



جامعة الانبار
كلية علوم الحاسوب وتكنولوجيا المعلومات
قسم أنظمة شبكات الحاسوب

برمجة كيانية OOP
المرحلة الثانية
الفصل الدراسي الأول والثاني

مدرس المادة
م.د. سميه عبدالله حمد

Object Oriented Programming (OOP)

Pointers in Python

Why Doesn't Python Have Pointers?

Could pointers in Python exist natively? Pointers encourage implicit changes rather than explicit. Often, they are complex instead of simple, especially for beginners. Even worse, they do something really dangerous like read from a section of memory you were not supposed to.

Python tends to try to abstract away implementation details like memory addresses from its users. Python often focuses on usability instead of speed. As a result, pointers in Python don't really make sense. But Python does, by default, give you some of the benefits of using pointers.

Understanding pointers in Python requires a short detour into Python's implementation details. Specifically, you'll need to understand:

Immutable vs mutable objects

Python variables/names

Objects in Python

In Python, everything is an object. For proof, you can explore using `isinstance()`:

```
>>> isinstance(1, object)
True
>>> isinstance(list(), object)
True
>>> isinstance(True, object)
True
>>> def foo():
...     pass
...
>>> isinstance(foo, object)
True
```

This code shows you that everything in Python is indeed an object. Each object contains at least three pieces of data:

Dr. Sumaya Abdulla Hamad

2022

Object Oriented Programming (OOP)

Reference count

Type

Value

The reference count is for memory management.

The type is used to ensure type safety during runtime. Finally, there's the value, which is the actual value associated with the object.

Not all objects are the same though. There is one other important distinction you'll need to understand: immutable vs mutable objects. Understanding the difference between the types of objects really helps clarify pointers in Python.

Immutable vs Mutable Objects

In Python, there are two types of objects:

Immutable objects can't be changed.

Mutable objects can be changed.

Understanding this difference is the first key to navigating the landscape of pointers in Python. Here's a breakdown of common types and whether or not they are mutable or immutable:

Type Immutable?

int Yes

float Yes

bool Yes

complex Yes

tuple Yes

frozenset Yes

str Yes

list No

set No

dict No

Object Oriented Programming (OOP)

As you can see, lots of commonly used primitive types are immutable. You'll need a couple of tools from the Python standard library:

id() returns the object's memory address.

is returns True if and only if two objects have the same memory address.

Once again, you can use these:

```
>>> x = 5
>>> id(x)
94529957049376
```

In the above code, you have assigned the value 5 to x. If you tried to modify this value with addition, then you'd get a new object:

```
>>> x += 1
>>> x
6
>>> id(x)
94529957049408
```

Even though the above code appears to modify the value of x, you're getting a new object as a response.

The str type is also immutable:

```
>>> s = "real_python"
>>> id(s)
140637819584048
>>> s += "_rocks"
>>> s
'real_python_rocks'
>>> id(s)
140637819609424
```

Again, s ends up with a different memory addresses after the += operation.

Object Oriented Programming (OOP)

Bonus: The += operator translates to various method calls.

For some objects like list, += will translate into `__iadd__()` (in-place add). This will modify self and return the same ID. However, str and int don't have these methods and result in `__add__()` calls instead of `__iadd__()`.

Trying to directly mutate the string s results in an error:

```
>>> s[0] = "R"
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: 'str' object does not support item assignment

The above code fails, and Python indicates that str doesn't support this mutation, which is in line with the definition that the str type is immutable.

Contrast that with a mutable object, like list:

```
>>> my_list = [1, 2, 3]
```

```
>>> id(my_list)
```

```
140637819575368
```

```
>>> my_list.append(4)
```

```
>>> my_list
```

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

```
>>> id(my_list)
```

```
140637819575368
```

This code shows a major difference in the two types of objects. `my_list` has an id originally. Even after 4 is appended to the list, `my_list` has the same id. This is because the list type is mutable.

Another way to demonstrate that the list is mutable is with assignment:

```
>>> my_list[0] = 0
```

```
>>> my_list
```

```
[0, 2, 3, 4]
```

```
>>> id(my_list)
```

```
140637819575368
```

Object Oriented Programming (OOP)

In this code, you mutate `my_list` and set its first element to 0. However, it maintains the same id even after this assignment. With mutable and immutable objects, the next step to Python is understanding Python's variable ecosystem.

Understanding Variables

Python variables are fundamentally different than variables in C or C++. In fact, Python doesn't even have variables. Python has names, not variables.

Most of the time, it's perfectly acceptable to think about Python names as variables, but understanding the difference is important. This is especially true when you're navigating the tricky subject of pointers in Python.

We can take a look at how variables work in C, what they represent, and then contrast that with how names work in Python.

Variables in C

Let's say you had the following code that defines the variable `x`:

```
int x = 2337;
```

This one line of code has several, distinct steps when executed:

- Allocate enough memory for an integer
- Assign the value 2337 to that memory location
- Indicate that `x` points to that value

Shown in a simplified view of memory, it might look like this:

X	
Location	0x7f1
Value	2337

In-Memory representation of X (2337)

Here, you can see that the variable `x` has a fake memory location of 0x7f1 and the value 2337. If, later in the program, you want to change the value of `x`, you can do the following:

```
x = 2338;
```

Object Oriented Programming (OOP)

The above code assigns a new value (2338) to the variable x, thereby overwriting the previous value. This means that the variable x is mutable. The updated memory layout shows the new value:

X	
Location	0x7f1
Value	2338

New In-Memory representation of X (2338)

Notice that the location of x didn't change, just the value itself. This is a significant point. It means that x is the memory location, not just a name for it.

Another way to think of this concept is in terms of ownership. In one sense, x owns the memory location. x is, at first, an empty box that can fit exactly one integer in which integer values can be stored.

When you assign a value to x, you're placing a value in the box that x owns. If you wanted to introduce a new variable (y), you could add this line of code:

```
int y = x;
```

This code creates a new box called y and copies the value from x into the box. Now the memory layout will look like this:

X		Y	
Location	0x7f1	Location	0x7f5
Value	2338	Value	2338

In-Memory representation of X (2338) and Y (2338)

Notice the new location 0x7f5 of y. Even though the value of x was copied to y, the variable y owns some new address in memory. Therefore, you could overwrite the value of y without affecting x:

```
y = 2339;
```

Now the memory layout will look like this:

Object Oriented Programming (OOP)

X		Y	
Location	0x7f1	Location	0x7f5
Value	2338	Value	2339

Updated representation of Y (2339)

Again, you have modified the value at y, but not its location. In addition, you have not affected the original x variable at all. This is in stark contrast with how Python names work.

Names in Python

Python does not have variables. It has names. Yes, this is a pedantic point, and you can certainly use the term variables as much as you like. It is important to know that there is a difference between variables and names.

Let's take the equivalent code from the above C example and write it in Python:

```
>>> x = 2337
```

Much like in C, the above code is broken down into several distinct steps during execution:

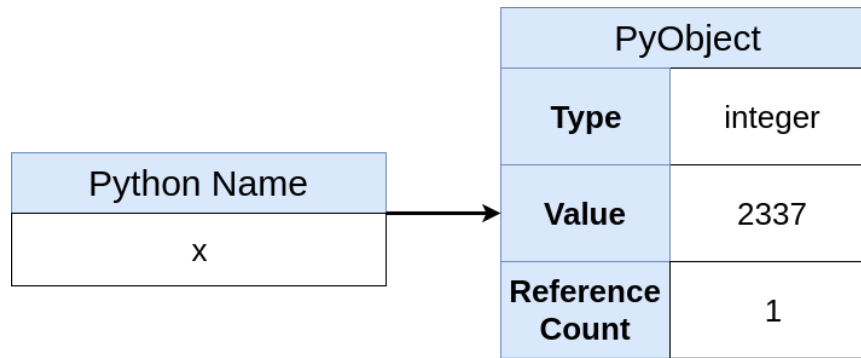
- Create a PyObject
- Set the typecode to integer for the PyObject
- Set the value to 2337 for the PyObject
- Create a name called x
- Point x to the new PyObject
- Increase the refcount of the PyObject by 1

Note: The PyObject is not the same as Python's object. It's specific to CPython and represents the base structure for all Python objects.

PyObject is defined as a C struct, so if you're wondering why you can't call typecode or refcount directly, it's because you don't have access to the structures directly. Method calls like `sys.getrefcount()` can help get some internals.

In memory, it might look something like this:

Object Oriented Programming (OOP)



Python In-Memory representation of X (2337)

You can see that the memory layout is vastly different than the C layout from before. Instead of `x` owning the block of memory where the value 2337 resides, the newly created Python object owns the memory where 2337 lives. The Python name `x` doesn't directly own any memory address in the way the C variable `x` owned a static slot in memory.

If you were to try to assign a new value to `x`, you could try the following:

```
>>> x = 2338
```

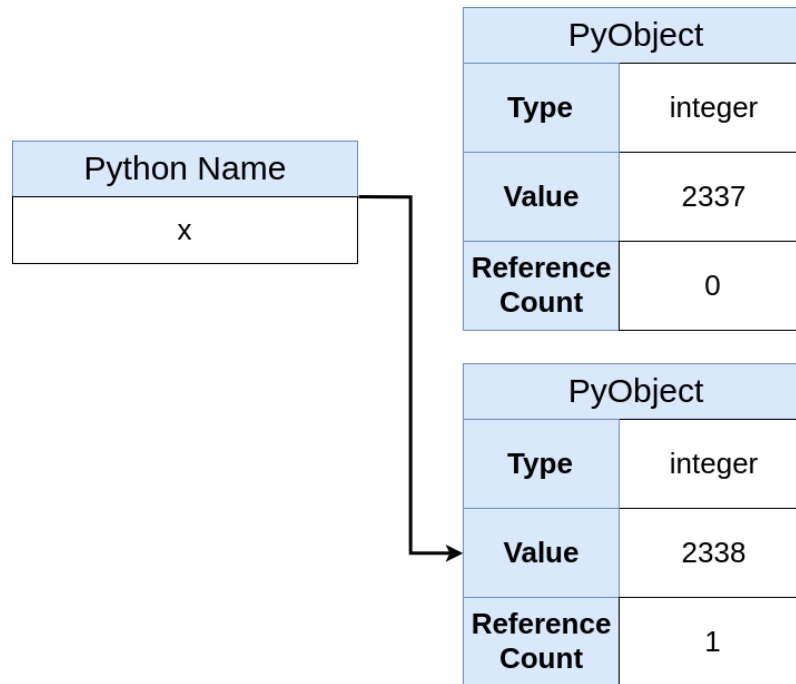
What's happening here is different than the C equivalent, but not too different from the original bind in Python.

This code:

- Creates a new PyObject
- Sets the typecode to integer for the PyObject
- Sets the value to 2338 for the PyObject
- Points `x` to the new PyObject
- Increases the refcount of the new PyObject by 1
- Decreases the refcount of the old PyObject by 1

Now in memory, it would look something like this:

Object Oriented Programming (OOP)



Python Name Pointing to new object (2338)

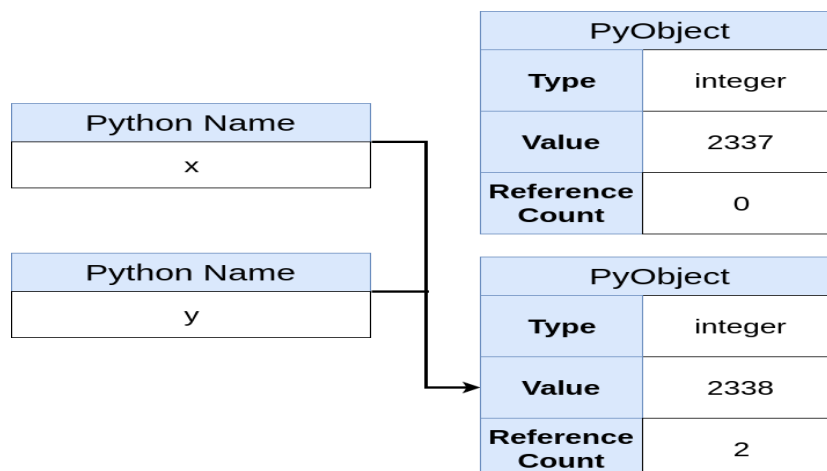
This diagram helps illustrate that `x` points to a reference to an object and doesn't own the memory space as before. It also shows that the `x = 2338` command is not an assignment, but rather binding the name `x` to a reference.

In addition, the previous object (which held the 2337 value) is now sitting in memory with a ref count of 0 and will get cleaned up by the garbage collector.

You could introduce a new name, `y`, to the mix as in the C example:

```
<<<y = x
```

In memory, you would have a new name, but not necessarily a new object:



X and Y Names pointing to 2338

Object Oriented Programming (OOP)

Now you can see that a new Python object has not been created, just a new name that points to the same object. Also, the object's refcount has increased by one. You could check for object identity equality to confirm that they are the same:

```
<<<y is x
```

True

The above code indicates that x and y are the same object. Make no mistake though: y is still immutable.

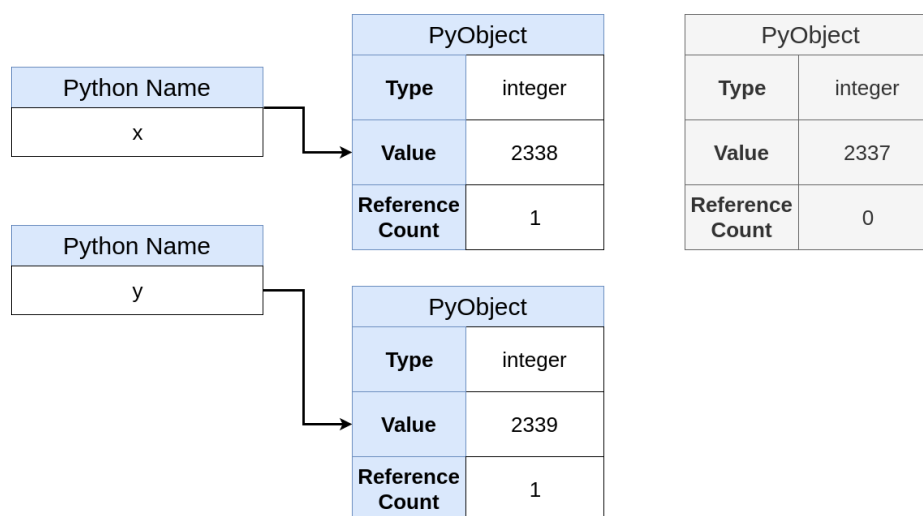
For example, you could perform addition on y:

```
<<<y += 1
```

```
<<<y is x
```

False

After the addition call, you are returned with a new Python object. Now, the memory looks like this:



x name and y name different objects

A new object has been created, and y now points to the new object. Interestingly, this is the same end-state if you had bound y to 2339 directly:

```
<<<y = 2339
```

The above statement results in the same end-memory state as the addition. To recap, in Python, you don't assign variables. Instead, you bind names to references.

Object Oriented Programming (OOP)

A Note on Intern Objects in Python

Now we understand how Python objects get created and names get bound to those objects, its time to understand the name of interned objects.

Suppose you have the following Python code:

```
>>>
```

```
>>> x = 1000
```

```
>>> y = 1000
```

```
>>> x is y
```

```
True
```

As above, x and y are both names that point to the same Python object. But the Python object that holds the value 1000 is not always guaranteed to have the same memory address. For example, if you were to add two numbers together to get 1000, you would end up with a different memory address:

```
>>>
```

```
>>> x = 1000
```

```
>>> y = 499 + 501
```

```
>>> x is y
```

```
False
```

This time, the line x is y returns False. This is confusing. Here are the steps that occur when this code is executed:

1. Create Python object(1000)
2. Assign the name x to that object
3. Create Python object (499)
4. Create Python object (501)
5. Add these two objects together
6. Create a new Python object (1000)
7. Assign the name y to that object

Technical Note: The above steps occur only when this code is executed inside a REPL (Read-Eval-Print Loop, or REPL, is a computer environment where user inputs are read and evaluated, and then the results are returned to the user). If you were to take the example above, paste it into a file, and run the file, then you would find that the x is y line would return True.

The core Python developers decided to make a few optimizations. These optimizations result in behavior that can be surprising to newcomers:

```
>>>
```

Object Oriented Programming (OOP)

```
>>> x = 20
>>> y = 19 + 1
>>> x is y
True
```

In this example, you see nearly the same code as before, except this time the result is True. This is the result of interned objects. Python pre-creates a certain subset of objects in memory and keeps them in the global [namespace](#) for everyday use.

Which objects depend on the implementation of Python?

CPython 3.7 interns the following:

1. Integer numbers between -5 and 256
2. Strings that contain ASCII letters, digits, or underscores only

The reasoning behind this is that these variables are extremely likely to be used in many programs. By interning these objects, Python prevents memory allocation calls for consistently used objects.

Strings that are less than 20 characters and contain ASCII letters, digits, or underscores will be interned. The reasoning behind this is that these are assumed to be some kind of identity:

```
>>>
>>> s1 = "realpython"
>>> id(s1)
140696485006960
>>> s2 = "realpython"
>>> id(s2)
140696485006960
>>> s1 is s2
True
```

Here you can see that s1 and s2 both point to the same address in memory. If you were to introduce a non-ASCII letter, digit, or underscore, then you would get a different result:

```
>>>
>>> s1 = "Real Python!"
>>> s2 = "Real Python!"
>>> s1 is s2
False
```

Object Oriented Programming (OOP)

Because this example has an exclamation mark (!) in it, these strings are not interned and are different objects in memory.

Interned objects are often a source of confusion. Just remember, if you're ever in doubt, that you can always use `id()` and `is` to determine object equality.

Simulating Pointers in Python

Just because pointers in Python don't exist natively doesn't mean you can't get the benefits of using pointers. In fact, there are multiple ways to simulate pointers in Python. You'll learn two in this section:

1. Using mutable types as pointers
2. Using custom Python objects

Using Mutable Types as Pointers

You've already learned about mutable types. Because these objects are mutable, you can treat them as if they were pointers to simulate pointer behavior. Suppose you wanted to replicate the following c code:

```
void add_one(int *x) {  
    *x += 1;  
}
```

This code takes a pointer to an integer (*x) and then increments the value by one. Here is a main function to exercise the code:

```
#include <stdio.h>  
  
int main(void) {  
    int y = 2337;  
    printf("y = %d\n", y);  
    add_one(&y);  
    printf("y = %d\n", y);  
    return 0;  
}
```

In the above code, you assign 2337 to y, [print out](#) the current value, increment the value by one, and then print out the modified value. The output of executing this code would be the following:

```
y = 2337  
y = 2338
```

One way to replicate this type of behavior in Python is by using a mutable type. Consider using a list and modifying the first element:

Object Oriented Programming (OOP)

```
>>>
```

```
>>> def add_one(x):  
...     x[0] += 1  
...  
>>> y = [2337]  
>>> add_one(y)  
>>> y[0]  
2338
```

Here, `add_one(x)` accesses the first element and increments its value by one. Using a list means that the end result appears to have modified the value. So pointers in Python do exist? Well, no. This is only possible because list is a mutable type. If you tried to use a tuple, you would get an error:

```
>>>
```

```
>>> z = (2337,)
>>> add_one(z)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 2, in add_one

TypeError: 'tuple' object does not support item assignment

The above code demonstrates that tuple is immutable. Therefore, it does not support item assignment. list is not the only mutable type. Another common approach to mimicking pointers in Python is to use a dict.

Let's say you had an application where you wanted to keep track of every time an interesting event happened. One way to achieve this would be to create a dict and use one of the items as a counter:

```
>>>
```

```
>>> counters = {"func_calls": 0}
>>> def bar():
...     counters["func_calls"] += 1
...
>>> def foo():
...     counters["func_calls"] += 1
...     bar()
...
>>> foo()
>>> counters["func_calls"]
```

Object Oriented Programming (OOP)

2

In this example, the counters [dictionary](#) is used to keep track of the number of function calls. After you call `foo()`, the counter has increased to 2 as expected. All because dict is mutable.

Keep in mind, this is only *simulates* pointer behavior and does not directly map to true pointers in C or C++. That is to say, these operations are more expensive than they would be in C or C++.

Using Python Objects

The dict option is a great way to emulate pointers in Python, but sometimes it gets tedious to remember the key name you used. This is especially true if you're using the dictionary in various parts of your application. This is where a custom Python class can really help.

To build on the last example, assume that you want to track metrics in your application. Creating a class is a great way to abstract the pesky details:

```
class Metrics(object):
    def __init__(self):
        self._metrics = {
            "func_calls": 0,
            "cat_pictures_served": 0,
        }
```

This code defines a Metrics class. This class still uses a dict for holding the actual data, which is in the `_metrics` member variable. This will give you the mutability you need. Now you just need to be able to access these values. One nice way to do this is with properties:

```
class Metrics(object):
    # ...

    @property
    def func_calls(self):
        return self._metrics["func_calls"]

    @property
    def cat_pictures_served(self):
        return self._metrics["cat_pictures_served"]
```


Object Oriented Programming (OOP)

This code makes use of `@property`. If you're not familiar with decorators, you can check out this [Primer on Python Decorators](#). The `@property` decorator here allows you to access `func_calls` and `cat_pictures_served` as if they were attributes:

```
>>>
```

```
>>> metrics = Metrics()
>>> metrics.func_calls
0
>>> metrics.cat_pictures_served
0
```

The fact that you can access these names as attributes means that you abstracted the fact that these values are in a dict. You also make it more explicit what the names of the attributes are. Of course, you need to be able to increment these values:

```
class Metrics(object):
    # ...

    def inc_func_calls(self):
        self._metrics["func_calls"] += 1

    def inc_cat_pics(self):
        self._metrics["cat_pictures_served"] += 1
```

You have introduced two new methods:

1. `inc_func_calls()`
2. `inc_cat_pics()`

These methods modify the values in the metrics dict. You now have a class that you modify as if you're modifying a pointer:

```
>>>
```

```
>>> metrics = Metrics()
>>> metrics.inc_func_calls()
>>> metrics.inc_func_calls()
>>> metrics.func_calls
2
```

Here, you can access `func_calls` and call `inc_func_calls()` in various places in your applications and simulate pointers in Python. This is useful when you have something like metrics that need to be used and updated frequently in various parts of your applications.

Object Oriented Programming (OOP)

Note: In this class in particular, making `inc_func_calls()` and `inc_cat_pics()` explicit instead of using `@property.setter` prevents users from setting these values to an arbitrary int or an invalid value like a dict.

Here's the full source for the Metrics class:

```
class Metrics(object):
    def __init__(self):
        self._metrics = {
            "func_calls": 0,
            "cat_pictures_served": 0,
        }

    @property
    def func_calls(self):
        return self._metrics["func_calls"]

    @property
    def cat_pictures_served(self):
        return self._metrics["cat_pictures_served"]

    def inc_func_calls(self):
        self._metrics["func_calls"] += 1

    def inc_cat_pics(self):
        self._metrics["cat_pictures_served"] += 1
```

Real Pointers With ctypes

Okay, so maybe there are pointers in Python, specifically CPython. Using the builtin ctypes module, you can create real C-style pointers in Python. If you are unfamiliar with ctypes, then you can take a look at [Extending Python With C Libraries and the “ctypes” Module](#).

The real reason you would use this is if you needed to make a function call to a C library that requires a pointer. Let's go back to the `add_one()` C-function from before:

```
void add_one(int *x) {
    *x += 1;
}
```

Here again, this code is incrementing the value of `x` by one. To use this, first compile it into a shared object. Assuming the above file is stored in `add.c`, you could accomplish this with `gcc`:

Object Oriented Programming (OOP)

```
$ gcc -c -Wall -Werror -fpic add.c
```

```
$ gcc -shared -o libadd1.so add.o
```

The first command compiles the C source file into an object called add.o. The second command takes that unlinked object file and produces a shared object called libadd1.so.

libadd1.so should be in your current directory. You can load it into Python using ctypes:

```
>>>
```

```
>>> import ctypes
```

```
>>> add_lib = ctypes.CDLL("./libadd1.so")
```

```
>>> add_lib.add_one
```

```
<_FuncPtr object at 0x7f9f3b8852a0>
```

The ctypes.CDLL code returns an object that represents the libadd1 shared object. Because you defined add_one() in this shared object, you can access it as if it were any other Python object. Before you call the function though, you should specify the function signature. This helps Python ensure that you pass the right type to the function.

In this case, the function signature is a pointer to an integer. ctypes will allow you to specify this using the following code:

```
>>>
```

```
>>> add_one = add_lib.add_one
```

```
>>> add_one.argtypes = [ctypes.POINTER(ctypes.c_int)]
```

In this code, you're setting the function signature to match what C is expecting. Now, if you were to try to call this code with the wrong type, then you would get a nice warning instead of undefined behavior:

```
>>>
```

```
>>> add_one(1)
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
ctypes.ArgumentError: argument 1: <class 'TypeError'>: \
expected LP_c_int instance instead of int
```

Python throws an error, explaining that add_one() wants a pointer instead of just an integer. Luckily, ctypes has a way to pass pointers to these functions. First, declare a C-style integer:

```
>>>
```

```
>>> x = ctypes.c_int()
```

Object Oriented Programming (OOP)

```
>>> x  
c_int(0)
```

The above code creates a C-style integer `x` with a value of 0. `ctypes` provides the handy `byref()` to allow passing a variable by reference.

Note: The term **by reference** is opposed to passing a variable **by value**.

When passing by reference, you're passing the reference to the original variable, and thus modifications will be reflected in the original variable. Passing by value results in a copy of the original variable, and modifications are not reflected in the original.

For more information on passing by reference in Python, check out [Pass by Reference in Python: Background and Best Practices](#).

You can use this to call `add_one()`:

```
>>>
```

```
>>> add_one(ctypes.byref(x))  
998793640  
>>> x  
c_int(1)
```

Nice! Your integer was incremented by one. Congratulations, you have successfully used real pointers in Python.

Conclusion

You now have a better understanding of the intersection between Python objects and pointers. Even though some of the distinctions between names and variables seem pedantic, fundamentally understanding these key terms expands your understanding of how Python handles variables.

You've also learned some excellent ways to simulate pointers in Python:

- Utilizing mutable objects as low-overhead pointers
- Creating custom Python objects for ease of use
- Unlocking real pointers with the `ctypes` module

These methods allow you to simulate pointers in Python without sacrificing the memory safety that Python provides.