Programming in Python

GAVRILUT DRAGOS

COURSE 13

ltegrating with C/C++

ctype is a module design for the following:

- Create wrappers over the most used C/C++ basic types (int, long, char, ….)
- Allows to load a dynamic library and access some of its exported function
- Methods for working with pointers
- Methods for working with strings (char* or wchar_t*)

Details about these modules can be found on:

- Python 2: https://docs.python.org/2/library/ctypes.html
- Python 3: https://docs.python.org/3/library/ctypes.html

ctypes

ctypes module provides a list of types that correspond to basic C/C++ types.

Object	C Type
c_char, c_byte	char
c_wchar	wchar_t
c_ubyte	unsigned char
c_bool	bool
c_short	short
c_ushort	unsigned short
c_int	int
c_uint	unsigned int
c_long	long
c_ulong	unsigned long

Object	C Type
c_longlong	long long
c_ulonglong	unsigned long long
c_size_t	size_t
c_float	float
c_double	double
c_long_double	long double
c_char_p	char*
c_wchar_p	wchar_t*
c_void_p	void*

ctypes

Example on how to use ctypes C/C++ types

Python 2.x/3.x

```
import ctypes

i = ctypes.c_int(10)
print (i.value)
i.value = 100
print (i.value)

i = ctypes.c_char_p(b"Python")
print (i.value)

i = ctypes.c_wchar_p("Python")
print (i.value)
```

Output

```
10
100
Python
Python
```

ctypes

Example on how to use ctypes C/C++ types

Python 2.x/3.x

```
import ctypes

i = ctypes.c_int(10)
print (i.value)
i.value = 100
print (i.value)

i = ctypes.c_char_p(b"Python")
print (i.value)

i = ctypes.c_wchar_p("Python")
print (i.value)
```

c_char_p expects a byte array for constructor in Python 3.

ut

on

hon

ctypes

Example on how to use ctypes C/C++ types

Python 2.x/3.x

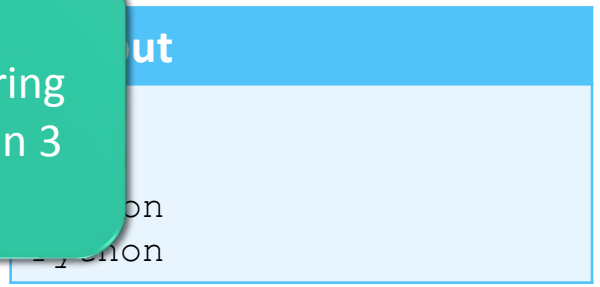
```
import ctypes

i = ctypes.c_int(10)
print (i.value)
i.value = 100
print (i.value)

i = ctypes.c_char_p(b"Python")
print (i.value)

i = ctypes.c_wchar_p("Python")
print (i.value)
```

c_wchar_p expects a string for constructor in Python 3



ctypes

ctypes also includes a function called **pointer** that can create a pointer to another **ctypes** object. This is useful when a pointer of some sort is required for a specific C/C++ function.

Python 2.x/3.x

```
import ctypes

i = ctypes.c_int(10)
ptr_i = ctypes.pointer(i)
print(ptr_i.contents.value)
ptr_i.contents.value = 20
print(i.value)
j = ctypes.c_int(30)
ptr_i.contents = j
print(ptr_i.contents.value)
```

Output

```
10
20
30
```

In this example, **ptr_i** is a pointer to the variable **i**. Each pointer object has a property (**contents**) that give access to the object the pointer points to (in our case, **ptr_i.contents** is **i**)

ctypes

References can also be used. ctypes provides a function **byref** that can be used to obtain a reference to an ctypes object.

Python 2.x/3.x

```
import ctypes

i = ctypes.c_int(10)
ref_i = ctypes.byref(i)
print(ref_i._obj.value)
ref_i._obj.value = 20
print(i.value)
j = ctypes.c_int(30)
```

Output

```
10
20
```

Building a reference is faster in Python that building a pointer. If speed is a concern **byref** should be used instead of **pointer**.

ctypes

References can also be used. ctypes provides a function **byref** that can be used to obtain a reference to an ctypes object.

Python 2.x/3.x

```
import ctypes

i = ctypes.c_int(10)
ref_i = ctypes.byref(i)
print(ref_i._obj.value)
ref_i._obj.value = 20
print(i.value)
j = ctypes.c_int(30)

ref_i._obj = j
```

An error will occur. _obj parameter is read only and can not be assigned anymore.

ctypes

C like structures can be build and pass as a parameter using *ctypes.Structure*. Fields will be created just like in a C/C++ structure.

Python 2.x/3.x

```
import ctypes

class Point(ctypes.Structure):
        _fields_ = [("x",ctypes.c_float),("y",ctypes.c_float)]

p = Point(1.5,3.5)
p.x += p.y
print(p.x,p.y)
print(Point.x.offset,Point.x.size)
print(Point.y.offset,Point.y.size)
```

C Structure

```
struct Point
{
    float x;
    float y;
}
```

Output

```
5.0 3.5
0 4
4 4
```

ctypes

ctype.Structure accepts multiple parameters for initialization (one for each field). If one is missing the default value (0 for numbers) will be considered.

Python 2.x/3.x

```
import ctypes

class Point(ctypes.Structure):
        _fields_ = [("x",ctypes.c_float),("y",ctypes.c_float)]

p = Point(1.5)
print(p.x,p.y)
print(Point.x.offset,Point.x.size)
print(Point.y.offset,Point.y.size)
```

C Structure

```
struct Point
{
    float x;
    float y;
}
```

Output

```
1.5 0.0
0 4
4 4
```

ctypes

ctype.Union can be used to described a C/C++ union where all the fields start from the same address (occupy the same memory space).

Python 2.x/3.x

```
import ctypes

class CUnion(ctypes.Union):
        _fields_ = [("i",ctypes.c_long),("s",ctypes.c_short)]
p = CUnion()
p.s = 10
print(p.i,p.s)
print(CUnion.i)
print(CUnion.s)
```

C Union

```
union CUnion
{
    long i;
    short s;
}
```

Output

```
10 10
<Field type=c_long, ofs=0, size=4>
<Field type=c_short, ofs=0, size=2>
```

In this case, as p.i and p.s are 0 from the initialization moment (no values provided to the constructor), once p.s becomes 10, the value of p.i will be modified as well (in this case 10).

ctypes

In case of **ctypes.Structure** the operator * can be used to specify a 1-dimensional array of a specific type.

Python 2.x/3.x

```
import ctypes
class Numbers(ctypes.Structure):
        _fields_ = [("n",ctypes.c_long * 10),("count",ctypes.c_long)]
p = Numbers()
p.n[0] = 10
p.n[1] = 20
s = []
for i in range(0,10):s += [p.n[i]]
p.count = 3
print(s,p.count)
print(Numbers.n)
print(Numbers.count)
```

C Structure

```
struct Numbers {
    long n[10];
    long count;
}
```

Output

```
[10, 20, 0, 0, 0, 0, 0, 0, 0, 0] 3
<Field type=c_long_Array_10, ofs=0, size=40>
<Field type=c_long, ofs=40, size=4>
```

ctypes

Pointer can also be used in a structure (ctypes provide the POINTER member for this). The default value will be NULL so they will have to be instantiated).

Python 2.x/3.x

```
import ctypes
class Numbers(ctypes.Structure):
        _fields_ = [("n", ctypes.POINTER(ctypes.c_long)),
                    ("count",ctypes.c_long)]
p = Numbers()
p.n = (ctypes.c_long * 5)(1,2,3,4,5)
p.count = 5
s = []
for i in range(0,p.count):
        s += [p.n[i]]
print(s)
```

C Structure

```
struct Numbers {
    long *n;
    long count;
}
```

Output

```
[1, 2, 3, 4, 5]
```

ctypes

A member in a structure can also be another structure previously defined. This way complex data structure can be created.

Python 2.x/3.x

```
import ctypes

class Point(ctypes.Structure):
        _fields_ = [("x",ctypes.c_float),
                    ("y",ctypes.c_float)]
class Triangle(ctypes.Structure):
        _fields_ = [("pct",Point * 3)]
t = Triangle()
t.pct[0].x = 10
t.pct[0].y = 20
print (Triangle.pct)
```

C Structures

```
struct Point {
    float x;
    float y;
};
struct Triangle {
    Point pct[3];
};
```

Output

```
<Field type=Point_Array_3, ofs=0, size=24>
```

ctypes

Bit sets are also possible by adding the 3rd parameter to the tuple used to describer the _fields_ member.

Python 2.x/3.x

```
import ctypes

class BitField(ctypes.Structure):
        _fields_ = [("bit_0", ctypes.c_int, 1),
                    ("next_3_bytes", ctypes.c_int, 3)]
b = BitField()
b.bit_0 = 1
b.next_3_bytes = 2;
print ( BitField.bit_0, BitField.next_3_bytes )
```

Output

```
<Field type=c_long, ofs=0:0, bits=1>
<Field type=c_long, ofs=0:1, bits=3>
```

ctypes

Casting between basic types is also allowed (in the same manner as in C/C++). **ctype** provides a **cast** method that can be used to convert to a pointer of a different kind.

Python 2.x/3.x

```
import ctypes

i = (ctypes.c_int * 2)(0x11223344,0x12345678)
b = ctypes.cast(i,ctypes.POINTER(ctypes.c_byte))
for tr in range(0,8):
        print(hex(b[tr]))
```

C code

```
int i[2];
i[0] = 0x11223344;
i[1] = 0x12345678;
unsigned char *b = (unsigned char *)&i[0];
for (tr=0;tr<8;tr++)
      printf("%02x ",b[tr]);
```

Output

```
0x44
0x33
0x22
0x11
0x78
0x56
0x34
0x12
```

Itegrating with C/C++

Usage of ctypes:

- Create a library (usually with C/C++)
- Export some functions from that library
- Load that library in python using **ctypes.cdll.LoadLibrary** function. This function works differently in Windows and Linux systems (for full compatibility it is best to use the full path of the library)
- Once the library is loaded, exported functions will be available
- Exported function can be called using ctypes types as parameters and resulted value.

Usually, a separate module (a python module) that uses ctypes is created. This module load the C/C++ library, instantiate variables and offers a python interface that does not require ctypes knowledge.

C library

Example on how to use ctypes C/C++ types

C library (windows)

```
extern "C"
{
        int __declspec(dllexport) Add( int x, int y)
        {
                return x+y;
        }
}
```

Compile information:

- File name: test.cpp
- Compiler: cl.exe (Windows)
- Compiler command: cl.exe test.cpp /MD /link /OUT:"test.dll" /DLL /NODEFAULTLIB /NOENTRY
- Resulted library: test.dll

Python usage

Example on how to use ctypes C/C++ types

Python 3.x

```
import ctypes

lib = ctypes.cdll.LoadLibrary("test.dll")
x = lib.Add(10,20)
print (x)
```

Output

```
30
```

While this example works, some things must be pointed out:

- Exported function don't always have a signature (especially if <<extern "C" >> is used). Therefor, Python can not check and see if you send the correct parameters when you call that function.
- It is recommended to use ctypes types instead of python type
- Use wrapper against any exported functions to validate the parameters

Python usage

Recommended call:

Python 3.x

```
import ctypes
lib = ctypes.cdll.LoadLibrary("test.dll")
x = lib.Add(ctypes.c_int(10), ctypes.c_int(20))
print (x)
```

Output

```
30
```

Use *argtypes* and *restype* to describe the type of parameters such a method will receive and the result.

Python 3.x

```
import ctypes
lib = ctypes.cdll.LoadLibrary("test.dll")
fn_add = lib.Add
fn_add.argtypes = [ctypes.c_int,ctypes.c_int]
fn_add.restype = ctypes.c_int
print (fn_add(10,30))
```

Output

```
40
```

C library

A more complex example

C library (windows)

```
extern "C"
{
        int __declspec(dllexport) Add( int *x, int count)
        {
                int sum = 0;
                for (int tr=0;tr<count;tr++)
                        sum += x[tr];
                return sum;
        }
}
```

Python usage

Python 3.x

```
import ctypes

lib = ctypes.cdll.LoadLibrary("test.dll")
p = (ctypes.c_int * 5)(1,2,3,4,5)
x = lib.Add(ctypes.pointer(p),5)
print (x)
```

Output

```
15
```

```
import ctypes

lib = ctypes.cdll.LoadLibrary("test.dll")
p = (ctypes.c_int * 5)(1,2,3,4,5)
x = lib.Add(ctypes.byref(p),5)
print (x)
```

C library

Working with strings:

C library (windows)

```
extern "C"
{
	__declspec(dllexport) const char* ParseString(const char* str,
                                                  int &size) {
		while (((*str)!=0) && (((*str)<'0') || ((*str)>'9')))
			str++;
		if ((*str)==0) return 0; //NULL
		size = 0;
		while ((str[size]>='0') && (str[size]<='9'))
			size++;
		return str;
	}
}
```

Python usage

Python 3.x

```
import ctypes

lib = ctypes.cdll.LoadLibrary("test.dll")
size = ctypes.c_int()
lib.ParseString.restype = ctypes.c_char_p
result = lib.ParseString(ctypes.c_char_p(b"Telefon 123456 !!!") ,
                         ctypes.byref(size))
if result is None:       # NULL in C is converted to None in Python
        raise SystemExit
print (result, size.value)
s = ""
for i in range(0,size.value):
        s+=chr(result[i])
print (s)
```

Output

```
b'123456 !!!' 6
123456
```