Python as a Binding Language

Using Python to manage and control libraries & functionality from other languages.
Why Python?

- An interpreter (as oppose to compiler):
  - Much easier to develop and test new code and functionality.
  - In many cases, (i.e. Python) offer a very large set of libraries and tools (modules).
  - Does garbage collection (no need to manage dynamic memory allocations).
  - Execution is likely to be slower as a side effect.

- But how do we add a new module (library) to Python?
  - Need to link functions and exchange data between Python-space and in C-space.
## Python vs. C

<table>
<thead>
<tr>
<th>C</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Compiler (like C++, C#, Pascal, Java)</strong></td>
<td><strong>Interpreter (like PHP, Perl)</strong></td>
</tr>
<tr>
<td>Process whole module at a time</td>
<td>Process one line at a time</td>
</tr>
<tr>
<td>Compiled sources into stand alone executables</td>
<td>Scripting that always require python to be executed</td>
</tr>
<tr>
<td>High performance (does not need the sources during execution)</td>
<td>Slower (parses/interprets the source lines while execution)</td>
</tr>
<tr>
<td>Allows for global (inter- and intra-) module optimization</td>
<td>Can do only, if at all, optimization at the single expression level</td>
</tr>
<tr>
<td>System programming</td>
<td></td>
</tr>
<tr>
<td>Dyn. Mem. Management</td>
<td>Garbage collection</td>
</tr>
<tr>
<td></td>
<td>Rapid testing</td>
</tr>
<tr>
<td></td>
<td>Rapid prototyping</td>
</tr>
<tr>
<td></td>
<td>Gluing</td>
</tr>
</tbody>
</table>
A first example of using python for an interactive session, invoking C code, is in *geometric modeling*
- building geometry on the fly.

We employ work developed, on a geometric kernel called *irit* ([www.cs.technion.ac.il/~irit](http://www.cs.technion.ac.il/~irit)).

The next few slides are snapshots of a python interactive session, using *irit*. 
```
>>> import math
>>> import irit
>>> t1 = irit.box( ( -2, -0.35, 0 ), 4, 0.7, 0.4 )
>>> t2 = irit.cyl( ( 0, 0, 0 ), ( 0, 0, 0.4 ), 1.4, 3 )
>>> s1 = t1 * t2
>>> t3 = irit.cyl( ( 0, 0, 0 ), ( 0, 0, 0.4 ), 0.9, 3 )
>>> s2 = s1 + t3
```

The point set operations are employing regular python operators, but they operate on/between irit geometric object!
>>> t4 = irit.cylinder( (1.45, -0.5, 1), (0, 1, 0), 0.8, 3 )
>>> t5 = irit.cylinder( (-1.45, -0.5, 1), (0, 1, 0), 0.8, 3 )
>>> s3 = s2 - t4 - t5
>>> t6 = irit.cylinder( (1.2, 0, -0.1), (0, 0, 0.5), 0.1, 3 )
>>> t7 = irit.cylinder( (-1.2, 0, -0.1), (0, 0, 0.5), 0.1, 3 )
>>> s4 = s3 - t6 - t7
>>> t8 = irit.cylinder( (0, 0, -0.2), (0, 0, 0.9), 0.3, 3 )
>>> t9 = irit.box( (-0.6, -0.15, -0.1), 1.2, 0.3, 0.7 )
>>> s5 = t8 + t9
>>> s6 = s4 - s5

Point set subtraction
Consider *C function* named `Master` that calls *C function* named `Slave`.

The following is performed:

- The (addresses of the) parameters to be provided to `Slave`, are pushed onto the stack.
- The return address of `Master` is pushed onto the stack.
- The program jumps to function `Slave` that is executed and creates its own local (automatic) variables.
On completion of **Slave**’s executions:

- The return result, if any, of **Slave** is pushed back onto the stack.
- After returning to **Master**, return result is fetched from the stack and the stack is cleaned.
  - updating the stack pointer.

Now what about a Python function **Master** calling a C function **Slave**?
A Python function named **Master** calling a C function named **Slave** must go a similar process to a C $\rightarrow$ C call.

However, data is now in different (Python/C) spaces and may have a completely different representation.

Luckily, these difficulties are taken into considerations and alleviated by some provided support.

* Low level Python support (to be discussed first).
* High level tools (SWIG) to ease such bindings (discussed second).
For Python function "Master" to call a C function "Slave", the Python environment must be aware of "Slave"’s. This means the name "Slave" but also its interface (parameters and return value).

The Python interpreter can import (typically dynamic/shared) external libraries (i.e. modules).

Typically with a suffix of (.so/.dll/.pyd).

For Python to recognize the library, each such C library named “XXXX.so” must have an external C function, called “initXXXX”, as an entry point.
Calling a function (Python -> C) III

- Python can be asked, at any time, to load a new module ‘XXXX’ via the Python command:
  
  \[ \text{import XXXX} \]

- Python will then look, at a special directory, for a module named `XXXX.{so,dll,pyd}`.
  
  \* Typically `/usr/lib/pythonZ.Z/lib-dynload` on Unix and `PythonZ.Z\Lib\site-packages` on Windows (python Z.Z).

- If found, module `XXXX` will be loaded and function `initXXXX` will be sought and automatically executed.

- Failing to find module `XXXX` (at that directory) or function `initXXXX` in `XXXX` constitutes a Python error...
Python Provide a set of low level functions to convert data from C space to Python space and vice versa:

- In C space, Python space data is encapsulated in PyObjects objects.
- **PyArg_ParseTuple** - Converts Python space data to C space.
- **Py_BuildValue** - Converts C space data to Python space.
- **Py_InitModule** – Initialization routine (from initXXXX...).
- These Python structs/functions are defined in dedicated headers (Python.h) and Python libraries.
  - That must be included and linked to.
Consider the following C functions, in one file:

```c
#include <ctype.h>
#include <Python.h>

static PyObject *matam_toupper(PyObject *self, PyObject *args) {
    unsigned int i;
    const char *str;
    char *UStr;
    PyObject *RetStr;
    if (!PyArg_ParseTuple(args, "s", &str))
        return NULL;
    UStr = strdup(str);
    for (i = 0; i < strlen(UStr); i++)
        UStr[i] = toupper(UStr[i]);
    RetStr = Py_BuildValue("s", UStr);
    free(UStr);
    return RetStr;
}

static PyObject *matam_system(PyObject *self, PyObject *args) {
    const char *command;
    int sts;
    if (!PyArg_ParseTuple(args, "s", &command))
        return NULL;
    sts = system(command);
    return Py_BuildValue("i", sts);
}
```
Consider the following C functions, in one file:

```c
static PyMethodDef Matam_methods[] = {
    {"system", matam_system, METH_VARARGS,
        "Execute a shell command."},,
    {"toupper", matam_toupper, METH_VARARGS,
        "Convert a string to upper case."},,
    {NULL, NULL, 0, NULL}
};
```

The name in Python space

Some helpful description

- The METH* flags tells Python how to process parameters.
#include <Python.h>

PyMODINIT_FUNC initmatam_simple(void)
{
    fprintf(stderr, "matam_simple is initialized.\n");
    (void) Py_InitModule("matam_simple", Matam_methods);
}

- The only external function (that must exists) in the module is `initmatam_simple`.
- All Py* C functions/types are python related.
We created a (shared) library/module called matam_simple.

The only entry point to the module is `initmatam_simple`.

`Initmatam_simple` references the `Matam_methods` table.

`Matam_methods` lists all functions Python can invoke.
Once compiled and linked, matam_simple.so must be placed at the directory Python expects.

Then, we can do the following:

```bash
[gershon-VirtualBox] ~/matam , 98 > python
Python 2.7.12 (default, Dec 4 2017, 14:50:18)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.

>>> import matam_simple
matam_simple is initialized.
```

And we are ready to use the functions in this C shared library!
Using the C functions in Python II

```python
>>> print matam_simple
<module 'matam_simple' from 'matam_simple.so'>

>>> dir(matam_simple)
['__doc__', '__file__', '__name__', '__package__', 'system', 'toupper']

>>> matam_simple.system("ls -sFC");
total 12712
  4 makefile.unx   9292 matam_simpleD.so*   24 matam_simple.oD
  4 makefile.wnt   4 matam_simple.gcno  3376 matam_simple.so*
  4 matam_simple.c   4 matam_simple.o

>>> matam_simple.toupper("This is some string");
'THIS IS SOME STRING'

>>> 
```

And also try “help(matam_simple)” and “help(matam_simple.toupper)”
While general, the low level interface that Python is offering is quite tedious to use.

**SWIG (Simplified Wrapper and Interface Generator)** is an example of a high level tool.

SWIG processes C function declarations (header files).
- Automatically generates the proper (not just Python) interface.
For our original `matam_simple.c` module, with functions toupper & system, we like to exploit in Python:

- Write a SWIG interface file, `matam_simple.i`, with these two functions’ declarations.
- Only to “swig –python matam_simple.i” to generate `matam_simple_wrap.c`.
- Compile original module `matam_simple.c` along with new `matam_simple_wrap.c` to build `matam_simple_wrap.so`. 

Calling a function (Python → C)
So far we saw how Python can invoke C function.
- Python provides low level special functions to map C space data/function to Python Space.
- High level tools, such as SWIG, exists as a viable alternative.

Additional Comments:
- Some OSs, like windows, require one to explicitly declare functions (and variables) that are made external in a DLL.
- Static linking is possible too – require rebuild of Python.
- One can also embed Python in C, but we will leave it to you to see how to invoke Python from C, if interested.
“Surely the most powerful stroke for software productivity, reliability, and simplicity has been the progressive use of high-level languages for programming. Most observers credit that development with at least a factor of five in productivity, and with concomitant gains in reliability, simplicity, and comprehensibility.”

- Frederick P. Brooks, Jr.