Hacking PyLongObject on Python 3.2

Cesare Di Mauro

EuroPython 2011 – Florence

June 2011

Integers on Python

Before Python 3

- 32 Bits ints: ± 2Gi = ≈ 2.14×10^9
- 64 Bits ints: ± 8Ei = ≈ 9.22×10^{18}

Depends on CPU architecture

- Longs: variable (limited by memory)

Int overflow => conversion to long

`PyIntObject`

`PyLongObject`

From Python 3

- Longs: variable (limited by memory)

PyIntObject

```
typedef struct {
  Py_ssize_t ob_refcnt;
  struct _typeobject *ob_type;
  long ob_ival;
} PyIntObject;
```

PyObject

"Common base"

Concrete (signed) integer value (32 or 64 bits)

Points to object's "type" structure (**PyInt_Type**)

Keeps track of object's reference counting

PyLongObject

```
typedef unsigned int digit;
typedef unsigned long twodigits;
#define PyLong_SHIFT    15 /* or 30 in Python >= 3.0 */

typedef struct _longobject PyLongObject;
/* Revealed in longintrepr.h */

struct _longobject {
  Py_ssize_t ob_refcnt;
  struct _typeobject *ob_type;
  Py_ssize_t ob_size;
  digit ob_digit[1];
};
```

PyObject_VAR_HEAD

"Common base"

The "digits"

Number of items in variable part

Some examples

Value	ob_size	ob_digit[0]	ob_digit[1]	Digit Size
0	0	Not used	Not used	Any
1000000	2	16960	30	15 bits
1000000	1	1000000	Not used	30 bits
-1000000	-1	1000000	Not used	30 bits
-10000000000	-2	336323584	9	30 bits

New long object structure

```
typedef unsigned int digit;
typedef unsigned long twodigits;
#define PyLong_SHIFT    30

typedef struct _longobject PyLongObject;
/* Revealed in longintrepr.h */

struct _longobject {
  Py_ssize_t ob_refcnt;
  struct _typeobject *ob_type;
  Py_ssize_t ob_size;
  digit ob_digit[1];
};
```

Changed meaning!

The idea

- **`ob_size`** always >= 1 (“regular” usage; same as lists, etc.)
- **`ob_digit[ob_size - 1]`** = Most Significant Digit (MSD)
- MSD “holds” the sign for non-zero values (2 complement)

“Short” integers (1 digit) work like old ints

- **`ob_size`** = 1
- **`ob_digit[0]`** = MSD = 31 bits signed integer

“Long” integers (2+ digits) work like longs

- **`ob_size`** > 1
- **`ob_digit[<= ob_size - 2]`** = 30 bits unsigned integers
- **`ob_digit[ob_size - 1]`** = MSD = 31 bits signed integer

Comparing the structures

Old PyLongObject

Value	ob_size	ob_digit[0]	ob_digit[1]	Digit Size
0	0	Not used	Not used	Any
~~1000000~~	~~2~~	~~16960~~	~~30~~	~~15 bits~~
1000000	1	1000000	Not used	30 bits
-1000000	-1	1000000	Not used	30 bits
-10000000000	-2	336323584	9	30 bits

New PyLongObject

Value	ob_size	ob_digit[0]	ob_digit[1]	Digit Size
0	1	0	Not used	30 bits
1000000	1	1000000	Not used	30 bits
-1000000	1	-1000000	Not used	30 bits
-10000000000	2	336323584	-9	30 bits

Make the common case fast…

```
#define ABS(x) ((x) < 0 ? -(x) : (x))
#define MEDIUM_VALUE(x)(Py_SIZE(x) < 0 ? -(sdigit)(x)->ob_digit[0] : \
  (Py_SIZE(x) == 0 ? (sdigit) 0 : (sdigit)(x)->ob_digit[0]))
static PyObject * long_add(PyLongObject *a, PyLongObject *b)
{
    CHECK_BINOP(a, b);
    if (ABS(Py_SIZE(a)) <= 1 && ABS(Py_SIZE(b)) <= 1)
        return PyLong_FromLong(MEDIUM_VALUE(a) + MEDIUM_VALUE(b));
```

Check for “short” int

Get “short” int value

```
#define LSD(v) ((sdigit) (v)->ob_digit[0])
static PyObject * long_add(PyLongObject *a, PyLongObject *b)
{
    CHECK_BINOP(a, b);
    if (Py_SIZE(a) == 1 && Py_SIZE(b) == 1)
        return PyLong_FromLong(LSD(a) + LSD(b));
```

Check for “short” int

Get “short” int value

…slowing down the “long” one

```
if (Py_SIZE(a) < 0) {
  if (Py_SIZE(b) < 0) {
    z = x_add(a, b);
    if (z != NULL && Py_SIZE(z) != 0)
      Py_SIZE(z) = -(Py_SIZE(z));
  }
  else
    z = x_sub(b, a);
}

else {
  if (Py_SIZE(b) < 0)
    z = x_sub(a, b);
  else
    z = x_add(a, b);
}
```

Set abs value

Restore the MSD

```
#define MSD(v) \
  ((sdigit) (v)->ob_digit[Py_SIZE(v) - 1])
sdigit msd_a = MSD(a);
sdigit msd_b = MSD(b);
if (msd_a < 0) {
  MSD(a) = -msd_a;
  if (msd_b < 0) {
    MSD(b) = -msd_b;
    z = x_add(a, b);
    if (z != NULL)
      SIGN_CHANGE(z);
    MSD(b) = msd_b;
  }
  else
    z = x_sub(b, a);
  MSD(a) = msd_a;
}
```

Save the MSD!

```
else {
  if (msd_b < 0) {
    MSD(b) = -msd_b;
    z = x_sub(a, b);
    MSD(b) = msd_b;
  }
  else
    z = x_add(a, b);
}


#define SIGN_CHANGE(v)\
  MSD(v) = -MSD(v);
```

maybe_small_long

```
static PyLongObject * maybe_small_long(PyLongObject *v)
{
    if (v && ABS(Py_SIZE(v)) <= 1) {
        sdigit ival = MEDIUM_VALUE(v);
        […]
    }
    return v;
}
```

Check for “short” int

Get “short” int value

```
static PyLongObject * maybe_small_long(PyLongObject *v)
{
    if (v && Py_SIZE(v) == 1) {
        sdigit ival = LSD(v);
        […]
    }
    return v;
}
```

Check for “short” int

Get “short” int value

NEGATE

```
/* If a freshly-allocated long is already shared, it must
   be a small integer, so negating it must go to PyLong_FromLong */
#define NEGATE(x) \
    do if (Py_REFCNT(x) == 1) Py_SIZE(x) = -Py_SIZE(x); \
       else { PyObject* tmp = PyLong_FromLong(-MEDIUM_VALUE(x)); \
           Py_DECREF(x); (x) = (PyLongObject*)tmp; } \
    while(0)
```

Get “short” int value

```
/* If a freshly-allocated long is already shared, it must
   be a small integer, so negating it must go to PyLong_FromLong */
#define NEGATE(x) \
    do if (Py_REFCNT(x) == 1) {SIGN_CHANGE(x);} \
       else { PyObject* tmp = PyLong_FromLong(-LSD(x)); \
           Py_DECREF(x); (x) = (PyLongObject*)tmp; } \
    while(0)
```

long_normalize

```
static PyLongObject * long_normalize(register PyLongObject *v)
{
    Py_ssize_t j = ABS(Py_SIZE(v));    Py_ssize_t i = j;
    while (i > 0 && v->ob_digit[i - 1] == 0)
        --i;
    if (i != j)
        Py_SIZE(v) = (Py_SIZE(v) < 0) ? -(i) : i;
    return v;
}
```

Get size == num digits

```
static PyLongObject * long_normalize(register PyLongObject *v)
{
    Py_ssize_t i = Py_SIZE(v);
    while (i > 1 && v->ob_digit[i - 1] == 0)
        --i;
    Py_SIZE(v) = i;
    return v;
}
```

PyLong_Copy

```
PyObject * _PyLong_Copy(
    PyLongObject *src)
{
  PyLongObject *result;  Py_ssize_t i;
  i = Py_SIZE(src);
  if (i < 0)
    i = -(i);
  if (i < 2) {
    sdigit ival = src->ob_digit[0];
    if (Py_SIZE(src) < 0)
      ival = -ival;
    CHECK_SMALL_INT(ival);
  }
  […]
  return (PyObject *)result;
}
```

Size holds sign

```
PyObject *_PyLong_Copy(
    PyLongObject *src)
{
  PyLongObject *result;  Py_ssize_t i;
  i = Py_SIZE(src);
  if (i == 1) {
    sdigit ival = LSD(src);
    CHECK_SMALL_INT(ival);
  }
  […]
  return (PyObject *)result;
}
```

If negative int, change sign

Comparisons: a pain in the neck

```
static int long_compare(
  PyLongObject *a, PyLongObject *b)
{
  Py_ssize_t sign;
  if (Py_SIZE(a) != Py_SIZE(b)) {
    sign = Py_SIZE(a) - Py_SIZE(b);
  }
  else { /* Equal sized */
    […]
  }
  return sign < 0 ?
    -1 : sign > 0 ? 1 : 0;
}
```

Change size sign if negative

```
static int long_compare(
  PyLongObject *a, PyLongObject *b)
{
  Py_ssize_t sign;
  Py_ssize_t i = Py_SIZE(a), j = Py_SIZE(b);
  if (i != j)
    sign = (FAST_MSD(a, i) < 0 ? -i : i) -
           (FAST_MSD(b, j) < 0 ? -j : j);
  else { /* Equal sized */
    […]
  }
  return sign < 0 ?
    -1 : sign > 0 ? 1 : 0;
}
#define FAST_MSD(v, size) \
  ((sdigit) (v)->ob_digit[(size) - 1])
```

…again!

```
else {
  Py_ssize_t i = ABS(Py_SIZE(a));
  while (--i >= 0 &&
    a->ob_digit[i] == b->ob_digit[i])
    ;
  if (i < 0)
    sign = 0;
  else {
    sign = (sdigit)a->ob_digit[i] -
           (sdigit)b->ob_digit[i];
    if (Py_SIZE(a) < 0)
      sign = -sign;
  }
}
```

```
else {
  i--; /* Decrease size. */
  sign = (sdigit)a->ob_digit[i] -
         (sdigit)b->ob_digit[i];
  if (i && sign == 0) {
    while (--i >= 0 &&
      a->ob_digit[i] == b->ob_digit[i])
      ;
    if (i < 0)
      sign = 0;
    else {
      sign = (sdigit)a->ob_digit[i] -
             (sdigit)b->ob_digit[i];
      if (IS_NEGATIVE(a))
        sign = -sign;
    }
  }
}
```

More digits, same MSDs?

#define IS_NEGATIVE(v) (MSD(v) < 0)

Changing sign is easier…

```
static PyObject *
long_neg(PyLongObject *v)
{
  PyLongObject *z;
  if (ABS(Py_SIZE(v)) <= 1)
    return PyLong_FromLong(
      -MEDIUM_VALUE(v));
  z = (PyLongObject *)_PyLong_Copy(v);
  if (z != NULL)
    Py_SIZE(z) = -(Py_SIZE(v));
  return (PyObject *)z;
}
```

```
static PyObject *
long_neg(PyLongObject *v)
{
  PyLongObject *z;
  if (Py_SIZE(v) == 1)
    return PyLong_FromLong(-LSD(v));
  z = (PyLongObject *)_PyLong_Copy(v);
  if (z != NULL)
    SIGN_CHANGE(z);
  return (PyObject *)z;
}
```

…absolute & bool a bit slower

```
static PyObject *
long_abs(PyLongObject *v)
{
  if (Py_SIZE(v) < 0)
    return long_neg(v);
  else
    return long_long((PyObject *)v);
}

static int
long_bool(PyLongObject *v)
{
  return Py_SIZE(v) != 0;
}
```

```
static PyObject *
long_abs(PyLongObject *v)
{
  if (IS_NEGATIVE(v))
    return long_neg(v);
  else
    return long_long((PyObject *)v);
}

static int
long_bool(PyLongObject *v)
{
  return MSD(v) != 0;
}
```

Hashing: good for “short” ints

```
static Py_hash_t
long_hash(PyLongObject *v)
{
  Py_uhash_t x; Py_ssize_t i;
  int sign;
  i = Py_SIZE(v);
  switch(i) {
    case -1: return v->ob_digit[0]==1 ?
      -2 : -(sdigit)v->ob_digit[0];
    case 0: return 0;
    case 1: return v->ob_digit[0];
  }
  sign = 1; x = 0;
  if (i < 0) {
    sign = -1; i = -(i);
  }
```

```
static Py_hash_t
long_hash(PyLongObject *v)
{
  Py_uhash_t x; Py_ssize_t i;
  int sign; sdigit msd;
  i = Py_SIZE(v);
  if (i == 1) {
    msd = LSD(v);
    return msd == -1 ? -2 : msd;
  }
  msd = FAST_MSD(v, i); sign = 1; x = 0;
  if (msd < 0) {
    sign = -1; FAST_MSD(v, i) = -msd;
  }
  […]
  MSD(v) = msd;
```

Inside PyLong_FromLong

```
PyObject * PyLong_FromLong(long ival)
{ unsigned long abs_ival, t;
  int ndigits = 0; int sign = 1;
  CHECK_SMALL_INT(ival);
  if (ival < 0) {
    abs_ival = 0U-(unsigned long)ival;
    sign = -1;
  } else abs_ival = (unsigned long)ival;
  if (!(abs_ival >> PyLong_SHIFT)) {
    PyLongObject *v = _PyLong_New(1);
    if (v) {
      Py_SIZE(v) = sign;
      v->ob_digit[0] = Py_SAFE_DOWNCAST(
        abs_ival, unsigned long, digit);
    }
    return (PyObject*)v; }
```

```
PyObject * PyLong_FromLong(long ival)
{ unsigned long abs_ival;
  sdigit sign; int sign = 1;
  CHECK_SMALL_INT(ival);
  if (ival < 0) {
    abs_ival = 0U-(unsigned long)ival;
    sign = -1;
  } else abs_ival = (unsigned long)ival;
  if (abs_ival < PyLong_BASE) {
    PyLongObject *v = _PyLong_New(1);
    if (v)
      LSD(v) = (sdigit) ival;
    return (PyObject*)v;
  }
```

… unrolled!

```
t = abs_ival;
while (t) { /* Loop to determine */
  ++ndigits; /* number of digits */
  t >>= PyLong_SHIFT;
}
v = _PyLong_New(ndigits);
if (v != NULL) {
  digit *p = v->ob_digit;
  Py_SIZE(v) = ndigits*sign;
  t = abs_ival;
  while (t) {
    *p++ = Py_SAFE_DOWNCAST(
  t & PyLong_MASK, unsigned long, digit);
    t >>= PyLong_SHIFT;
  }
}
```

```
#ifdef LONG64BITS
if (abs_ival <= PyLongLong_MASK) {
#endif
  if (v = _PyLong_New(2)) {
    v->ob_digit[0] = (digit)
      abs_ival & PyLong_MASK;
    v->ob_digit[1] = (sdigit)
      (abs_ival >> PyLong_SHIFT) * sign;
  }
#ifdef LONG64BITS
} else { v = _PyLong_New(3);
  if (v) {
 v->ob_digit[0] = (digit)
        abs_ival & PyLong_MASK;
 v->ob_digit[1] = (digit)(abs_ival >>
         PyLong_SHIFT) & PyLong_MASK;
 v->ob_digit[2] = (sdigit)(abs_ival >>
         PyLong_SHIFT * 2) * sign;
  }
} #endif
```

Inside PyLong_AsLongAndOverflow

```
long PyLong_AsLongAndOverflow(
    PyObject *vv, int *overflow)
{ register PyLongObject *v;
  unsigned long x, prev;
  long res = -1;
  Py_ssize_t i;  int sign;
  *overflow = 0;
  v = (PyLongObject *)vv;
  switch (i = Py_SIZE(v)) {
    case -1:
  res = -(sdigit)v->ob_digit[0]; break;
    case 0:
  res = 0; break;
    case 1:
  res = v->ob_digit[0]; break;
 default:
```

```
long PyLong_AsLongAndOverflow(
    PyObject *vv, int *overflow)
{ register PyLongObject *v;
  unsigned long x;

  Py_ssize_t i; long sign;
  *overflow = 0;
  v = (PyLongObject *)vv;
  i = Py_SIZE(v);
  if (i-- == 1)
    x = (long) LSD(v);
  else {
```

…unrolled too!

```
sign = 1;
x = 0;
if (i < 0) {
  sign = -1;
  i = -(i);
}
while (--i >= 0) {
  prev = x;
  x = (x << PyLong_SHIFT) |
       v->ob_digit[i];
  if ((x >> PyLong_SHIFT) != prev) {
    *overflow = sign;
    goto exit;
  }
}
```

```
sdigit msd = v->ob_digit[i--]; sign = 1;
if (msd < 0) {
  sign = -1; msd = -msd;
}
#ifdef LONG64BITS
if (i <= 1) {
  x = ((long) msd << PyLong_SHIFT) + v->ob_digit[i];
  if (i) {
    unsigned long prev = x;
    x = (x << PyLong_SHIFT) + v->ob_digit[0];
    if (x >> PyLong_SHIFT != prev)
      goto overflow;
  }
}
#else
if (!i) {
  x=((long) msd << PyLong_SHIFT) + v->ob_digit[0];
  if (x >> PyLong_SHIFT != msd)
    goto overflow;
}
#endif
else
  goto overflow;
```

Some “numbers”…

- 6 files changed
- ~230 changes
- ~5.000 lines “involved”

Changed files:

- `Include/longintrepr.h` (1)
- `Lib/test/test_sys.py` (1)
- `Modules/mathmodule.c` (1)
- `Objects/longobject.c` (215)
- `Python/import.c` (2)
- `Python/marshal.c` (9)

Code coupling & dependencies

From Include/longobject.h:

```
typedef struct _longobject PyLongObject;
/* Revealed in longintrepr.h */
```

Public interface: Include/longobject.h

Private interface: Include/longintrepr.h

Interface violation: mathmodule.c, marshal.c (, test_sys.py)

Getting mad with it (5000 lines to analyze and “follow”)!

Coupling / dependencies should be well documented!

Biggest benefits

- Immediate access to long size (no ABS macro)
- Immediate check for common case (“short ints”, Size == 1)
- Immediate value for common case (no MEDIUM_VALUE mac.)

Biggest drawbacks

- More complex internal sign access (must get MSD)
- More complex internal sign change (ditto)
- Longs require MSD “backup” (internal routines work on absolute digits) and “restore”
- More complex comparisons

Some benchmarks*

Test	Minimum		Result	Average		Result
call_method	0.561	0.530	6% faster	0.570807	0.557547	2% faster
call_method_slots	0.530	0.531	0% slower	0.540573	0.561497	4% slower
call_method_unknown	0.592	0.592	0% faster	0.616513	0.603877	2% faster
nbody	0.453	0.436	4% faster	0.471910	0.450370	5% faster
nqueens	0.390	0.359	9% faster	0.399990	0.375650	6% faster
slowpickle	0.655	0.670	2% slower	0.667210	0.681410	2% slower
iterative_count	0.218	0.202	8% faster	0.229320	0.215590	6% faster
threaded_count	0.218	0.202	8% faster	0.227290	0.215750	5% faster

*only significant results reported

Taken from: http://speed.python.org/ v. 62e754c57a7f

python.exe perf.py -r -b bench_name standard_32_bits/python.exe new_32_bits/python.exe

Windows 7 x64, AMD Phenom II X4 955, 8GB DDR3 PC3-10700

Compiled with Microsoft Visual C++ 2008 Express Edition

Again!

Test	Minimum		Result	Average		Result
call_method	0.561	0.530	6% faster	0.573613	0.544483	5% faster
call_method_slots	Not significant results					
call_method_unknown	Not significant results					
nbody	0.452	0.436	4% faster	0.475180	0.450370	6% faster
nqueens	0.390	0.374	4% faster	0.407780	0.389690	5% faster
slowpickle	Not significant results					
iterative_count	No change			0.233370	0.222300	5% faster
threaded_count	Not significant results					
richards	0.250	0.265	6% slower	0.268790	0.274560	2% slower

ONE line changed (long_compare). Newer code optimized!

Got a strange behavior…

Better (micro)benchmark needed!

Wider digits?

- 64 bits processor may use 60 bits digits
- 60 bits digits -> 61 bits “short” ints (± 1Ei = ≈ 1.15×10^{18})
- Wider digits means less “longs” usage
- Requires 128 bits integer math (long long long? __int128?)

Some (non mutually-exclusive) solutions:

- Implement 128 bits math (expensive, slow)
- Split 60 bits digit into two 30 bits digits (duplicate some code)
- Use assembly (can avoid some 128 bits calculations)

Future directions

Fast code path for “common cases” (“short” ints):

- Left and right shifts
- Bitwise operators (and, or, xor)
- Division and modulus (and power?)
- Comparisons

Intern some widely used constants (0, 1, 10)

Group PyLong_As* and PyLong_From* APIs

Introduce assembly (x86, ARM) for some critical code

Rethink memory allocation logic

Thanks to

Martin v. Löwis

Hints on taking advantage of the GIL

Python Developers

Python! And the GREAT test suite

Microsoft

Visual Studio!

My family

To stand me...