Chrono::Python

Python Interoperability Module
Chrono::Python

• Use Python in Chrono? Yes!
  The **PYTHON module** is a wrapper to Chrono classes and functions

• How to build it:
  • Install Python (suggested v.3.3 or later), **64 bit** if you compiled Chrono in 64 bit
  • Install the SWIG tool
  • Enable PYTHON module in Chrono CMake
  • Set directories to Python libs etc. in the Cmake
  • Build Chrono – generating the PYTHON module might require few minutes...

• The **PYTHON module** of Chrono contains
  a) The **ChronoEngine_pyparser.dll** library for parsing Python **from the C++ Chrono side**
  b) The **Chrono::PyEngine .pyd modules** for calling the API **from the Python side**
Chrono::Python
a) Call Python from the C++ side
a) Call Python from the C++ side:

- Use the ChronoEngine_pyparser.dll library

- Create a python engine for parsing:

  ```cpp
  ChPythonEngine my_python;
  // figure out what version of Python is run under the hood
  my_python.Run("import sys");
  GetLog() << "Python version run by Chrono:\n";
  my_python.Run("print (sys.version)\n");
  ```

- Execute Python instructions:

  ```cpp
  my_python.Run("a =8.6");
  my_python.Run("b =4");
  my_python.Run("c ='blabla' ");
  my_python.Run("print('In:Python - A computation:', a/2)\n");
  ```
a) Call Python from the C++ side:

// TEST - fetch a value from a python variable (in __main__ namespace)
GetLog() << "\n\nChrono::PyEngine Test 3.\n";
double mfval;
my_python.GetFloat("a", mfval);
GetLog() << "In:C++ - Passed float variable 'a' from Python, a=" << mfval << "\n";

// TEST - set a value into a python variable (in __main__ namespace)
my_python.SetFloat("d", 123.5);
my_python.Run("print('In:Python - Passed variable d from c++, d=', d)");

// In the previous examples we didn't have any syntax errors.
// In general, it is wise to enclose Python commands in a try-catch block
// because errors are handled with exceptions:

try {
    my_python.Run("a= this_itGoInG_TO_giVe_ErroRs()");
} catch (ChException myerror) {
    GetLog() << "Ok, Python parsing error caught as expected.\n";
}
a) Call Python from the C++ side:

- Load a mechanical system in a .py file:

  ```cpp
  # load a mechanical system, previously saved to disk from SolidWorks add-in
  ChSystem my_system;
  try {
    my_python.ImportSolidWorksSystem(GetChronoDataFile("solid_works/swiss_escapement").c_str(),
                                      my_system);  // note, don't type the .py suffix in filename..
    my_system.ShowHierarchy(GetLog());
  }
  
  How to generate the .py models? ⇒ See the Chrono::SolidWorks add-in
b) Use Chrono from the Python side
b) Use Chrono from the Python side:

• Have you built Chrono::PyEngine? (see build instructions on the web site)

• Ok, now you can call Chrono API functions from a Python command line!

• Suggested: use PyScripter or similar IDEs for editing/running Python programs

• Hint: look at .py examples in chrono\src\demos\python
b) Use Chrono from the Python side:

• Important!!! All Python programs must import Chrono::PyEngine Python modules using the import statement:

```python
import ChronoEngine_python_core as chrono
```

• Chrono classes will be accessed via the `chrono.xxx.yyy.zzz` Python namespace.

• If you use additional modules, for example, add also:

```python
import ChronoEngine_python_postprocess as postprocess
import ChronoEngine_python_irrlicht as chronoirr
```
b) Use Chrono from the Python side:

- Let's create a 3D vector object:
  ```python
  my_vect1 = chrono.ChVectorD()
  ```

- Modify the properties of that vector object; this is done using the . dot operator:
  ```python
  my_vect1.x = 5
  my_vect1.y = 2
  my_vect1.z = 3
  ```

- Some classes have build parameters, for example another vector can be built by passing the 3 coordinates for quick initialization:
  ```python
  my_vect2 = chrono.ChVectorD(3,4,5)
  ```

- Most operator-overloading features that are available in C++ for the Chrono::Engine vectors and matrices are also available in Python, for example:
  ```python
  my_vect4 = my_vect1*10 + my_vect2
  ```
b) Use Chrono from the Python side:

- Member functions of an object can be called using the . dot operator, like in C++:
  ```python
  my_len = my_vect4.Length()
  print ('vector length =', my_len)
  ```

- You can use most of the classes that you would use in C++, for example let's play with quaternions and matrices:
  ```python
  my_quat = chrono.ChQuaternionD(1,2,3,4)
  my_qconjugate = ~my_quat
  print ('quat. conjugate =', my_qconjugate)
  print ('quat. dot product=', my_qconjugate ^ my_quat)
  print ('quat. product=', my_qconjugate % my_quat)
  ma = chrono.ChMatrixDynamicD(4,4)
  ma.FillDiag(-2)
  mb = chrono.ChMatrixDynamicD(4,4)
  mb.FillElem(10)
  mc = (ma-mb)*0.1;
  print (mc);
  mr = chrono.ChMatrix33D(); ...
  ```
b) Use Chrono from the Python side:

* Differences respect to the C++ API:

  - Not all C++ classes/functions are wrapped in Python
  - Templated classes are instanced with type ‘double’ by appending ‘D’ at the name:

<table>
<thead>
<tr>
<th>PYTHON</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>chrono.ChVectorD</td>
<td>ChVector&lt;double&gt;</td>
</tr>
<tr>
<td>chrono.ChQuaternionD</td>
<td>ChQuaternion&lt;double&gt;</td>
</tr>
<tr>
<td>chrono.ChMatrix33D</td>
<td>ChMatrix33&lt;double&gt;</td>
</tr>
<tr>
<td>chrono.ChMatrixNMD</td>
<td>ChMatrixNM&lt;double&gt;</td>
</tr>
</tbody>
</table>
b) Use Chrono from the Python side:

* Differences respect to the C++ API:

  - Shared pointers are handled automatically:
    
    **C++:**
    ```cpp
    std::shared_ptr<ChLinkLockRevolute> my_link_BC(new ChLinkLockRevolute);
    ```
    
    **PYTHON:**
    ```python
    my_link_BC = chrono.ChLinkLockRevolute()
    ```

  
  - Upcasting is automatic, like in C++, but downcasting? There are no dynamic_cast.... But we added some helper functions called `CastToChClassNameShared()` :
    
    **C++:**
    ```cpp
    myvis = std::dynamic_pointer_cast<ChVisualization>(myasset);
    ```
    
    **PYTHON:**
    ```python
    myvis = chrono.CastToChVisualizationShared(myasset)
    ```