Using the LINGO Python API (19.0.6)

The LINGO Python interface allows you to send a model and it's supporting data to LINGO for solution. The interface can also return solution values to your Python code.

To learn more about the LINGO modeling language, see the LINGO manual. In particular, Chapter 11, *Interfacing with Other Applications,* may prove helpful.

For a quick start, there are several downloadable examples here: https://github.com/lindosystems/lingoapi-python/tree/main/examples that illustrate calling the LINGO API from Python.

Installing the LINGO Python API

You will need to install the LINGO API as an add-in package to your version of Python. You can do this from the command line with pip as follows (user input in ***bold italics***):

```
C:\pywork> pip install lingo_api
Collecting lingo_api
  Using cached lingo_api-19.0.6-cp310-cp310-win_amd64.whl (16 kB)
Requirement already satisfied: numpy>=1.19 in
c:\python\python310\lib\site-packages (from lingo_api) (1.22.3)
Installing collected packages: lingo_api
Successfully installed lingo_api-19.0.6
```

You may then test your installation as follows:

```
C:\pytest> python -m lingo_test
LINGO API is Working.
```

To run one of the Python examples, unpack the Chess example from GitHub, then enter the following:

```
C:\pytest\Chess> python chess.py

Global optimum found!
Brand       Peanut       Cashew     Produce
=========================================
Pawn         721.1538    48.0769   769.2308
Knight         0.0000     0.0000     0.0000
Bishop         0.0000     0.0000     0.0000
King          28.8462   201.9231   230.7692
=========================================
Totals          750.0      250.0     1000.0
```

Code Sample

Below is a Python code fragment that illustrates creating a LINGO model object. This code was extracted from the Chess blending example downloadable from GitHub.

```
lngFile = "chess.lng"
NUTS   = np.array(["Peanut","Cashew"])
BRANDS = np.array(["Pawn", "Knight", "Bishop", "King"])
SUPPLY = np.array( [750, 250])      # Total supply of each type
PRICE = np.array( [2,3,4,5])        # price that each brand charge
FORMULA = np.array( [[15,10, 6, 2],
                     [1, 6,10,14]])  # formula matrix
PRODUCE = np.zeros(len(PRICE))      # variables
STATUS = -1                         # LINGO status of model

# Pass the model to the LINGO API
lingo_api.Model(lngFile, "log")

# Pointers used in the model for passing data and solution values
model.set_pointer("Pointer1",NUTS,lingo.SET)
model.set_pointer("Pointer2",BRANDS,lingo.SET)
model.set_pointer("Pointer3",SUPPLY,lingo.PARAM)
model.set_pointer("Pointer4",PRICE,lingo.PARAM)
model.set_pointer("Pointer5",FORMULA,lingo.PARAM)
model.set_pointer("Pointer6",PRODUCE,lingo.VAR)
model.set_pointer("Pointer7",STATUS,lingo.VAR)

lingo.solve(model)
```

Model

Model data is passed in a Python object called `lingo.api` using the `Model` method. The calling sequence for `Model` is: `lingo_api.Model(lngFile, logFile="model.log", cbSolver=None, cbError=None, uData=None)`, where:

- **`lngFile`**: A string containing the path to the LINGO model file. The model must be saved in LINGO in LNG (text) format. LG4 (binary) format model files may not currently be passed to the LINGO API. The model's expressions must also be bracketed with a `MODEL:` statement at the start and an `END` statement at the end of the model (refer to any of the sample LNG model files to see the placement of these commands).
- **`logFile`**: An optional string path to a logfile that will created by the API. The log file is useful for debugging. Whenever you experience problems calling the LINGO API, be sure to review the contents of this log file for any errors. By default it will be named "`model.log`", and will be saved to the same directory as the python script running the model.
- **`cbSolver`**: An optional callback function written in pure Python that will be called by LINGO periodically. There are three LINGO getter functions that can be used to return information from the solver.
- **`cbError`**: An optional callback function written in pure Python that will execute when LINGO raises an error. This will allow for the user to raise the error in Python terminating the script and providing some detail on what should be fixed.
- **`uData`**: Is data passed to the callback functions and must be set to something other than `None` for the callback functions to be passed to the API.

Setting Pointers

To set pointers to a Model object so that they can be passed to LINGO use set_pointer(ptrName, ptrData, ptrType). Pointers must be set in the same order as they appear in the LINGO script.

- **`ptrName`**: A string name that describes the pointer and can be used to retrieve the data associated with it. It is helpful to name the pointer "PointerN" for the Nth pointer in the LINGO model.
- **`ptrData`**: Data to be sent to Lingo. This can be a NumPy array, floats or ints. For variable data send an NumPy array of zeros of the same length of the set that is associated with. If the ptrData is for naming set members send a NumPy array of strings.
- **`ptrType`**: A lingo_api constant that is used to indicate whether the pointer is a variable, parameter, or a set.
  - Lingo_api.SET: Use if ptrData is for naming set indexes.
  - Lingo_api.Param: Use if ptrData is constant model data.
  - Lingo_api.VAR: Use if ptrData is for a variable.

Solve

Once a `Model` object has been created, and all the pointers have been set, then use the `solve` function to call the LINGO API and process the model.

```
lingo_api.solve(model)
```

Getters

To get the model's data, there are two functions: `get_pointer(ptrName)` returns pointer data and its type:

```
price, ptrType = model.get_pointer("Price_Pointer")
```

To get the file path to the LINGO model file use `get_lngFile()`, and use `get_logFile()` to get the log file path, e.g.:

```
lngFN = model.get_lngFile()
```

Setters

To change the LINGO file to a different file path, use the function `set_lngFile(lngFile)`. To set or change the path of the log file use the function `set_logFile(logFile)`.

```
lngFile = "path/to/model.lng"
logFile = "path/to/modelLog.log"
model.set_lngFile(lngFile)
model.set_logFile(logFile)
```

To set callback functions use `set_cbSolver(cbSolver)` to set the solver callback function and `set_cbError(cbError)` to set the error callback function. To set user data use `set_uData(uData)`

```
model.set_cbSolver(cbSolver)
model.set_cbError(cbError)
model.set_uData(uData)
```

Sending Data To and Receiving the Solution Back From Lingo

For sending and receiving data in the LINGO API, the `@POINTER(i)` statement is used. If data is being sent to Lingo, the `@POINTER()` statement is placed on the righthand-side: `SUPPLY = @POINTER(3)`. If data is being sent back to Python from Lingo, the `@POINTER()` statement is placed on the lefthand-side: `@POINTER(6) = PRODUCE`.

Note that to get the solution status of the model, you can return the value of the `@STATUS()` function to LINGO as was done in the *Code Sample* section above. Possible status conditions are:

@STATUS() Code	Interpretation
0	Global Optimum - The optimal solution has been found, subject to current tolerance settings.
1	Infeasible - No solution exists that satisfies all constraints.
2	Unbounded - The objective can be improved without bound.
3	Undetermined - The solution process failed.
4	Feasible - A feasible solution was found that may, or may not, be the optimal solution.
5	Infeasible or Unbounded - The preprocessor determined the model is either infeasible or unbounded. Turn off presolving and re-solve to determine which.
6	Local Optimum - Although a better solution may exist, a locally optimal solution has been found.
7	Locally Infeasible - Although feasible solutions may exist, LINGO was not able to find one.
8	Cutoff - The objective cutoff level was achieved.
9	Numeric Error - The solver stopped due to an undefined arithmetic operation in one of the constraints.

In general, if `@STATUS()` does not return a code of 0, 4, 6, or 8, then the solution is of little use and should not be trusted. In many cases LINGO will not even export data to the `@POINTER()` memory locations if `@STATUS()` does not return one of these three codes.

Callback Functions

The LINGO API supports two types of callback functions that are defined by the user. The first is a solver callback that is called throughout the time that the solver is running. The second is an error callback that is called when the solver encounters and error. Both callback functions must follow a preset order of parameters or else they will not work properly. The next two subsections will explain in more detail how to use the callback functions.

User Data

User data is set set_uData() and is required to be set to something other than None. A useful data type is a dictionary since can be made filled with any type of data that can be accessed with a key.

```
uData = {"Prefix": "LINGO API", "Postfix": "…", "LastIter":-1,
"nVars"=Nvars}
model.set_uData(uData)
```

The example above uses a dictionary to pass three pieces of data to the callback functions. The `Prefix` and `Postfix` can be used in the callback printout. The `LastIter` can be used when the solver callback is sending data from the same iteration, and you only want to printout that iteration once.

Solver Callback

The solver callback is set with `set_cbSolver(cbSolver)` and must be a function written in Python. This callback function can request data from three different getter functions included in the LINGO API. The Python function has a few requirements that it must conform to in order to run properly. The first is the function parameters and the order in which they appear in the definition. The second is that the function must return 0 to continue or -1 to interrupt the LINGO solver.

```
def cbSolver(pEnv, nReserved, uData):

      # your code here

      return 0
```

The three parameters are passed to the call back function by LINGO from the API.

- **`pEnv`**: The environment pointer for the model and is used as an argument for the API callback getter functions.
- **`nReserved`**: An integer reserved for future versions of Lingo. This will always be 0 if printed out and is not used in any arguments for any API getter functions.
- **`uData`**: The user data that is set by the user.

Solver Callback Getter Functions

The two solver getter functions are:

```
pyLSgetIntCallbackInfoLng(penv, nobject, result)
pyLSgetDouCallbackInfoLng(penv, nobject, result)
```

The main difference between these two functions is the type of data they return, indicated by `Int` for `integer` and `Dou` for `double`.

- **`penv`:** The pointer to the LINGO environment that is solving the model.
- **`nobject`**: An integer that indicates what information will be inserted into result. The name of `nobject` can be used as well for example `0` is `lingo_api.LS_INFO_VARIABLES_LNG`.
- **`result:`** A NumPy array with a specified data type corresponding to which getter function is being used. If Int is being used then the data type of the array must be set to `numpy.int32`, and if it is Dou then the type must be set to `numpy.double`.
- **`Returns:`** An error code that should be checked before proceeding see the error code section for a detailed table of possible returns. To raise an exception use: `LingoError(errorcode)`, as we illustrate here:

```
nIters = numpy.array([-1], dtype=numpy.int32)
errorcode = lingo_api.pyLSgetIntCallbackInfoLng(penv,
lingo_api.LS_IINFO_ITERATIONS_LNG, nIters)
if errorcode != lingo_api.LSERR_NO_ERROR_LNG:
      raise lingo_api.LingoError(errorcode)
```

nobject	*Name*	*Type*	*Information Item*
0	LS_IINFO_VARIABLES_LNG	*Int*	Total number of variables
1	LS_IINFO_VARIABLES_INTEGER_LNG	*Int*	Number of integer variables
2	LS_IINFO_VARIABLES_NONLINEAR_LNG	*Int*	Number of nonlinear variables
3	LS_IINFO_CONSTRAINTS_LNG	*Int*	Total number of constraints
4	LS_IINFO_CONSTRAINTS_NONLINEAR_LNG	*Int*	Number of nonlinear constraints
5	LS_IINFO_NONZEROS_LNG	*Int*	Total nonzero matrix elements

6	LS_IINFO_NONZEROS_NONLINEAR_LNG	*Int*	Number of nonlinear nonzero matrix elements
7	LS_IINFO_ITERATIONS_LNG	*Int*	Number of iterations
8	LS_IINFO_BRANCHES_LNG	*Int*	Number of branches (IPs only)
9	LS_DINFO_SUMINF_LNG	*Double*	Sum of infeasibilities
10	LS_DINFO_OBJECTIVE_LNG	*Double*	Objective value
11	LS_DINFO_MIP_BOUND_LNG	*Double*	Objective bound (IPs only)
12	LS_DINFO_MIP_BEST_OBJECTIVE_LNG	*Double*	Best objective value found so far (IPs only)

To retrieve data specific to a variable use `LSgetCallbackVarPrimalLng(penv, varName, values)`. This variable can be any variable set in the Lingo script and does not need to be assigned to any pointers.

- **`penv:`** The pointer to the LINGO environment that is being used to solve the model.
- **`varName:`** A NumPy array of a string type that is accessible by the C API |s1024. Where s is string and 1024 is an arbitrary buffer size that needs to be long enough to hold the entire string.
- **`values:`** A NumPy array of type double that is at least the length of the number of values being returned.

An example follows:

```
varName = np.array(["X"], dtype="|s1024")
val     = np.zeros(uData["Nvars"],dtype=np.double)
errorcode = LSgetCallbackVarPrimalLng(penv, varName, val)
if errorcode != lingo_api.LSERR_NO_ERROR_LNG:
     raise lingo_api.LingoError(errorcode)
```

Error Callback

The error callback is set with `set_cbError(cbError)` and must be a function written in Python. The Python function has a few requirements that it must conform to in order to run properly. The first is the function parameters and the order in which they appear in the definition. The second is that the function must return nothing, and when it is called it is best to raise an exception to stop the program from running and display the error message as illustrated here:

```
def cbError(penv, uData, nErrorCode, errorText):
    raise lingo_api.CallBackError(nErrorCode, errorText)
```

The error callback's arguments are as follows:

- **`penv:`** The pointer to the LINGO environment used to solve the model.
- **`uData:`** The user's data, which is set by the user.
- **`nErrorCode:`** The error code number that corresponds to the error.
- **`errorText:`** A string with the reason for the error and some information on fixing it.

Troubleshooting

64-bit LINGO vs 32-bit Lingo

The LINGO API is configured to work with both 64- and 32-bit versions of Lingo. However, to use the 64-bit version of LINGO a 64-bit version of Python must be used. Similarly, the 32-bit version of LINGO requires a 32-bit version of Python. When `pip install lingo_api` runs, the version of Python associated with pip will install the appropriate bit-level version of LINGO API. To determine the version of Python associated with pip use the command: `pip -V`.

Possible errors due to misconfiguration

No Environment Variable

For 64-bit versions of LINGO the environment variable `LINGO64_19_HOME` must be set before using the LINGO API. If it is not set, you will see the error “Environment variable LINGO64_19_HOME should be set to the Lingo64_19 directory".

Similarly for 32-bit versions of LINGO the environment variable `LINGO_19_HOME` must be set before using the LINGO API. If it is not set, the error "Environment variable LINGO_19_HOME should be set to the Lingo19 directory".

Normally, Lingo’s installation program sets these environment variables, so they will not normally be of concern.

Fix Using Windows

On the command line for Windows 64:

```
> setx LINGO_19_HOME "C:\LINGO64_19"
```

On the command line for Windows 32

```
> setx LINGO64_19_HOME "C:\LINGO64_19"
```

Fix Using Linux

For administrative users:

```
$ export LINGO64_19_HOME="/opt/lingo19"
```

For standard (non-administrative) users:

```
$ export LINGO64_19_HOME="~/opt/lingo19"
```

To have this variable set automatically, add the above line to your `~/.bashrc` or `~/.bash_profile` file.

LINGO Import Error

This error will occur when the .dll (Windows), or .so (Linux) files are not where they are expected. If the .dll, or .so files are never moved or deleted this error will not occur. If, however the files have been moved then when `import lingo_api` is ran. For example, this is what the error looks like for windows 64-bit versions.

```
LINGO Import Error:
                        Make sure all the following files are present
in C:\LINGO64_19:
                         Chartdir60.dll
                         Cilkrts20.dll
                         Conopt3.dll
                         Conopt464.dll
                         Libifcoremd.dll
                         Libiomp5md.dll
                         Libmmd.dll
                         Lindo64_13_0.dll
                         Lindopr64_8.dll
                         Lingd64_19.dll
                         Lingdb64_3.dll
                         Lingf64_19.dll
                         Lingfd64_19.dll
                         Lingj64_19.dll
                         Lingoau64_14.dll
                         Lingr64_1.dll
                         Lingxl64_5.dll
                         Mosek64_9_2.dll
                         Msvcr120.dll
>>>
```

The directory `C:\LINGO64_19` is the same directory that the environment variable `LINGO64_19_HOME` points to. The .dll files are all of the files that where present in that directory when LINGO was initially installed and need to remain in that directory.

LingoError

The LINGO API function `solve()` makes the API calls to LINGO to allocate memory, solve the model, and to deallocate the memory. These calls return an error code if that error code is nonzero then a `LingoError` is raised. If not caught an error message will be displayed, and the Python script will exit. `LingoError` has two fields:

- error: The value of the errorcode.
- message: A string that describes the error.

To catch this error put the solve function in a try/except block like so:

```
try:
    lingo.solve(model)
except lingo.LingoError as e:
    if(e.error == 73):
       print(f" A user interrupt occurred.”)
    eles:
        print(e.message)
    exit(1)
```

Here is an example of what is displayed in the terminal if the error is not caught:

```
File "C:\Users\James\Documents\GitHub\lingoapi-
python\examples\CHESS\chess.py", line 72, in <module>
    lingo.solve(model)
File "C:\Users\James\Desktop\myenv\lib\site-
packages\lingo_api\modelLoader.py", line 79, in solve
    raise LingoError(1)
lingo_api.lingoExceptions.LingoError: 1 -> Out of dynamic system
memory.
```

The table below includes all the errors that may occur.

Value	*Name*	*Descriptions*
0	LSERR_NO_ERROR_LNG	No error.
1	LSERR_OUT_OF_MEMORY_LNG	Out of dynamic system memory.
2	LSERR_UNABLE_TO_OPEN_LOG_FILE_LNG	Unable to open the log file.
3	LSERR_INVALID_NULL_POINTER_LNG	A NULL pointer was passed to a routine that was expecting a non-NULL pointer.

4	LSERR_INVALID_INPUT_LNG	An input argument contained invalid input.
5	LSERR_INFO_NOT_AVAILABLE_LNG	A request was made for information that is not currently available.
6	LSERR_UNABLE_TO_COMPLETE_TASK_LNG	Unable to successfully complete the specified task.
7	LSERR_INVALID_LICENSE_KEY_LNG	The license key passed to *LScreateEnvLicenceLng()* was invalid.
8	LSERR_INVALID_VARIABLE_NAME_LNG	A variable name passed to *LSgetCallbackVarPrimal()* was invalid.
73	LSERR_USER_INTERRUPT_LNG	The error callback function raised an exception.
1000	LSERR_JNI_CALLBACK_NOT_FOUND_LNG	A valid callback function was not found in the calling Java
1001	LSERR_CALLBACK_ERROR_SET	A user interrupt occurred.

TypeNotSupportedError

The `ptrData` set by `model.set_pointer()` needs to be NumPy arrays, floats, or ints. Otherwise, a `TypeNotSupportedError` exception will be raised by the `solve()` function. If not caught an error message will be displayed, and the Python script will exit.
`TypeNotSupportedError` has two fields:

- error: The type of the pointer that caused the error.
- message: A string that describes the accepted datatypes.

To catch this error put the solve function in a try/except block like so:

```
try:
    lingo.solve(model)
except lingo.TypeNotSupportedError as e:
    print(e)
```

Here is an example of what is displayed in the terminal if the error is not caught:

```
Traceback (most recent call last):
  File "errorTest.py", line 151, in <module>
    lingo.solve(model)
  File "C:\Users\James\Desktop\myenv37\lib\site-
packages\lingo_api\modelLoader.py", line 147, in solve
    raise TypeNotSupportedError(error)
lingo_api.lingoExceptions.TypeNotSupportedError: Pointer2 [0.0, 0.0,
0.0] type: <class 'list'> -> Unsupported type
Excepted For VAR/PARAM: NumPy array of numbers, Int, floats
Excepted For       SET: NumPy array of String or Int
```

PointerTypeNotSupportedError

The ptrType set by `model.set_pointer()` needs to be of the three lindo_api constants available

- lingo_api.SET: Use if ptrData is for naming set indexes.
- lingo_api.Param: Use if ptrData is constant model data.
- lingo_api.VAR: Use if ptrData is for a variable.

Otherwise, a `PointerTypeNotSupportedError` exception will be raised by `solve()`, and if not caught the Python script will be terminated. `PointerTypeNotSupportedError` has two fields:

- error: The type of the pointer that caused the error.
- message: A string that lists the three pointer types.

To catch this error put the solve function in a try/except block like so:

```
try:
    lingo.solve(model)
except lingo.PointerTypeNotSupportedError as e:
    print(e)
```

Here is an example of what is displayed in the terminal if the error is not caught:

```
Traceback (most recent call last):
  File "errorTest.py", line 144, in <module>
    lingo.solve(model)
  File "C:\Users\James\Desktop\myenv37\lib\site-
packages\lingo_api\modelLoader.py", line 178, in solve
    raise PointerTypeNotSupportedError(ptrType)
lingo_api.lingoExceptions.PointerTypeNotSupportedError: 5345 -> is
not a supported pointer type\Supported types:
lingo_api.SET
lingo_api.PARAM
lingo_api.VAR
```

EmptyPointer

The pointer data `set by model.set_pointer()` can not be empty, and if it is `solve()` will raise am `EmptyPointer` exception. `If not caught an error message will be displayed, and the Python script will exit. EmptyPointer` has two fields:

- error: The name of the pointer that caused the exception to be raised.
- message: A string that describes the error.

To catch this error put the solve function in a try/except block like so:

```
try:
    lingo.solve(model)
except lingo.EmptyPointer as e:
    print(e)
```

Here is an example of what is displayed in the terminal if the error is not caught:

```
Traceback (most recent call last):
  File "errorTest.py", line 156, in <module>
    lingo.solve(model)
  File "C:\Users\James\Desktop\myenv38\lib\site
packages\lingo_api\modelLoader.py", line 150, in solve
    raise EmptyPointer(key)
                lingo_api.lingoExceptions.EmptyPointer:

Pointer2 -> is an empty pointer. Allocate memory needed
```

Catching All Errors

Note that more then one exception can be handled in a try/except block bellow is an example of each of the LINGO exceptions and the base exception being caught:

```
try:
    lingo.solve(model)

except lingo.LingoError as e:
    # Your Code Here

except lingo.TypeNotSupportedError as e:
    # Your Code Here

except lingo.EmptyPointer as e:
    # Your Code Here

except lingo.PointerTypeNotSupportedError as e:
    # Your Code Here

except Exception as e:
    # Catch any non-LINGO errors
    # Your Code Here
```

How to Build Wheel and Install (for package managers)

To build the python package on any operating system first start by creating a whl file. From the top of the `lingoapi-python` directory run the command.

```
python -m build
```

If the command is successful, a new directory named `dist` is created in the `lingoapi-python` directory. The new directory will have two files with extension .whl and .tar.gz. For example, if you build on Windows using Python 3.10 the new directory will look like this.

```
├── dist
| ├── lingo_api-x.y.z-cp310-cp310-win_amd64.whl
| └── lingo-x.y.z.tar.gz
```

The package can now be installed locally using the command.

```
> pip install dist/*.whl
```