《软件工程》期末大作业开发验收汇报

WinPython - a Simple Python Interpreter

WU-SUNFLOWER

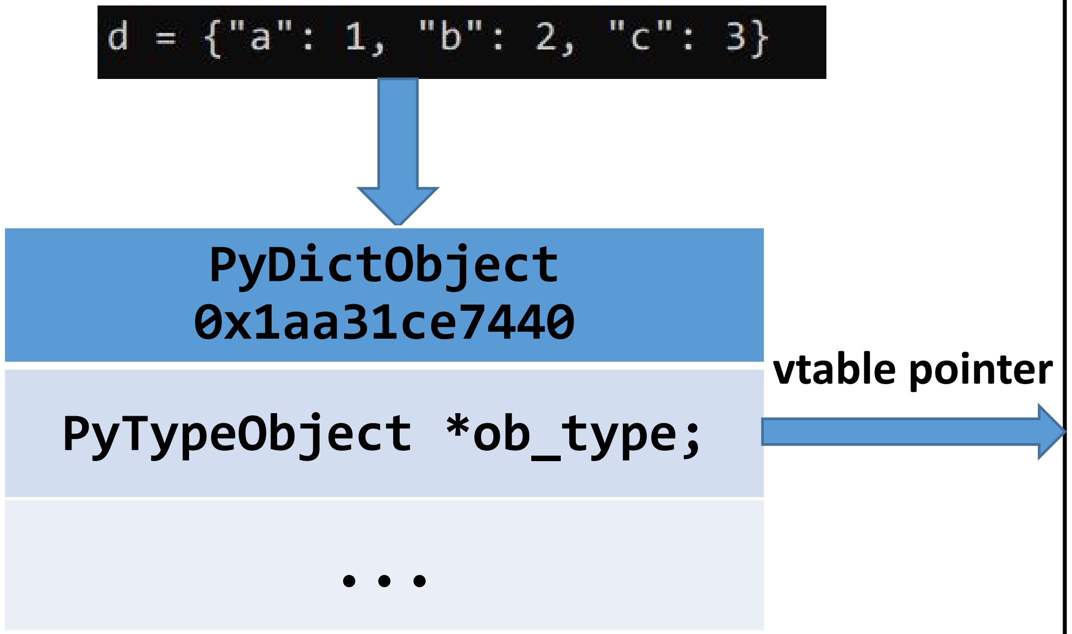
Summary

I will share our **exploration experiences** about making Python better on the following topics:

- Introduce Oop-Klass Model to Python
- Introduce Small Integers(Smis) to Python
- Introduce Copy Garbage Collection Algorithm to Python

Oop-Klass Model——bring up problem

- To implement polymorphism, **virtual function tables** are a commonly used method.
- CPython does indeed do this.



python/cpython/Objects/dictobject.c

```
PyTypeObject PyDict_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "dict",
    sizeof(PyDictObject),
    PyObject_HashNotImplemented, /*
tp_hash */
    0, /* tp_call */
    0, /* tp_str */
    dict_traverse,  /* tp_traverse */
    dict_richcompare, /* tp_richcompare */
    dict_iter, /* tp_iter */
    mapp_methods, /* tp_methods */
    // ...
};
```

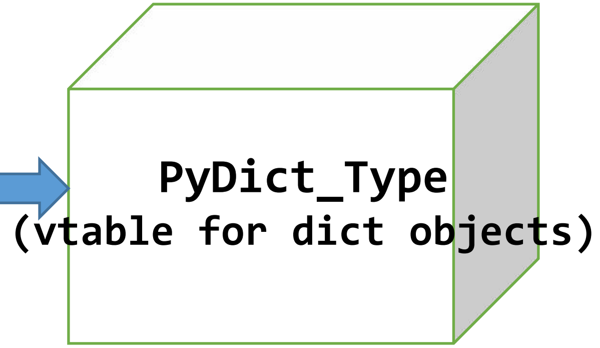
Oop-Klass Model——bring up problem

- However, what about 'dict' itself in CPython?

```
>>> dict
<class 'dict'>
>>> type(dict)
<class 'type'>
>>>
```

python/cpython/Python/bltinmodule.c

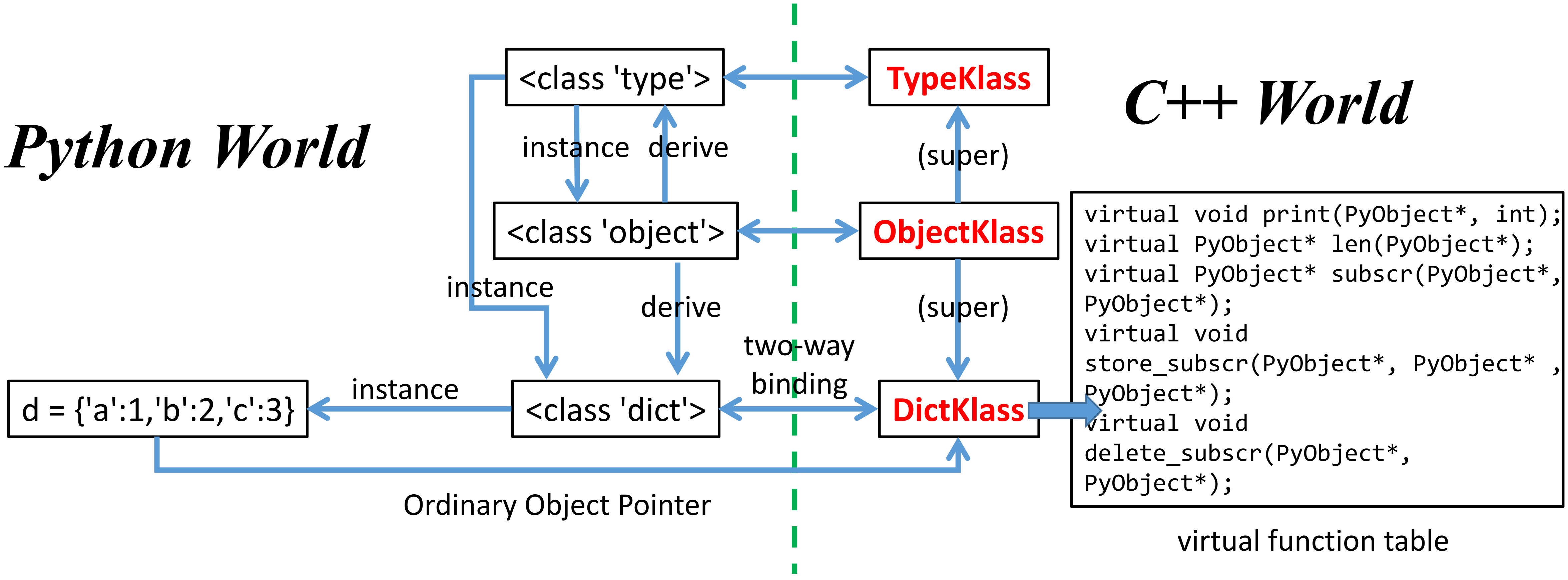
```
//...
SETBUILTIN("False", Py_False);
SETBUILTIN("True", Py_True);
SETBUILTIN("bool", &PyBool_Type);
SETBUILTIN("dict", &PyDict_Type);
SETBUILTIN("float", &PyFloat_Type);
SETBUILTIN("int", &PyLong_Type);
SETBUILTIN("list", &PyList_Type);
// ...
```



- So, *PyDict_Type* serves as both the vtable for dictionary objects within the interpreter and the type object (what we refer to as the class object for dictionaries) in the Python ecosystem.

Oop-Klass Model——improvement

- Since I think this design can easily lead to confusion and is difficult to maintain, I try to introduce the *Oop-Klass* model into WinPython.



Small Integers

- As we know, CPython treats integers as Python objects, and they have to be allocated in heap...

```
(base) C:\Users\Administrator>python
Python 3.11.7 | packaged by Anaconda, Inc. |
Type "help", "copyright", "credits" or "lice
>>> x = 12345678
>>> y = 12345678
>>> id(x)
2514494272496
>>> id(y)
2514494268944
>>>
```

python/cpython/Objects/longobject.c

```c
PyLongObject * _PyLong_New(Py_ssize_t size) {
   assert(size >= 0);
   PyLongObject *result;
   if (size > (Py_ssize_t)MAX_LONG_DIGITS) {
      PyErr_SetString(PyExc_OverflowError,
                  "too many digits in integer");
      return NULL;
   }
   Py_ssize_t ndigits = size ? size : 1;
   result = PyObject_Malloc(offsetof(PyLongObject,
long_value.ob_digit) +
                  ndigits*sizeof(digit));
   if (!result) {
      PyErr_NoMemory();
      return NULL;
   }
   // ...
```

Small Integer——bring up problem

This has led to at least two issues in CPython:

- 1. During the process of performing mathematical calculations, **a large number of temporary integer objects will be created**, which costs much time.

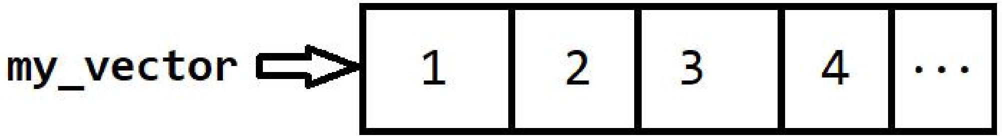
- 2. For some data structures(like List in Python), **CPU's data cache is difficult to function effectively.**

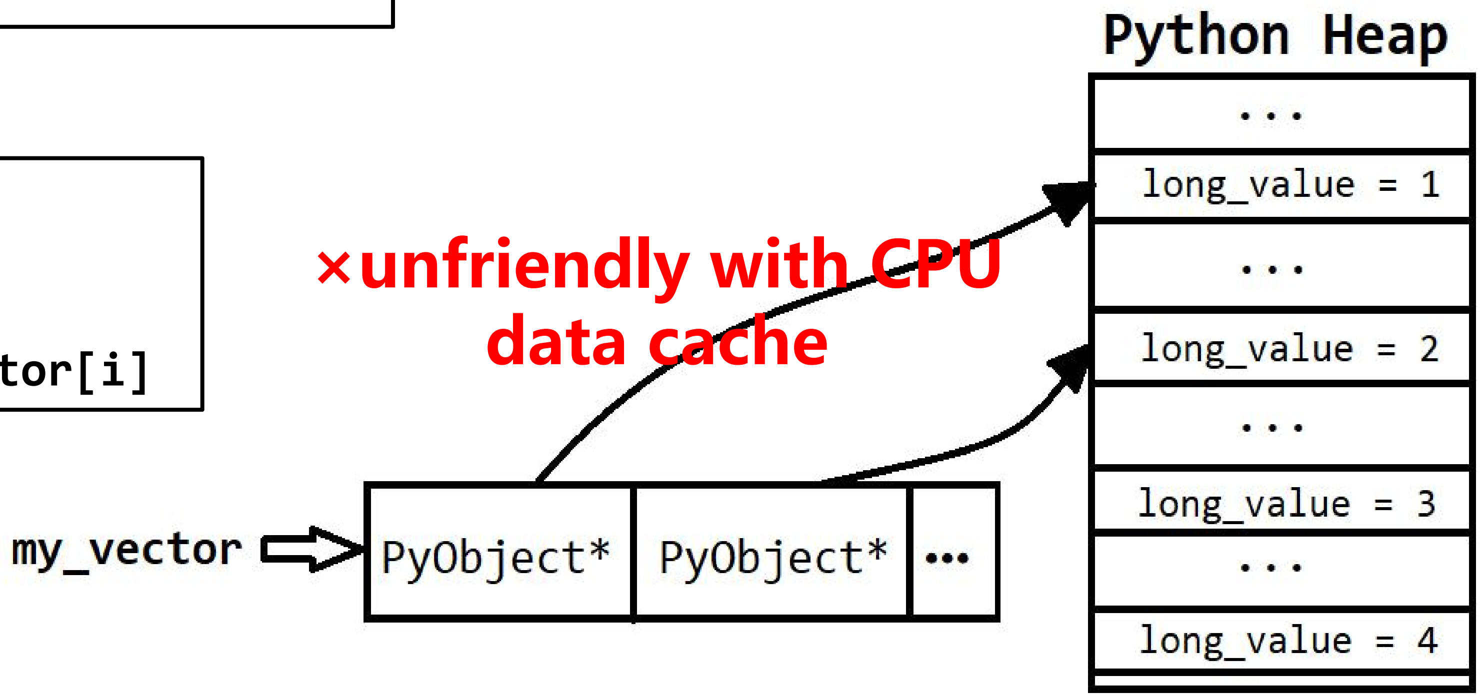
Small Integer——bring up problem

```
/* In C/C++ */
int my_vector[] = {1, 2, 3, ...};
for (int i = 0; i < n; ++i) {
    // do something with my_vector[i]
}
```

```
# In CPython
my_vector = [1, 2, 3, ...]
for i in range(len(my_vector)):
    // do something with my_vector[i]
```

✓friendly with CPU data cache





Small Integer——improvement

- To solve this problem, I proposed the concept of *small integers* (*Smis*), which are signed integers that occupy 63bits, to replace traditional integer objects in CPython.
- In WinPython, objects are allocated in the heap at addresses aligned by 8 bytes, which allows we to use its 3 least bits for tagging.
- So I decided to use the least significant bit to distinguish Smis from heap object pointers.

```
                 |----- 64 bits -----|
Heap Pointer:    |_____address______0|
Small Integer:   |___int63_value____1|
```

Small Integer——implementation & disadvantage

WinPython/framework/Universe.hpp

```
#define isPyInteger(x) ((uintptr_t)(x) & 1)
#define toRawInteger(x) ((int64_t)((int64_t)(x) >> 1))
#define toPyInteger(x) ((PyObject*)(((uintptr_t)(x) << 1) | 1))
```

WinPython/framework/interpreter.cpp

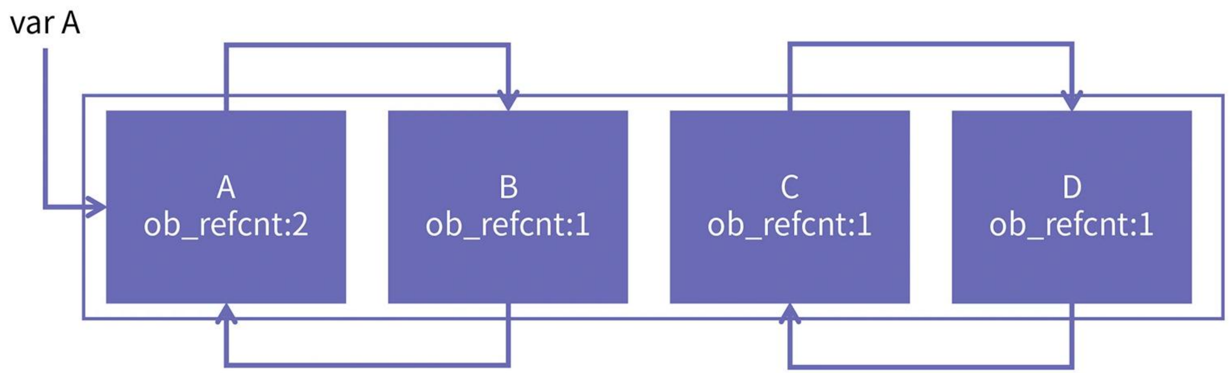
```
case ByteCode::Binary_Multiply:
    rhs = POP();
    lhs = POP();
    // deal with the situation of multiply two integer.
    if (isPyInteger(lhs) && isPyInteger(rhs)) {
        PUSH(toPyInteger(toRawInteger(lhs) * toRawInteger(rhs)));
    }
    // deal with other situations like [1,2,3]*3
    else {
        PUSH(lhs->mul(rhs));
    }
    break;
```

Copy Garbage Collection——bring up problem

- As we know, CPython uses ***reference counting*** internally to implement automatic memory management. And it has a large flaw...

```
class Base:
    pass

A = Base(); B = Base()
C = Base(); D = Base()
var_A = A
A.ref = B; B.ref = A
C.ref = D; D.ref = C
```



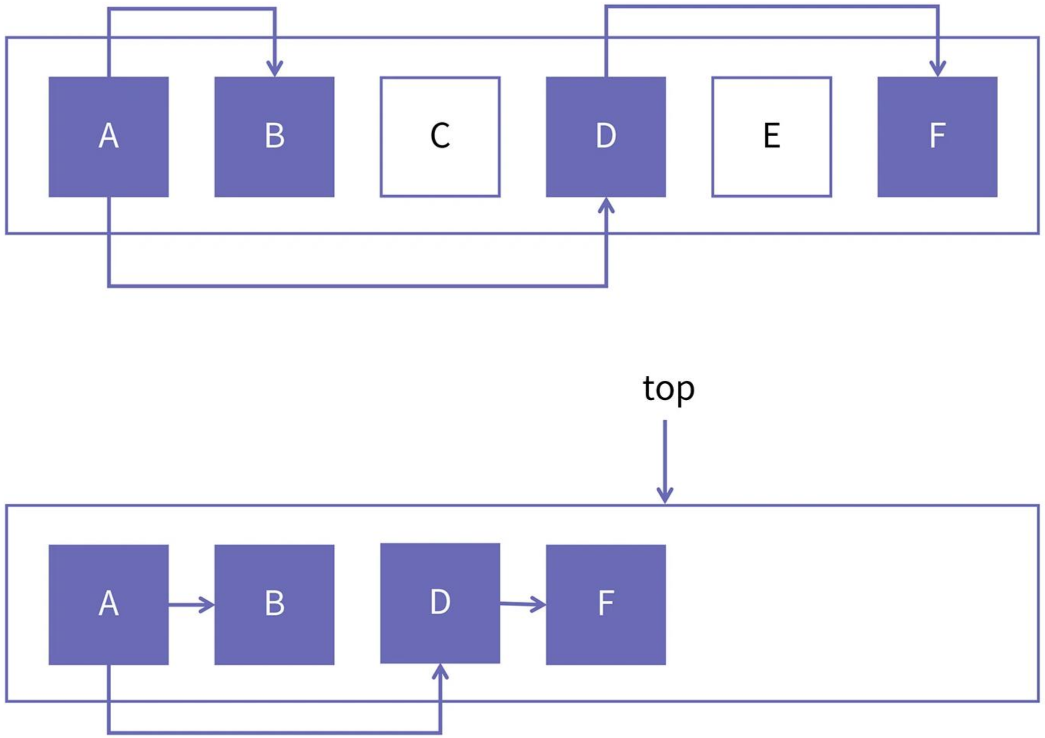
- While the latest version of CPython has effective means of handling cyclic references, **I still want to explore if there is a method to eradicate this issue.**

Copy Garbage Collection——idea

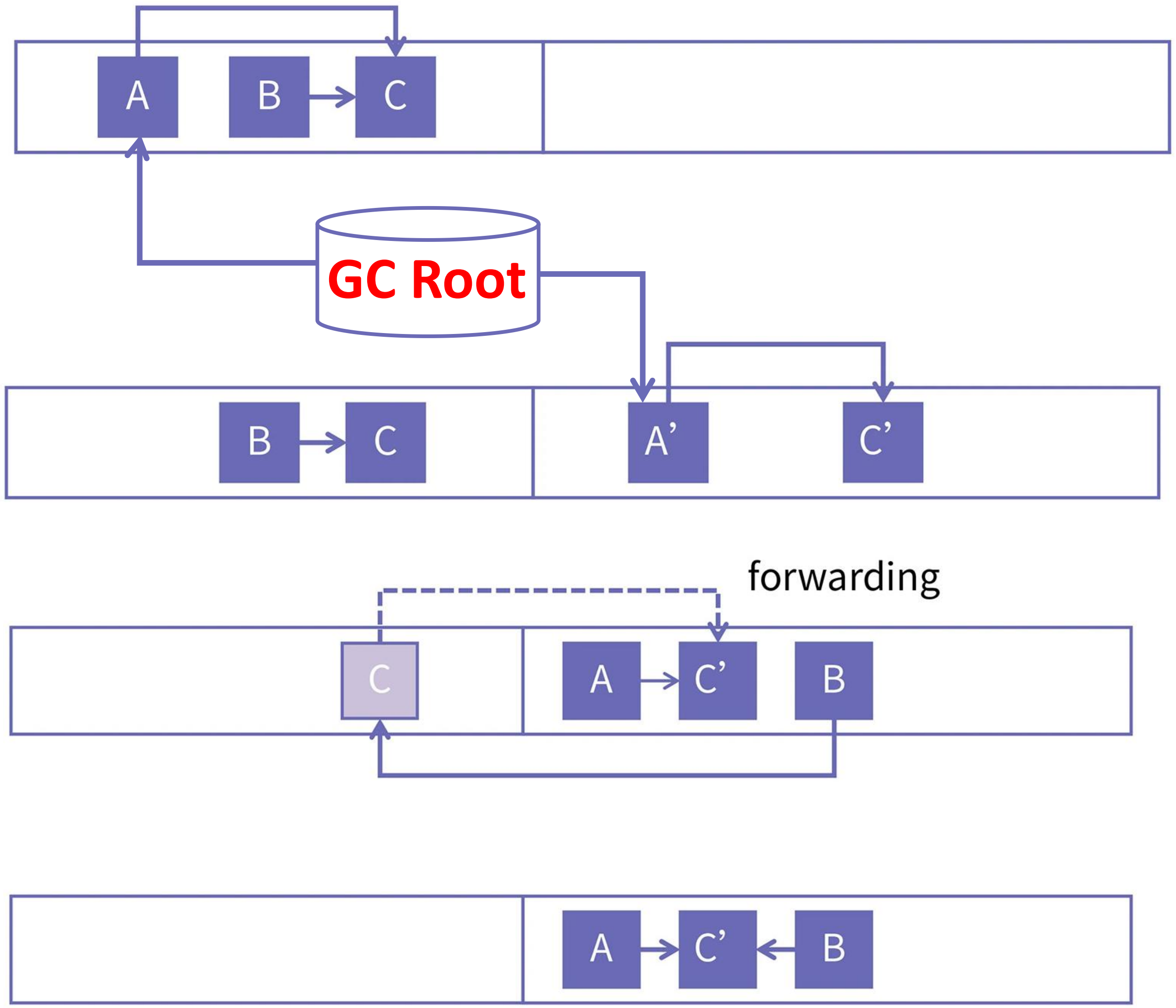
- **The Generational Hypothesis:** Most objects in memory tend to die young, meaning they are short-lived and are often collected soon after being allocated.

Copy Garbage Collection——idea

- Split the heap space into ***the eden space*** and ***the survivor space***.
- We allocate Python objects in ***the eden space***.
- When the eden space is full, we just copy all the accessible objects into ***the survivor space***, and turn it into ***the new eden space***.
- Finally, we clear ***the old eden space***, and turn it into ***the new survivor space***.
- Repeat the above...



Copy Garbage Collection——implement




```
class PyObject {
// store forwarding pointer here
uintptr_t _mark_word;
// get/set forwarding pointer
void* getNewAddr();
void setNewAddr(void*);
// get the size of python object
size_t getSize();
```

```
void ScavengeOopClosure::scavenge() {
  process_roots();
  while (!_oop_stack->isEmpty()) {
    PyObject* object = _oop_stack->pop();
    if (...) object->oops_do(this);
  }
}
void ScavengeOopClosure::process_roots() {
  Universe::oops_do(this);
  Interpreter::getInstance()->oops_do(this);
}
```

Copy Garbage Collection——disadvantages

- After completing the development of WinPython, I reflected and realized that my Copy GC algorithm still has the following shortcomings, **which are obvious in recursive programs**:
- 1. **Old generation objects** generated during the user program's execution will be **repeatedly copied**.
- 2. For complex user programs, **the time taken for DFS search during each copy operation will be lengthy**, leading to "world pause" in the user program.

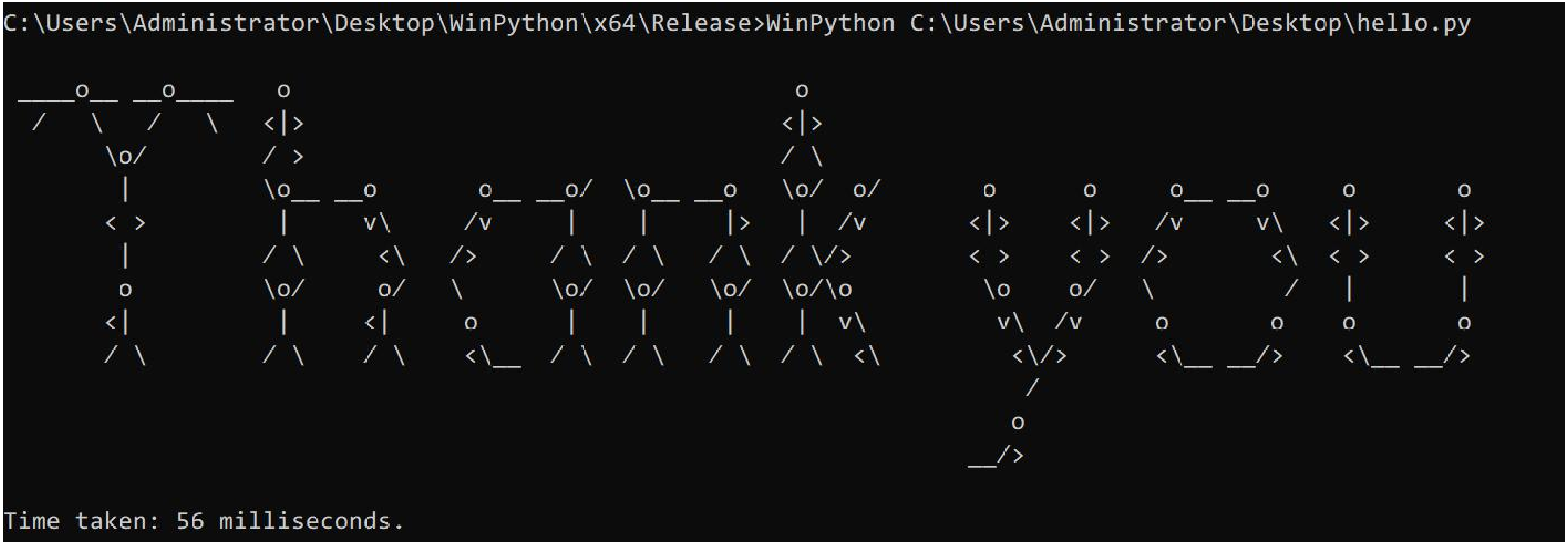
What's more

One of the main goals for developing WinPython is to provide a powerful platform for undergraduate computer science students to explore and learn the basic principles of how Python language work.

Therefore, after the end of this semester, I will publish the source code of the WinPython project.

Welcome to follow

https://github.com/WU-SUNFLOWER


C:\Users\Administrator\Desktop\WinPython\x64\Release>WinPython C:\Users\Administrator\Desktop\hello.py
Time taken: 56 milliseconds.