are the same for Tcl and Python. The commands generate the same messages, warnings, and errors in both the Python and Tcl implementations.

The Tcl `log_trace` command creates a flat replay Tcl script that can be used to replay every application command issued in a session.

Note: When the application commands are evaluated in Python mode, the generated trace log contains the equivalent Tcl commands, rather than the Python syntax.

Feedback

7

Python Script Examples

The following table shows the Tcl command and the corresponding Python commands:

Table 3 Tcl Commands and Their Corresponding Python Commands

TCL Command	Python Command
`create_cell`	`cmd.create_cell()`
`create_net`	`cmd.create_net()`
`connect_net`	`cmd.connect_net()`
`create_bound`	`cmd.create_bound()`
`move_object`	`cmd.move_object()`
`add_to_bound`	`cmd.add_to_bound()`
`get_cells`	`cmd.get_cells()`
`get_lib_cells`	`cmd.get_lib_cells()`
`get_nets`	`cmd.get_nets()`

Here are a few examples of the Python codes from the tool.

Example 1 Plot a bar graph that shows the number of cells mapped of different types

```
import matplotlib.pyplot as plt
    plot = {}
    seq = cmd.all_registers('-edge_triggered')
    plot['sequential'] = len(seq)
    buf = cmd.get_cells("*", '-hier', filter = "function_id==a1.0 ||
 function_id== Ia1.0")
    plot['Buffer'] = len(buf)
    xor = cmd.get_cells("*", '-hier', filter = "function_id== xor2")
    plot['XOR'] = len(xor)
    xnor = cmd.get_cells("*", '-hier', filter = "function_id== Ixor2")
   plot['XNOR'] = len(xnor)
   mux = cmd.filter_collection(cmd.get_cells("*", '-hier'), '-regexp',
 '-nocase', " ref_name =~.*mux.*")
```

```
    plot['MUX'] = len(mux)
    print("Combinational = ",len(xor)+len(buf))
    cmd.start_gui()
    plt.bar(plot.keys(), plot.values())
    plt.title("Cells in the Design")
    plt.show()
```

Example 2 Find the total number and area occupied by cells of different library types

```
total_cells = cmd.get_flat_cells()
hvt_cells = cmd.get_flat_cells().filter("ref_name =~ *HVT*")
lvt_cells = cmd.get_flat_cells().filter("ref_name =~ *LVT*")
print ('Total cells is', len(total_cells))
#Total cells is 567
print ('Total cells is', len(lvt_cells))
#Total cells is 495
print ('Total cells is', len(hvt_cells))
#Total cells is 72
area_total = 0
for cell_area in total_cells['area']: area_total = area_total + cell_area
print ('Total Area occupied by all cells', area_total)
#Total Area occupied by all cells 1654.2233107089996
area_lvt = 0
for cell_area in lvt_cells['area']: area_lvt = area_lvt + cell_area
print ('Total Area occupied by LVT cells', area_lvt)
#Total Area occupied by LVT cells 1559.9358863830566
area_hvt = 0
for cell_area in hvt_cells['area']: area_hvt = area_hvt + cell_area
print ('Total Area occupied by HVT cells', area_hvt)
#Total Area occupied by HVT cells 94.287424325943
```

Example 3 Get lib cell candidates with unique voltage structure

```
from matplotlib.path import Path
import re

def vt(lc):
    cmd.current_lib(lc['lib_name'])
    cmd.current_block('{}.frame'.format(lc['name']))
    for vt in vt_layers:
        aa = cmd.get_shapes().filter("layer.name == {}".format(vt))
        if aa:
            pl = Path([(-0.001, -0.001), (0.001, -0.001), (0.001, h +
 0.001), (-0.001, h + 0.001)])
            res = re.findall(r'{(.*?)}', pp)
            for i in res:
                pp = i.split()
                r = pl.contains_points([(float(pp[0]), float(pp[1]))])[0]
                if r:
                    cc.append(pp[1])
        else:
            bb.append(0)
    bb = '|'.join([str(b) for b in bb])
```

```
        return bb

    vt_layers = list(cmd.get_layers().filter("layer_type ==
     implant")['name'].to_array())
    vt_layers = [x.replace('{', '').replace('}','') for x in vt_layers]
    lib_cells = cmd.get_lib_cells()['lib_name', 'name', 'width',
     'height'].to_pandas()
    lib_cells['vt'] = lib_cells.apply(lambda x: vt(x), axis=1)
    for v in range(len(vt_layers)):
    lib_cells[vt_layers[v]] = lib_cells['vt'].map(lambda x: x.split('|')[v])
    ll = lib_cells.drop(columns=['lib_name', 'name', 'width', 'height', 'vt']
    ll.drop_duplicates(inplace=True)
    print(ll)
```

Example 4   Query timing arc count for all lib cells that are used in one block

```
    cells = cmd.get_cells('-hierarchical').filter('is_hierarchical==false')
    pp = cells['full_name', 'ref_name'].to_pandas()
    pp['ref_name'] = pp['ref_name'].astype("string")
    lc_indexs = pp['ref_name'].drop_duplicates().index.to_list()
    pl = pp.loc[lc_indexs, :]
    pl['timing_arc_count'] = pl['full_name'].apply(lambda x:
     len(cmd.get_timing_arcs('-of_objects', cmd.get_cells(x))))
    print(pl)
```

Example 5   Query and update dont_touch attribute for all cells

```
    def query_update(c):
        cell = cmd.get_cells(c)
        s = cell.get('dont_touch')
        cell.set('dont_touch', 'false')

        return s
    cells = cmd.get_cells(physical_context=True)
    dont_touch = cells['dont_touch'].to_array()
    cells.set('dont_touch', False)
```