Ten thousand meters

Diving deep, flying high to see why

about blog materials

Python behind the scenes #8: how Python integers work

Published: Mon 08 February 2021
By Victor Skvortsov
tags: Python behind the scenes Python CPython

In the previous parts of this series we studied the core of the CPython interpreter and saw how the most fundamental aspects of Python are implemented. We made an overview of the CPython VM, took a look at the CPython compiler, stepped through the CPython source code, studied how the VM executes the bytecode and learned how variables work. In the two most recent posts we focused on the Python object system. We learned what Python objects and Python types are, how they are defined and what determines their behavior. This discussion gave us a good understanding of how Python objects work in general. What we haven't discussed is how particular objects, such as strings, integers and lists, are implemented. In this and several upcoming posts we'll cover the implementations of the most important and most interesting built-in types. The subject of today's post is `int`.

Note: In this post I'm referring to CPython 3.9. Some implementation details will certainly change as CPython evolves. I'll try to keep track of important changes and add update notes.

Why Python integers are interesting

Integers require no introduction. They are so ubiquitous and seem so basic that you may doubt whether it's worth discussing how they are implemented at all. Yet, Python integers are interesting because they are not just 32-bit or 64-bit integers that CPUs work with natively. Python integers are arbitrary-precision integers, also known as bignums. This means that they can be as large as we want, and their sizes are only limited by the amount of available memory.

Bignums are handy to work with because we don't need to worry about such things as integer overflows and underflows. They are extensively used in fields like cryptography and computer algebra where large numbers arise all the time and must be represented precisely. So, many programming languages have bignums built-in. These include Python, JavaScript, Ruby, Haskell, Erlang, Julia, Racket. Others provide bignums as a part of the standard library. These include Go, Java, C#, D, PHP. Numerous third-party libraries implement bignums. The most popular one is the GNU Multiple Precision Arithmetic Library (GMP). It provides a C API but has bindings for all major languages.

There are a lot of bignum implementations. They're different in detail, but the general approach to implement bignums is the same. Today we'll see what this approach looks like and use CPython's implementation as a reference example. The two main questions we'll have to answer are:

- how to represent bignums; and
- how to perform arithmetic operations, such as addition and multiplication, on bignums.

We'll also discuss how CPython's implementation compares to others and what CPython does to make integers more efficient.

Bignum representation

Think for a moment how you would represent large integers in your program if you were to implement them yourself. Probably the most obvious way to do that is to store an integer as a sequence of digits, just like we usually write down numbers. For example, the integer `51090942171709440000` could be represented as `[5, 1, 0, 9, 0, 9, 4, 2, 1, 7, 1, 7, 0, 9, 4, 4, 0, 0, 0, 0]`. This is essentially how bignums are represented in practice. The only important difference is that instead of base 10, much larger bases are used. For example, CPython uses base 2^15 or base 2^30 depending on the platform. What's wrong with base 10? If we represent each digit in a sequence with a single byte but use only 10 out of 256 possible values, it would be very memory-inefficient. We could solve this memory-efficiency problem if we use base 256, so that each digit takes a value between 0 and 255. But still much larger bases are used in practice. The reason for that is because larger base means that numbers have less digits, and the less digits numbers have, the faster arithmetic operations are performed. The base cannot be arbitrary large. It's typically limited by the size of the integers that the CPU can work with. We'll see why this is the case when we discuss bignum arithmetic in the next section. Now let's take a look at how CPython represents bignums.

Everything related to the representation of Python integers can be found in `Include/longintrepr.h`. Technically, Python integers are instances of `PyLongObject`, which is defined in `Include/longobject.h`, but `PyLongObject` is actually a typedef for `struct _longobject` that is defined in `Include/longintrepr.h`:

```
struct _longobject {
    PyVarObject ob_base; // expansion of PyObject_VAR_HEAD macro
    digit ob_digit[1];
};
```

This struct extends `PyVarObject`, which in turn extends `PyObject`:

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

So, besides a reference count and a type that all Python objects have, an integer object has two other members:

- `ob_size` that comes from `PyVarObject`; and
- `ob_digit` that is defined in `struct _longobject`.

The `ob_digit` member is a pointer to an array of digits. On 64-bit platforms, each digit is a 30-bit integer that takes values between 0 and 2^30-1 and is stored as an unsigned 32-bit int (`digit` is a typedef for `uint32_t`). On 32-bit platforms, each digit is a 15-bit integer that takes values between 0 and 2^15-1 and is stored as an unsigned 16-bit int (`digit` is a typedef for `unsigned short`). To make things concrete, in this post we'll assume that digits are 30 bits long.

The `ob_size` member is a signed int, whose absolute value tells us the number of digits in the `ob_digit` array. The sign of `ob_size` indicates the sign of the integer. Negative `ob_size` means that the integer is negative. If `ob_size` is 0, then the integer is 0.

Digits are stored in a little-endian order. The first digit (`ob_digit[0]`) is the least significant, and the last digit (`ob_digit[abs(ob_size)-1]`) is the most significant.

Finally, the absolute value of an integer is calculated as follows:

$$val = ob\_digit[0] \times (2^{30})^0 + ob\_digit[1] \times (2^{30})^1 + \cdots + ob\_digit[|ob\_size| - 1] \times (2^{30})^{|ob\_size|-1}$$

Let's see what all of this means with an example. Suppose we have an integer object that has `ob_digit = [3, 5, 1]` and `ob_size = -3`. To compute its value, we can do the following:

```
$ python -q
>>> base = 2**30
>>> -(3 * base**0 + 5 * base**1 + 1 * base**2)
-1152921509975556099
```

Now let's do the reverse. Suppose we want to get the bignum representation of the number `51090942171709440000`. Here's how we can do that:

```
>>> x = 51090942171709440000
>>> x % base
952369152
>>> (x // base) % base
337507546
>>> (x // base // base) % base
44
>>> (x // base // base // base) % base
0
```

So, `ob_digit = [952369152, 337507546, 44]` and `ob_size = 3`. Actually, we don't even have to compute the digits, we can get them by inspecting the integer object using the `ctypes` standard library:

```
import ctypes


MAX_DIGITS = 1000

# This is a class to map a C `PyLongObject` struct to a Python object
class PyLongObject(ctypes.Structure):
    _fields_ = [
        ("ob_refcnt", ctypes.c_ssize_t),
        ("ob_type", ctypes.c_void_p),
        ("ob_size", ctypes.c_ssize_t),
        ("ob_digit", MAX_DIGITS * ctypes.c_uint32)
    ]


def get_digits(num):
    obj = PyLongObject.from_address(id(num))
    digits_len = abs(obj.ob_size)
    return obj.ob_digit[:digits_len]


>>> from num_digits import get_digits
>>> x = 51090942171709440000
>>> get_digits(x)
[952369152, 337507546, 44]
```

As you might guess, the representation of bignums is an easy part. The main challenge is to implement arithmetic operations and to implement them efficiently.

Bignum arithmetic

We learned in part 6 that the behavior of a Python object is determined by the object's type. Each member of a type, called slot, is responsible for a particular aspect of the object's behavior. So, to understand how CPython performs arithmetic operations on integers, we need to study the slots of the `int` type that implement those operations.

In the C code, the `int` type is called `PyLong_Type`. It's defined in `Objects/longobject.c` as follows:

```
PyTypeObject PyLong_Type = {
    PyVarObject_HEAD_INIT(&PyType_Type, 0)
    "int",                                      /* tp_name */
    offsetof(PyLongObject, ob_digit),           /* tp_basicsize */
```

```
        sizeof(digit),                              /* tp_itemsize */
        0,                                          /* tp_dealloc */
        0,                                          /* tp_vectorcall_offset */
        0,                                          /* tp_getattr */
        0,                                          /* tp_setattr */
        0,                                          /* tp_as_async */
        long_to_decimal_string,                     /* tp_repr */
        &long_as_number,                            /* tp_as_number */
        0,                                          /* tp_as_sequence */
        0,                                          /* tp_as_mapping */
        (hashfunc)long_hash,                        /* tp_hash */
        0,                                          /* tp_call */
        0,                                          /* tp_str */
        PyObject_GenericGetAttr,                    /* tp_getattro */
        0,                                          /* tp_setattro */
        0,                                          /* tp_as_buffer */
        Py_TPFLAGS_DEFAULT | Py_TPFLAGS_BASETYPE |
            Py_TPFLAGS_LONG_SUBCLASS,               /* tp_flags */
        long_doc,                                   /* tp_doc */
        0,                                          /* tp_traverse */
        0,                                          /* tp_clear */
        long_richcompare,                           /* tp_richcompare */
        0,                                          /* tp_weaklistoffset */
        0,                                          /* tp_iter */
        0,                                          /* tp_iternext */
        long_methods,                               /* tp_methods */
        0,                                          /* tp_members */
        long_getset,                                /* tp_getset */
        0,                                          /* tp_base */
        0,                                          /* tp_dict */
        0,                                          /* tp_descr_get */
        0,                                          /* tp_descr_set */
        0,                                          /* tp_dictoffset */
        0,                                          /* tp_init */
        0,                                          /* tp_alloc */
        long_new,                                   /* tp_new */
        PyObject_Del,                               /* tp_free */
    };
```

We can see the `long_new()` function that creates new integers, the `long_hash()` function that computes hashes and the implementations of some other important slots. In this post, we'll focus on the slots that implement basic arithmetic operations: addition, subtraction and multiplication. These slots are grouped together in the `tp_as_number` suite. Here's what it looks like:

```
    static PyNumberMethods long_as_number = {
        (binaryfunc)long_add,       /*nb_add*/
        (binaryfunc)long_sub,       /*nb_subtract*/
        (binaryfunc)long_mul,       /*nb_multiply*/
        long_mod,                   /*nb_remainder*/
        long_divmod,                /*nb_divmod*/
        long_pow,                   /*nb_power*/
        (unaryfunc)long_neg,        /*nb_negative*/
        long_long,                  /*tp_positive*/
        (unaryfunc)long_abs,        /*tp_absolute*/
        (inquiry)long_bool,         /*tp_bool*/
        (unaryfunc)long_invert,     /*nb_invert*/
        long_lshift,                /*nb_lshift*/
        long_rshift,                /*nb_rshift*/
        long_and,                   /*nb_and*/
        long_xor,                   /*nb_xor*/
        long_or,                    /*nb_or*/
        long_long,                  /*nb_int*/
        0,                          /*nb_reserved*/
        long_float,                 /*nb_float*/
        0,                          /* nb_inplace_add */
        0,                          /* nb_inplace_subtract */
        0,                          /* nb_inplace_multiply */
        0,                          /* nb_inplace_remainder */
        0,                          /* nb_inplace_power */
        0,                          /* nb_inplace_lshift */
        0,                          /* nb_inplace_rshift */
        0,                          /* nb_inplace_and */
        0,                          /* nb_inplace_xor */
        0,                          /* nb_inplace_or */
        long_div,                   /* nb_floor_divide */
        long_true_divide,           /* nb_true_divide */
        0,                          /* nb_inplace_floor_divide */
        0,                          /* nb_inplace_true_divide */
        long_long,                  /* nb_index */
    };
```

We'll begin by studying the `long_add()` function that implements integer addition.

Addition (and subtraction)

First note that a function that adds two integers can be expressed via two other functions that deal with absolute values only:

- a function that adds the absolute values of two integers; and
- a function that subtracts the absolute values of two integers.

It's possible because:

$$-|a|+(-|b|) = -(|a|+|b|)$$

$$|a|+(-|b|) = |a|-|b|$$

$$-|a|+|b| = |b|-|a|$$

CPython uses these simple identities to express the `long_add()` function via the `x_add()` function that adds the absolute values of two integers and the `x_sub()` function that subtracts the absolute values of two integers:

```
static PyObject *
long_add(PyLongObject *a, PyLongObject *b)
{
    PyLongObject *z;

    CHECK_BINOP(a, b);

    if (Py_ABS(Py_SIZE(a)) <= 1 && Py_ABS(Py_SIZE(b)) <= 1) {
        return PyLong_FromLong(MEDIUM_VALUE(a) + MEDIUM_VALUE(b));
    }
    if (Py_SIZE(a) < 0) {
        if (Py_SIZE(b) < 0) {
            z = x_add(a, b); // -|a|+(-|b|) = -(|a|+|b|)
            if (z != NULL) {
                /* x_add received at least one multiple-digit int,
                   and thus z must be a multiple-digit int.
                   That also means z is not an element of
                   small_ints, so negating it in-place is safe. */
                assert(Py_REFCNT(z) == 1);
                Py_SET_SIZE(z, -(Py_SIZE(z)));
            }
        }
        else
```

```
            z = x_sub(b, a); // -|a|+|b| = |b|-|a|
        }
        else {
            if (Py_SIZE(b) < 0)
                z = x_sub(a, b); // |a|+(-|b|) = |a|-|b|
            else
                z = x_add(a, b);
        }
        return (PyObject *)z;
    }
```

So, we need to understand how `x_add()` and `x_sub()` are implemented.

It turns out that the best way to add the absolute values of two bignums is the column method taught in the elementary school. We take the least significant digit of the first bignum, take the least significant digit of the second bignum, add them up and write the result to the least significant digit of the output bignum. If the result of the addition doesn't fit into a single digit, we write the result modulo base and remember the carry. Then we take the second least significant digit of the first bignum, the second least significant digit of the second bignum, add them to the carry, write the result modulo base to the second least significant digit of the output bignum and remember the carry. The process continues until no digits are left and the last carry is written to the output bignum. Here's CPython's implementation of this algorithm:

```
// Some typedefs and macros used in the algorithm:
// typedef uint32_t digit;
// #define PyLong_SHIFT    30
// #define PyLong_BASE     ((digit)1 << PyLong_SHIFT)
// #define PyLong_MASK     ((digit)(PyLong_BASE - 1))


/* Add the absolute values of two integers. */
static PyLongObject *
x_add(PyLongObject *a, PyLongObject *b)
{
    Py_ssize_t size_a = Py_ABS(Py_SIZE(a)), size_b = Py_ABS(Py_SIZE(b));
    PyLongObject *z;
    Py_ssize_t i;
    digit carry = 0;

    /* Ensure a is the larger of the two: */
    if (size_a < size_b) {
        { PyLongObject *temp = a; a = b; b = temp; }
        { Py_ssize_t size_temp = size_a;
            size_a = size_b;
            size_b = size_temp; }
    }
    z = _PyLong_New(size_a+1);
    if (z == NULL)
        return NULL;
    for (i = 0; i < size_b; ++i) {
        carry += a->ob_digit[i] + b->ob_digit[i];
        z->ob_digit[i] = carry & PyLong_MASK;
        carry >>= PyLong_SHIFT;
    }
    for (; i < size_a; ++i) {
        carry += a->ob_digit[i];
        z->ob_digit[i] = carry & PyLong_MASK;
        carry >>= PyLong_SHIFT;
    }
    z->ob_digit[i] = carry;
    return long_normalize(z);
}
```

First note that Python integers are immutable. CPython returns a new integer as a result of an arithmetic operation. The size of the new integer is initially set to the maximum possible size of the result. Then if, after the operation is performed, some leading digits happen to be zeros, CPython shrinks the size of the integer by calling `long_normalize()`. In the case of addition, CPython creates a new integer that is one digit longer than the larger operand. Then if, after the operation is performed, the most significant digit of the result happens to be 0, CPython decrements the size of the result by one.

Note also that a digit takes lower 30 bits of a 32-bit int. When we add two digits, we get at most 31-bit integer, and a carry is stored at the bit 30 (counting from 0), so we can easily access it.

Subtraction of the absolute values of two bignums is done in a similar manner except that carrying is replaced with borrowing. We also need to ensure that the first bignum is the larger of the two. If this is not the case, we swap the bignums and change the sign of the result after the subtraction is performed. As it is implemented in CPython, the borrowing is easy because according to the C specification, unsigned ints are a subject to a modular arithmetic:

Otherwise, if the new type is unsigned, the value is converted by repeatedly adding or subtracting one more than the maximum value that can be represented in the new type until the value is in the range of the new type.

This means that when we subtract a larger digit from a smaller one, the maximum possible int is added to the result to get a value in the valid range. For example, `1 - 2 = -1 + (2**32 - 1) = 4294967294`. To get the effect of borrowing, we just write the bits 0-29 to the result and check the bit 30 to see if the borrowing happened. Here's how CPython does all of that:

```c
// Some typedefs and macros used in the algorithm:
// typedef uint32_t digit;
// #define PyLong_SHIFT    30
// #define PyLong_BASE     ((digit)1 << PyLong_SHIFT)
// #define PyLong_MASK     ((digit)(PyLong_BASE - 1))


static PyLongObject *
x_sub(PyLongObject *a, PyLongObject *b)
{
    Py_ssize_t size_a = Py_ABS(Py_SIZE(a)), size_b = Py_ABS(Py_SIZE(b));
    PyLongObject *z;
    Py_ssize_t i;
    int sign = 1;
    digit borrow = 0;

    /* Ensure a is the larger of the two: */
    if (size_a < size_b) {
        sign = -1;
        { PyLongObject *temp = a; a = b; b = temp; }
        { Py_ssize_t size_temp = size_a;
            size_a = size_b;
            size_b = size_temp; }
    }
    else if (size_a == size_b) {
        /* Find highest digit where a and b differ: */
        i = size_a;
        while (--i >= 0 && a->ob_digit[i] == b->ob_digit[i])
            ;
        if (i < 0)
            return (PyLongObject *)PyLong_FromLong(0);
        if (a->ob_digit[i] < b->ob_digit[i]) {
            sign = -1;
            { PyLongObject *temp = a; a = b; b = temp; }
        }
        size_a = size_b = i+1;
    }
    z = _PyLong_New(size_a);
    if (z == NULL)
        return NULL;
    for (i = 0; i < size_b; ++i) {
        /* The following assumes unsigned arithmetic
           works module 2**N for some N>PyLong_SHIFT. */
        borrow = a->ob_digit[i] - b->ob_digit[i] - borrow;
        z->ob_digit[i] = borrow & PyLong_MASK;
        borrow >>= PyLong_SHIFT;
        borrow &= 1; /* Keep only one sign bit */
    }
    for (; i < size_a; ++i) {
        borrow = a->ob_digit[i] - borrow;
        z->ob_digit[i] = borrow & PyLong_MASK;
        borrow >>= PyLong_SHIFT;
        borrow &= 1; /* Keep only one sign bit */
    }
    assert(borrow == 0);
    if (sign < 0) {
        Py_SET_SIZE(z, -Py_SIZE(z));
    }
    return maybe_small_long(long_normalize(z));
}
```

The `long_sub()` function that implements integer subtraction delegates the work to `x_add()` and `x_sub()`, just like `long_add()` does. Here it is:

```
static PyObject *
long_sub(PyLongObject *a, PyLongObject *b)
{
    PyLongObject *z;

    CHECK_BINOP(a, b);

    if (Py_ABS(Py_SIZE(a)) <= 1 && Py_ABS(Py_SIZE(b)) <= 1) {
        return PyLong_FromLong(MEDIUM_VALUE(a) - MEDIUM_VALUE(b));
    }
    if (Py_SIZE(a) < 0) {
        if (Py_SIZE(b) < 0) {
            z = x_sub(b, a);
        }
        else {
            z = x_add(a, b);
            if (z != NULL) {
                assert(Py_SIZE(z) == 0 || Py_REFCNT(z) == 1);
                Py_SET_SIZE(z, -(Py_SIZE(z)));
            }
        }
    }
    else {
        if (Py_SIZE(b) < 0)
            z = x_add(a, b);
        else
            z = x_sub(a, b);
    }
    return (PyObject *)z;
}
```

Arithmetic operations on bignums are much slower that the same arithmetic operations on native integers performed by a CPU. In particular, bignum addition is much slower than CPU addition. And it is slower not only because CPU performs multiple arithmetic operations to add two bignums but mainly because bignum addition usually involves multiple memory accesses, and a memory access can be quite expensive, i.e. hundreds of times more costly than an arithmetic operation. Fortunately, CPython employs an optimization to add and subtract small integers faster. This optimization is done by the following check:

```
static PyObject *
long_sub(PyLongObject *a, PyLongObject *b)
{
    //...

    if (Py_ABS(Py_SIZE(a)) <= 1 && Py_ABS(Py_SIZE(b)) <= 1) {
        // MEDIUM_VALUE macro converts integer to a signed int
        return PyLong_FromLong(MEDIUM_VALUE(a) - MEDIUM_VALUE(b));
    }

    //...
}
```

If both integers comprise of at most one digit, CPython doesn't call `x_add()` or `x_sub()` but simply computes the result with a single operation. If the result also fits into a single digit, no more calculations are required, and bignums are effectively added (or subtracted) as if they were native integers.

Multiplication

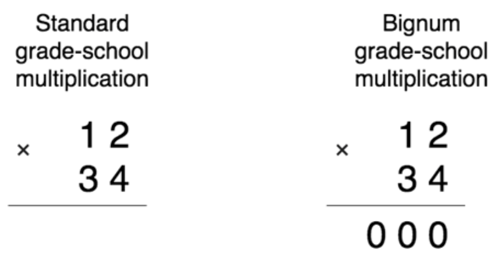
There is no silver-bullet algorithm for bignum multiplication. Several algorithms are used in practice because some perform better on relatively small bignums and others perform better on large and extremely large bignums. CPython implements two multiplication algorithms:

- the grade-school multiplication algorithm that is used by default; and
- the Karatsuba multiplication algorithm that is used when both integers have more than 70 digits.

Wikipedia summarizes the grade-school multiplication algorithm as follows:

> *Multiply the multiplicand by each digit of the multiplier and then add up all the properly shifted results.*

The bignum implementation has one important difference. Instead of storing the results of multiplying by each digit and then adding them up in the end, we add these results to the output bignum as soon as we compute them. The following gif illustrates the idea:


Standard grade-school multiplication
× 12
34
Bignum grade-school multiplication
× 12
34
000

This optimization saves both memory and time. The best way to understand other details of the algorithm is to look at the actual implementation. Here's CPython's one:

```c
// Some typedefs and macros used in the algorithm:
// typedef uint32_t digit;
// typedef uint64_t twodigits;
// #define PyLong_SHIFT    30
// #define PyLong_BASE     ((digit)1 << PyLong_SHIFT)
// #define PyLong_MASK     ((digit)(PyLong_BASE - 1))


/* Grade school multiplication, ignoring the signs.
 * Returns the absolute value of the product, or NULL if error.
 */
static PyLongObject *
x_mul(PyLongObject *a, PyLongObject *b)
{
    PyLongObject *z;
    Py_ssize_t size_a = Py_ABS(Py_SIZE(a));
    Py_ssize_t size_b = Py_ABS(Py_SIZE(b));
    Py_ssize_t i;

    // The size of the result is at most size_a + size_b
    z = _PyLong_New(size_a + size_b);
    if (z == NULL)
        return NULL;

    memset(z->ob_digit, 0, Py_SIZE(z) * sizeof(digit));
    if (a == b) {
        // ... special path for computing a square
    }
    else { /* a is not the same as b -- gradeschool int mult */

        // Iterate over the digits of the multiplier
        for (i = 0; i < size_a; ++i) {
            twodigits carry = 0;
            twodigits f = a->ob_digit[i];
            digit *pz = z->ob_digit + i;
            digit *pb = b->ob_digit;
            digit *pbend = b->ob_digit + size_b;

            // ... signal handling

            // Iterate over the digits of the multiplicand
            while (pb < pbend) {
                carry += *pz + *pb++ * f;
                *pz++ = (digit)(carry & PyLong_MASK);
                carry >>= PyLong_SHIFT;
                assert(carry <= PyLong_MASK);
            }
            if (carry)
                *pz += (digit)(carry & PyLong_MASK);
            assert((carry >> PyLong_SHIFT) == 0);
        }
    }
    return long_normalize(z);
}
```

Note that when we multiply two 30-bit digits, we can get a 60-bit result. It doesn't fit into a 32-bit int, but this is not a problem since CPython uses 30-bit digits on 64-bit platforms, so 64-bit int can be used to perform the calculation. This convenience is the primary reason why CPython doesn't use larger digit sizes.

The grade-school multiplication algorithm takes $O(n^2)$ time when multiplying two n-digit bignums. The Karatsuba multiplication algorithm takes $O(n^{\log_2 3}) = O(n^{1.584...})$. CPython uses the latter when both operands have more than 70 digits.

The idea of the Karatsuba algorithm is based on two observations. First, observe that each operand can be splited into two parts: one consisting of low-order digits and the other consisting of high-order digits:

$$x = x_1 + x_2 \times base^{len(x_1)}$$

Then a multiplication of two n-digit bignums can be replaced with four multiplications of smaller bignums. Assuming the splitting is done so that $len(x_1) = len(y_1)$,

$$xy = (x_1 + x_2 \times base^{len(x_1)})(y_1 + y_2 \times base^{len(x_1)}) = x_1y_1 + (x_1y_2 + x_2y_1) \times base^{len(x_1)} + x_2y_2 \times base^{2len(x_1}$$

The results of the four multiplications can then be calculated recursively. This algorithm, however, also works for $O(n^2)$. We can make it asymptotically faster using the following observation: four multiplications can be replaced with three multiplications at the cost of a few extra additions and subtractions because

$$x_1y_2 + x_2y_1 = (x_1 + x_2)(y_1 + y_2) - x_1y_1 - x_2y_2$$

So, we only need to calculate x_1y_1, x_2y_2 and $(x_1 + x_2)(y_1 + y_2)$. If we split each operand in such a way so that its parts have about half as many digits, then we get an algorithm that works for $O(n^{\log_2 3})$. Success!

The bignum division is a bit harder to implement. We won't discuss it here, but it's essentially the familiar long division algorithm. Check out `Objects/longobject.c` to see how bignum division and other arithmetic operations are implemented in CPython. The descriptions of the implemented algorithms can be found in the chapter 14 of the Handbook of Applied Cryptography by Alfred Menezes (it's free!).

CPython's bignums vs other bignum implementations

How fast is CPython's implementation of bignums compared to other implementations? While it's not the easiest task to come up with a totally representative test, we can get some idea. The Benchmarks Game has a pidigits benchmark that measures the performance of bignums in different programming languages. The benchmark asks to implement a specific algorithm for generating digits of pi. You can find the results here. One important thing to know about these results is that the fastest programs use bignums provided by the GMP library and not the bignums provided by the language. If we exclude the programs that use GMP bindings, we get the following results:

#	source	secs
1	Haskell GHC #5 *	0.75
2	Chapel #2 *	0.76
3	Julia *	1.56
4	Go #8	2.68
5	Dart #2	3.33
6	**Python 3 #4**	**3.85**
7	OCaml #5	4.36
8	Lisp SBCL #2	5.77
9	Node js #4	6.15
10	Java	7.61
11	Erlang HiPE #3	7.94
12	VW Smalltalk #4	8.02

#	source	secs
13	Racket	11.40
14	Free Pascal	14.65
15	Ruby	17.10
16	PHP	5 min

Some languages rely on GMP to implement built-in bignums. They are marked with an asterisk (*).

The results show that CPython's implementation has decent performance. Still, GMP proves that bignums can be implemented even more efficiently. The natural question to ask is: What makes GMP's bignums faster than CPython's bignums? I can think of three main reasons:

1. Some parts of GMP are written in assembly language. The code is highly optimized for different CPU architectures.
2. GMP uses bigger digit sizes. It uses 64-bit digits on 64-bit platforms and 32-bit digits on 32-bit platforms. As a result, GMP represents the same integers with fewer digits. Thus, arithmetic operations are performed faster. This is possible due to reason 1. For example, GMP can read the carry flag or use the `adc` instruction to add with carry. It can also get the 128-bit result of mutiplying two 64-bit integers with the `mul` instruction.
3. GMP uses more sophisticated algorithms to do bignum arithmetic. For example, the Karatsuba algorithm is not the asymptotically fastest multiplication algorithm. And GMP implements seven different multiplication algorithms. Which one is used depends on the operands' sizes.

The performance of CPython's bignums should be enough for most applications. When it's not enough, GMP's bignums can be used in a Python program via the `gmpy2` module.

For more comments on the results of the pidigits benchmark, check out this article.

Memory usage considerations

Python integers take a considerable amount of memory. Even the smallest integers take 28 bytes on 64-bit platforms:

- a reference count `ob_refcnt`: 8 bytes
- a type `ob_type`: 8 bytes
- an object's size `ob_size`: 8 bytes
- `ob_digit`: 4 bytes.

Allocating a list of a million integers requires allocating the integers themselves plus a million references to them, which is about 35 megabytes in total. Compare it to 4 megabytes required to allocate an array of a million 32-bit ints. So, sometimes it makes sense to use the `array` module or `numpy` to store large amounts of homogenous data.

We said before that CPython creates a new integer object on every arithmetic operation. Fortunately, it employs an optimization to allocate small integers only once during the interpreter's lifetime. The integers in the range [-5, 256] are preallocated when CPython starts. Then, when CPython needs to create a new integer object, it first checks if the integer value is in the range [-5, 256] and, if it is in the range, returns the preallocated object. The elimination of extra memory allocations saves both memory and time.

The range [-5, 256] is chosen because the values in this range are extensively used throughout CPython and the Python standard library. For more details on the choice, check out this article.

Conclusion

The design of built-in types has certainly contributed to the Python's success. Python integers serve as an example of a quite efficient and, at the same time, accessible bignum implementation. We made use of this fact today to learn both about Python integers and about bignums. Next time we'll continue to study Python built-in types. Stay tuned to learn about how Python strings work.

If you have any questions, comments or suggestions, feel free to contact me at victor@tenthousandmeters.com