Topic 2.1: Integers (`int`)

Python integers are arbitrary-precision, meaning they can grow as large as memory allows. There is no equivalent to Java's `int`, `long`, `short`, or `byte` – just `int`.

In Python, an `int` object is able to store an integer of variable size because, as mentioned previously, all variables in Python are objects, and we previously showed a diagrammatic representation of an `int` object that stores the value `42` in CPython.

For reasons of simplifying the implementation (preventing overflow when performing intermediate calculations), CPython actually only uses 30 of the 32 bits in each 4-bytes of integer storage. Thus the maximum value CPython stores in a 28-byte object is the largest integer that fits into 30 bits: the value $2^{30} - 1$ (Python: `my_int = 2**30 - 1`). An `int` object that stores this value is shown diagrammatically below. Notice the `obj_data` contains the expected binary value.

```
                 my_int
                 ┌──────────────────┐
                 │ 000000C1_A027C42C │
                 └──────────────────┘
                             │
                             ▼  PyObject
         obj_refcnt  ┌──────────────────┐
                     │ 00000000_00000001 │
           obj_type  ├──────────────────┤
                     │ 000000C3_1B7C6910 │───▶ PyTypeObject
           obj_size  ├──────────────────┤
                     │ 00000000_00000004 │
           obj_data  ├──────────┐
                     │ 3FFFFFFF │
                     └──────────┘
```

As discussed previously, the variable – in this case `my_int` – stores the reference to the `int` object, and that object contains the object information, such as the object type, along with the bits required to store the integer value.

For an integer that will not fit into 30 bits, a larger `int` object is created. For example, if we added one to `my_int` (Python: `my_int = my_int + 1`), the `int` object will use two 4-byte storage location for the integer. This is shown diagrammatically below. Notice that `obj_data` is 8-bytes in length and the 31st bit of the number is stored as the least significant bit of the second 4-byte storage (the least significant word comes first).

```
                 my_int
                 ┌──────────────────┐
                 │ 000000C1_A027C448 │
                 └──────────────────┘
                             │
                             ▼  PyObject
         obj_refcnt  ┌──────────────────┐
                     │ 00000000_00000001 │
           obj_type  ├──────────────────┤
                     │ 000000C3_1B7C6910 │───▶ PyTypeObject
           obj_size  ├──────────────────┤
                     │ 00000000_00000004 │
           obj_data  ├──────────────────┤
                     │ 00000000_00000001 │
                     └──────────────────┘
```

You should not feel you need to memorize the details of the implementation. These details are presented here so that you have a conceptual understanding of how an integer of arbitrary size can be implemented in software, even though the underlying hardware only implements instructions that operate on specific and limited numbers of bits.

The following is some code that demonstrates some of the concepts presented above.

```
>>> import sys
>>> sys.getsizeof(1)
28

>>> x = 2**30-1
>>> x
1073741823
>>> sys.getsizeof(x)
28

>>> x = x + 1
>>> x
1073741824
>>> sys.getsizeof(x)
32

>>> 2**100
1267650600228229401496703205376

>>> 2**1000
10715086071862673209484250490600018105614048117055336074437503
88370351051124936122493198378815695858127594672917553146825187
14528569231404359845775746985748039345677482423098542107460506
23711419775418215304647498358194126739876755916554394607706291
45711964776865421676604298316526243868372056680693376
```