



# Python Programming for Beginners

**-Advanced-**

## Python Data Structure

2022학년도 2학기  
Suk-Hwan Lee

# Artificial Intelligence

Creating the Future

**Dong-A University**

Division of **C**omputer **E**ngineering &  
**A**rtificial **I**ntelligence

## References

- Pointers in Python: What's the Point? :  
<https://realpython.com/pointers-in-python/>
- Data size in memory vs. on disk :  
<https://stackoverflow.com/questions/22999766/data-size-in-memory-vs-on-disk>
- Why python is Slow: Looking Under the Hood :  
<https://nbviewer.jupyter.org/url/jakevdp.github.io/downloads/notebooks/WhyPythonIsSlow.ipynb>  
<https://medium.com/@cookatrice/why-python-is-slow-looking-under-the-hood-7126baf936d7>  
<https://www.fatalerrors.org/a/why-python-is-slow-looking-under-the-hood.html>
- Understanding Data Types in Python :  
<https://jakevdp.github.io/PythonDataScienceHandbook/02.01-understanding-data-types.html>
- Python Data Science Handbook



## Pointers in Python: What's the Point?

[참고] <https://realpython.com/pointers-in-python/>

### 학습 목표

- Learn why pointers in Python don't exist
  - Explore the difference between **C variables** and **Python names**
  - Simulate pointers in Python
  - Experiment with real pointers using **ctypes**
- 
- 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 open up a REPL(Read-Eval-Print Loop) and explore using **isinstance()**:

```
isinstance(1, object)
```

True

```
isinstance(list(), object)
```

True

```
isinstance(True, object)
```

True

```
def foo():  
    pass
```

```
isinstance(foo, object)
```

True

- Each object contains at least three pieces of data:
  - Reference count
  - Type
  - Value
- The **reference count** is for memory management. For an in-depth look at the internals of memory management in Python, you can read [Memory Management in Python](#).
- The **type** is used at the **CPython layer** to ensure type safety during runtime.
- The **value** is the actual value associated with the object.

## Pointers in Python: What's the Point?

### Immutable vs Mutable Objects

- **Immutable objects** can't be changed. : **int, float, bool, complex, tuple, frozenset, str**
- **Mutable objects** can be changed. : **list, set, dict**

- `id()` returns the object's memory address.
- `is` returns True if and only if two objects have the same memory address.

Python

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

Python

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

- ✓ If you tried to modify this value with addition, then you'd get a **new object**:

**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__()`.

For more detailed information, check out the Python [data model docs](#).

- The `str` type is also immutable:

Python

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

- ✓ `s` ends up with a *different memory address* after the `+=` operation.

Python

```
>>> s[0] = "R"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

## Pointers in Python: What's the Point?

- a mutable object, like list:

Python

```
>>> my_list = [1, 2, 3]
>>> id(my_list)
140637819575368
>>> my_list.append(4)
>>> my_list
[1, 2, 3, 4]
>>> id(my_list)
140637819575368
```

- ✓ 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*:

Python

```
>>> my_list[0] = 0
>>> my_list
[0, 2, 3, 4]
>>> id(my_list)
140637819575368
```

### ❖ PyObject

- 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.

<https://github.com/python/cpython/blob/v3.7.3/Include/object.h#L101>

```
typedef struct _object {
    PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;
```

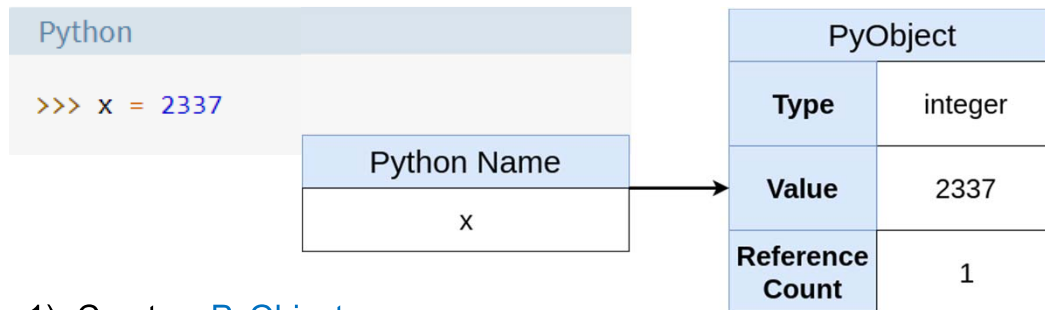
```
/* Nothing is actually declared to be a PyObject, but every pointer to
 * a Python object can be cast to a PyObject*. This is inheritance built
 * by hand. Similarly every pointer to a variable-size Python object can,
 * in addition, be cast to PyVarObject*.
 */
```

## Pointers in Python: What's the Point?

### Understanding Variables

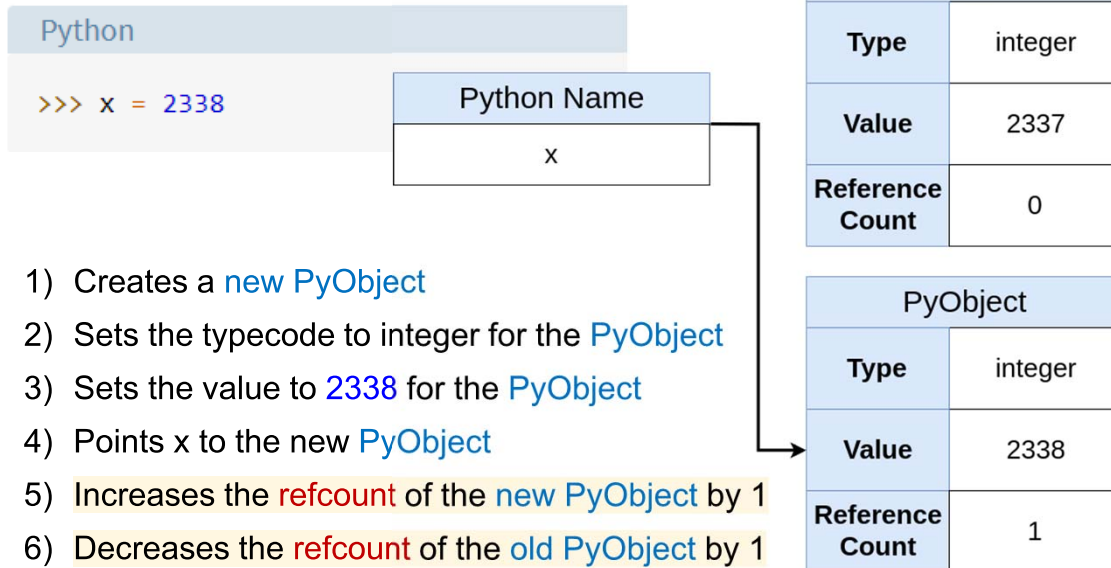
- Python doesn't even have *variables*. Python has *names*, not *variables*.
- Most of the time, it's perfectly acceptable to think about Python names as variables, but understanding the difference is important. This is especially true when you're navigating the tricky subject of pointers in Python.

#### 1) Names in Python



- 1) Create a **PyObject**
- 2) Set the typecode to integer for the **PyObject**
- 3) Set the value to 2337 for the **PyObject**
- 4) Create a name called `x`
- 5) Point `x` to the new **PyObject**
- 6) Increase the **refcount** of the **PyObject** by 1

- Assign a new value to `x`



- 1) Creates a new **PyObject**
- 2) Sets the typecode to integer for the **PyObject**
- 3) Sets the value to 2338 for the **PyObject**
- 4) Points `x` to the new **PyObject**
- 5) Increases the **refcount** of the new **PyObject** by 1
- 6) Decreases the **refcount** of the old **PyObject** by 1

- `x` points to a reference to an object and doesn't own the memory space as before
- `x = 2338` command is not an assignment, but rather binding the name `x` to a reference.
- 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**.

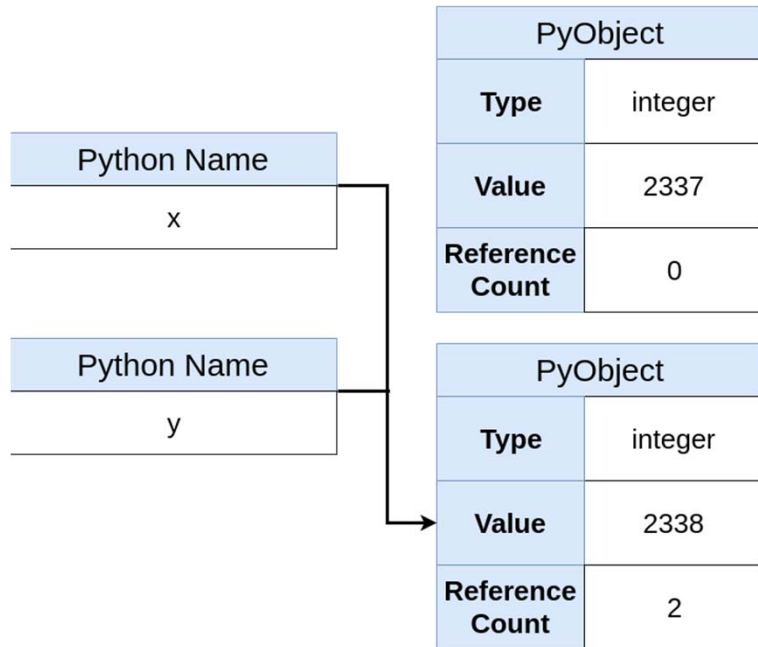
## Pointers in Python: What's the Point?

- Introduce a new name, `y`, to the mix as in the C example

Python

```
>>> y = x
```

- 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.
- `x` and `y` are the same object. `y` is still immutable.



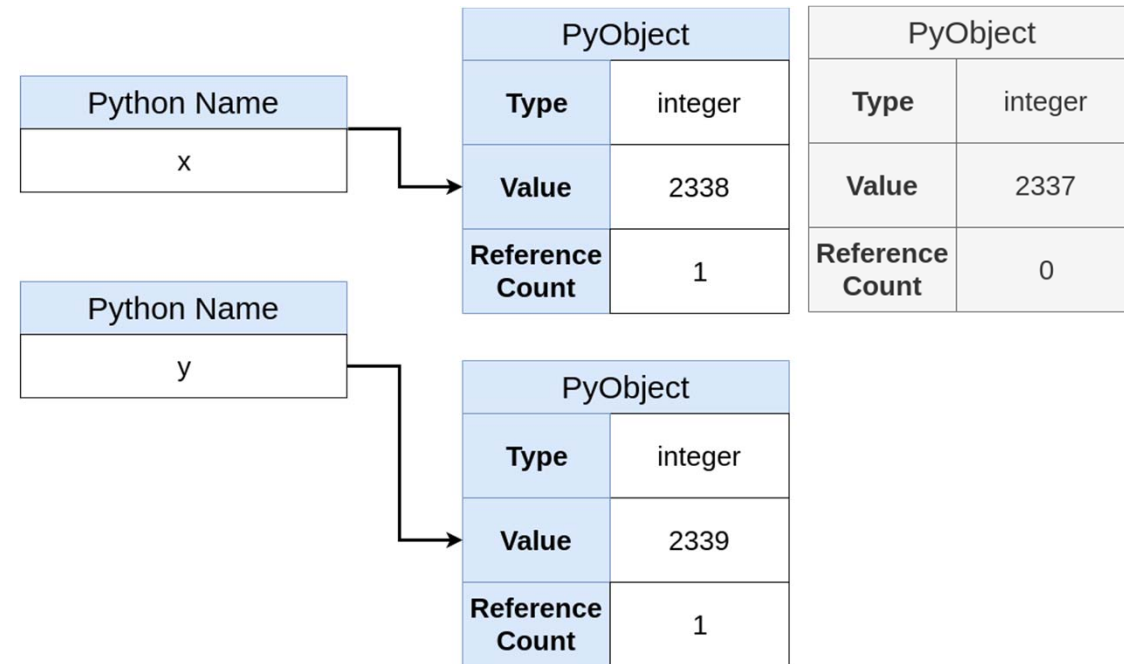
Python

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

- Perform addition on `y`: After the addition call, 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:

Python

```
>>> y += 1  
>>> y is x  
False
```



## Pointers in Python: What's the Point?

### 2) A Note on Intern Objects in Python

Python

```
>>> x = 1000
>>> y = 1000
>>> x is y
True
```

- 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.

Python

```
>>> x = 1000
>>> y = 499 + 501
>>> x is y
False
```

- if you were to add two numbers together to get 1000, you would end up with a different memory address:

- 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

- 객체 인터닝(Object Interning)이란, 특정 불변(Immutable) 객체의 값을 하나의 메모리에 저장해놓은 후 그 메모리를 계속해서 재활용하는 최적화 방식

```
x = 1000
y = 499+501
x is y
```

False

```
x = 100
y = 49+51
x is y
```

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.
- CPython 3.7 interns the following:
  - ✓ Integer numbers between -5 and 256
  - ✓ Strings that contain ASCII letters, digits, or underscores only
- Because 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:



## Pointers in Python: What's the Point?

### 2) A Note on Intern Objects in Python

Python

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

- 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:

Python

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

- If you were to introduce a non-ASCII letter, digit, or underscore, then you would get a different result:
- s1 and s2 have an exclamation mark (!)

- **Bonus:** If you really want these objects to reference the same internal object, then you may want to check out [sys.intern\(\)](#). One of the use cases for this function is outlined in the documentation:

Interning strings is useful to gain a little performance on dictionary lookup—if the keys in a dictionary are interned, and the lookup key is interned, the key comparisons (after hashing) can be done by a pointer compare instead of a string compare. (Source)

## Pointers in Python: What's the Point?

### Simulating Pointers in Python

- Multiple ways to simulate pointers in Python
  - ✓ Using mutable types as pointers
  - ✓ Using custom Python objects

#### 1) Using Mutable Types as Pointers

- Consider using a *list* and modifying the first element:

Python

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

- `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:

Python

```
>>> 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
```

Python

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

## Pointers in Python: What's the Point?

### 2) 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.
- A custom Python class can really help

Python

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

- 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](#):
- 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.

Python

```
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"]
```

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

Python

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

## Pointers in Python: What's the Point?

- You need to be able to increment these values:

Python

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

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

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

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

Python

```
>>> 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.

- Here's the full source for the Metrics class:

Python

```
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
```

## Pointers in Python: What's the Point?

### Real Pointers With ctypes

- There are pointers in Python, specifically [CPython](#).
- Using the *builtin* **ctypes** module, you can create real C-style pointers in Python.

C

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

- 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:

Shell

```
$ 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:

Python

```
>>> 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:

Python

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

## Pointers in Python: What's the Point?

- if you were to try to call this code with the wrong type, then you would get a nice warning instead of undefined behavior:.

Python

```
>>> 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:

Python

```
>>> x = ctypes.c_int()
>>> 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.
- You can use this to call `add_one()`:

Python

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

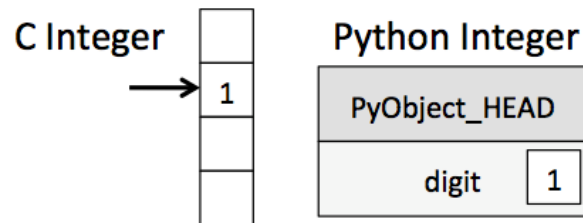
- The integer was incremented by one. Have successfully used real pointers in Python.

## Why Python is slow?

- Python is slower than Fortran and C for a variety of reasons:

### 1. Python is Dynamically Typed rather than Statically Typed.

- 프로그램 실행 시, 인터프리터는 정의된 변수의 유형을 알고 있지 않다는 것을 의미합니다. C변수와 파이썬 변수의 차이는 아래 그림으로 요약됩니다.(컴파일 된 언어의 표준으로 C언어를 사용합니다.)



- C 언어 변수의 경우, 컴파일러는 단지 그 정의만으로도 변수의 유형을 알고 있다. 파이썬 변수의 경우는 프로그램 실행시의 변수는 파이썬 object의 일부 종류라는 것이다.

```
/* C code */  
int a = 1;  
int b = 2;  
int c = a + b;
```

1. Assign <int> 1 to a
2. Assign <int> 2 to b
3. call binary\_add<int, int>(a, b)
4. Assign the result to c

- 파이썬에서 인터프리터는 1과 2는 object라는 것만을 알고, 그들의 타입은 알지 못한다. 그래서 인터프리터는 타입 정보를 찾기 위해 각 변수의 PyObject\_HEAD를 검사 한 후, 두 타입의 적절한 덧셈 루틴을 호출해야 한다. 마지막으로 반환 값을 보관 유지하는 새로운 Python object를 만들고 초기화해야 한다. 이벤트의 순서는 대략 다음과 같다.

```
# python code  
a = 1  
b = 2  
c = a + b
```

#### 1. Assign 1 to a

- 1a. Set a->PyObject\_HEAD->typecode to integer
- 1b. Set a->val = 1

#### 2. Assign 2 to b

- 2a. Set b->PyObject\_HEAD->typecode to integer
- 2b. Set b->val = 2

#### 3. call binary\_add(a, b)

- 3a. find typecode in a->PyObject\_HEAD
- 3b. a is an integer; value is a->val
- 3c. find typecode in b->PyObject\_HEAD
- 3d. b is an integer; value is b->val
- 3e. call binary\_add<int, int>(a->val, b->val)
- 3f. result of this is result, and is an integer.

#### 4. Create a Python object c

- 4a. set c->PyObject\_HEAD->typecode to integer
- 4b. set c->val to result

- 동적 타이핑은 모든 작업에 더 많은 단계가 포함되어 있다는 것을 의미함. 이것이 숫자데이터에 관한 연산에서 C언어와 비교했을때 파이썬이 느린 가장 큰 이유임.

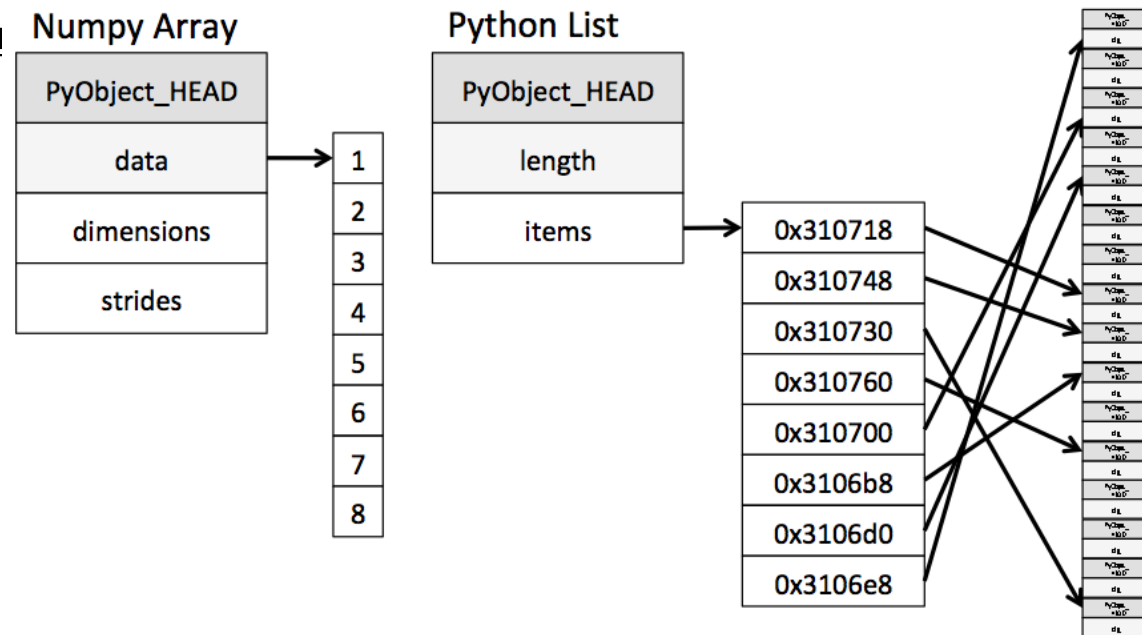
## Why Python is slow?

### 2. Python is interpreted rather than compiled.

- 스마트한 컴파일러는 결과의 속도를 높일 수 있도록, 반복되거나 불필요한 연산을 미리 내다보고 최적화 할 수 있다.
- 컴파일러 최적화는 컴파일러만의 특성이고, 이에 대한 몇가지 예제들은 [Numba](#)와 [Cython](#)에 관해 확인 할 수 있다.

### 3. Python's object model can lead to inefficient memory access

- 일련의 정수 자료들을 처리한다고 할 때, C언어는 어떤 종류의 버퍼 기반의 배열을 사용하는 동안, 파이썬에서는 표준 **List object**를 사용할 수 있다.
- Numpy 배열**은 C의 배열과 유사한 파이썬 object이다. 이 배열은 **값들의 연속되는 데이터 버퍼를 위한 포인터**를 가지고 있다. 달리 말하자면, **파이썬 리스트**는 **포인터의 연속되는 버퍼를 위한 포인터(double pointer)**를 가지고 있다. 각각의 포인터들은 그것들의 데이터(지금의 경우는 정수) 주소를 가지고 있고 그것들을 가르키고 있다.
- Numpy 레이아웃은 저장과 액세스 측면 모두에서 파이썬의 레이아웃보다 훨씬 더 효율적이다



### So Why Use Python?

- 동적인 타이핑은 파이썬을 C보다 사용하기 쉽게 해 준다.
- 파이썬은 매우 **Flexible**하고 **Forgiving**합니다. 이 유연함은 개발시간의 효율적인 사용을 이끌어내고, C나 포트란의 최적화가 절실히 필요할 경우에도 파이썬을 통해 쉽게 컴파일된 라이브러리에 접근(**Python offers easy hooks into compiled libraries.**)할 수 있다



### 파이썬 동작의 내부 과정 알아보기

- 기준 : Python 3.8, 64-bit CPU

```
import sys
print("Python version =", sys.version[:5])
```

Python version = 3.8.1

### Digging into Python Integers

- 파이썬의 정수형은 생성하기와 사용하기가 아래와 같이 쉽다.

```
x = 40
print(x)
```

40

- 인터페이스의 단순함은 내부적으로 일어나고 있는 복잡성과는 모순된다.
- 앞 단락에서 파이썬 정수형의 메모리 레이아웃에 대해서 간단히 이야기 했었다. 여기에서는 파이썬 인터프리터 자체에서 파이썬의 정수형의 내부를 조사하기 위해 [파이썬의 내장 ctypes 모듈](#)을 사용한다.
- 하지만 먼저, 우리는 파이썬의 정수가 어떻게 생겼는지 C의 API 수준과 같이 정확히 알아야 한다.

- CPython의 실제 x 변수는 [Include/longintrepr.h](#) 내부에 있는 CPython 소스코드 안에 정의되어 있는 구조체에 저장된다.

```
struct _longobject {
    PyObject_VAR_HEAD # standard header for variable length objects
    digit ob_digit[1]; # array of numbers
};
```

- PyObject\_VAR\_HEAD는 [Include/object.h](#)에 정의된, 다음과 같은 구조를 가진 object를 시작하는 매크로이다.

```
typedef struct {
    PyObject ob_base;
    Py_ssize_t ob_size; /* Number of items in variable part */
} PyVarObject;
```

- PyObject 요소 또한 Include/object.h의 정의에 포함되어 있다.

```
typedef struct _object {
    _PyObject_HEAD_EXTRA
    Py_ssize_t ob_refcnt;
    struct _typeobject *ob_type;
} PyObject;
```

- 여기 \_PyObject\_HEAD\_EXTRA는 일반적으로 파이썬 빌드에서 사용되지 않는 매크로이다.

## Python meta-hacking

- 이 모든 것과 typedefs/macros를 정확히 합치면, 정수 object는 아래의 구조와 같이 동작한다.

```
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

- `ob_refcnt`: 메모리 할당/반납 (deallocation) 핸들하게 도와주는 object의 참조 수,
- `ob_type`: object의 모든 형식 및 메서드의 정의가 들어있는 구조체에 대한 포인터, 객체 타입 부호화
- `ob_size`: 데이터 멤버들의 크기
- `ob_digit`: 실제 수치를 포함

- 실제 object 구조와 위의 정보의 일부를 확인하는데 `ctypes` 모듈을 사용할 것이다. C 구조체로 Python의 표현을 정의하면서 시작해 보자.

```
import ctypes

class IntStruct(ctypes.Structure):
    _fields_ = [("ob_refcnt", ctypes.c_long),
                ("ob_type", ctypes.c_void_p),
                ("ob_size", ctypes.c_ulong),
                ("ob_digit", ctypes.c_long)]

    def __repr__(self):
        return ("IntStruct(ob_digit={self.ob_digit}, "
                "refcount={self.ob_refcnt})").format(self=self)
```

- 이제 어떤 수(x) 40에 대한 내부 표현을 살펴보자.
- CPython에서 `id` 함수의 object의 메모리 위치 제공기능을 사용한다.

```
x = 40
IntStruct.from_address(id(40))
```

```
IntStruct(ob_digit=0, refcount=67)
```

- `ob_digit` 속성은 메모리의 정확한 위치를 가르킨다!
- 하지만 `refcount`는 어떤가요? 정확히 하나의 값을 생성했지만, 왜 참조 수가 하나 이상의 값을 가질까요?
- 파이썬이 작은 정수를 많이 사용하는 것은 아주 잘 알려져 있다. 이러한 정수의 각각을 위해 새로운 PyObject가 생성된 경우라면, 많은 양의 메모리를 사용하게 된다. 이 때문에 파이썬에서는 일반적인 정수값을 Singleton으로 구현한다. 즉, 이 숫자 중 하나의 복사본만 메모리에 존재한다. 달리 말하자면, 이 범위 내에서 새로운 파이썬 정수를 만들 때마다 단순히 그 값을 가지는 Singleton에 대한 reference가 만들어진다.

```
x = 40
y = 40
id(x) == id(y)
```

True

```
x = 1234
y = 1234
id(x) == id(y)
```

False

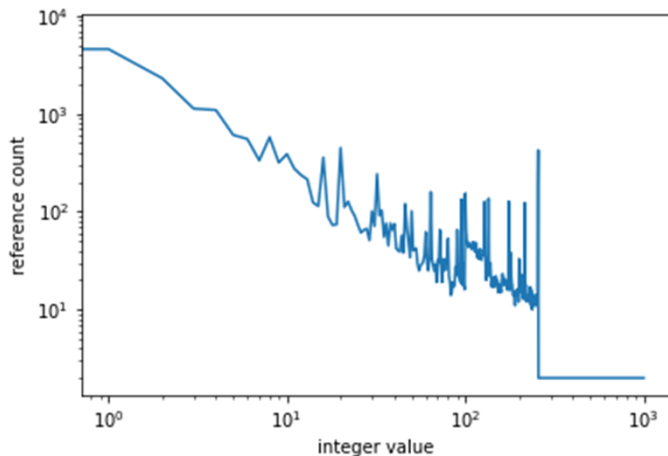
- 두 변수는 같은 메모리 주소를 가리키는 단순한 포인터이다.
- 만약 훨씬 더 큰 정수 (파이썬 3.4버전에서 255보다 큰 경우) 라면, 이것은 더 이상 참이지는 않습니다.

## Python meta-hacking

- 파이썬 인터프리터를 시작하면 많은 정수 객체들이 만들어진다.
- 각각에 얼마나 많은 reference들이 있는지를 살펴보면, 흥미로울 것이다.

```
%matplotlib inline
import matplotlib.pyplot as plt
import sys
plt.loglog(range(1000), [sys.getrefcount(i) for i in range(1000)])
plt.xlabel('integer value')
plt.ylabel('reference count')
```

Text(0, 0.5, 'reference count')



- 우리는 '0'이 몇 천 번을 reference되는것을 확인하고, 여러분이 예상하는 것처럼 reference frequency는 일반적으로 정수의 값이 증가함에 따라 감소하는것을 볼 수 있다.

- 더 나아가 우리의 예상대로 결과가 나타나는지 확인하기 위해, **ob\_digit 필드가 올바른 값을 가지고 있는지 확인해 보자**

```
all(i == IntStruct.from_address(id(i)).ob_digit
    for i in range(256))
```

True

- 256보다 큰 번호를 가지지 않는다는 것을 알아차릴 수 있을 것이다. [Objects/longobject.c](#)에서 수행되는 몇가지 비트-쉬프트 연산은 이러한 큰 정수가 메모리에 표시되는 방식을 변경한다.
- 정확하게 그것이 왜 일어나고 있는지 완전히 이해하고 말할 수는 없습니다만, 여기서 볼 수 있듯이 효율적으로 **long int 데이터 타입의 오버플로 한계를 넘어 정수처리를 하는 파이썬의 능력과 관계**있다고 생각한다.

```
2**100
```

1267650600228229401496703205376

- 이 숫자는 long 타입으로 표현하기에 너무 크다. long타입은 64비트로 표현 가능한 값들만 가질 수 있다. (64비트 표현은  $2^{64}$ )

### Digging into Python Lists

- 좀 더 복잡한 형식으로 위의 아이디어를 적용해 파이썬 리스트를 알아봅시다.
- 정수형과 유사하게, [Include/listobject.h](#) 안에 있는 리스트 object에서 그 정의를 찾을 수 있다.

```
typedef struct {
    PyObject_VAR_HEAD
    PyObject **ob_item;
    Py_ssize_t allocated;
} PyListObject;
```

- 또한, 구조체를 효과적으로 따라가는 것을 보기 위해 매크로와 애매한 타입들을 확장할 수 있습니다.

```
typedef struct {
    long ob_refcnt;
    PyTypeObject *ob_type;
    Py_ssize_t ob_size;
    PyObject **ob_item;
    long allocated;
} PyListObject;
```

- `PyObject **ob_item`은 리스트의 내용물을 가르키고,
- `ob_size`값은 리스트 안에 얼마나 많은 아이템들이 있는지 말해준다.

```
class ListStruct(ctypes.Structure):
    _fields_ = [("ob_refcnt", ctypes.c_long),
                ("ob_type", ctypes.c_void_p),
                ("ob_size", ctypes.c_ulong),
                ("ob_item", ctypes.c_long), # PyObject** pointer cast to long
                ("allocated", ctypes.c_ulong)]

    def __repr__(self):
        return ("ListStruct(len={self.ob_size}, "
                "refcount={self.ob_refcnt})").format(self=self)
```

```
L = [1,2,3,4,5]
ListStruct.from_address(id(L))
```

```
ListStruct(len=5, refcount=1)
```

- 제대로 수행되었는지 확인하기 위해 몇가지 추가 참조 리스트를 만들고, 참조 횟수에 어떻게 영향을 미치는지 알아보자.

```
tup = [L, L] # two more references to L
ListStruct.from_address(id(L))
```

```
ListStruct(len=5, refcount=3)
```


## Python meta-hacking

- 이제 리스트에서 실제 요소들을 찾는 것에 대해 살펴보자.
- 위에서 본 것처럼, 요소들은 PyObject 포인터들의 연속적인 배열 속에 저장되어 있다. ctypes를 사용해, IntStruct 객체로 구성된 복합 구조를 만들 수 있다.

```
# get a raw pointer to our list
Lstruct = ListStruct.from_address(id(L))

# create a type which is an array of integer pointers the same length as L
PtrArray = Lstruct.ob_size * ctypes.POINTER(IntStruct)

# instantiate this type using the ob_item pointer
L_values = PtrArray.from_address(Lstruct.ob_item)
```

- 각 항목의 값들을 살펴 보자. 

```
[ptr[0] for ptr in L_values] # ptr[0] dereferences the pointer
```

```
[IntStruct(ob_digit=1, refcount=5296),
 IntStruct(ob_digit=2, refcount=2887),
 IntStruct(ob_digit=3, refcount=932),
 IntStruct(ob_digit=4, refcount=1049),
 IntStruct(ob_digit=5, refcount=808)]
```

- 리스트에서 PyObject정수를 얻어낸 결과임

### Digging into NumPy arrays

- 이제 비교를 위해 똑같은 방법으로 Numpy 배열을 살펴 보도록 하자.
- NumPy C-API 배열 정의의 상세한 단계별 안내는 `numpy/core/include/numpy/ndarraytypes.h`에서 찾을 수 있다.

```
import numpy as np
np.__version__
```

'1.20.3'

- NumPy 배열 자체를 나타내는 구조체를 생성하는 것으로 시작하고,
- 여기에서 파이썬 버전에 접근하기 위해 사용자 속성 `shape`과 `stride`를 추가한다.

- 참조 수 확인

```
L = [x,x,x] # add three more references to x
xstruct
```

`NumpyStruct(shape=(10, 20), refcount=4)`

```
class NumpyStruct(ctypes.Structure):
    _fields_ = [("ob_refcnt", ctypes.c_long),
                ("ob_type", ctypes.c_void_p),
                ("ob_data", ctypes.c_long), # char* pointer cast to long
                ("ob_ndim", ctypes.c_int),
                ("ob_shape", ctypes.c_voidp),
                ("ob_strides", ctypes.c_voidp)]

    @property
    def shape(self):
        return tuple((self.ob_ndim * ctypes.c_int64).from_address(self.ob_shape))

    @property
    def strides(self):
        return tuple((self.ob_ndim * ctypes.c_int64).from_address(self.ob_strides))

    def __repr__(self):
        return ("NumpyStruct(shape={self.shape}, "
                "refcount={self.ob_refcnt})").format(self=self)
```

```
x = np.random.random((10, 20))
xstruct = NumpyStruct.from_address(id(x))
xstruct
```

`NumpyStruct(shape=(10, 20), refcount=1)`

## Python meta-hacking

- 이제 우리는 데이터 버퍼를 뽑는 부분을 할 수 있다. 단순화 하기 위해 `strides`를 무시하고, 연속적인 C 배열을 가정한다. 이것은 비트를 가지고 작업을 일반화 할 수 있다.

```
x = np.arange(10)
xstruct = NumpyStruct.from_address(id(x))
size = np.prod(xstruct.shape)

# assume an array of integers
arraytype = size * ctypes.c_long
data = arraytype.from_address(xstruct.ob_data)

[d for d in data]

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- `data`변수는 이제 NumPy배열 안의 메모리를 정의하는 연속적인 블록을 나타냅니다. 이를 보여주기 위해 배열속의 값을 바꾸어 보자.

```
x[4] = 555
[d for d in data]

[0, 1, 2, 3, 555, 5, 6, 7, 8, 9]
```

- `x`와 `data`는 둘 다 메모리의 같은 연속적인 블록을 가르킨다.
- 파이썬 리스트와 NumPy nd-배열(ndarray)의 내부를 비교하자면, NumPy 배열이 훨씬 더 확실하게 동일한 타입의 데이터 리스트를 나타내는데 보다 더 단순하다. 이 사실은 핸들링 뿐만 아니라 더 효율적인 컴파일러를 만드는 것과도 관계가 있다.