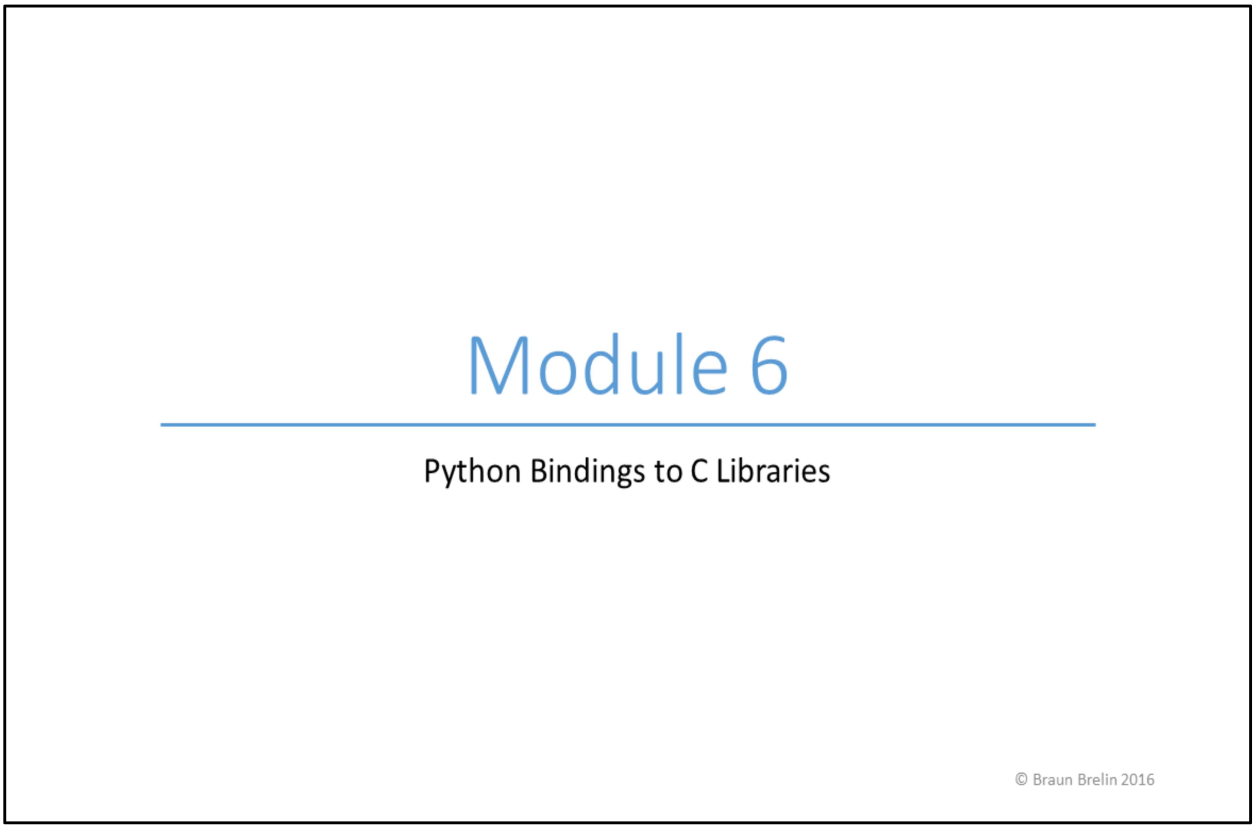
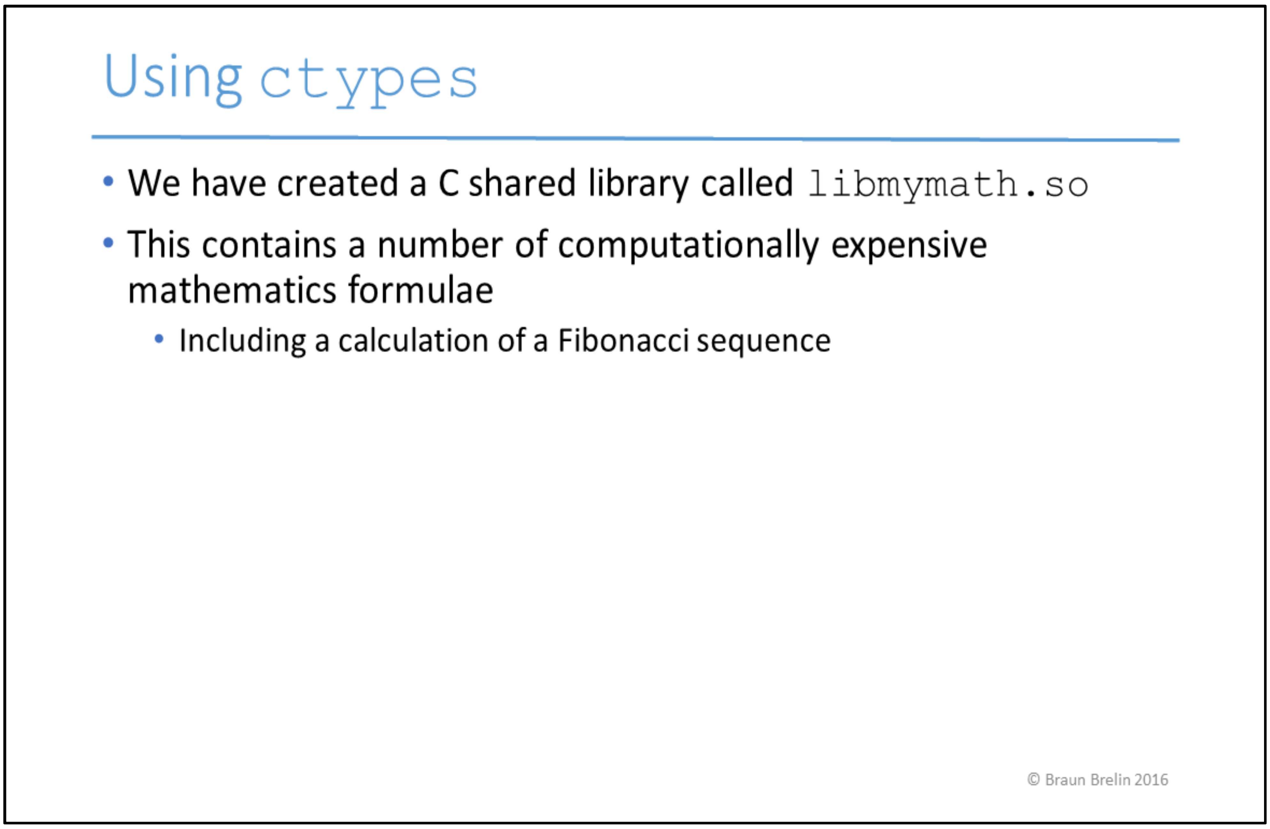


While Python is a wonderful language for writing applications, it is certainly slower than natively compiled code such as the C programming language. There are times when we'll want to call functions that are written in C for both performance and accessibility reasons.

We'll look at a python library called *ctypes* which we will use to call libraries and functions written in C.

This method uses Python's *Foreign Function Interface* library to allow us to directly call functions from a C shared library. In Windows, this is called a *Dynamic Link Library*.



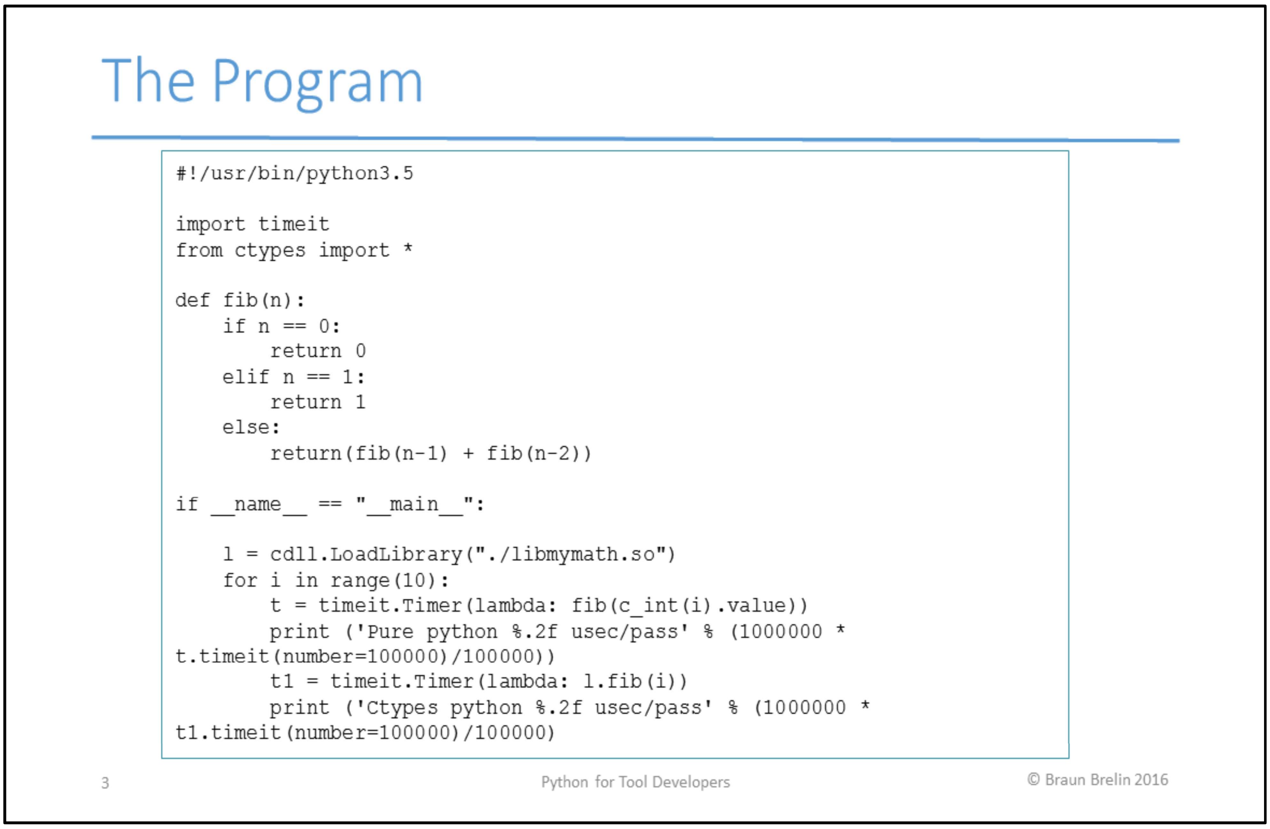
Using `ctypes`

Let's look at an example.

Here we have created a C shared library called `libmymath.so`. This library contains a number of computationally expensive mathematics formulae, including the fibonacci sequence calculation. Let's take a look at a Python program that does two things:

1. It calculates the fibonacci sequence on its own in Python
2. It calls the `libmymath.so` file, written in C, to do the same calculation

It also benchmarks the two functions using the `timeit` module.



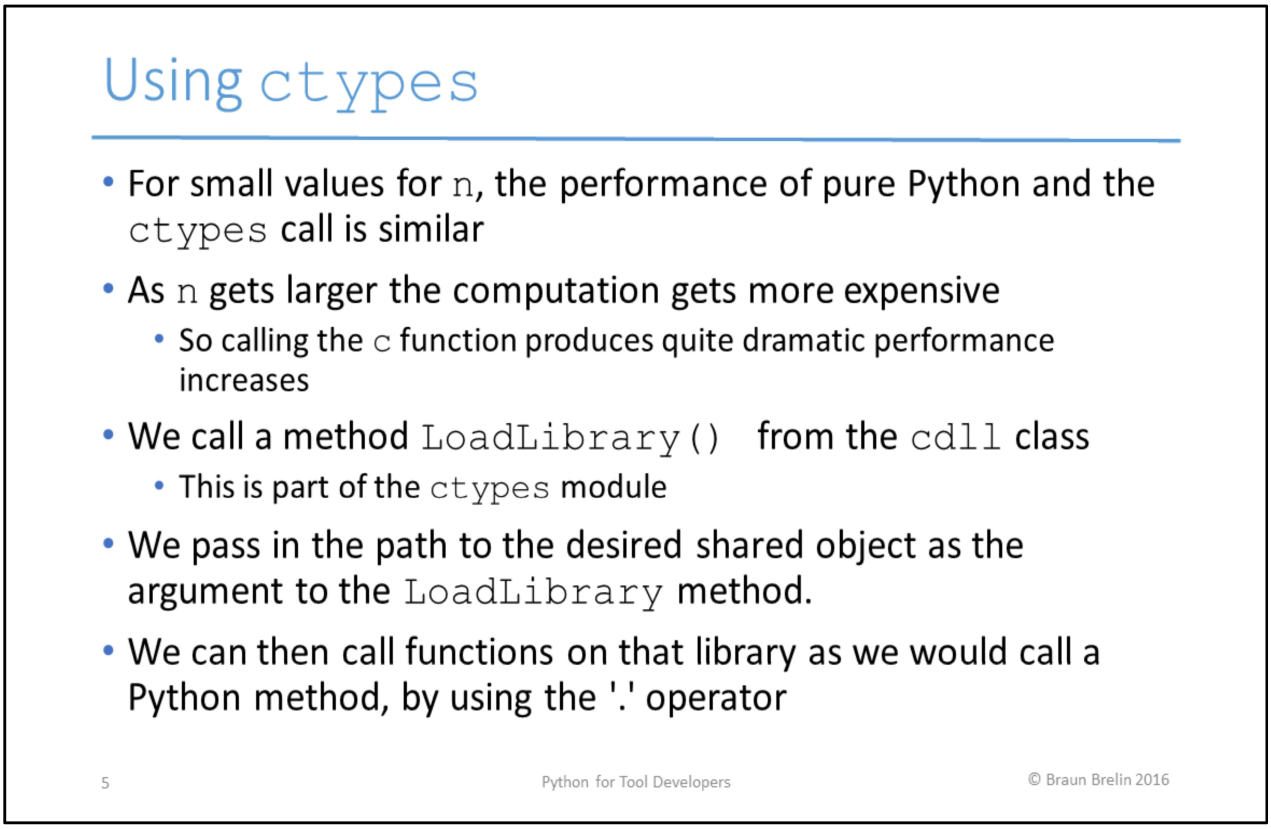
First let's look at the code.

```
#!/usr/bin/python3.5
 import timeit
from ctypes import *
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return(fib(n-1) + fib(n-2))
if __name__ == "__main__":
     l = cdll.LoadLibrary("./libmymath.so")
    for i in range(10):
        t = timeit.Timer(lambda: fib(c_int(i).value))
        print ('Pure python %.2f usec/pass' % (1000000 *
t.timeit(number=100000)/100000))
        t1 = timeit.Timer(lambda: l.fib(i))
        print ('Ctypes python %.2f usec/pass' % (1000000 *
t1.timeit(number=100000)/100000))
```

Here we see that we've declared a function *fib*, which calculates the sum of the first *n* numbers of the sequence in Python. We're also going to use `ctypes` to call the `libmymath.so fib` function to do the same thing.

Let's examine the output of this program.

```
Value of n = 0
Pure python 0.49 usec/pass
Value of n = 0
Ctypes python 0.50 usec/pass

Value of n = 1
Pure python 0.51 usec/pass
Value of n = 1
Ctypes python 0.50 usec/pass

Value of n = 2
Pure python 0.92 usec/pass
Value of n = 2
Ctypes python 0.52 usec/pass

Value of n = 3
Pure python 1.34 usec/pass
Value of n = 3
Ctypes python 0.53 usec/pass

Value of n = 4
Pure python 2.07 usec/pass
Value of n = 4
Ctypes python 0.55 usec/pass

Value of n = 5
Pure python 3.23 usec/pass
Value of n = 5
Ctypes python 0.58 usec/pass

Value of n = 6
Pure python 5.20 usec/pass
Value of n = 6
Ctypes python 0.64 usec/pass

Value of n = 7
Pure python 8.64 usec/pass
Value of n = 7
Ctypes python 0.72 usec/pass

Value of n = 8
Pure python 13.95 usec/pass
Value of n = 8
Ctypes python 0.84 usec/pass

Value of n = 9
Pure python 26.26 usec/pass
Value of n = 9
Ctypes python 1.03 usec/pass
```



Note that for small values for `n`, the performance of the pure Python and the `ctypes` call is quite similar. However, as `n` gets larger, and the computation gets more expensive, calling the c function produces quite dramatic performance increases.

If we examine the program, we see that we call a method `LoadLibrary()` from the `cdll` class which is part of the `ctypes` module. We pass in the path to the desired shared object as the argument to the `LoadLibrary` method. We can then call functions on that library as we would call a Python method, by using the '.' operator.

Some ctypes

- `ctypes` defines a number of intrinsic types that map to Python objects
- Here is a small sampling of them

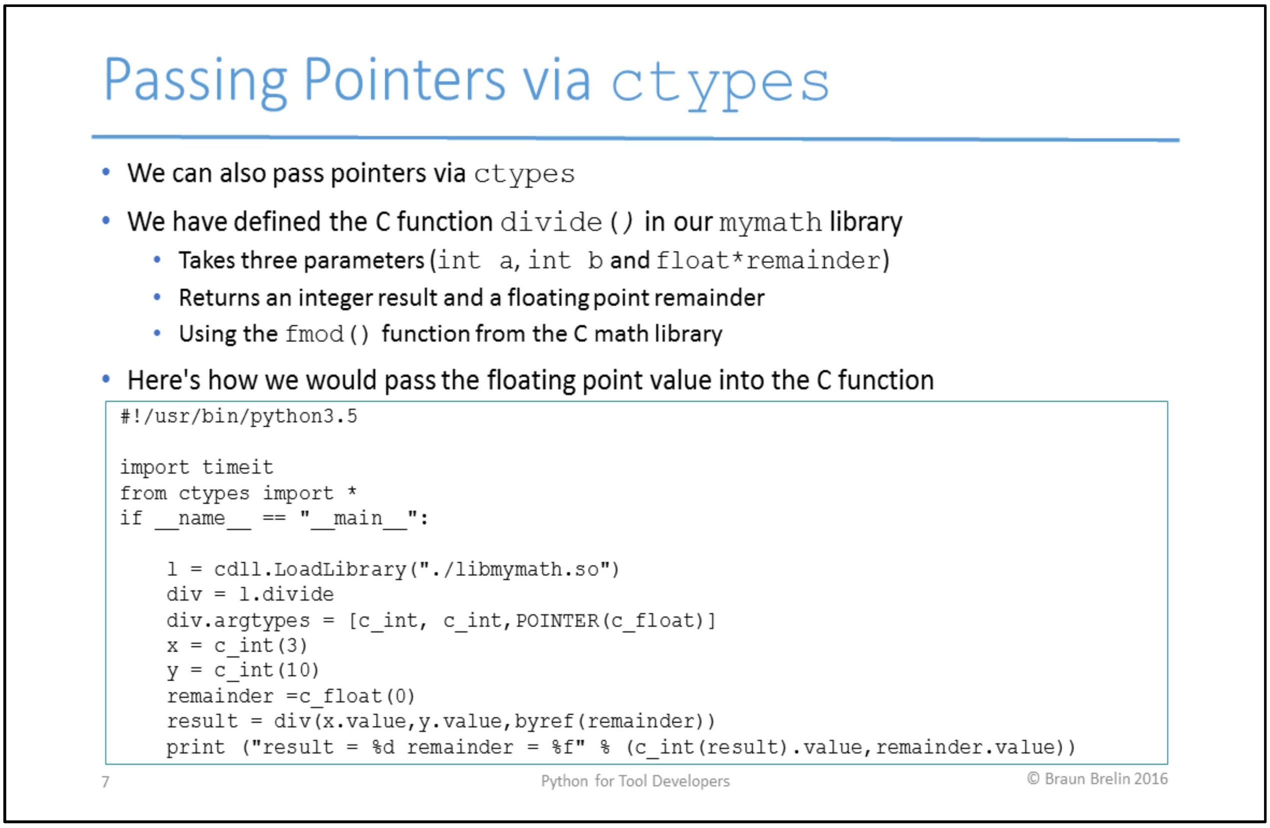
`ctypes` type	C type	Python type
c_bool	_Bool	bool
c_byte	char	One character byte object
c_short	short	int
c_int	int	int
c_long	long	int
c_float	float	float
c_double	Double	float
c_char_p	char * (NULL terminated)	bytes object or None
c_void_p	void *	int or None

6 Python for Tool Developers © Braun Brelin 2016

Ctypes defines a number of intrinsic types that map to Python objects. Here is a small sampling of them.

`ctypes` type	C type	Python type
c_bool	_Bool	bool
c_byte	char	One character byte object
c_short	short	int
c_int	int	int
c_long	long	int
c_float	float	float
c_double	Double	float
c_char_p	char * (NULL terminated)	bytes object or None
c_void_p	void *	int or None

This is an incomplete list. For the full table of C to Python intrinsic mappings refer to the Python.org documentation on `ctypes`.

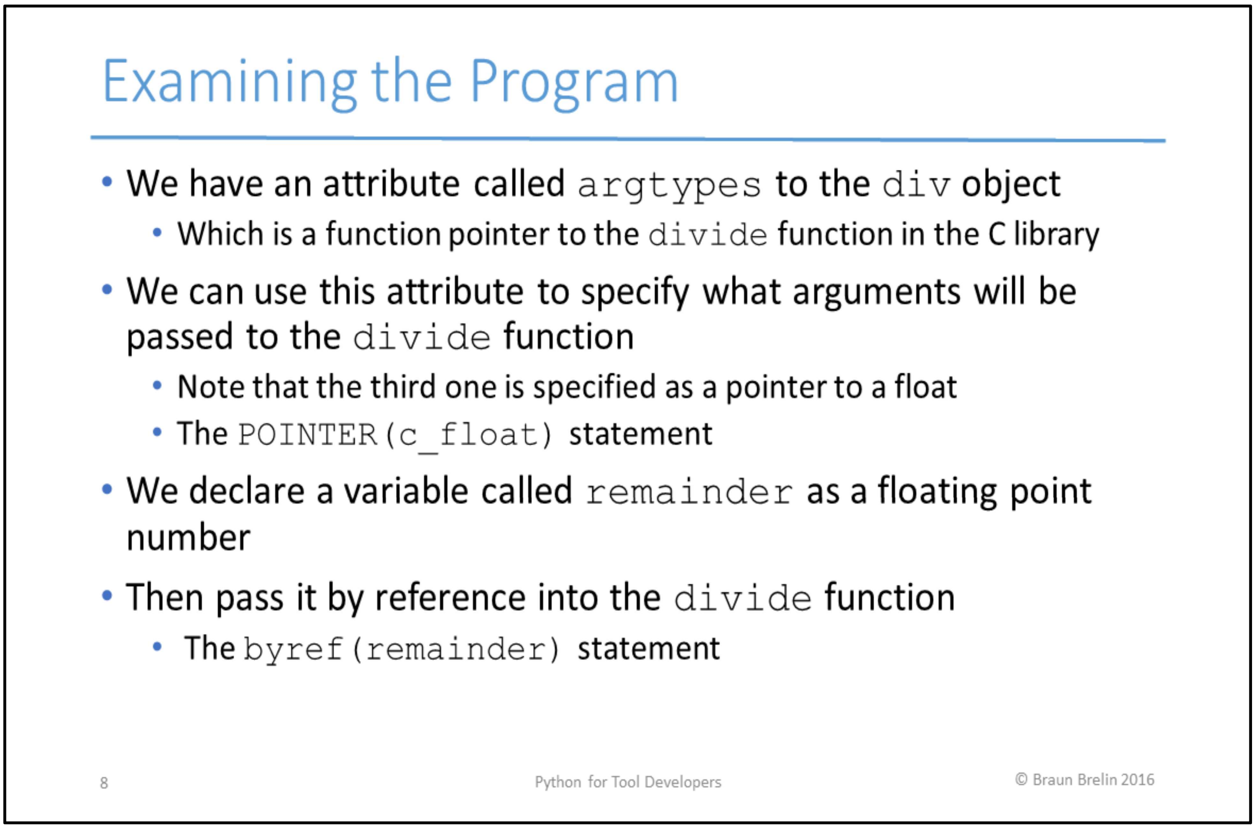


We can also pass pointers via `ctypes`.

Consider the following C function `divide()` which we have defined in our `mymath` library as taking three parameters (`int a, int b` and `float * remainder`) and returning an integer result and a floating point remainder using the `fmod()` function from the C math library.

Here's how we would pass the floating point value into the C function.

```
#!/usr/bin/python3.5

import timeit
from ctypes import *
if __name__ == "__main__":

    l = cdll.LoadLibrary("./libmymath.so")
    div = l.divide
    div.argtypes = [c_int, c_int,POINTER(c_float)]
    x = c_int(3)
    y = c_int(10)
    remainder =c_float(0)
    result = div(x.value,y.value,byref(remainder))
    print ("result = %d remainder = %f" %
(c_int(result).value,remainder.value))
```



Let's look at this program.

We note that we have an attribute to the `div` object (which is a function pointer to the `divide` function in the C library) called `argtypes`. We can use this attribute to specify what arguments will be passed to the `divide` function. Note that the third one is specified as a pointer to a float (The `POINTER(c_float)` statement).

We declare a variable called `remainder` as a floating point number and then pass it by reference into the `divide` function (the `byref(remainder)` statement).