

GiD

**The pre and postprocessing
system for computer analysis
in science and engineering**

Using Python in GiD

1 Tohil Python package	4
1.1 Tohil 4.3 documentation	5
1.1.1 What's New in Tohil	6
1.1.1.1 What's New In Tohil 4.3	7
1.1.1.2 What's New In Tohil 4.2	8
1.1.1.3 What's New In Tohil 4.1	9
1.1.1.4 What's New In Tohil 4.0	10
1.1.1.5 What's New In Tohil 3.2	12
1.1.1.6 What's New In Tohil 3.0	13
1.1.2 The Tohil Tutorial	14
1.1.2.1 1. The Tohil Tutorial	15
1.1.2.2 2. Using Tcl from Python	16
1.1.2.3 3. Using Python From Tcl	20
1.1.2.4 4. Tohil's Tclobj Python Data Type	23
1.1.2.5 5. Tohil's Tcdict Python Data Type	26
1.1.2.6 6. TclProcs	28
1.1.2.7 7. Shadow Dictionaries	29
1.1.3 Tohil Reference	30
1.1.3.1 Tohil Introduction	31
1.1.3.2 Tohil Python Functions	32
1.1.3.3 Tohil Tcl Functions	35
1.1.3.4 Tohil Types	36
1.1.3.5 Tohil Exceptions	40
1.1.3.6 Tohil Tcl Errors	41
1.1.4 Building and Installing Tohil	42
1.1.4.1 Building and Installing on Linux	43
1.1.4.2 Building and Installing on macOS	44
1.1.4.3 Building and Installing on FreeBSD	46
1.1.5 Dealing with Bugs	48
1.1.6 Tohil Copyright and License	49
2 Install more Python modules	50
3 Run Python as external process	51
4 Run Python inside GiD	52
4.1 From the lower entry	53
4.2 IDLE shell	57
4.3 From an user-macro button	58
4.4 From a problemtype or plugin	59
5 Real example: meshio GiD plugin	60
6 Debug Python code	66
7 Debug Python from VS Code editor	68
8 Windows 7 issues	71
9 macOS issues	72
10 Future work	73

Using Python in GiD

GiD uses Tcl/Tk as scripting language and to create its GUI

Python is other scripting language, with simple syntax and very popular (and exists a lot of modules implementing features)

It is very interesting to allow call Python code from GiD, and interact with GiD (asking data, or doing actions), and this is now possible using the tohil package.

- It is possible to re-use our current Python code in GiD without need to be re-written in Tcl language.
- It is possible to use a lot of Python modules to expand the current possibilities of GiD, (e.g. numpy, scipy, TensorFlow, matplotlib, ...)
- Developer users with experience in Python can develop code without need to learn Tcl language (except the minimum to call the Python code)
- GiD-problemtypes and plugins Python-based become self-contained and will use a known Python version (without the need to globally install Python or use other incompatible version)

Tohil Python package

GiD 16.1.2d developer will include a Tcl package called tohil that provides ways to exchange data and execute code between the Python and Tcl interpreters.

The documentation of tohil is at <https://flightaware.github.io/tohil-docs/>

A python interpreter (initial version 3.10.5) will be included inside this package and some common modules

- numpy 1.23.3 : efficient use of arrays
- matplotlib 3.6.1 : to plot graphs
- meshio 5.3.4 : to convert between several mesh formats
- h5py 3.7.0 : to read/write HDF5 scientific data format
- netCDF4 1.6.1 : to read/write netCDF4 scientific data format
- debugpy 1.6.3 : to allow debug of Python from VSCode editor in case of embedded Python interpreter

Tohil 4.3 documentation

Welcome! This is the documentation for Tohil 4.3. Tohil, a feathered serpent, powerfully joins Python and Tcl.

**Parts of the documentation:**[What's new in Tohil 4.3?](#)[Tutorial
start here](#)[Tohil Reference](#)[Building and Installing Tohil](#)*Building Tohil on Unix systems such as Linux, MacOS and
FreeBSD***Meta information:**[Reporting bugs](#)[About the documentation](#)[Tohil Copyright and License](#)**Other resources**

- [Tohil github repo](#)
- [FlightAware](#)
- [Python](#)
- [TCL](#)

What's New in Tohil

"What's New in Tohil" describes the big changes between major Tohil versions.

What's New In Tohil 4.3

Tohil 4.3 is mostly a maintenance release. One nice improvement, TclProcs now return the tclobj datatype by default. If you haven't seen them, TclProcs are the slickest way to invoke Tcl procs from Python using tohil. A few releases back we changed tohil.call and tohil.eval to return Python tclobj objects by default (which can be overridden). Now TclProcs work that way too.

Since tclobjs behave like strings when used as strings in Python, very little if any code, even code that makes extensive use of TclProcs, should require changes.

Additional Improvements

- Much faster Python-to-Tcl floating point and integer conversions.
- Tohil previously stored a Python capsule containing a pointer to its corresponding Tcl interpreter in `__main__.interp` and you could make Python crash by doing like `interp = ""` before importing tohil. This renames `interp` to something way less likely to have a conflict.
- Many new tests

Bug Fixes

- Thread state handling improvements aka bug and crash fixes when using Tohil/Python from multiple Tcl interpreters.
- Tohil tclobj integer math is now always performed at 64-bits, even on 32-bit machines.
- Fixed infinite recursion when Tohil's exception handler caused an exception.

Improved Build Support

- configure script improvements to permit building Tohil with nix (<https://github.com/flightaware/tohil/pull/65>).
- tests can now be run via nix.

For release notes on github, visit [the Tohil github repo](#).

For the full changelog, visit the [Tohil github changelog between 4.2.0 and 4.3.0](#).

What's New In Tohil 4.2

Welcome to Tohil 4.2.

4.2 is primarily a maintenance release, but includes at least one really nice new feature:

Python code passed to `tohil::exec` is now unindented before being passed to Python

Tohil's #1 new feature request! Up until now, the argument to `tohil::exec` had to obey Python indentation rules including there being no indentation at all for the top level, leading to ugly stuff like:

```
tohil::exec {
def new_validate(self, data):
    return json.loads(base64.b64decode(data))
}
```

^ The "def" here has to occur at the beginning of the line, i.e. not be preceded by any spaces or tabs, or Python will raise an exception. This nesting does not "read" well.

To make it easier to make your code read well and comply with Python indentation rules, if the first nonblank line starts with whitespace, `tohil::exec` will un-indent the code block such that the first line is not indented at all and following lines are unindented to match, all done lickety split, natively in C.

So you can now nest your embedded Python code in a more standard way:

```
tohil::exec {
    def new_validate(self, data):
        return json.loads(base64.b64decode(data))
}
```

Additional Improvements

- Added `-nonevalue` option to `tohil::call`, allowing the "none" sentinel to be specified arbitrarily (Retains the default value of `tohil::NONE`.)
- Cleaned up tohil namespace so `dir(tohil)` doesn't show modules tohil imported as if it had created them.
- Made package forget tohil work.
- Added support for the `Tcl unload` command to be able to unload the Tohil shared library. (Consider it risky, though.)

Bug Fixes

- Fixed crash when `register_callback`-registered functions raised a Python exception

Improved Build Support

- Added support for building tohil as a Debian package
- Homebrew formula for building with homebrew

What's New In Tohil 4.1

tclobjs returned from more places

- `to=dict` conversions now returns tclobjs for the dictionary values.
- `tclobj.as_dict()` does so as well.

Python-side callback function registration

The new `register_callback` function provides a nice way to create Tcl commands that directly call corresponding Python functions. This is useful for processing asynchronous callbacks from the Tcl event loop using Python, and may be useful for other stuff as well.

Several other improvements

- Pass `None` from Tcl to Python functions called via `tohil::call` by using the `tohil::NONE` sentinel.
- More precise error messages when Tohil startup fails should help with troubleshooting installation problems.
- Many new tests.

Numerous bug fixes

- Python builtins can be called from Tcl-side `tohil::call` without resorting to explicitly specifying the builtins namespace.
- Fixed bugs in how `tclobjs` (tclobjs bound to vars) handled some methods, such as `insert`, `pop`, `append`, and `extend`.
- Correct behavior of `tclobj` iterators, also fixes a crash.
- If Python is initializing Tcl, it now does so using `package require` rather than a `Tohil_Init` to cause Tohil's Tcl package code to get sourced. (It also requires the exact version of Tohil that it is, to reduce the risk of it loading some other version of the library when multiple versions are installed.)

Considerably improved documentation

We've considerably improved and extended the Tohil tutorial and reference, in Python-standard RST format, and are serving it out at <https://flightaware.github.io/tohil-docs/>

What's New In Tohil 4.0

tclobj default return

This is a biggie. Many Tohil functions accept a `to=` argument where you can specify a Python data type to convert a tcl object returned from doing a call or accessing tcl data. You can set a return type of `str`, `int`, `bool`, `float`, `list`, `set`, `dict`, `tuple`, `tohil.tclobj` or `tohil.tcldict`.

Prior to Tohil 4, if you didn't set a `to=` return type, the default return type was string, `str`. This seemed perfectly reasonable; after all, in Tcl, despite it having internal objects and maintaining in them a cache of a conversion to a data type such as integer, list, etc, in Tcl "every value is a string."

However, as we have enhanced and extended Tohil's `tclobj` type, it has become ever easier to use `tclobjs` directly from Python with no funny business. You can get a `tclobj`'s string representation with `str()`, integer with `int()`, float with `float()`, list with `list()`, and others. You can use Python list notation to access and manipulate elements of `tclobjs` when they contain lists, can iterate over them, etc.

Since Tohil's `tclobj` type implements Python's number protocol, if `tclobjs` contain numbers, they can be used in calculations without conversion via `int()` and `float()`.

Consequently starting in Tohil 4, the default `to` return is now `tohil.tclobj`. In our experience, and a little bit to our surprise, most Python code that uses Tohil will "just work" without modifications.

If, though, for instance, you didn't specify a default return and then knowing you would get a `str` invoked string methods on the `str` that was returned, you'll probably get an error because the `tclobj` doesn't implement all of the `str` datatype's methods. In this case, adding a `to=str` to the Tohil call will be sufficient to get your code working under Tohil 4.

Python Subinterpreter Support

Full Python subinterpreter support!

First, starting with version 4, Tohil properly supports multi-phase init, meaning that multiple Python interpreters (the Python interpreter and any subinterpreters) can import tohil and they will get their own instance of Tohil, so there is no "crosstalk" between the interpreters.

Second, Tohil recognizes when a second Tcl interpreter within the same process has done a `package require tohil` and will create and exclusively interact with a separate, distinct Python subinterpreter for each corresponding Tcl interpreter.

Say for instance you create a new Tcl interpreter from Tcl, using something like `set interp [interp create]` and then do `$interp eval "package require tohil"`, that second interpreter doing the package require causes a new Python subinterpreter to be created and initialized.

And it works great.

When any of the Tcl interpreters exercises their Python interpreter, Tohil will automatically switch Python's executing interpreter to that interpreter (swap its thread state), if needed.

Upon deletion of a Tcl interpreter, if there is an attached Python subinterpreter, it is deleted as well.

Implementation Note: This was pretty tricky, because we previously had global variables, in particular one pointing to the Tcl interpreter. We had to figure out ways to stash the pointer to the Tcl interpreter in Python using C such that we could find it later when we didn't have control over how we were called, for example we are being called from Python with to do some Python thing, you only get what it calls you with. So we stashed the interpreter pointer in a capsule in Tohil's Python data types' dictionaries and in `__main__`'s dictionary. It turned out really nice.

Support for Separate Virtual Interpreters in Rivet

A nice bit of fallout from the above, if you're running the Apache webserver with the [Apache Rivet](#) module installed and running in the mode where different virtual hosts run in separate Tcl interpreters, known as separate virtual interpreters, each vhost that does a `package require tohil` will get its own Python subinterpreter, isolating Python between the vhosts just as Tcl is.

A function can now be specified in a to= arg

The `to=` argument to a Tohil function such as `tohil.eval`, `tohil.call`, etc, has until now been required to specify a Python data type such as `int`, `float`, `str`, `tohil.tclobj`, etc. It can now also be specified as a callable function.

If the `to` argument is not a recognized data type but is a callable function, Tohil will call that function with one argument, a `tclobj` object containing the object to be returned, and it is expected that the function will manipulate the object in some way and then return a result. Whatever the function returns is what the relevant tohil function will return.

This provides an additional way for a Tohil developer to customize the return of some Tcl activity in order to make it more standard and readily useful to the Python caller.

User-Facing Behavior Changes

- When Tcl is the parent, `package require tohil` will, in addition to initializing Python, automatically import tohil on the Python side.

Internal Changes

- When Tcl is the parent and Tohil initializes Python from scratch, we use `PyInitializeEx(0)` instead of `Py_Initialize` to prevent Python from registering signal handlers. (Signal handling probably ought to be Tcl's business under this circumstance.)
- Internal code refactoring and cleanup.

Documentation Improvements

- Greatly improved documentation in Python-standard format.
- Makefile and docs for building the docs.
- `make serve` target to serve the docs via http (for devs)

What's New In Tohil 3.2

tclobjs

Tohil Tcl objects, *tclobjs*, are a data type Tohil creates in Python. *Tclobjs* have gained considerable new power. Among them, they now implement the number protocol.

This means *tclobjs* can be used for number in numeric calculations without needing to pass through *int()* or *float()*.

Testing of *tclobjs* for boolean value now provides tcl semantics. 'f', 'F', 'n', 'N', 0, substrings case-insensitively matching "false" or "no", evaluate as false; 't', 'T', 'y', 'Y', any number other than 0, substrings matching "true" or "yes", evaluate true.

Tclobjs can be used as one or both operands for addition, subtraction, multiplication, division, remainder, divmod, bitwise or, and, xor, left shift, right shift, etc. Unary ops invert, negative, position, absolute value all work.

Tclobjs can be used for "inplace" number calculations such as *+=*, */=*, *<=<*, etc.

Tclobj iterator code was rewritten from Python into C.

Removed almost all of the *as_* methods of *tclobjs* that convert *tclobjs* into various Python data types. *tclobj.as_int()* has been replaced by *int(tclobj)*, *as_bool* by *bool()*, *as_float* by *float()*, *as_str* by *str()*. *llength()* has been removed; you can use *len()* to get the same thing.

Removed *tclobj*'s *as_tuple()* method; use *tuple(tclobj)* instead. Likewise removed *as_tclobj()* method; use *tohil.tclobj(tclobj)* instead. Remove *as_tcldict()*; use *tohil.tcldict(tclobj)* instead.

What's cool is these functions are provided by the *tclobj* type implementation, so that are real efficient in terms of how they interact with the underlying Tcl objects.

The *tclobj.reset()* method has been renamed to *clear()* for consistency with Python lists and dicts. It also works for *tcldicts*.

Tclobj lappend method has been renamed to *append* and *lappend_list* renamed to *extend*, for compatibility with Python's lists.

- *tclobjs* can now ingest python sets (in addition to lists, tuples, etc, which it already could do.)

Tclobj Shadow Vars

Another new feature, *Tclobj* shadow vars, *t = tohil.tclvar('t')*, makes *t* a *tclobj* that shadows a variable *t* in the Tcl interpreter. Any changes to the variable from the Tcl side are "seen" from the Python side, and vice versa. The variable can also be an array element.

Tcldicts

- *Tcldict* objects now provide many methods that standard Python dicts provide, such as *keys()*, *values()*, *items()*. Because of this, *dict(tcldict)* now works.
- The *clear()* method is now supported to empty the Tcl dict.
- A new *tcldict pop* method behaves the same as *pop* for standard Python dicts, popping the last item in the list if no position is specified, else popping the specified position, i.e. removing it from the list and returning it.
- A new *.insert(i, x)* method will insert item *x* at position *i*.

ShadowDicts

ShadowDicts implement many additional methods implemented by standard Python dicts.

ShadowDicts now have a *get* method that behaves as standard dicts do. A new *clear* method removes all items from the shadow diction, i.e. it unsets the shadowed Tcl array.

Python exception improvements

We now raise more standard Python *TypeError*, *KeyError* and *ValueError* exceptions in places where we used to just raise *RuntimeError*.

Tohil method improvements and changes

Tohil.unset can now take an arbitrary number of arguments of variable names and array elements to unset, include zero. As before, it is fine to unset something that doesn't exist.

Testing Improvements

- Lots of new tests.
- Also we're now using the hypothesis testing framework and have found and fixed a number of problems because of it.
- All tests pass now on 32-bit ARM Linux.
- Linux CI automated testing using Github Actions

Build Improvements

What's New In Tohil 3.0

Welcome to Tohil 3.

Tohil 3 brings forward all the slick stuff from Tohil 2, plus it provides the means of accessing Tcl functions from Python in such a way that they very much look and behave like native Python functions. Not only that, but for Tcl procs made available to Python by tohil, every parameter can be specified by position or by name, something few native Python functions or Python C functions can do.

TclProcs

Any Tcl proc or C command can be defined as a Python function simply by creating a TclProc object and then calling it.

```
>>> import tohil
>>> tohil.package_require("Tclx")
'8.6'
>>> intersect = tohil.TclProc("intersect")
>>> intersect([1, 2, 3, 4, 5, 6], [4, 5, 6, 7, 8, 9], to=list)
['4', '5', '6']
```

It's pretty fun to play with them this way from the command line.

While TclProcs are directly callable, as seen above, they are fully fledged Python object and have a number of interesting and potentially useful attributes and method, including the Python function name, Tcl proc name, whether the Tcl function being shadowed is a proc or not (if not, it's a command written in C), a Python dictionary specifying any default arguments and their values, and the proc's arguments.

TclNamespaces

But wait, there's more. `tohil.import_namespace(my_namespace)` will create a TclNamespace object and import all the procs and C commands as methods of that namespace, recursively importing any subordinate namespaces and their procs and C commands as well. Namespaces and function calls can be chained, so you get the hierarchy of Tcl namespaces and procs and C commands created after loading all of your packages, chainable from Python.

It's a convenient way to leverage TclProcs across all of your Tcl procs and commands.

```
>>> import tohil
>>> tohil.package_require("clock::rfc2822")
'0.1'
>>> tcl = tohil.import_tcl()
>>> tcl.clock.rfc2822.parse_date('Wed, 14 Apr 2021 12:04:48 -0500', to=int)
1618419888
```

TclError Exception Class

Tohil 3 also adds a sweet TclError exception class, and any Tcl errors that bubble back all the way to Python without any Tcl code having caught the error will be thrown in Python as TclError exceptions. The TclError object can be examined to find out all the stuff Tcl knows about the error... the result, the error code, code level, error stack, traceback, and error line.

New helpers Functions

`tohil.package_require` is real useful. The others ones tohil needs for itself and they're not as useful, but maybe for some people for some purposes.

- `tohil.package_require(package_name, version=version)`
- `tohil.info_procs()` - return a list of procs. pattern arg optional.
- `tohil.info_commands()` - return a list of commands, includes procs and C commands.
- `tohil.info_body()` - return the body of a proc.
- `tohil.info_default()` - return the default value for an argument of a proc
- `tohil.info_args(proc)` - return a list of the names of the arguments for a proc
- `tohil.namespace_children(namespace)` - return a list of all the child namespaces of a namespace

Tests

- Dozens of new tests.
- "make test" runs both the Python ones and the Tcl ones

The Tohil Tutorial

1. The Tohil Tutorial

Tohil is simultaneously a Python extension and a TCL extension that makes it really seamless to move data around and invoke functions in one from the other.

Tohil is open source software, available for free including for profit and/or for redistribution, under the permissive 3-clause BSD license (See [_copyright-and-license](#)).

It is written in C, Python and Tcl, and makes use of the Python and Tcl C APIs to let Python call Tcl, Tcl call Python, give Python access to Tcl's objects and give Tcl access to Python's objects.

Not just strings and ints and float, but lists, dicts, sets, tuples, and more, flow freely, intuitively and largely unencumbered between the two languages.

Tohil is efficient when moving data between the languages. Integers and floats are copied natively; they do not suffer an intermediate conversion through strings. Likewise lists, tuples, dicts, etc, are accessed through the C language mechanisms provided by the two languages, ergo accessed and manipulated in the most native and efficient ways.

Tohil is freely available in source form from the Tohil github website, <https://github.com/flightaware/tohil> , and may be freely distributed.

This tutorial is intended to provide an introduction to Tohil, and typical ways of using it.

To follow along and start experimenting with Tohil you'll want to have working Python and Tcl interpreters, and have Tohil built and installed such that `import tohil` works from Python and `package require tohil` works from Tcl.

2. Using Tcl from Python

Here we'll introduce using Tcl from Python.

Hopefully you've got Tcl and Python and Tohil installed and you can follow along and try stuff out.

2.1. *tohil.eval*

```
>>> import tohil
>>> tohil.eval('puts "Hello, world."')
Hello, world.
```

Not bad. You can actually do a lot with that.

Anything the Tcl code returns can be gotten by Python.

```
>>> t = tohil.eval('return "Hello, world."')
>>> t
<tohil.tclobj: 'Hello, world.'>
>>> str(t)
'Hello, world.'
```

Here we'll use Tcl's *clock format* function to format a Unix epoch seconds-since-1970 clock into a Posix standard time in the Spanish locale:

```
>>> clock = 1616182348
>>> tohil.eval(f'clock format {clock} -locale es -gmt 1")
'vie mar 19 19:32:28 GMT 2021'
```

2.2. *helper functions*

We can load in Tcl packages by doing

```
>>> tohil.eval('package require Tclx')
```

but we do this so often that tohil provides a shortcut:

```
>>> tohil.package_require('Tclx')
```

You can specify the version as an optional argument, either by positional or named parameter. The following two statements are equivalent:

```
>>> tohil.package_require('Tclx', '8.6')
>>> tohil.package_require('Tclx', version='8.6')
```

Experienced Python developers without a lot of Tcl experience may be surprised by Tcl's leniency when it comes to data types.

Here we request a Tcl package with the version number specified as floating point. It works fine.

```
>>> tohil.package_require('Tclx', 8.6)
```

Another one you'd end up doing a lot is `tohil.eval("source file.tcl")`. For that we provide the slightly less paper-cutty...

```
>>> tohil.source("file.tcl")
```


2.3. *tohil.call*

You get fancy and start using f-strings to create Tcl commands with arguments, maybe you're doing something like

```
tohil.eval(f"register_user {user_id} {user_name} {user_fullname}")
```

If any of those variables being substituted contain dollar signs, quotes, or square brackets, you're not going to have a good time, because Tcl is going to try to interpret that stuff, and that could lead to errors up to and including remote code execution.

Consequently, Tohil provides *tohil.call*, a function that takes an arbitrary number of arguments and passes them one-for-one to the corresponding Tcl function in a way that keeps Tcl from trying to interpret any of the arguments.

```
>>> import tohil
>>> clock = 1616182348
>>> tohil.call('clock', 'format', clock, '-locale', 'fr')
'ven. mars 19 19:32:28 UTC 2021'
```

The key thing in the above is `tohil.call('clock', 'format', clock, '-locale', 'fr')`, equivalent to `tohil.eval(f"clock format {clock} -locale fr")` but without the risk of inadvertent misinterpretation of arguments.

2.4. *tohil.expr*

You can also evaluate Tcl expressions from Python using *tohil.expr*. As with many other tohil functions, `to=` can be used to request conversion to a specific Python datatype.

```
>>> tohil.expr('5+5')
'10'
>>> tohil.expr('5**5')
'3125'
>>> tohil.expr('1/3')
'0'
>>> tohil.expr('1/3.')
'0.3333333333333333'
>>> tohil.expr('1/3.',to=float)
0.3333333333333333
>>> tohil.expr('[clock seconds] % 86400')
'25571'
>>> tohil.expr('[clock seconds] % 86400',to=int)
25571
```

2.5. *tohil.getvar* and *tohil.setvar*

Python has direct access to Tcl variables and array elements using *tohil.getvar*. Likewise, *tohil.setvar* can set them.

```
>>> import tohil
>>> tohil.setvar("foo", "bar")
>>> tohil.getvar("foo")
'bar'
>>> tohil.setvar(var="happy", value="lamp")
>>> tohil.getvar("happy")
'lamp'

>>> tohil.eval("array set x [list a 1 b 2 c 3 d 4]")
''
>>> tohil.getvar('x(a)')
'1'
>>> tohil.getvar('x(a)', to=int)
1
>>> tohil.getvar(var='x(b)', to=float)
2.0
>>> tohil.getvar("x(e)")
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: can't read "x(e)": no such element in array
```

As you can see, it's an error to try to get a variable or array element that doesn't exist. You can use *tohil.exists* to see if the variable exists, or trap the Python exception, or make use of *tohil.getvar*'s handy *default* keyword-only argument:

```
>>> tohil.getvar("x(e)", default="0")
'0'
>>> tohil.getvar("x(e)", default=0, to=int)
0
>>> tohil.getvar("x(d)", default=0, to=int)
4
```

2.6. *tohil.exists*

You can use *tohil.exists* to see if a variable or array element exists:

```
>>> tohil.eval("array set x [list a 1 b 2 c 3 d 4]")
''
>>> tohil.exists("x(c)")
True
>>> tohil.exists("x(e)")
False
>>>
>>> tohil.exists("banana")
False
```

2.7. *tohil.incr*

tohil.incr takes a Tcl variable name or array element and attempts to increment it.

If the contents of the variable preclude it being used as an int, a Python `TypeError` exception is thrown.

An optional position argument specifies the amount to increment by. The default increment is 1. Negative increments are permitted. The increment amount can also be specified as a keyword argument, using "incr".

```
tohil.incr('var')
tohil.incr('var',2)
tohil.incr('var',incr=-1)
```

2.8. *tohil.unset*

tohil.unset can be used to unset variables, array elements, and even entire arrays in the Tcl interpreter.

```
>>> tohil.setvar("x(e)", "5")
>>> tohil.getvar("x(e)")
'5'
>>> tohil.unset("x(e)")
>>> tohil.getvar("x(e)")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: can't read "x(e)": no such element in array
```

- Unset takes an arbitrary number of arguments, including zero.
- Unsetting an array element uses Tcl subscript notation, for example `tohil.unset('x(e)')`.
- Unsetting an array by name without a subscript will unset the entire array.
- It is not an error to attempt to unset a variable that doesn't exist.

2.9. *tohil.subst*

Tcl's *subst* command is pretty cool. By default it performs Tcl backslash, command and variable substitutions, but doesn't evaluate the final result, like *eval* would. So it's handy for generating some kind of string, but with embedded `$`-substitution and square bracket evaluation.

```
>>> import tohil
>>> tohil.eval("set name karl")
'karl'
>>> tohil.subst("hello, $name")
'hello, karl'
```

2.10. *tohil.convert*

tohil.convert will convert some Python thing passed to it, into a Tcl object, and then back to some other Python type, any type supported in accordance with the *to=* argument.

The “to=” way of requesting a type conversion is supported. Although you might not care about converting to int or float or something, you might want a *tohil.tclobj* for your efforts, anirite?

2.11. *tohil.interact*

Run the Tcl interactive command loop on stdin, hopefully a terminal, until you send an EOF, at which point you'll be returned to the Python command line. See also *tohil::interact*.

THis is handy if you're using Python interactively and you find yourself making a lot of *tohil.eval* calls to manipulate the Tcl interpreter, you can flip to the Tcl interpreter, interact with it directly, then flip back by sending an end-of-file.

2.12. *tohil.tcl_stdout_to_python()*

Redirect Tcl's standard output to pass through Python's I/O subsystem.

Among other things, if using [Jupyter Notebook](#), invoking *tohil.tcl_stdout_to_python()* will cause output from the Tcl interpreter to appear in the notebook.

tclvar

3. Using Python From Tcl

In this section we'll introduce using Python from Tcl.

Hopefully you've got Tcl and Python and Tohil installed and you can follow along and try stuff out.

Let's fire up Tcl and mess around with Python:

```
$ tclsh
% package require tohil
4.0.0
```

OK, good news, we've got a working Tcl and Tohil.

If the package-require failed then please visit the installation instructions and get tohil built and installed on your computer.

3.1. *tohil::eval*

```
% tohil::eval "37 + 5"
42
```

That may not look like much, but Tohil got Python to add 37 and 5 for us and returned the result back to tcl.

```
% set answer [tohil::eval "37 + 5"]
42
% puts $answer
42
```

3.2. *tohil::exec*

In Python, *eval* evaluates a single expression and returns the result, so even trying to eval something like `answer = 42` is an error. Python provides *exec*, which can evaluate an arbitrary code block, and Tohil hews to that by providing *tohil::exec*.

```
% tohil::exec "answer = 37 + 5"
% tohil::exec "print(answer)"
42
```

OK, that's pretty cool. Tohil is getting Python to do stuff for us from Tcl. Yay.

(A quick note, *tohil::eval* and *tohil::exec* are named and work the way Python's *eval* and *exec* work. Tcl has its own *eval* for evaluating Tcl stuff. It is for Tcl something closer to Python's *exec*, except that Tcl's *eval* returns a result, while Tcl's *exec* runs programs and returns their output, something much different.)

```
% expr 5 / 4
1
% tohil::eval "5 / 4"
1.25
% tohil::eval "5 // 4"
1
% expr 5 // 4
missing operand at _@_
in expression "5 /_@_/ 4"
while evaluating expr 5 // 4
```

Yep, it's Python we're talking to, all right. See how Tcl division of two integers yielded an integer result while Python, a float? Then we used Python's integer division `//` to get integer division, while trying that with Tcl was an error because Tcl doesn't have that operator.

3.3. *tohil::import*

OK, we can start doing Python stuff from Tcl, like import a module.

```
% tohil::exec "import numpy"
```

We do this often enough that Tohil provides a shortcut:

```
% tohil::import numpy
```

3.4. notes about exec

One thing that can trip people up is it can be surprising that `tohil::exec` never returns anything.

```
% tohil::exec "answer = 42"
% tohil::exec "answer"
```

The above returns without an error, but doesn't provide anything.

You instead need to use `tohil::eval` in this example. You can call functions using `tohil::eval`, by the way.

Though possibly a bit surprising, this behavior is consistent with how `exec` works in Python. It probably shouldn't be a surprise that Tohil is using Python's `eval` and `exec` mechanisms at the C level to provide these capabilities to Tcl.

3.5. tohil::run

`tohil::run` is a special version of `tohil::exec` that grabs anything Python emits to `stdout` while the `exec` is running, and returns it to the caller.

3.6. tohil::call

If you start creating from Tcl, Python to be executed with `eval` and `exec`, you may notice there's a risk that if you use substitute-in data, you know, such as names, addresses, cities or whatever, that unless you are very careful, various characters can cause your Python not to parse properly. For example, a single quote in a name, quotes in general, and other stuff.

Tohil provides `tohil::call` to provide a way to call a Python function while making sure that the arguments you pass to the function are not interpreted by Python along the way.

`tohil::call` provides a way to invoke one Python function, with zero or more arguments, without having to pass it through Python's `eval` or `exec` and running the risk that Python metacharacters appearing in the data will cause quoting problems, accidental code execution, etc.

When you use `tohil::call`, Tohil converts all of your arguments to Python Unicode, unless an argument is comprised of a special sentinel, normally `tohil::NONE`, in which case that argument is replaced by the Python "None" data type.

This sentinel can be changed with the `-nonevalue` argument.

3.7. tohil::interact

Take tohil to eleven. You're on ten here... all the way up... You're on ten on your guitar... where can you go from there? Where? Nowhere. Exactly. What we do is if we need that extra... push over the cliff... you know what we do?

We run `tohil::interact` from Tcl and enter the Python interactive loop. When we're done, we send end of file (^D) to end the Python loop and return to the Tcl one.

```
% tohil::interact
>>> def foo():
...     print("bar")
...
>>> ^D
% tohil::eval foo()
bar
```

3.8. Using tohil from Rivet

[Apache Rivet](#) is an Apache webserver module that provides among other things a way for webpages to be made from HTML files with embedded Tcl code that executes when the page is requested.

From a Rivet page, in some of your Tcl code, invoke package require tohil.

If you run `tohil_rivet` it will plug tohil's Python interpreter such that everything Python writes to `stdout` using `print`, or whatever, will go through Tcl's `stdout` and thereby into your Rivet page.

```
<?
package require tohil; tohil_rivet

puts "calling out to Python to add 5 + 5: [::tohil::eval "5 + 5"]"

tohil::exec {
    print('hello, world')
    print("<hr>")
}

?>
```

4. Tohil's Tclobj Python Data Type

To provide powerful and facile interactions between Python and Tcl, Tohil provides a new Python data type, the Tcl object, or *tclobj*, aka *tohil.tclobj*.

It's a Python-wrapped Tcl object.

It's pretty insanely powerful.

See, Tcl has these objects, fairly similar to Python at the C level, that it uses throughout.

They can be a string or an int or a float or a list or a nested dictionary or a bunch of other things.

Each tclobj maintains a pointer to a Tcl object and can do things to and with that Tcl object.

4.1. Creating Tclobj Objects From Python

You can create an empty tclobj just like creating any other object from a class in Python:

```
t = tclobj()
```

Something pretty cool is you can pass many different Python data types, including lists and dicts, to *tohil.tclobj*, and it will and it will pretty much "do the right thing," i.e. it will produce something expected and straightforward that Tcl can make sense of.

For example you can pass *tclobj()* None, bools, numbers, bytes, unicode, sequences, maps, and even other tclobjs.

4.2. .getvar() and .setvar() methods

You can also attach a tclobj to a Tcl variable or array element, or set a variable or array element from the contents of the tclobj using its *getvar()* and *setvar()* methods.

4.3. Get stuff from tclobjs as Other Python Objects

Tclobjs have methods to convert the tclobj to Python strings, ints, floats, bools, lists, sets, tuples, dicts, byte arrays, and, again, tclobjs.

- *bool(t)* - Return the contents of the tclobj object as a Python bool
- *float(t)*, *int(t)* - as a python float and int (long), respectively.
- *list(t)* - as a python list
- *str(t)* - as a python str
- *tuple(t)* - as a python tuple object
- *tohil.tclobj(t)* - as a new python tclobj object
- *tohil.tcldict(t)* - as a new python tcldict object
- *t.as_byte_array()* - a Python byte array
- *t.as_dict()* - a Python dict

4.4. set() and reset()

t.set() tries to covert whatever Python object is passed to it and store it in the tclobj as a Tcl object, while *t.reset()* resets the tclobj object to contain an empty Tcl object.

4.5. incr()

t.incr() tries to increment the tclobj object. If the contents of the object preclude it from being used as an integer, a *TypeError* exception is thrown.

t.incr takes an optional positional argument, which is the increment amount. It can also be specified using the "incr" named argument.

```
t = tohil.tclobj(0)
t.incr()
t.incr(1)
t.incr(incr=-1)
```

4.6. Tclobjs Containing Tcl lists

When a tclobj contains Tcl lists, cool stuff comes into play.

Accessing a tclobj containing a list from Python is nearly identical to accessing a standard Python list. You can access and change elements, use slice notation, etc, and most standard Python list methods are provided as well.

You can get the length of the tclobj list with *len(obj)*, while *obj.index(i)* will return the *i*'th element.

You can also just use `l[i]` to get the *i*'th element of *l*, although the *lindex* supports Tohil's *to=type* conversion as well.

`obj.append()` will append to the list stored in the `tclobj`.

`obj.extend()` will append a Tcl object comprising a list, or a Python list, to a list, making it flat, i.e. each element of the list is appended to `obj`'s list.

`obj.pop(x)` will pop the *x*'th element from a tcl obj comprising a list. If no index is specified, `obj.pop()` removes and returns the last item in the list.

`obj.insert(i, x)` will insert item *x* at position *i*. So as with Python lists, `obj.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `obj.append(x)`

`obj.clear()` removes all items from the list.

You can use Python's indexing syntax to access and replace list elements.

```
>>> x = tohil.eval("list 1 2 3 4 5 6", to=tohil.tclobj)
>>> x
<tohil.tclobj: '1 2 3 4 5 6'>
>>> x[2]
'3'
>>> x[3:]
['4', '5', '6']
>>> x[-2:]
['5', '6']
>>> x[-2:-1]
['5']
```

4.7. Comparing Tclobjs w/o Each Other

Tclobjs can be compared. If equality check is requested, first their internal `tclobj` pointers are compared for absolute equality. Following that, and for all other cases (`<`, `<=`, `>`, `>=`), their string representations are obtained and compared.

Not something you probably should rely on for complicated objects but should be fine for simple ones.

Comparisons are really permissive, too, in what the `tclobj` implementation accepts from Python.

It seems pretty good, but this is new stuff, so be careful and let us know how it's going.

4.8. Get the `tclobj`'s Tcl Type and Reference Count

`t._tcltype` will tell you the tcl object type of the tcl object stored within the `tclobj`. Note that you may get nothing back even though there is some valid thing there, say for instance a dict, but you haven't accessed it as a dict, so it's just a string or list or some other data type until you do.

`t._refcount` will tell you the reference count of the `tclobj`'s tcl object. This isn't probably useful for production code but it is kind of cool for poking around and trying to understand what objects are shared and how and when and stuff.

`t._pyrefcount` likewise will return the python reference count of the `tclobj`.

Note that if you're poking around, that sometimes you might think the reference count is one higher than it should be, but frequently the object you just set the value of also happens to be the Tcl interpreter result (i.e. you used the interpreter to make it). Once the interpreter does something else and produces a new result, your object's reference count will go down by one.

If this doesn't make sense, don't worry about it. You probably don't need it and don't care anyway.

You can create a `tclobj` from most Python stuff.

...a list:

```
>>> l = [1, 2, 3, 4, 5]
>>> type(l)
<class 'list'>
>>> kl = tohil.tclobj(l)
>>> str(kl)
'1 2 3 4 5'
>>> kl.llength()
5
```

...a tuple:

```
>>> z = tohil.tclobj((1, 2, 3))
>>> str(z)
'1 2 3'
```


...a dict:

```
>>> d = {'a': 0, 'b': 1, 'c': 2, 'd': 3}
>>> z = tohil.tclobj(d)
>>> str(z)
'a 0 b 1 c 2 d 3'
```

5. Tohil's Tcldict Python Data Type

Tcldicts are Python-wrapped Tcl dictionaries.

While they have the same internal structure as *tclobjs* (a Python object pointing to a Tcl object), *tcldicts* are a distinct data type in Python because *tcldicts* have different implementations of sequences and maps and *whatnot*, to provide a Pythonic feel to Tcl dictionaries, which can also be hierarchies of key-value data.

You can create a *tcldict* object similarly to creating a *tclobj*:

```
>>> d = tohil.tcldict("a 1 b 2 c 3 d 4 e 5")
```

5.1. Accessing Elements in a Tcldict

You can access elements using normal Python dict access techniques.

For instance, *d["a"]* returns 1, *d.get('a')* does the same. With the “get” approach you can specify a *to=type* to control what Python type is returned. Also you can set the Python type returned by doing *d.to = type*.

5.2. Setting Values into a Tcldict

Setting values will do a Tcl *dict set* on a *tcldict*. It takes a key and a value. The value can be one among a number of different Python objects. Two options are to pass a *tclobj* or *tcldict* object, in which case *tohil* will do the right thing and grab a reference to the object rather than copying it.

The key can be a list of keys, in which case instead of working with dict as a single-level dictionary, it will treat it as a nested tree of dictionaries, with inner dictionaries stored as values inside outer dictionaries.

```
` d[['airport', 'KHOU', 'name']] = 'Houston Hobby' d[['airport', 'KHOU', 'lat']] = 29.6459 d
[['airport', 'KHOU', 'lon']] = -95.2769 `
```

Standard Python dicts can't do this.

The *td_get* method will also do a *dict get* on a *tclobj*. It returns the object in the style requested, *tclobj* by default, but *to=* can be specified, as in:

```
>>> x = tohil.eval("list a 1 b 2 c 3", to=tohil.tcldict)
>>> x.get('a')
<tohil.tclobj: '1'>
>>> x.get('a',to=int)
1
```

5.3. get()

Likewise, *get* will accept a list of keys, treating the Tcl object as a nested tree of dictionaries, with inner dictionaries stored as values inside outer dictionaries. It is an error to try to get a key that doesn't exist.

t.get() supports our *to=datatype* technique to get the contents of the *tcldict* as one of a number of different datatypes (the same ones supported for *tohil.getvar*, etc.)

One kind of annoyance about Tcl dicts is having to use *dict exist* to traverse the hierarchy of dicts to see if something exists before traversing it a second time to actually get it, or to try the *get* and catch the error.

get() offers a nice alternative approach, where you invoke it and specify the optional *default=* argument, where you can specify a value that *get* will return if the requested key doesn't exist.

If a *to=datatype* is specified and the default value is used, *Tohil* will coerce the default value to the *to*-specified datatype, if possible, or an exception is raised if not.

5.4. Checking for Existence

You can do the usual Python 'a' in *mydict* check for existence.

The thing being checked for can be a list, in order to navigate a hierarchy of Tcl dictionaries. For example, ["airport", "KHOU"] in *mydict*

5.5. len()

len(t) returns the size of the Tcl dict, or throws an error if the contents of the object can't be treated as a Tcl dict.

5.6. Removing Elements

Standard `del t[key]` Python stuff.

`td_remove` can also accept a list of elements and in that case it will delete a hierarchy of subordinate Tcl dictionaries. In the list case, if more than one element is specified in the list, it is an error if any of the keys don't exist.

5.7. Assembling Tcldicts from Tcldicts and Tclobjs

You can create new tclobjs or tcldicts as the contents of sub-parts of dictionaries and use them as dictionaries in their own right, or whatever.

Say you have a tcldict `t` containing a dictionary of elements, one of which, `a`, contains a dictionary of elements, one of which, `c`, contains a dictionary of elements, `d`.

If you want a dictionary consisting of everything below `c`, you might do

```
x = t[['a', 'b', 'c']]
```

...or...

```
x = t.get(['a','b','c'], to=tohil.tclobj)
```

Likewise you can compose more complicated dictionaries by attaching a dictionary to a point within another dictionary, simply by assigning a tclobj or tcldict that itself contains a dictionary.

5.8. iterators

You can iterate over a tcldict with normal Python semantics.

For example, something like:

```
>>> t = tohil.tcldict("a 1 b 2 c 3 d 4 e 5 f 6")
>>> for i in t:
...     print(i)
```

If you pass a `to=` conversion to `iter`, the iterator returns tuples comprising the key and the value as well, with the value converted to the `to=` conversion.

```
for key, value in t.iter(to=int):
    print(f"key {key} value {value}")
```

6. TclProcs

TclProcs brings the integration of Tcl functions into Python to a new level of transparency, simplicity and versatility. Using TclProcs, Most Tcl procs now look and behave just like Python functions in most cases.

Using Tcl's introspection capabilities, tohil traverses a hierarchy of Tcl namespaces, identifying all the procs and C-commands in each one, and for the procs, sussing out their arguments and default values, and stashing it so that we can create entrypoints for all of the Tcl procs in Python to invoke Tohil's trampoline function to call the Tcl proc and return the result.

It's awesome!

While tohil can't determine arguments and defaults for Tcl commands that are implemented in C, Tohil still makes entrypoints for them, making them available from Python. Since many Tcl commands and extensions are implemented in C and provide their functionality with a hybrid of Tcl procs and C commands, wrapping the C functions can be important for providing a way for Python to have access to everything such a package provides.

`tohil.import_tcl()` returns a `TclNamespace` object corresponding to whatever namespace you point it at ("`::`" is a good one), and all of the procs and commands found in that namespace are defined as methods of the `TclNamespace` object, and can be executed as such methods. It's very natural and pythonic.

This means you can do stuff like:

```
import tohil
k = tohil.import_namespace("::")
```

...and then invoke top level procs like `k.intersect3()`. And you can chain namespaces.

```
>>> tohil.package_require("Tclx")
>>> k = tohil.import_tcl()
list1 = ["a", "b", "c", "d", "e", "f"]
list2 = ["d", "e", "f", "g", "h", "i"]
a_only, in_both, b_only = k.intersect3(list1, list2, to=tuple)
```

And the subordinate namespaces are created in there too, and they're chainable too.

```
k.clock('format', k.clock("seconds", to=seconds), "-format", "%D %T", "-gmt", 1)
```

7. Shadow Dictionaries

Shadow Dictionaries, aka ShadowDicts, create a Python dict-like object that shadows a Tcl array.

Tcl arrays are basically the Tcl equivalent of Python's dicts, by the way.

Let's assume we have an array `x` in Tcl that we want to shadow as a dictionary `x` in Python, we would write `x = tohil.ShadowDict("x")`.

If you just specify a variable name without any namespace qualifiers, the array references the current Tcl execution frame, like if a Tcl proc had called Python and in our Python we did the `x` equals thing for a shadow dict then the `x` array would exist in the proc's frame. In other words, the array is local to the caller on the Tcl side.

If we're invoking it not from Tcl code called from Python, just from Python or the top level of Python or whatever, then `x` is in the global ("::") namespace. You can always provide namespace qualifiers to identify the global or some subordinate namespace, like `::cryptolib::x`

Once created, shadowdict elements can be gotten as a string using `str()` or `print()`, etc.

Elements can be read from the Python side using dictionary notation, for example `x['d']`, set in a standard way (`x['e'] = '5'`), and deleted in a standard way using `del` (`del x['e']`). Also you can iterate on the keys as with dicts.

Changes made from the Python side occur on the Tcl side, and all accesses, traversals, etc, are made using the actual Tcl array. In other words, ShadowDicts never cache values from the Tcl array on the Python side.

In the example below we set up a Tcl array, create a ShadowDict of it in Python, get a string representation of the dict, read from the dict, insert into it, delete from it, and demonstrate that the changes we made are present on the Tcl side. Finally, it iterates over the shadow dict, showing the same keys from Python that Tcl was shown to have.

```
>>> tohil.eval("array set x [list a 1 b 2 c 3 d 4]")
<tohil.tclobj: ''>
>>> x = tohil.ShadowDict("x", to=int)
>>> x
{'d': '4', 'e': '5', 'a': '1', 'b': '2', 'c': '3'}
>>> x['d']
4
>>> x['e'] = '5'
>>> x['e']
5
>>> del x['d']
>>> tohil.eval("parray x")
x(a) = 1
x(b) = 2
x(c) = 3
x(e) = 5
<tohil.tclobj: ''>
>>> for i in x:
...     print(i)
...
a
b
c
e
```

ShadowDict support many of the capabilities of regular python dicts. For example, `len(x)` will return the length of the shadow dict i.e. the size of the shadowed Tcl array.

`x.keys()` return the keys, `x.values()` returns the values, and `x.items()` returns the keys and items as a list of two-element tuples. However, unlike regular Python dicts, they are not mutable, i.e. if you have captured a reference to `x.keys()` the contents of `x.keys()` does not change when the corresponding dict is changed.

`x.get(key)` will return the element of the array indexed by key if it exists, else it will raise a `KeyError` exception. However if a named parameter, `default`, is specified with a value, in the event key is not found in `x`, the default value will be returned instead.

Finally the `to=` named parameter can be used to specify a Python return type such as `list`, `set`, `dict`, `int`, `float`, `str`, `tohil.tclobj`, `tohil.tcldict`, etc.

`x.pop(key)`, if `key` is in the shadow dictionary, removes it and returns it. A default value can be specified as an optional second argument. If a default is not specified and the key is not in the dictionary, a `KeyError` exception is raised. As with so many other functions, the `to=` named parameter can be specified to state what data type you want the data returned to Python as.

Tohil Reference

This library reference manual describes the library that is Tohil.

Tohil Introduction

This reference describes Tohil, the Python module and Tcl package that are two great tastes that taste great together.

Tohil provides ways to exchange data and execute code between the Python and Tcl interpreters.

Python executes Tcl code using *tohil.eval* and *tohil.call*, and some minor variants.

Tcl executes Python code using *tohil::eval*, *tohil::exec*, and some variants.

Tohil's *tclobj* and *tcldict* Python type objects provide Python access to Tcl's native objects. Tclobjs and tcldicts behave in a very Python-like way. For instance, tclobjs can be used as numbers in calculations (if they contain valid numbers), treated as lists and accessed using familiar Python list syntax and methods, be automatically constructed by ingesting familiar Python data types, including recursive ingestion of lists, dicts, tuples, sets, etc.

Meanwhile, uncaught Tcl errors resulting from Tcl code invoked from Python are propagated back through Python as *TclError* exceptions, while uncaught exceptions raised from Python code invoked from Tcl are propagated back through Tcl as a Tcl error, the traceback interspersing Tcl and Python as the error/exception unwinds.

Tohil Python Functions

Tohil has a number of functions and data types that it provides when the tohil package has been imported.

tohil.alias(*name*, *callback*)

Identical to `register_callback` and under consideration to replace it as the command used to alias Tcl commands to Python commands, although backwards compatibility would be maintained.

tohil.call(** args*[, *to=type*])

Invoke a Tcl command while specifying each argument explicitly, and returns the result.

Using `tohil.call`, even if some arguments contain Tcl metacharacters such as dollar sign, backslash, and square brackets, Tcl will not evaluate them.

Zero or more arguments can be specified. If one or more arguments are specified, the first argument is the command name (which could be the name of a proc or a Tcl C function or whatever), and whatever additional positional arguments are passed as arguments to the command.

If no arguments are specified, that's legal for Tcl. The Tcl interpreter will evaluate an empty string, and return an empty result.

The optional *to=* named parameter can specify a Python data type to return, such as *str*, *int*, *float*, *bool*, *list*, *set*, *dict*, *tuple*, *tohil.tclobj*, *tohil.tclDict*, or a function that takes one argument and returns a result.

If the evaluation results in a Tcl error and the error is not caught by inline Tcl code using Tcl's *try* or *catch*, that is to say if an uncaught Tcl error is received by Tohil from the attempt, Tohil uses information about the Tcl error to create, populate and raise a `TclError` exception to Python.

tohil.convert(*python_object*[, *to=type*])

Convert some Python object into a Tcl object and then convert back to a Python object, a `tohil.tclobj` by default, but it can be converted to any optional *to=* destination type or be passed through a *to=* function.

tohil.eval(*[tcl_code=]code*[, *to=type*])

Given a string of valid Tcl code, including at the caller's discretion multiple statements separated by semicolons, or multiline blocks, evaluate *tcl_code* using the Tcl interpreter, and return its result, by default as a *tohil.tclobj* data type.

As with *tohil.call*, above, if the evaluation results in an uncaught Tcl error, Tohil will construct and raise a `TclError` exception to Python.

tohil.exists(*[var=]varString*)

Returns True if the variable named by *varString* exists, or False if it doesn't.

varString can be an element of a Tcl array by using Tcl array notation, for example 'airports(KHOU)', and `tohil.exists` will return based on the existence of the specified element in the specified array.

tohil.expr(*[expression=]exprString*[, *to=type*])

Evaluate *exprString* as a Tcl expression, and returns the result.

The optional *to=* named parameter can be supplied to specify one of the supported Python data types or functions.

tohil.getvar(*[var=]varString*, *to=tohil.tclobj*[, *default=defVal*])

Get a Tcl variable or array element and return it to the caller.

The variable is accessed from the current Tcl context, which may be global.

The name of the variable or array element is in *varString*.

varString can include namespace qualifiers to ensure a reference is global or to explicitly access a variable within a specific namespace.

The optional *to=* named parameter can be supplied to specify one of the supported Python data types or functions.

An optional default value can be specified using the *default=* named parameter. If a default value is specified and the specified variable or array element doesn't exist in the Tcl interpreter, the default value will be returned instead. *default=None* is a valid default value and is distinct from not providing a default value.

If the variable doesn't exist and a default value was not provided, Tohil will raise a Python `NameError` exception.

Note that default values are coerced to the *to=* data type, a `tohil.tclobj` by default.

tcl = tohil.import_tcl()

Using Tcl's introspection capabilities, traverse all Tcl namespaces, and identify all procs and C commands in each one.

Create a hierarchy of `TclNamespace` objects returning the top-level namespace object.

For each proc, suss out its arguments and default values, if any, and attach, to each namespace, entrypoints for each proc and C command so that calling the Tcl procs looks very much like calling any Python function.

tohil.incr(*[var=]varName*[, *[incr=]increment*])

Take a Tcl variable name or array element as specified by the *varName* string, and attempt to increment it.

The optional increment amount can be specified positionally or using the *incr=* keyword. Its value is **1** by default. The increment amount can be negative.

If the variable doesn't exist, it is created and set to the increment amount.

If the contents of the variable preclude it being used as an integer, Tohil will raise a Python `TypeError` exception.

tohil.interact()

Run the Tcl interactive command loop on stdin, which hopefully is a terminal, until the user sends EOF, at which point they'll be returned to the Python command line, or whatever the Python code that called *tohil.interact()* does next.

tohil.package_require(packageName[, [version=]versionID])

Load the specified package. A specific package version can be specified, either positionally or by name using the *version=* parameter.

This is a shortcut for `tohil.eval(f"package require {packageName} {versionID}")` or `tohi..call("package", "require", packageName, versionID)`.

tohil.register_callback(name, callback)

Create a Tcl command with the given name linked to the given Python callable. When the Tcl-side command is invoked, tohil will directly invoke the corresponding Python function, passing along any arguments. This is useful in cases where the Tcl event loop is being used to execute code asynchronously and you want to handle the callbacks using Python, but in general allows a Python function to be made directly callable as a Tcl function.

tohil.result([to=type])

Return the Tcl interpreter result object.

The Tcl interpreter has a "result object." It contains the result of the last thing the interpreter did.

It's not something you would likely normally need to access, because you would have gotten the result by doing something like `set myResult [myFunction myArg1 myArg2]`.

Nonetheless we make it available because it's been useful for the Tohil devs to be able to see what's in there.

tohil.run()

Perform *tohil.exec*, but redirect stdout emitted while python is running it into a string and return the string to *tohil.run*'s caller after the *exec* has finished.

Python users are often surprised that *exec* doesn't return anything.

tohil.setvar([var=]varName[, [value=]value)

Set a variable or array element referenced by *varName* to the value specified by *value*.

A few errors are possible, such as trying to set an array element of a scalar variable or set a scalar variable that is actually an array. Tohil raises these as a Python `RuntimeError` exception.

tohil.source(fileName)

Take the contents of the file specified by *fileName* and evaluate it using the Tcl interpreter. The return value is the value of the last command executed in the script.

This is the equivalent of `tohil.call("source", fileName)`.

tohil.subst(substString)

Perform Tcl backslash, command and variable substitutions, and return the result of doing that without evaluating it.

This is handy for generating some kind of string while substituting parts of it with embedded `$`-substitutions of Tcl variables and evaluation of Tcl code enclosed in square brackets.

See also the Tcl "subst" manual page.

tohil.tclvar([var=]varName)

Create a `tclobj` object that shadows a Tcl variable or array element.

Any accesses of the resulting `tclobj` from Python will always begin with a (noncopying) access of the Tcl variable or array element's contents, and any writing of the variable from Python (by doing things with the `tclobj` such as invoking methods on them, using Python list notation to update `tclobj` list elements, etc.) will likewise store the value into the corresponding variable or array element in the Tcl interpreter.

tohil.unset(*args)

tohil.unset is used to unset variables, array elements, and even entire arrays in the Tcl interpreter.

Zero or more arguments specify names to unset.

Unsetting an array element uses subscript notation, for example `x(e)`.

Unsetting an array by name without a subscript will unset the entire array.

It is not an error to attempt to unset variables, arrays and array elements that don't exist.

`tohil.tcl_stdout_to_python()`

Tcl normally uses its own I/O system to read and write data.

As *tohil.rivet()* can be used from Python to redirect Python's writing to standard output to go through Tcl's I/O subsystem (and, hence, to Rivet), *tohil.tcl_stdout_to_python* does the opposite, configuring the Tcl interpreter to redirect its standard output, *stdout*, away from Tcl's I/O subsystem and instead send whatever is written through Python's.

If using [Jupyter Notebook](#), invoking `tohil.tcl_stdout_to_python()` will cause any Tcl output written to standard output to appear in the notebook rather than in the log file or stdout of the command running Jupyter notebook.

Tohil Tcl Functions

Once a package `require Tohil` has been performed from Tcl interpreter, the following commands are available:

tohil::call [-kwlist list] [-nonevalue word] [obj.]function [arg...]

tohil::call provides a way to invoke a Python function from Tcl, with zero or more positional parameters and zero or more named parameters, without having to pass the parameters through Python's *eval* or *exec* and running the risk that Python metacharacters appearing in the data will cause quoting problems, unintentional code execution, etc.

Whatever the Python function returns is returned to Tcl.

If *-kwlist list* is specified, *list* contains key-value pairs that will be passed to the function as named parameters.

When you use *tohil::call*, Tohil converts all of your arguments to Python Unicode, unless an argument is comprised of a special sentinel (normally *tohil::NONE*, or the argument to the *-nonevalue* option), in which case the Python "None" data type is substituted in place of that argument.

If *-nonevalue word* is specified, then this overrides the default sentinel string.

tohil::eval evalString

evalString contains a valid Python expression. Tohil evaluates the string using the Python interpreter and returns to Tcl whatever Python returned.

If an exception is thrown and not caught by any Python code before getting back to Tohil, Tohil traps the exception, converts it into a Tcl error, and returns that error to the caller.

tohil::exec

tohil::exec evaluates the code passed to it, similarly to Python's *exec* function. Nothing is returned.

If the Python code prints anything, it goes to stdout using Python's I/O subsystem. However you can easily redirect Python's output to go to a string, or whatever, in the normal Python manner. *tohil::run*, in fact, provides a way to do this.

To make it easier to comply with Python indentation rules, if the first nonblank line starts with whitespace, *exec* will un-indent the code block so the first line is not indented at all and following lines are undented to match.

tohil::import module

Import the specified module into the globals of the Python interpreter.

The name of the module may be of the form *module.submodule*.

You can do the same thing using *exec* and, currently, exercise more control, for example *tohil::exec "from io import StringIO"*. However, this reads cleanly and is often enough.

tohil::interact

We run *tohil::interact* from the Tcl command prompt to enter the Python interactive loop. When we're done, we send end of file (^D) to end the Python loop and return to the Tcl one.

tohil::run

tohil::run evaluates the code passed to it as if with Python's *exec*, but unlike *tohil::exec*, anything emitted by the Python code to Python's stdout (print, etc) is captured by *tohil::run* and returned to the caller.

tohil::redirect_stdout_to_python

Redirects Tcl's standard output to be sent through Python's I/O subsystem.

Works by pushing a custom Tcl channel handler onto Tcl's stdout channel. The handler passes everything written to Tcl's stdout to Python using Python's *sys.stdout.write*.

This allows, among other things, Tcl output to show up in Jupyter Notebook.

tohil_rivet()

tohil_rivet redirects data written from Python to standard output to be delivered through Tcl's standard output instead.

When Tcl is being executed from within the [Apache Rivet](#) webserver module, the output of Python code invoked from Tcl using Tohil will be written into webpage Apache is constructing.

Tohil Types

The following sections describe the types that importing tohil makes available to the Python interpreter.

The principal tohil types are *tclobj* and *tcldict*. There are a few additional types for iterators and exceptions.

Tclobjs and tcldicts are mutable. As with native Python types, methods that add, subtract, or rearrange their members in place, don't return a specific item, returning *None* rather than the collection instance itself.

Some operations are supported by both object types; in particular, they can be compared for equality, tested for truth value, and converted to a string with the `str()` function, while with the `repr()` produces a perhaps somewhat developer-friendly string representation of the object. Tclobjs can be used freely as float or integer values in numeric calculations (when the contents of the tclobj are numeric), including as a source or target of in-place arithmetic.

Tclobjs and tcldicts are very flexible in terms of what they can be constructed from. A tclobj can be created as an empty Tcl object, or from a Python *None* object, a Python boolean, int, float, or string, a Python list, tuple, set, dict, sequence or map, and Unicode/UTF-8 translations should work fine.

Testing Tclobj Truth Values

Any tclobj can be tested for truth value, for use in an if or while condition or as operand of a Boolean operations.

Interpretation of the boolean is according to Tcl rules. These are very close to Python rules, however.

```
>>> import tohil
>>> tohil.tclobj(True)
<tohil.tclobj: '1'>
>>> tohil.tclobj(False)
<tohil.tclobj: '0'>
```

```
>>> bool(tohil.tclobj(1))
True
>>> bool(tohil.tclobj(0))
False
```

```
>>> tohil.tclobj('y')
<tohil.tclobj: 'y'>
>>> bool(tohil.tclobj('y'))
True
>>> bool(tohil.tclobj('t'))
True
>>> bool(tohil.tclobj('f'))
False
>>> bool(tohil.tclobj('F'))
False
>>> bool(tohil.tclobj('not-a-boolean'))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: expected boolean value but got "not-a-boolean"
```

Comparisons

Tclobjs and tcldicts can be compared. When they are asked to be compared, their string representations are compared. If the Tcl objects don't have strings available, they will be marshalled, and this could be high overhead for large and/or very complicated structures.

Using Tohil tclobjs as Numeric Types

Tohil tclobjs can be freely used in Python code where integers or floating point numbers are needed. The underlying Tcl object will be requested using Tcl standard library routines, causing an efficient fetch of the Tcl object's cached representation if the cached representation is of the correct type, or by causing an attempt by the Tcl library to convert the contents of the Tcl object to a python Boolean, integer or float.

Tohil will raise a *TypeError* exception if the Tcl object can't be converted to the Python type that's needed.

Both Python and Tcl support arbitrarily large numbers, and you can freely assign tclobjs from arbitrarily large numbers produced by Python, and vice versa.

Note that Python calculations performed using Tohil's tclobjs are limited to 64 bits (or whatever width a C language "long long" is on the machine tohil was compiled for.) While this should be fine in the overwhelming majority of cases, if you are manipulating numbers that are wider than 64 bits (i.e. less than -9,223,372,036,854,775,808 or greater than 9,223,372,036,854,775,807), you will need to move them from tclobjs to native Python ints, first, by invoking `int()` on the tclobjs of interest.

Bitwise Operations on Tohil Types

Tohil `tclobj` objects can be freely used as a source for boolean operations and shift counts. Bitwise and, or, exclusive or, left and right shift, invert, and absolute value are supported.

Attempting bitwise operations on a `tclobj` that isn't or can't be converted into an integer will fail with a `TypeError` exception raised.

tclobjs as lists

`Tclobjs` whose internal contents are valid Tcl lists can be largely treated as Python lists.

`Tclobjs-as-lists` can be created from Python based on strings, lists, tuples, sets, even dicts. It's pretty cool.

The common sequence operations of `in` and `not in` work fine, while the notation `s[i]` returns the *i*th item of `tclobj` `s`.

Slices are supported, for example `s[i:j]` returns a slice of `s` from *i* to *j* while `s[i:j:k]` yields a slice of `s` from *i* to *j* with step *k*.

`len(s)` returns the length of `s`'s list, while `min(s)` returns the smallest item and `max(s)` the largest. Beware these'll be treated like strings even if they're numbers.

`Tclobjs` are mutable; you can assign an element with `s[i] = x`, append an element with `s.append(x)`, extend `s` with the contents of a Python list, set, tuple, int, float, etc, or another `tclobj`, with `s.extend(x)`.

Because `tclobjs` are mutable, they cannot be directly used as a key in a dictionary, or a value in a set. If you need to use one as a key, wrap it with `str()` or something.

You can clear a `tclobj` or `tcldict` using `s.clear()`, and pop items from the list using `s.pop([i])`.

list.append(x)

Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

list.extend(iterable)

Extend the list by appending all the items from the iterable. Equivalent to `a[len(a):] = iterable`.

list.insert(i, x)

Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

list.remove(x)

Remove the first item from the list whose value is equal to `x`. It raises a `ValueError` if there is no such item.

list.pop([i])

Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

list.clear()

Remove all items from the list. Equivalent to `del a[:]`.

list.index(x[, start[, end]])

Return zero-based index in the list of the first item whose value is equal to `x`. Raises a `ValueError` if there is no such item.

The optional arguments *start* and *end* are interpreted as in the slice notation and are used to limit the search to a particular subsequence of the list. The returned index is computed relative to the beginning of the full sequence rather than the *start* argument.

Some standard Python list methods are not implemented, such as `count`, `reverse`, `sort`, and `copy`.

An example that uses most of the list methods:

```
>>> fruits = tohil.tclobj(['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana'])
>>> fruits
<tohil.tclobj: 'orange apple pear banana kiwi apple banana'>
>>> len(fruits)
7
>>> fruits.append('watermelon')
>>> fruits
<tohil.tclobj: 'orange apple pear banana kiwi apple banana watermelon'>
>>> fruits.insert(1, 'cantaloupe')
>>> fruits
<tohil.tclobj: 'orange cantaloupe apple pear banana kiwi apple banana watermelon'>
>>> fruits.pop()
'watermelon'
>>> fruits.pop(5)
'kiwi'
```

Mapping Types — *tcldict*

Tcldicts are a Python type that manages a Tcl object of a dictionary structure. Most things you can do with a Python dict you can do with a *tcldict*.

However, unlike dicts, *tcldicts* are recursive. From Python, if a key is specified as a Python list, the Tcl dictionary is managed as a hierarchy of dictionaries.

Tcldicts can be created by the *tcldict* constructor.

class *tcldict*(*val*[, *kwargs*])

Return a new *tcldict* initialized from an optional positional argument and a possibly empty set of keyword arguments.

Tcldicts can be created by passing a Python *list*, *dict*, *tuple*, or *set*, a Tcl *list*, a *tclobj* or *tcldict* object, or create one aliased to a variable in the Tcl interpreter using *tohil.tcldictvar*.

If no positional argument is given, an empty *tcldict* is created. If a positional argument is given and it is a mapping object, a dictionary is created with the same key-value pairs as the mapping object. Otherwise, the positional argument must be an iterable object. Each item in the iterable must itself be an iterable with exactly two objects. The first object of each item becomes a key in the new dictionary, and the second object the corresponding value. If a key occurs more than once, the last value for that key becomes the corresponding value in the new dictionary.

Keywords can be *default*, *to*, and/or *var*. Specifying a default using the keyword is the same as doing it using a positional parameter.

The *to* keyword specifies a default type conversion to be applied when retrieving an item from the dict. To-types can be *str*, *bool*, *int*, *float*, *list*, *set*, *dict*, *tuple*, *tohil.tclobj* or *tohil.tcldict*.

These are the operations that dictionaries support (and therefore, custom mapping types should support too):

***list*(*d*)**

Return a list of all the keys used in the *tcldict* *d*.

***len*(*d*)**

Return the number of items in the *tcldict* *d*.

***d*[*key*]**

Return the item of *d* with key *key*. Raises a *KeyError* if *key* is not in the map.

The *__missing__*() method supported by native Python dicts is not supported by *tohil* *tcldicts*.

d*[*key*] = *value

Set *d*[*key*] to *value*.

***del d*[*key*]**

Remove *d*[*key*] from *d*. Note that while native Python dicts raise a *KeyError* if *key* is not in the map, it is not an error to attempt to delete a key from a *tohil* dict.

key* in *d

Return *True* if *d* has a key *key*, else *False*.

key* not in *d

Equivalent to *not key in d*.

***iter*(*d*)**

Return an iterator over the keys of the dictionary. This is a shortcut for *iter(d.keys())*.

***clear*()**

Remove all items from the dictionary.

***get*(*key*[, *default*])**

Return the value for *key* if *key* is in the dictionary, else *default*. If *default* is not given, it defaults to *None*, so that this method never raises a *KeyError*.

***items*()**

Return a new view of the *tcldict*'s items ((*key*, *value*) pairs). Note that unlike native Python dict items, *tcldict* items are not mutable. You probably didn't even know that dict items are mutable. See the [documentation of view objects](#).

***keys*()**

Return a new view of the `tcldict`'s keys. As with items above, if you keep a reference to keys the keys doesn't change if the `tcldict` does. For more on keys in general, see the [documentation of view objects](#).

pop(*key*[, *default*])

If *key* is in the `tcldict`, remove it and return its value, else return *default*. If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

update([*other*])

Update the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.

`update()` accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

Note: Not implemented yet unless it has been and someone didn't update the docs.

values()

Return a new view of the `tcldict`'s values. Same notes apply. See the [documentation of view objects](#).

Dictionaries compare equal if and only if they are the exact same Tcl object or their Tcl string representations are identical.

Order comparisons ('<', '<=', '>=', '>') can be performed.

Please note that unlike modern Python dicts, `TclDicts` do **not** preserve insertion order. `TclDicts` are traversed in hash order, which you can consider to effectively be random. Sorry not sorry, not my fault.

Dictionary view objects

The objects returned by `tcldict.keys()`, `tcldict.values()` and `tcldict.items()` are fake *view objects*. Unlike native Python dicts, they do not provide a dynamic view on the `tcldict`'s entries, which means that when the `tcldict` changes, the view does **not** reflect these changes.

Dictionary views can be iterated over to yield their respective data, and support membership tests.

Tohil Exceptions

If Tcl code invoked from Python using Tohil gets a Tcl traceback, and no Tcl code traps the error, Tohil will receive the error and turn around and throw a `TclError` exception.

`TclError` is an exception class that is, as Python requires, a class derived from the `BaseException` class.

In a try statement with an except clause that mentions `TclError`, that clause will handle the Tcl exception.

What's nice about the `TclError` class is that it is populated by Tohil with useful information that Tohil gleaned from the Tcl interpreter, such as the interpreter result, traceback, Tcl error code, code level, and in some cases the file and line number.

Likewise, uncaught exceptions in the Python interpreter resulting from code invoked from Tcl using Tohil will propagate a TCL error including a stack trace of the Python code that was executing. As the exception continues up the stack, the Tcl stack trace will be appended to it.

The Tcl error code is set to a Tcl list comprising "PYTHON", the class name of the exception, and the base error message. This is experimental but likely to continue. We would like to add the class arguments, though.

Such Python errors may be caught (as per Tcl stack traces) with Tcl's `catch` or `try`, the same as any other TCL error.

```
>>> try:
...     tohil.eval("no")
... except tohil.TclError as err:
...     mine = err
...
>>> mine
<class 'tohil.TclError' 'invalid command name "no"' ['TCL', 'LOOKUP', 'COMMAND', 'no']>
>>> mine.code
1
>>> mine.errorcode
['TCL', 'LOOKUP', 'COMMAND', 'no']
>>> mine.errorline
1
>>> mine.errorinfo
'invalid command name "no"\n    while executing\n"no"'
>>> mine.errorstack
'INNER no'
>>> mine.level
0
>>> mine.result
'invalid command name "no"'
```

Here is a sample Tcl session catching an uncaught Python exception as a Tcl error:

```
>>> tohil.interact()
$ tclsh
% package require tohil
3.2.0
% catch {tohil::eval "no"} catchResult catchDict
1
% puts $catchResult
name 'no' is not defined
% puts $catchDict
-code 1 -level 0 -errorstack {INNER {invokeStk1 tohil::eval no} UP 1} -errorcode {PYTHON NameError
{name 'no' is not defined}} -errorinfo {name 'no' is not defined
from python code executed by tohil File "tohil", line 1, in <module>
    invoked from within
"tohil::eval "no""} -errorline 1
```


Tohil Tcl Errors

If Python code invoked from Tcl results in a Python exception that is not trapped by any of the Python code, Tohil will trap the exception, translate it into a Tcl error, and propagate the error back through Tcl.

If Tcl is being used interactively and the error isn't caught, it will make its way to an error reported in the interactive session.

Likewise if the Tcl error propagates all the way back to tohil due to Tcl code having been called from Python, the Tcl error will be propagated as a Python exception, see :ref:_tohil-exceptions.

The Tcl stack trace will include the Python stack trace of the Python code that was executing at the time of the exception being raised.

The Tcl error code, often referred to as the Tcl global *errorCode*, but also accessible via a Tcl dict created via an argument to Tcl's *catch* or *try*, is made available to Python through Tohil's *TclError* object.

Tcl's *errorCode* represents additional information about an error in the form of a Tcl list that is intended to be easy to process with programs, unlike, for instance, error messages, which may be localized into different languages and be difficult to interpret reliably using a program.

The first element of the *errorCode* list identifies a general class of errors. For Tcl errors generated by Tohil in response to uncaught Python exceptions, the first element of *errorCode* is set to *PYTHON*.

The next element is the Python class name of the exception, followed by the base error message.

```
>>> tohil.interact()
% tohil::exec missing_function
name 'missing_function' is not defined
while evaluating tohil::exec missing_function
% puts $errorCode
PYTHON NameError {name 'missing_function' is not defined}
% puts $errorInfo
name 'missing_function' is not defined
from python code executed by tohil File "tohil", line 1, in <module>
    invoked from within
    "tohil::exec missing_function"
    ("eval" body line 1)
    invoked from within
    "eval $::tclreadline::LINE"
```

Here is another sample Tcl session catching an uncaught Python exception as a Tcl error, where the *errorCode* is set into the catch options variable:

```
>>> $ tclsh % package require tohil 3.2.0 % catch {tohil::eval "no"} catchResult catchDict 1 % puts $catchResult name 'no'
is not defined % puts [dict get $catchDict -errorCode] PYTHON NameError {name 'no' is not defined}

% puts $catchDict -code 1 -level 0 -errorstack {INNER {invokeStk1 tohil::eval no} UP 1} -errorCode {PYTHON NameError
{name 'no' is not defined}} -errorinfo {name 'no' is not defined from python code executed by tohil File "tohil", line 1, in
<module> invoked from within "tohil::eval "no""} -errorline 1
```

Building and Installing Tohil

Building Tohil on Unix systems such as Linux, MacOS and FreeBSD

Below are instructions for building and installing Tohil from source code on Linux, the Mac, and FreeBSD.

The Tohil developers expect to make Tohil available as pip-installable binary packages for at least Intel/AMD64 and Arm procesors for Linux, the Mac, and FreeBSD in the not too distant future. In the meantime, you will need to build from source.

Building and Installing on Linux

Install Prerequisites

You should backup your machine regularly, and confirm you have made a full backup before proceeding.

First you need to install Python and the Python *pip* installer:

```
sudo apt install python3-dev python3-pip tcl8.6-dev
```

There are a few addition things that are probably nice to have:

```
sudo apt install tcl-doc tcl-tclreadline tclx8.4-dev tclx8.4-doc
sudo apt install tcllib tcllib-critcl
```

To run the test suite, you'll need Python's *hypothesis* module:

```
sudo pip3 install hypothesis
```

...and if you plan to build documentation, sphinx:

```
sudo pip3 install hypothesis
```

Build the Configure Script

Next you build the configure script:

```
autoreconf
```

Then run the configure script. The Python version must be specified.

Run the Configure Script

Run the configure script. The Python version must be specified.

```
./configure --with-python-version=3.7m
```

don't forget the "m" if your stuff has that, which Debian tends to.

Make

```
make
sudo make install
make test
```

There's a README.Linux file in the top-level tohil directory that might have some useful info in it.

Building and Installing on macOS

Install Prerequisites

You should backup your machine regularly, and confirm you have made a full backup before proceeding.

These instructions assume you are using MacPorts, an open source community initiative to provide a way to build and install open source software on a Mac, and handle their dependencies.

Please follow instructions on installing MacPorts at <https://www.macports.org>.

Caveat emptor: Remember in our License we are not responsible if this stuff screws up your computer or fails to work in the expected way for whatever reason.

Make sure you've got Xcode installed (Apple's developer tools; they're free. You can install it from Apple's App Store.)

Then install MacPorts for the version of macOS that you're using/building for.

Bring MacPorts Up to Date

You'll want to update MacPorts config to the latest, using its *selfupdate* feature, and upgrade any installed ports.

```
sudo port selfupdate
```

This next part feels a wee bit dangerous in that it might break some of your ports, but it's cool. It'll update all your installed ports, and their dependencies, to the latest version it can:

```
sudo port upgrade outdated
```

You can use `port outdated` to see what ports need updating.

Install Python

Install python and select it. If you expect `python` to start Python 2 instead of Python 3, don't execute the line below that sets *python* to point to *python39*.

```
sudo port install python39
sudo port select --set python python39
sudo port select --set python3 python39
sudo port install py39-setuptools
```

Install Tcl

The commands below will install Tcl and a number of widely used packages. You can, of course, install all the Tcl stuff you want.

```
sudo port install tcl tcl-tls tcllib
sudo port install tclreadline tclx
sudo port install sqlite3-tcl
```

Install Autoconf

Install autoconf, a little bit older version than the absolute latest, because autoconf 2.71 changed stuff quite a bit and the Tcl Extension Architecture (TEA) autoconf stuff that TOhil uses hasn't yet been updated to work with it, so if you try to use 2.71 to create the *configure* script, it'll fail with a whole bunch of errors.

```
sudo port install autoconf264
```

Run Autoconf

From the tohil top-level directory, run `autoconf264`. Like many Unix command line tools, it will produce no output if everything worked.

Run the Configure Script

It shouldn't be necessary to specify *--exec-prefix*, but it seems to be. We could use some help on this.

```
./configure --prefix=/opt/local --exec-prefix=/opt/local --with-python-version=3.9 --with-tcl=/opt/local/lib
```

...then *make* and *make install*:

Make

```
make
sudo make install
make test
```

There's *README.macOS* and *README.MacPorts* files in the top-level tohil directory that might have some useful info in them.

Building and Installing on FreeBSD

Install Prerequisites

You should backup your machine regularly, and confirm you have made a full backup before proceeding.

These instructions assume you are building FreeBSD ports from source. You can also install ports without building from source by using the pkg package manager. We're only covering doing it using ports at this time.

First you need to install Python and the Python *pip* installer:

```
cd /usr/ports/lang/python39
sudo make install

cd /usr/ports/devel/py-pip
sudo make install
```

Next install Tcl if you haven't already:

```
cd /usr/ports/lang/tcl86
sudo make install

cd /usr/ports/lang/tclX
sudo make install

cd /usr/ports/devel/tcllib
sudo make install

cd /usr/ports/devel/tcllibc
sudo make install
```

There are a few addition things that are probably nice to have such as ports *devel/tclreadline*, *databases/tcl-sqlite3*, *devel/tclbsd*, *devel/tcllauncher*, and *devel/tcltls*.

To run the test suite, you'll need Python's *hypothesis* module:

```
sudo pip3 install hypothesis
```

...and if you plan to build documentation, sphinx:

```
sudo pip3 install sphinx
```

Build the Configure Script

Next you build the configure script:

```
autoreconf
```

You might need to install *devel/autconf*.

Run the Configure Script

Run the configure script. The Python version must be specified.

This specification is a little trickier than usual because the approach the FreeBSD developers have taken toward packaging is a little more particular about where stuff is supposed to go.

This has advantages, though. For instance you can have multiple versions of Tcl installed and multiple versions of Python 3 installed at the same time.

```
./configure --with-tcl=/usr/local/lib/tcl8.6 --mandir=/usr/local/man --with-python-version=3.7m
```

In the above, we tell configure where to find the Tcl library because it's in a slightly nonstandard place. We tell it the Python version; Tohil's configure script will use python3.7m-config or whatever to find the Python library and includes.

Don't forget the "m" in the version name if your stuff has that.

Make

```
make
sudo make install
make test
```

There's a README.FreeBSD file in the top-level tohil directory that might have some useful info in it.

Dealing with Bugs

Python is a mature programming language which has established a reputation for stability, but Tohil is new and somewhat immature.

In order to become highly reliable, the Tohil developers would like to know of any deficiencies you find in it.

It can be sometimes faster to fix bugs yourself and contribute patches to Tohil. This can be done through <https://github.com/flightaware/tohil> .

Documentation bugs

If you find a bug in this documentation or would like to propose an improvement, please submit a bug report on aforelinked github site. If you have a suggestion on how to fix it, include that as well.

Using the Tohil issue tracker

Please use Tohil's issue tracker at <https://github.com/flightaware/tohil/issues> .

Getting started contributing to Tohil yourself

Beyond just reporting bugs that you find, you are also welcome to submit patches to fix them. We will review pull requests and work with you to get yours accepted, or something satisfactory, as long as what you are trying to do is reasonably aligned with how we think tohil should work.

You can always fork it if you want to go off on your own direction.

Tohil Copyright and License

Tohil and this documentation is:

Copyright (c) 2014, Aidan Hobson Sayers All rights reserved.

Copyright (C) 2021 FlightAware LLC All Rights Reserved

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. The name of the author may not be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Python and Tcl, build tooling, etc, are copyrighted by their respective developers.

Install more Python modules

Note: It is possible to install more Python modules using the pip standard way of python

```
<GiD>\scripts\tohil\python -m pip install <module>
```

e.g. to install the KratosMultiphysics module version 9.2:

```
<GiD>\scripts\tohil\python -m pip install KratosMultiphysics-all==9.2
```

Note: Kratos is a framework for building parallel multi-disciplinary simulation: <https://github.com/KratosMultiphysics>

The pip program is located at <GiD>\scripts\tohil\python\scripts\pip.exe

modules will be located at

```
<GiD>\scripts\tohil\python\lib\site-packages
```

Note: probably must run pip in a console opened in Windows 'as administrator' (otherwise Windows doesn't allow to copy the files if GiD is installed in 'Program files' as usual). In Linux do it as 'sudo'

Some Python modules that look interesting for GiD purposes

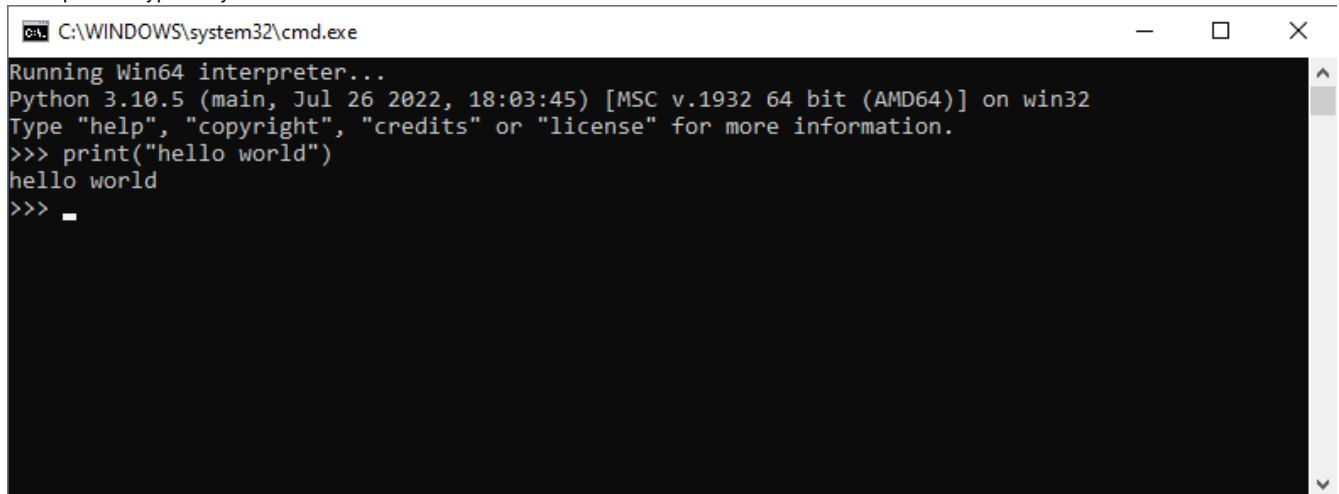
- NURBS-Python (geomdl) <https://github.com/orbingol/NURBS-Python> (pure-scripting use of NURBS)
- pythonOCC <https://dev.opencascade.org/project/pythonocc> (Python wrap of C++ OpenCascade CAD)
- vtk <https://vtk.org> (Python wrap of C++ Vtk scientific visualization library).
- SciPy <https://scipy.org> (fundamental algorithms for scientific computing)
- SymPy <https://docs.sympy.org/latest/modules/geometry/index.html> (library for symbolic mathematics)
- PyMesh <https://pymesh.readthedocs.io/en/latest> (mesh processing library)
- ...

Run Python as external process

It is possible to start Python running

<GiD>\scripts\tohil\python\python.bat

This opens the typical Python console

A screenshot of a Windows command prompt window. The title bar shows the path 'C:\WINDOWS\system32\cmd.exe'. The window content shows the output of running a Python script. It starts with 'Running Win64 interpreter...', followed by the Python version and build information: 'Python 3.10.5 (main, Jul 26 2022, 18:03:45) [MSC v.1932 64 bit (AMD64)] on win32'. It then prompts for help information. The user enters '>>> print("hello world")' and the output 'hello world' is displayed. The prompt '>>>' is followed by a cursor. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

```
C:\WINDOWS\system32\cmd.exe
Running Win64 interpreter...
Python 3.10.5 (main, Jul 26 2022, 18:03:45) [MSC v.1932 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("hello world")
hello world
>>> _
```

This Python process is not related with the gid.exe process (it is not an embedded Python)

It can be used for example to run a GiD problemtype Python-based solver, without need to install an extra Python, the one of GiD could be used, and its relative location is a priori known.

If this process is killed the gid process is not affected, and don't share any memory.

It is possible to import the tohil module in this Python interpreter to call Tcl commands, but will create a new Tcl interpreter with the standard commands. Cannot use here GiD commands like `tohil.call('GiD_Info','mesh','nodes','-array')` or `tcl.GiD_Info('mesh','nodes','-array')`

Run Python inside GiD

It is possible to run a Python script in several ways

- 1- From the lower entry
- 2- Python IDLE shell
- 3- From an user-macro button
- 4- Invoked from a problemtype or plugin

From the lower entry

From the entry that is placed at the bottom of the GiD window, can invoke Tcl code writing `-np-` followed some space and then some tcl code an press <Return>

Note: `-np-` mean 'No-Process' (that the words are not GiD process keywords) and instead the next command is expected to be Tcl code. It is used as a fast way to run a simple procedure or re-define code.

Use copy/paste code after `-np-`

In particular there are some predefined Tcl procs that run python code, like `GiD_Python_Exec` that expects the python code to be evaluated

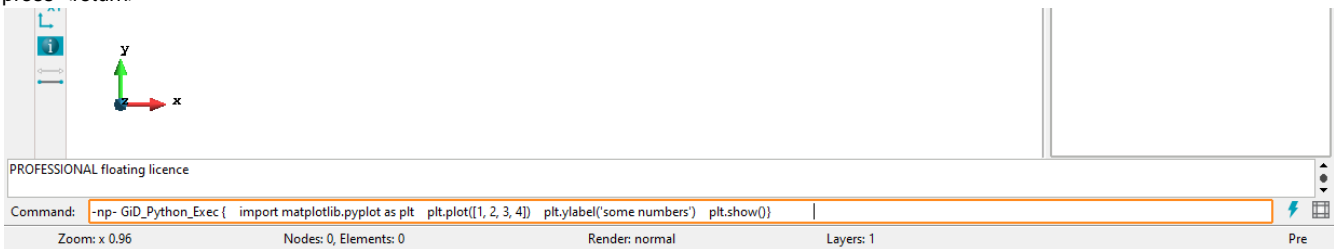
Example: matplotlib graph of a line

This is a simple example of call from Tcl a Python code that uses the matplotlib to show a graph (in new Tk window opened by tkinter)

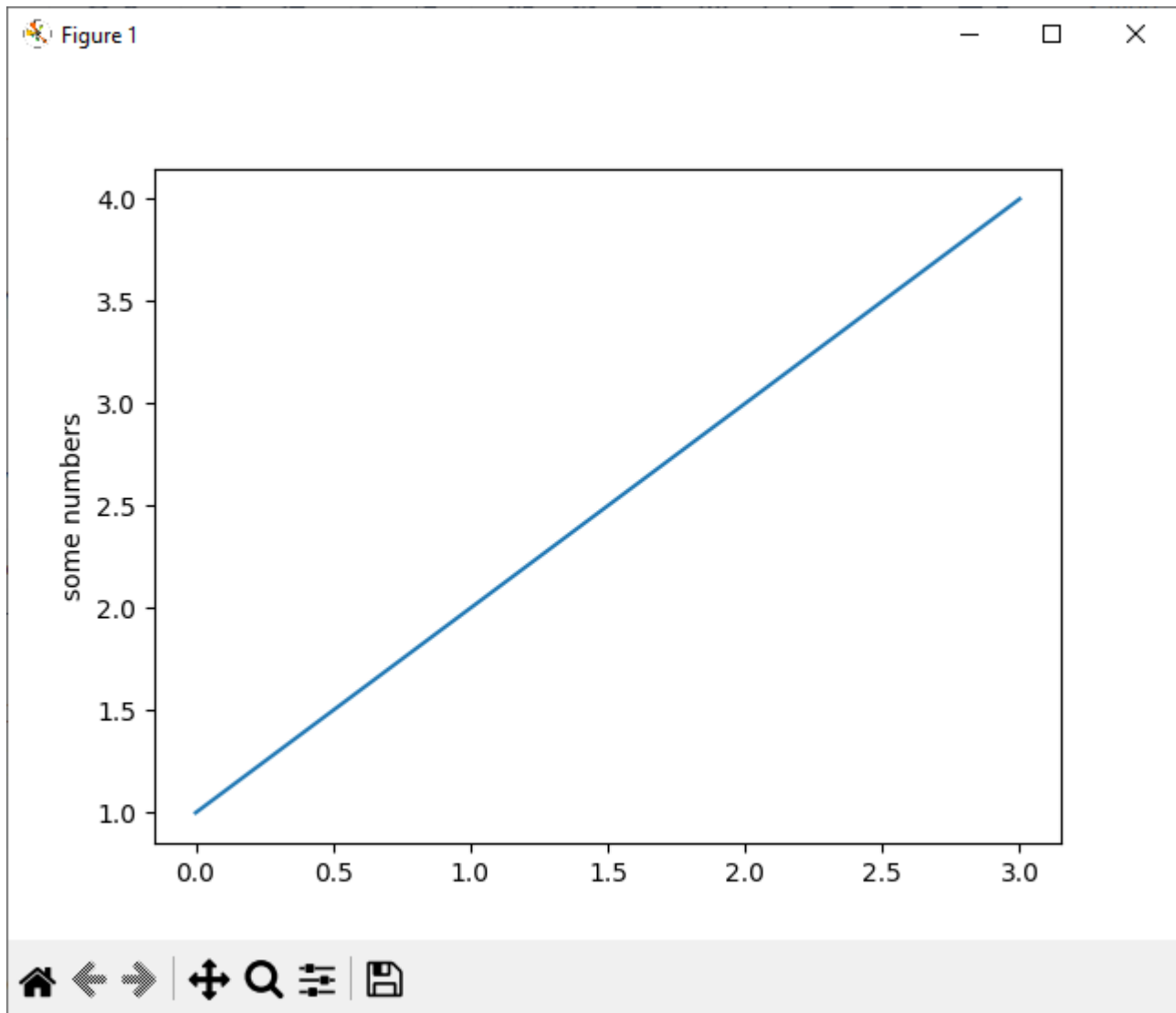
```
GiD_Python_Exec {
  import matplotlib.pyplot as plt
  plt.plot([1, 2, 3, 4])
  plt.ylabel('some numbers')
  plt.show()
}
```

`GiD_Python_Exec` do an implicit package require `tohil` and then `tohil::exec`

Can invoke this Tcl code for example pasting it after `-np-` in the command line (with one or more spaces separating the code pasted), and press <return>



then will appear a window like this



There are other procs similar to `GiD_Python_Exec`, like `GiD_Python_Eval` (for a single instruction and value is returned) or `GiD_Python_Call` (to invoke a function that must be previously defined)

Source a file with the code

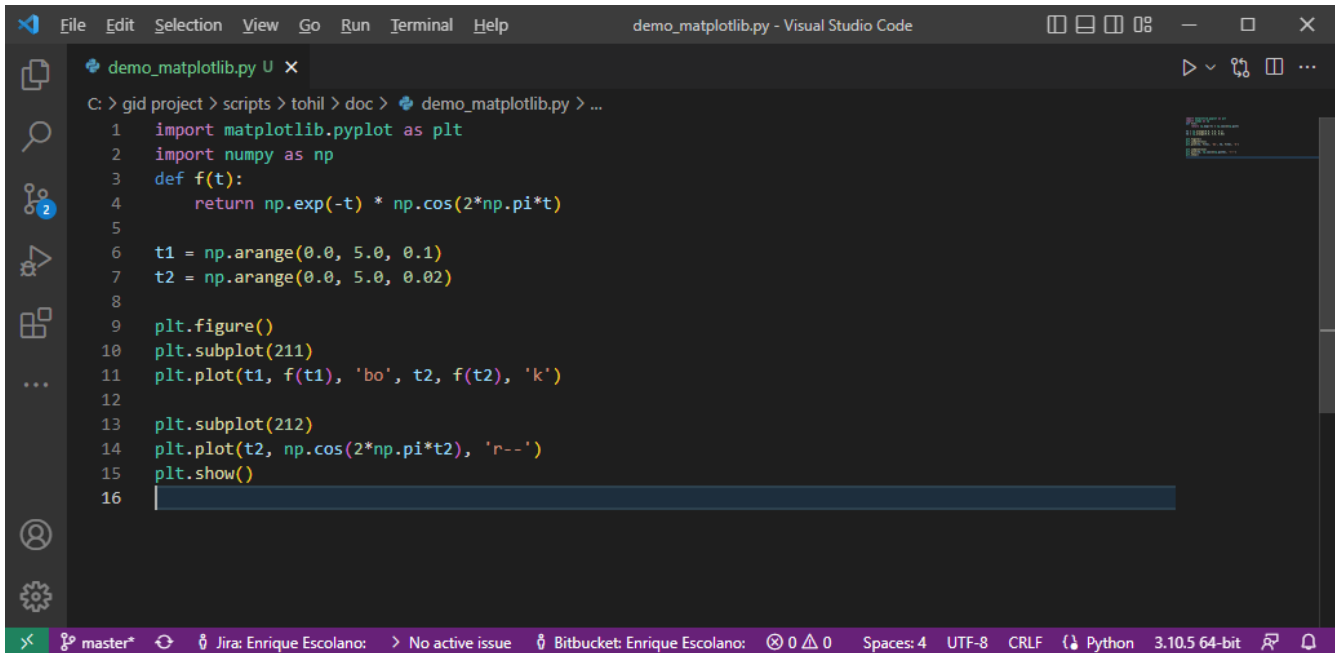
The normal way to write long Python code is write it in a .py file.

Note: Visual Studio Code is our recommended editor, installing the MS Python extension for this language.

`GiD_Python_Source` is a Tcl proc that expects the name of a python file with the code to be sourced

Example: matplotlib graph of two curves

The file `<GiD>/scripts/tohil/doc/demo_matplotlib.py` contain a code like the one of the image



```

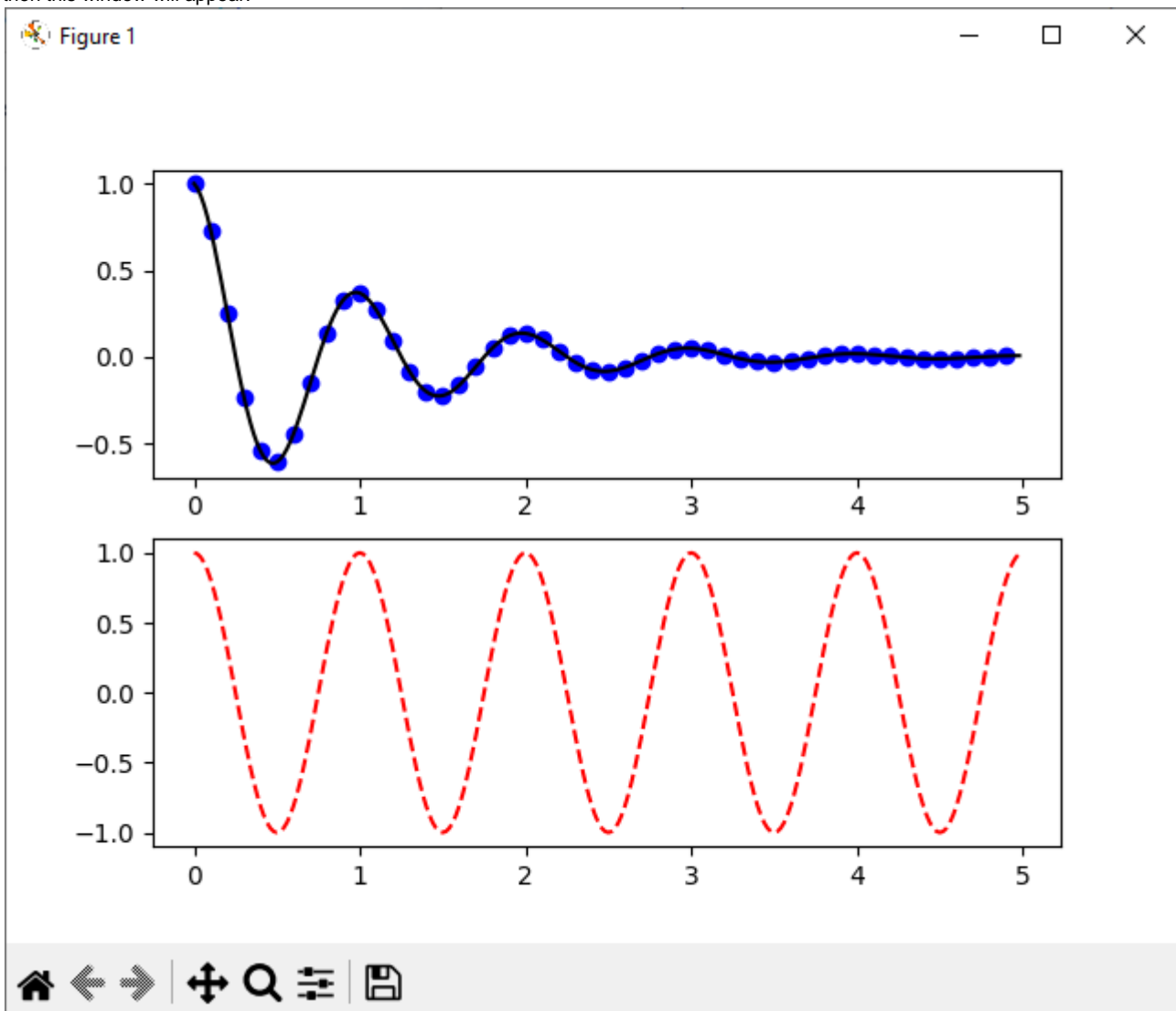
1 import matplotlib.pyplot as plt
2 import numpy as np
3 def f(t):
4     return np.exp(-t) * np.cos(2*np.pi*t)
5
6 t1 = np.arange(0.0, 5.0, 0.1)
7 t2 = np.arange(0.0, 5.0, 0.02)
8
9 plt.figure()
10 plt.subplot(211)
11 plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
12
13 plt.subplot(212)
14 plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
15 plt.show()
16

```

and can source this file writing this (<GiD> must be replaced by the true path)

```
-np- GiD_Python_Source {<GiD>/scripts/tohil/doc/demo_matplotlib.py}
```

then this window will appear:



Python force reload a file

Other Tcl procs related to source Python code are `GiD_Python_Import_File` and `GiD_Python_Import`

Using our Tcl command `GiD_Python_Import` (that do `tohil::import`) will import a module in Python from its tail name without `.py` extension, and must be found based on the path environment variable

`GiD_Python_Import_File` is similar to `GiD_Python_Import` but expects the full path.

But if we are developing and modify the `.py` file doing a new import won't refresh the code in the interpreter.

A trick to do it is to use the Tcl command `GiD_Python_Source`, then the new code of the file is used without need to restart GiD.

In fact it seems that this is similar to use in Python the function `reload` of the `importlib` module

```
import importlib
importlib.reload(module)
```


IDLE shell

It is possible to open an IDLE shell, e.g. to try interactive Python commands, with the Tcl proc `GiD_Python_Open_IDLE_Shell`

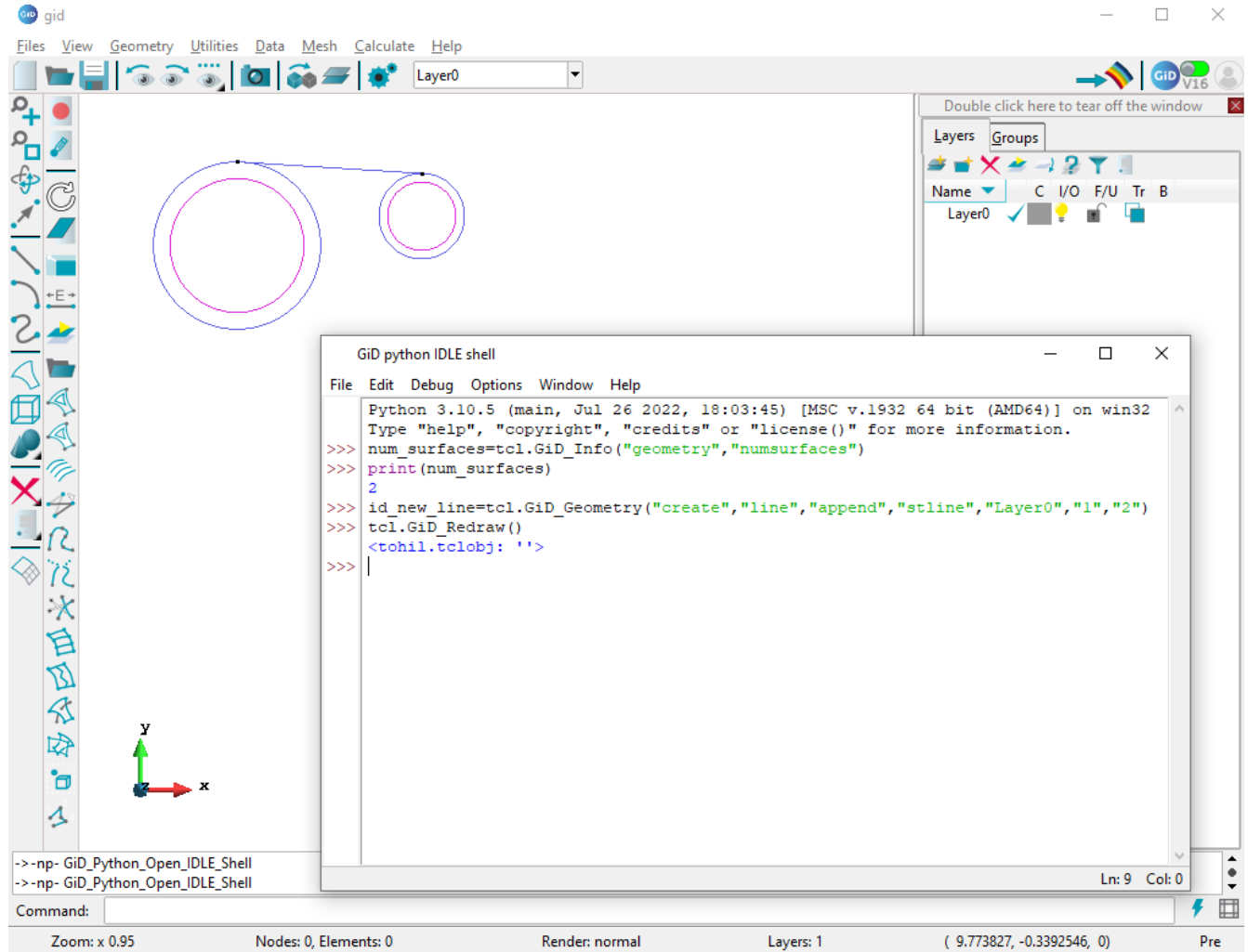
In practice go to GiD menu Utilities->Tools->Console Python...

e.g. the next picture show this IDLE console and the use of a Python command to set a variable with the current number of surfaces asked to GiD

```
num_surfaces=tcl.GiD_Info("geometry","numsurfaces")
```

and it creates in GiD a new straight line in the layer Layer0 from the point id=1 to the point id=2 and store the new line id in a variable

```
id_new_line=tcl.GiD_Geometry("create","line","append","stline","Layer0","1","2")
```



Note that the new line is not immediately shown, until the model is redrawn (can invoke it with `tcl.GiD_Redraw()`)

Note: The use of this console is recommended to interactively try several Python commands.

From an user-macro button

The macros toolbar have buttons that call the Tcl code. Most buttons and procs are predefined, for common use, but each user can add its own buttons (they are stored with its preferences), to do some interesting action.

See [Macros](#)

In particular the Tcl code can invoke Python code with the procs related previously: `GiD_Python_Exec`, `GiD_Python_Call`, `GiD_Python_Source`, ...

From a problemtype or plugin

A problemtype is a collection of files inside <GiD>/problemtypes that customize GiD for a particular kind of simulation. When a problemtype is loaded it can modify the whole GUI: menus, toolbars, etc, and can for example add a menu that invoke the Tcl code that run the Python command of interest

A plugin of GiD is simply a collection of files inside <GiD>/plugins that are loaded when GiD starts. They can modify the GUI, for example adding some menu to invoke some new feature. Like previously the Tcl code can run Python code (e.g. [Real example: meshio GiD plugin](#))

Note: the features added by plugins are available without need to load a problemtype, and are preserved swapping to other problemtype.

Real example: meshio GiD plugin

This plugin uses the features of the meshio Python module to allow GiD import/export the mesh in all formats supported by this module.

The module is written in python, but GiD uses Tcl as main scripting language. The Tohil package is the bridge to allow its use.

The plugin is placed at

<GiD>/plugins/Import/meshio

and like the rest of plugins is automatically loaded from Tcl when GiD starts.

It add to the Files Import/Export menus a new “meshio” item.

to import meshes the Python file gid_meshio.py define for example a function my_meshio_read_mesh

```
import numpy as np
import meshio

def my_meshio_read_mesh(filename):
    #to avoid that numpy truncate the printed representation of its arrays
    np.set_printoptions(threshold=np.inf)
    mesh=meshio.read(filename)
    return [mesh.points,mesh.cells,mesh.cells_dict]
```

and the Tcl file meshio.tcl invoke the import of this file in the Python interpreter to have defined the function, and then call the function to obtain the mesh data independently of the mesh file format read.

```
set filename_python [file join [gid_filesystem::get_folder_standard plugins] Import/meshio/gid_meshio.
py]
GiD_Python_Import_File $filename_python
...
set m [GiD_Python_Call gid_meshio.my_meshio_read_mesh $filename_mesh]
...
```

Here GiD_Python_Import_File is an auxiliary proc (see <GiD>/scripts/gid_python.tcl) that decorate the tohil syntax and basically do

package require tohil

tohil::import \$module_name

And GiD_Python_Call is like an alias of the command tohil::call

then the variable m has the data that define a collection of meshes (element type, coordinates of nodes and element connectivities), and this data is processed at Tcl level to have the desired final data.

this is the code of the proc that create GiD meshes from a file in a format supported by the meshio Python module:

```
proc MeshIo::Init { } {
    variable meshio_num_nodes_per_cell
    variable meshio_gid_element

    #linear and serendipit
    set meshio_num_nodes_per_cell(vertex) 1
    set meshio_num_nodes_per_cell(line) 2
    set meshio_num_nodes_per_cell(triangle) 3
    set meshio_num_nodes_per_cell(quad) 4
    set meshio_num_nodes_per_cell(quad8) 8
    set meshio_num_nodes_per_cell(tetra) 4
    set meshio_num_nodes_per_cell(hexahedron) 8
    set meshio_num_nodes_per_cell(hexahedron20) 20
    set meshio_num_nodes_per_cell(hexahedron24) 24
    set meshio_num_nodes_per_cell(wedge) 6
    set meshio_num_nodes_per_cell(pyramid) 5
    #quadratic
    set meshio_num_nodes_per_cell(line3) 3
    set meshio_num_nodes_per_cell(triangle6) 6
    set meshio_num_nodes_per_cell(quad9) 9
    set meshio_num_nodes_per_cell(tetra10) 10
    set meshio_num_nodes_per_cell(hexahedron27) 27
    set meshio_num_nodes_per_cell(wedge15) 15
    set meshio_num_nodes_per_cell(wedge18) 18
    set meshio_num_nodes_per_cell(pyramid13) 13
    set meshio_num_nodes_per_cell(pyramid14) 14
    #degree 3
    set meshio_num_nodes_per_cell(line4) 4
    set meshio_num_nodes_per_cell(triangle10) 10
```

```

set meshio_num_nodes_per_cell(quad16) 16
set meshio_num_nodes_per_cell(tetra20) 20
set meshio_num_nodes_per_cell(wedge40) 40
set meshio_num_nodes_per_cell(hexahedron64) 64
#degree 4
set meshio_num_nodes_per_cell(line5) 5
set meshio_num_nodes_per_cell(triangle15) 15
set meshio_num_nodes_per_cell(quad25) 25
set meshio_num_nodes_per_cell(tetra35) 35
set meshio_num_nodes_per_cell(wedge75) 75
set meshio_num_nodes_per_cell(hexahedron125) 125
#degree 5
set meshio_num_nodes_per_cell(line6) 6
set meshio_num_nodes_per_cell(triangle21) 21
set meshio_num_nodes_per_cell(quad36) 36
set meshio_num_nodes_per_cell(tetra56) 56
set meshio_num_nodes_per_cell(wedge126) 126
set meshio_num_nodes_per_cell(hexahedron216) 216
#degree 6
set meshio_num_nodes_per_cell(line7) 7
set meshio_num_nodes_per_cell(triangle28) 28
set meshio_num_nodes_per_cell(quad49) 49
set meshio_num_nodes_per_cell(tetra84) 84
set meshio_num_nodes_per_cell(wedge196) 196
set meshio_num_nodes_per_cell(hexahedron343) 343
#degree 7
set meshio_num_nodes_per_cell(line8) 8
set meshio_num_nodes_per_cell(triangle36) 36
set meshio_num_nodes_per_cell(quad64) 64
set meshio_num_nodes_per_cell(tetra120) 120
set meshio_num_nodes_per_cell(wedge288) 288
set meshio_num_nodes_per_cell(hexahedron512) 512
#degree 8
set meshio_num_nodes_per_cell(line9) 9
set meshio_num_nodes_per_cell(triangle45) 45
set meshio_num_nodes_per_cell(quad81) 81
set meshio_num_nodes_per_cell(tetra165) 165
set meshio_num_nodes_per_cell(wedge405) 405
set meshio_num_nodes_per_cell(hexahedron729) 729
#degree 9
set meshio_num_nodes_per_cell(line10) 10
set meshio_num_nodes_per_cell(triangle55) 55
set meshio_num_nodes_per_cell(quad100) 100
set meshio_num_nodes_per_cell(tetra220) 220
set meshio_num_nodes_per_cell(wedge550) 550
set meshio_num_nodes_per_cell(hexahedron1000) 1000
set meshio_num_nodes_per_cell(hexahedron1331) 1331
#degree 10
set meshio_num_nodes_per_cell(line11) 11
set meshio_num_nodes_per_cell(triangle66) 66
set meshio_num_nodes_per_cell(quad121) 121
set meshio_num_nodes_per_cell(tetra286) 286

#linear and serendipit
set meshio_gid_element(vertex) point
set meshio_gid_element(line) line
set meshio_gid_element(triangle) triangle
set meshio_gid_element(quad) quadrilateral
set meshio_gid_element(tetra) tetrahedra
set meshio_gid_element(hexahedron) hexahedra
set meshio_gid_element(wedge) prism
set meshio_gid_element(pyramid) pyramid
#quadratic
set meshio_gid_element(line3) line
set meshio_gid_element(triangle6) triangle
set meshio_gid_element(quad8) quadrilateral
set meshio_gid_element(quad9) quadrilateral
set meshio_gid_element(tetra10) tetrahedra
set meshio_gid_element(hexahedron20) hexahedra
set meshio_gid_element(hexahedron27) hexahedra
set meshio_gid_element(wedge15) prism
set meshio_gid_element(pyramid13) pyramid

...
}
proc MeshIo::ReadPreUnstructuredMesh { filename } {
    set fail 0
    variable meshio_gid_element
    variable meshio_num_nodes_per_cell
    MeshIo::Import_gid_meshio_py ;#to load in python the file gid_meshio.py to define its python
functions before be called
    set m [GiD_Python_Call gid_meshio.my_meshio_read_mesh $filename]
    set nodes_coordinates [lindex [MeshIo::PythonArrayToTclList [lindex $m 0]] 0]
    #e.g.[lindex $m 1] == {<meshio CellBlock, type: triangle, num cells: 156, tags: []>}
    set meshio_element_types_and_connectivities [MeshIo::PythonArrayToTclList [lindex $m 2]]
    set layer [GiD_Layers get to_use]
    set offset_nodes [GiD_Info mesh MaxNumNodes]
    set offset_elements [GiD_Info mesh MaxNumElements]
    set last_element_id $offset_elements

```

```

#better use GiD_MeshPre_Create with same syntax as GiD_MeshPost (and some day could be implemented
to be faster in C++)
set num_nodes [llength $nodes_coordinates]
set node_ids [objarray new_from_to intarray [expr $offset_nodes+1] [expr $offset_nodes+$num_nodes]]
set vertices [objarray new doublearray [expr $num_nodes*3]]
set i 0
foreach node $nodes_coordinates {
    foreach value $node {
        objarray set $vertices $i $value
        incr i
    }
}
foreach {meshio_element_type meshio_connectivities} $meshio_element_types_and_connectivities {
    set element_type ""
    set element_num_nodes 0
    if { [info exists meshio_gid_element($meshio_element_type)] } {
        set element_type $meshio_gid_element($meshio_element_type)
        set element_num_nodes $meshio_num_nodes_per_cell($meshio_element_type)
    } else {
        W "element $meshio_element_type not supported"
        continue
    }
    set elements [lindex $meshio_connectivities 0]
    set num_elements [llength $elements]
    set element_ids [objarray new_from_to intarray [expr $last_element_id+1] [expr
$last_element_id+$num_elements]]
    set element_vertex_indices [objarray new intarray [expr $num_elements*$element_num_nodes]]
    set i 0
    foreach element $elements {
        foreach node_id $element {
            objarray set $element_vertex_indices $i [expr $node_id+$offset_nodes+1]
            incr i
        }
    }
    set zero_based_array 0
    GiD_MeshPre_Create $element_type $element_num_nodes $node_ids $vertices $element_ids
    $element_vertex_indices $zero_based_array $layer
    incr last_element_id $num_elements
}
return $fail
}

```

For the import feature GiD is invoking from its Tcl code functions of Python files and modules like meshio and numpy.

For the export feature it is implemented in two alternative ways:

1. For Tcl-like programmers, with most code in Tcl
2. For Python-like programmers: with most code in Python

Note that only the second approach is bi-directional and will invoke from Python Tcl commands (of the GiD interpreter) importing in Python the tohil module

1. Ask to GiD its current mesh data with Tcl, and process this data with Tcl code to reach the format expected by the Python meshio function and then call a Python function that create a meshio.Mesh and write it to file with the desired format.
2. From Tcl call some Python code that ask GiD mesh information calling Tcl commands and process this data with Python to create a meshio.Mesh and write it to file with the desired format.

The implementation of 1. is something like this

meshio.tcl (big code)

```

proc MeshIo::Init { } {
    variable meshio_element_name
    ...
    set meshio_element_name(point,1) vertex
    set meshio_element_name(line,2) line
    set meshio_element_name(triangle,3) triangle
    set meshio_element_name(quadrilateral,4) quad
    set meshio_element_name(tetrahedra,4) tetra
    set meshio_element_name(hexahedra,8) hexahedron
    set meshio_element_name(prism,6) wedge
    set meshio_element_name(pyramid,5) pyramid
    #quadratic
    set meshio_element_name(line,3) line3
    set meshio_element_name(triangle,6) triangle6
    set meshio_element_name(quadrilateral,8) quad8
    set meshio_element_name(quadrilateral,9) quad9
    set meshio_element_name(tetrahedra,10) tetra10
    set meshio_element_name(hexahedra,20) hexahedron20
    set meshio_element_name(hexahedra,27) hexahedron27
    set meshio_element_name(prism,15) wedge15
    set meshio_element_name(pyramid,13) pyramid13
}

```

```

proc MeshIo::TclObjarrayToPythonArrayPoints { node_xyzs } {
    set points ""
    lassign $node_xyzs xs ys zs
    set num_nodes [objarray length $xs]
    for {set i_node 0} {$i_node<$num_nodes} {incr i_node} {
        set x [objarray get $xs $i_node]
        set y [objarray get $ys $i_node]
        set z [objarray get $zs $i_node]
        append points "\[$x,$y,$z\],"
    }
    return "\[$points\]"
}

proc MeshIo::TclObjarrayToPythonArrayConnectivities { element_num_nodes connectivities } {
    set cells ""
    set num_elements [expr [objarray length $connectivities]/$element_num_nodes]
    set i 0
    for {set i_element 0} {$i_element<$num_elements} {incr i_element} {
        set node_ids [list]
        for {set i_node 0} {$i_node<$element_num_nodes} {incr i_node} {
            lappend node_ids [objarray get $connectivities $i]
            incr i
        }
        append cells "\[[join $node_ids ,]\],"
    }
    return "\[$cells\]"
}

proc MeshIo::WritePreUnstructuredMesh { filename } {
    variable meshio_element_name
    MeshIo::Import_gid_meshio_py ;#to load in python the file gid_meshio.py to define its python
    functions before be called

    # coordinates
    lassign [GiD_Info mesh nodes -array] node_ids node_xyzs

    set max_id_nodes [objarray get $node_ids end]
    set num_nodes [objarray length [lindex $node_xyzs 0]]
    set new_node_ids ""
    set nodes_renumbered_for_meshio 0
    if { $max_id_nodes != $num_nodes } {
        set nodes_renumbered_for_meshio 1
        set new_node_ids [objarray new_from_to intarray 0 [expr $num_nodes-1]]
    }
    set points [MeshIo::TclObjarrayToPythonArrayPoints $node_xyzs]
    set cells "\["
    foreach element_type {linear triangle quadrilateral tetrahedra pyramid prism hexahedra } {
        set elements_data [lindex [GiD_Info mesh elements $element_type -array2] 0]
        if { [llength $elements_data] } {
            lassign $elements_data element_type_ret element_ids connectivities materials
            set num_elements_block [objarray length $element_ids]
            if { $num_elements_block } {
                if { $nodes_renumbered_for_meshio } {
                    objarray renumber $connectivities $node_ids $new_node_ids
                } else {
                    objarray incr $connectivities -1 ;#meshio is zero based
                }
                set some_element_id [objarray get $element_ids 0]
                set element_num_nodes [llength [GiD_Mesh get element $some_element_id connectivities]]
                set meshio_etype $meshio_element_name($element_type,$element_num_nodes)
                append cells "(\\"$meshio_etype\",[MeshIo::TclObjarrayToPythonArrayConnectivities
$element_num_nodes $connectivities]),"
            }
        }
    }
    append cells "\]"
    set result [GiD_Python_Call gid_meshio.my_meshio_write_mesh $points $cells $filename]
    return 0
}

```

gid_meshio.py (small code)

```

import meshio

def my_meshio_write_mesh(points,cells,filename):
    #trick, use ast.literal_eval to convert from string to list representation
    import ast
    points=ast.literal_eval(points)
    cells=ast.literal_eval(cells)
    mesh=meshio.Mesh(points,cells)
    result=mesh.write(filename)
    return result

```

The implementation of 2. is something like this

meshio.tcl (small code)

```
proc MeshIo::WritePreUnstructuredMesh2 { filename } {
    MeshIo::Import_gid_meshio_py ;#to load in python the file gid_meshio.py to define its python
    functions before be called
    set result [GiD_Python_Call gid_meshio.my_meshio_write_mesh2 $filename]
    return 0
}
```

gid_meshio.py (big code)

```
import numpy as np
import tohil
import meshio

#to create functions and variables for all tcl available ones
tcl=tohil.import_tcl()

gid_to_meshio_type = {
    "sphere,1":"vertex",
    "point,1":"vertex",
    "line,2":"line",
    "triangle,3":"triangle",
    "quadrilateral,4":"quad",
    "tetrahedra,4":"tetra",
    "hexahedra,8":"hexahedron",
    "prism,6":"wedge",
    "pyramid,5":"pyramid",
    #quadratic
    "line,3":"line3",
    "triangle,6":"triangle6",
    "quadrilateral,8":"quad8",
    "quadrilateral,9":"quad9",
    "tetrahedra,10":"tetra10",
    "hexahedra,20":"hexahedron20",
    "hexahedra,27":"hexahedron27",
    "prism,15":"wedge15",
    "pyramid,13":"pyramid13",
}

def gid_points_to_meshio_points(node_xyzs):
    xs,ys,zs=node_xyzs
    num_nodes=len(xs)
    points=np.empty((num_nodes,3))
    for i_node in range(num_nodes):
        points[i_node]=(float(xs[i_node]),float(ys[i_node]),float(zs[i_node]))
    #points[:, 0] = xs[:]
    #points[:, 1] = ys[:]
    #points[:, 2] = zs[:]
    return points

def gid_elements_to_meshio_cells(element_num_nodes,connectivities):
    cells=[]
    #operator // is for integer division
    num_elements=len(connectivities)//element_num_nodes
    i=0
    for i_element in range(num_elements):
        node_ids=[]
        for i_node in range(element_num_nodes):
            node_ids.append(connectivities[i])
            i+=1
        cells.append(node_ids)
    return cells

def tohil_obj_array_int_to_numpy(items):
    num_items=len(items)
    numpy_array=np.empty(num_items,np.int64)
    for i in range(num_items):
        numpy_array[i]=int(items[i])
    return numpy_array

def numpy_renumber(connectivities,old_node_ids,new_node_ids):
    fail=0
    length_connectivities=len(connectivities)
    length_old_ids=len(old_node_ids)
    length_new_ids=len(new_node_ids)
    if(length_old_ids==length_new_ids):
        max_old_id=np.max(old_node_ids)
        new_number=np.empty(max_old_id,np.int32)
        for i in range(length_old_ids):
            new_number[old_node_ids[i]]=new_node_ids[i]
```



```

        for i in range(length_connectivities) :
            connectivities[i]=new_number[connectivities[i]]
    else:
        fail=1
    return connectivities

def numpy_incr(connectivities,increment):
    connectivities=connectivities+increment
    return connectivities

#similar to my_meshio_write_mesh but asking GiD data from python and processing this data here
def my_meshio_write_mesh2(filename):
    info_nodes=tuple(tcl.GiD_Info('mesh','nodes','-array'))
    node_ids,node_xyzs=info_nodes
    #tcl.W(node_ids)
    #tcl.W(node_xyzs)
    max_id_nodes=int(node_ids[-1])
    num_nodes=len(node_ids)
    nodes_renumbered_for_meshio=False
    if (max_id_nodes != num_nodes):
        nodes_renumbered_for_meshio=True
        new_node_ids=np.arange(num_nodes)
    points=gid_points_to_meshio_points(node_xyzs)
    cells = []
    for element_type in
['linear','triangle','quadrilateral','tetrahedra','pyramid','prism','hexahedra']:
        info_elements=tuple(tcl.GiD_Info('mesh','elements',element_type,'-array2'))
        if (len(info_elements)):
            #tcl.W(info_elements)
            elements_data=info_elements[0]
            element_type_ret,element_ids_original,connectivities_original,materials=elements_data
            element_ids=tohil_obj_array_int_to_numpy(tuple(element_ids_original))
            connectivities=tohil_obj_array_int_to_numpy(tuple(connectivities_original))
            num_elements_block=len(element_ids)
            if (num_elements_block):
                if (nodes_renumbered_for_meshio):
                    connectivities=numpy_renumber(connectivities,node_ids,new_node_ids)
                else:
                    connectivities=numpy_incr(connectivities,-1)
                some_element_id=element_ids[0]
                element_num_nodes=len(tcl.GiD_Mesh('get','element',some_element_id,'connectivities'))
                key=element_type+'_'+str(element_num_nodes)
                meshio_etype=gid_to_meshio_type[key]
                cells.append((meshio_etype,gid_elements_to_meshio_cells(element_num_nodes,
connectivities)))
            mesh=meshio.Mesh(points,cells)
            result=mesh.write(filename)
            return result

```

Note that in this case use Python commands to call Tcl like these:

```

import tohil

tcl=tohil.import_tcl()

info_nodes=tuple(tcl.GiD_Info('mesh','nodes','-array'))

info_elements=tuple(tcl.GiD_Info('mesh','elements',element_type,'-array2'))

#tcl.W(node_ids) #to show information in a GiD message window for debug

element_num_nodes=len(tcl.GiD_Mesh('get','element',some_element_id,'connectivities'))

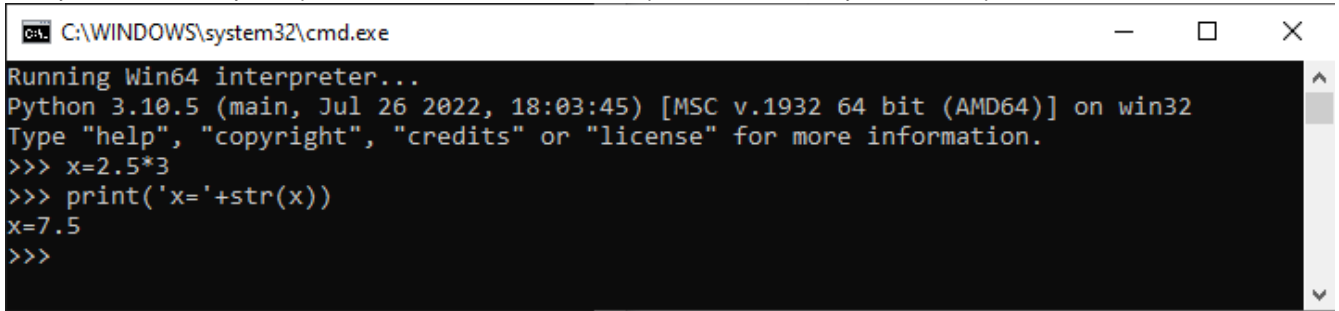
```

To know the syntax of GiD-Tcl added commands (like GiD_Info or GiD_Mesh) read the GiD Customization Manual: [TCL AND TK EXTENSION](#)

Debug Python code

Python Print

In a Python ran externally it is opened a DOS console, and then it is possible to use the Python function print to show values in this console



```

C:\WINDOWS\system32\cmd.exe
Running Win64 interpreter...
Python 3.10.5 (main, Jul 26 2022, 18:03:45) [MSC v.1932 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> x=2.5*3
>>> print('x='+str(x))
x=7.5
>>>

```

But running code inside GiD this console doesn't exists, and then this command cannot be used, except in case that the IDLE shell is opened and then the output of print is showed there.

Python print to file

To debug Python code it is always possible (inside and outside GiD) to print data of variables to a file, with something like this:

```

f=open('C:/temp/my_debug.txt','a')
f.write('hello world\n')
f.close()

```

Python show text with the W GiD proc

If Python is running in GiD then can call the Tcl GiD procedure called W that show text in a window

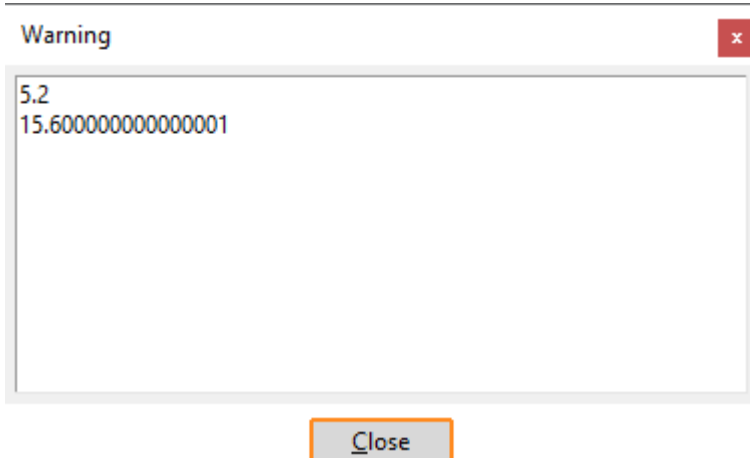
e.g.

```

GiD_Python_Exec {
    import tohil
    tcl=tohil.import_tcl()
    a=5.2
    tcl.W(a)
    b=a*3
    tcl.W(b)
}

```

will show a GiD window with the value of the variables 'a' and 'b'

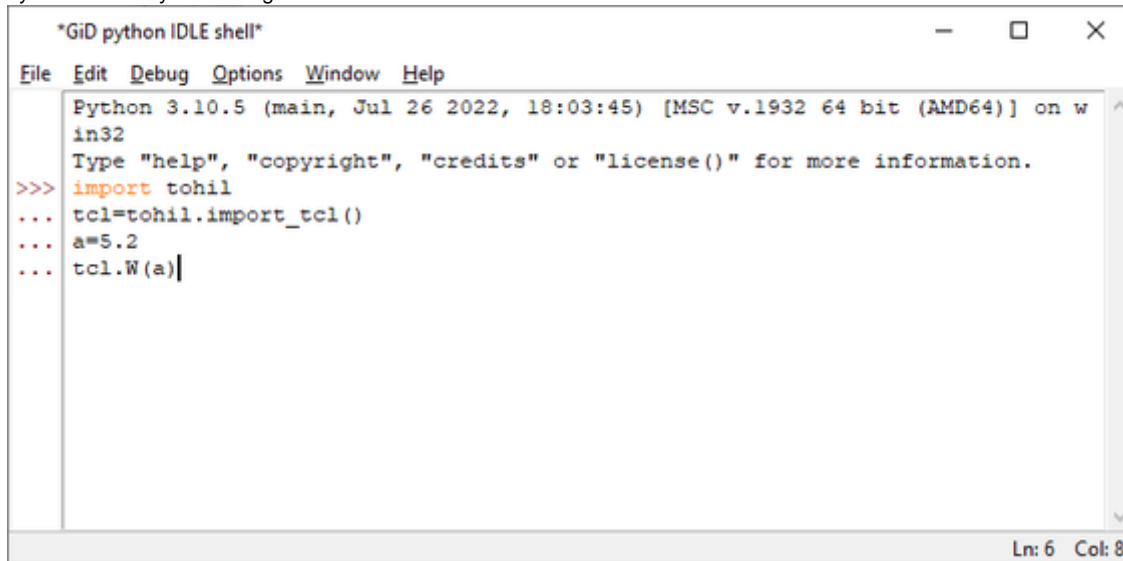


Note: `tcl = tohil.import_tcl()` is a tohil module command that uses Tcl's introspection capabilities to create Python object methods for each Tcl proc and command, so that calling the Tcl procs looks very much like calling any Python function.

Warning: the Python methods created depends on the procs existing when `tohil.import_tcl()` is invoked. If a proc like W is already not defined in Tcl during this call, the Python method `tcl.W('hello world')` won't exists, but `tohil.call('W','hello world')` will works.

Warning: It seems that `tcl.W()` cannot be called from the IDLE shell window or will crash

By now do not try something like this or GiD will crash:



```
*GiD python IDLE shell*
File Edit Debug Options Window Help
Python 3.10.5 (main, Jul 26 2022, 18:03:45) [MSC v.1932 64 bit (AMD64)] on w
in32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import tohil
... tcl=tohil.import_tcl()
... a=5.2
... tcl.W(a)|
Ln: 6 Col: 8
```

In this case it is less interesting to be used, because can use `print()`. The most interesting case is to use to debug code of `GiD_Python_Exec`, or `GiD_Python_Source`

Debug Python from VS Code editor

The scenery of use python in GiD is not usual, because the main process is not python.exe but gid.exe, and the python interpreter is embedded in GiD.

- The 'normal' case is to run a python.exe process that evaluate the code of a .py python file.

In the 'normal' case it is easy to use the Visual Studio Code editor, install the Python extension of Microsoft, and open a .py file, set breakpoints with <F9> and start debugging with <F5> and see the value of variables, stack trace, etc when the flow reaches a break point.

- The GiD case run a gid.exe that has embedded a Tcl interpreter and at run time is loaded the tohil package that create an embedded Python interpreter and add Tcl commands to call Python to this interpreter . At run time it is also possible that this Python interpreter import the tohil module that add Python functions to call the Tcl interpreter.

Note: Running python.exe and importing tohil will create a new Tcl interpreter (not related at all with the one embedded in GiD, then GiD commands cannot be used)

It is possible in this case to do a 'remote debugging' with Visual Studio Code, connecting the debugger with the embedded Python of a running gid.exe in a host and port (e.g. localhost and 5678)

but before gid must start a server on this host and port calling our Tcl proc GiD_Python_StartDebuggerServer

```
proc GiD_Python_StartDebuggerServer { } {
    tohil::import debugpy
    #tohil::exec "debugpy.log_to('[GidUtils::GetTmp]')"
    set my_python [file join [gid_filesystem::get_folder_standard scripts] tohil/python/bin/x64/python.exe]
    #to avoid a bug of debugpy that use sys.executable and try to run another gid.exe!! see
    https://github.com/microsoft/debugpy/issues/262
    tohil::exec "debugpy.configure(python='$my_python')"
    tohil::exec "debugpy.listen(5678)"
    #tohil::exec "debugpy.wait_for_client()"
    return 0
}
```

Note: The required Python module debugpy is pre-installed in our tohil's Python copy

Then open the folder with the .py file in VS Code

e.g. the folder <GiD>\plugins\Import\meshio

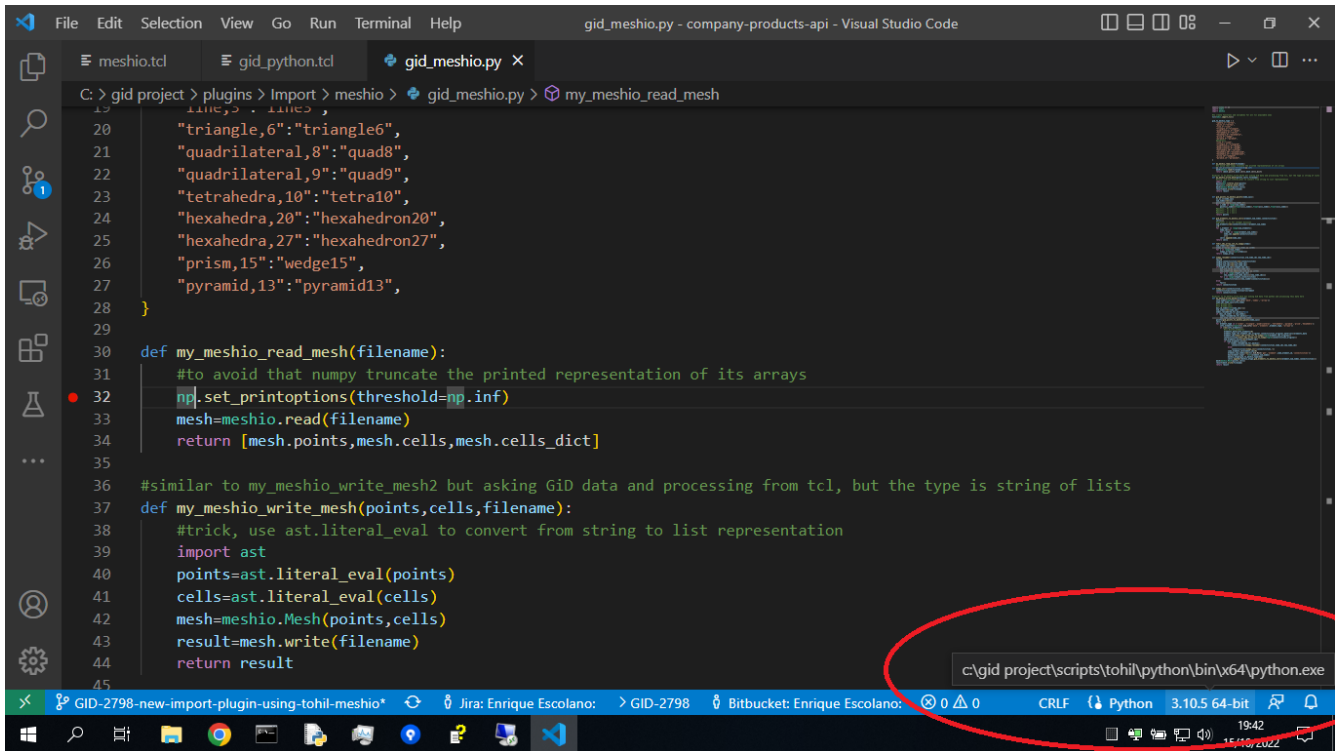
Note: be careful: must use File->Open folder..., not File->Open file, because opening the folder VS will create inside a hidden folder named '.vscode' where can save a file 'launch.json' to store the debug settings.

Select the file gid_meshio.py and set an stop <F9> in the line `np.set_printoptions(threshold=np.inf)`

Then click in the lower state bar to set as python interpreter the one of GiD (probably initially points to another Python of our system),

e.g. select

<GiD>\scripts\tohil\python\bin\x64\python.exe



Then start GiD and call `GiD_Python_StartDebuggerServer`

(e.g. can write `-np- GiD_Python_StartDebuggerServer`, or un-comment the line of `meshio.tcl` `#GiD_Python_StartDebuggerServer`)

And the first time that run the debug in VS Code <F5> it ask the configuration way to debug: select for example attach to remote debug, configuring the `host=localhost` and `port=5678`

this information is stored in the file `.vscode/launch.json`, with a content like this

```

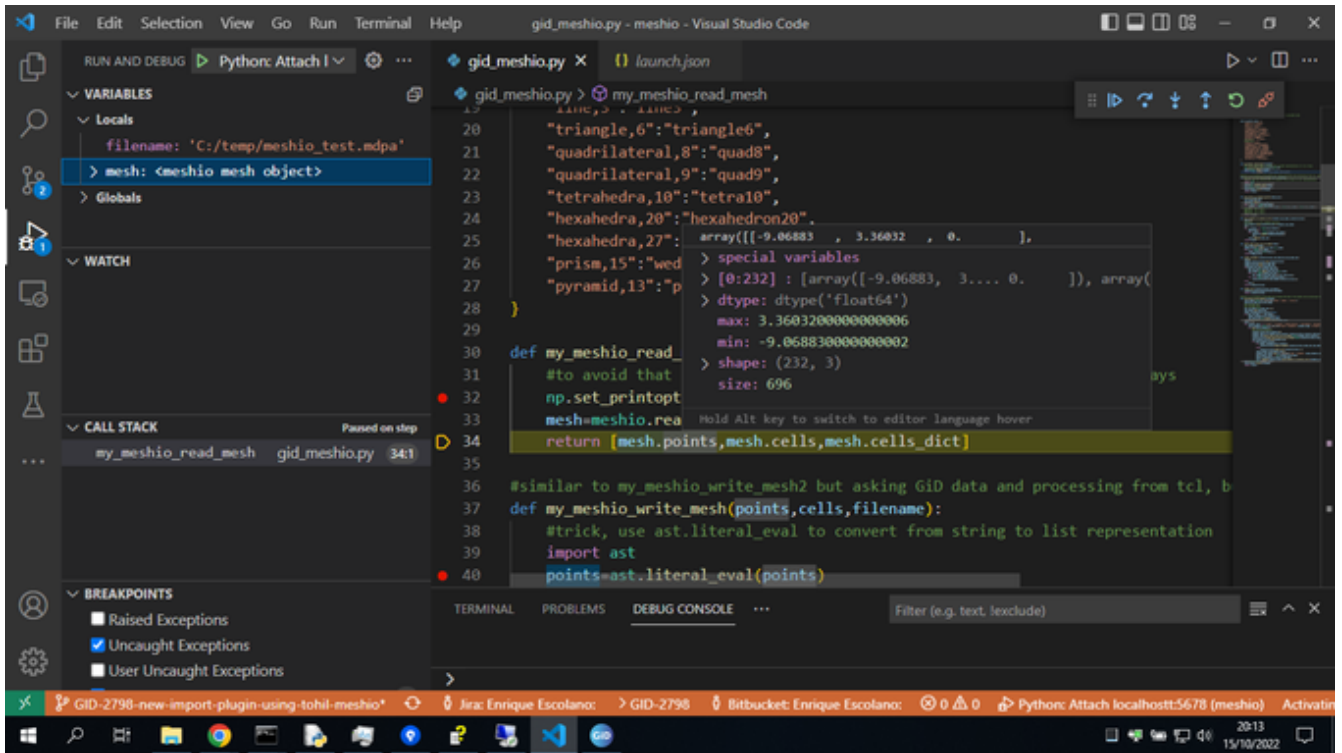
{
  // Use IntelliSense to learn about possible attributes.
  // Hover to view descriptions of existing attributes.
  // For more information, visit: https://go.microsoft.com/fwlink/?linkid=830387
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Python: Attach localhost:5678",
      "type": "python",
      "request": "attach",
      "connect": {
        "host": "localhost",
        "port": 5678
      },
      // "processId": "${command:pickProcess}",
      // "justMyCode": true
    }
  ]
}

```

In GiD go to menu `Files->Import->Meshio`

and select some mesh file, like a Kratos `.mdpa` file to be imported.

then the debugger will be stopped in our break point and can inspect variables, etc.



Python show text with W

If python is running in GiD then can call the Tcl GiD procedure called W that show text in a window

e.g.

```
import tohil
tcl=tohil.import_tcl()
a=5.2
tcl.W(a)
```

will show a GiD window with the value of the variable a

Python force reload a file

Using our Tcl command `GiD_Python_Import_File` (that do `tohil::import`) will import a module in Python, but if we are developing and modify the .py file doing a new import don't refresh the code in the interpreter.

A trick to do it is to use the Tcl command `GiD_Python_Source`, then the new code of the file is used without need to restart GiD.

In fact it seems that this is similar to use in Python the function `reload` of the `importlib` module

```
import importlib
importlib.reload(module)
```

Windows 7 issues

Python 3.10 is not supported in Windows 7, it requires a missing system library `api-ms-win-core-path-l1-1-0.dll`

We supply this missing feature compiling this alternative: <https://github.com/nalexandru/api-ms-win-core-path-HACK>

But Windows 7 has other special issues, it seems that don't use the PATH environment variable to find DLLs, then in order to be able to load the Python dependencies of `tohil.dll` is necessary to copy to the <GiD> main folder these files (can be located at <GiD>\scripts\tohil\python\bin\x64)

- `python3.dll`
- `python310.dll`
- `api-ms-win-core-path-l1-1-0.dll`

The same happens to load `gid.exe`, it is necessary to copy to <GiD> the DLLs:

- `tcl86t.dll`
- `tk86t.dll`
- `objarray.dll`

macOs issues

The IDLE shell opened from GiD show the menus on the top, like usual in macOS, but trying to close it is closing the whole GiD.

Future work

- objarray

Implement in tohil C/C++ source code the use of the GiD TclObj type objarray, to store efficiently vectors for the nodes and elements of the mesh, and do automatic conversion from/to the appropriated Python object. Now it is used a implicit conversion to string that is less efficient that use the native types of float, double, int, long, etc.

- W or other procs that open GiD Tk windows seems that cannot be called in the IDLE window, that is a TkInter window opened from python. It is crashing.
- It seems the the IDLE shell opened inside GiD is not able to debug the Python code, must be studied...
- It seems that the remote debug from VS Code editor only can handle some function, and the debug step by step cannot enter in other functions. Must be studied...