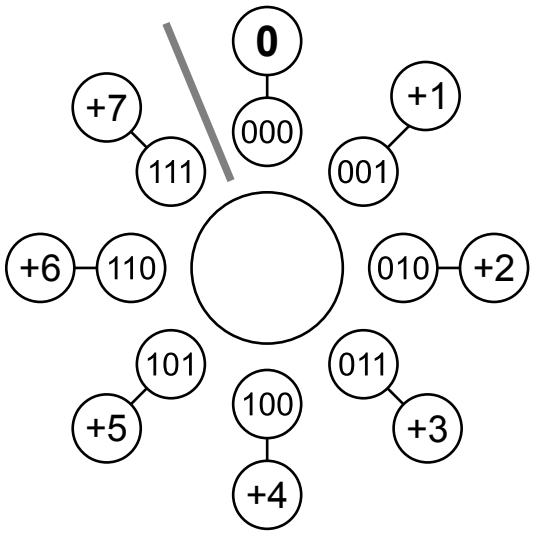
Topic 2.1: Integers (`int`)

Unsigned Binary Representation of Integers

A natural number (any non-negative integer) can be represented in binary using *unsigned binary* representation. In this case, we simply assign increasing integer values to increasing binary values, as shown in the table, below.

binary	***denary***
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7



Unsigned Binary Assignment of Denary Values

The diagram above shows these assignments in a strange form wrapped around a circle. Let me explain why.

A computer's physical hardware can store a limited number of bits. It is common for computer hardware to be limited to either 32 or 64 bits per data item (for a 32-bit processors and 64-bit processors, respectively).

To make calculations more simple, consider what would happen if we only have 3-bits to store an unsigned integer. The maximum number of different values (number of *symbols*) that can be stored in 3 bits is 8 values (2^3 values). If the value zero is included, then the natural numbers that can be stored in 3 bits ranges from 0 to 7. Examine the table above to see the most logical assignment of denary values to the binary numbers.

What will happen if an operation results in a value that is not within the range of 0 to 7? Examine the following example where 1 is added to 7:

```
  111      7
+ 001    + 1
 1000      0
```

If there were 8 bits to store the number, the binary representation would be correct. However, since there are only 3 bits to store the result of the operation, the upper 1 bit is "lost". This is called ***overflow***. If we add 1 to 7, the resulting number is 0. For the circular diagram, above, you can see addition moves clockwise around the circle from 1 to 2 to 3, and so on. If we move clockwise from

the value 7, the next position is 0. If we add 1 to the maximum unsigned integer that can be stored, the result "wraps around" and becomes 0. Similarly, if we subtract 1 from 0 (the minimum unsigned integer that can be stored), the result "wraps around" the other way and becomes the maximum unsigned integer value. For a 3-bit unsigned number, that value is 7.

Use the circular diagram above to examine what the result is if we start with denary 6 and move clockwise 5 spaces around the circle.

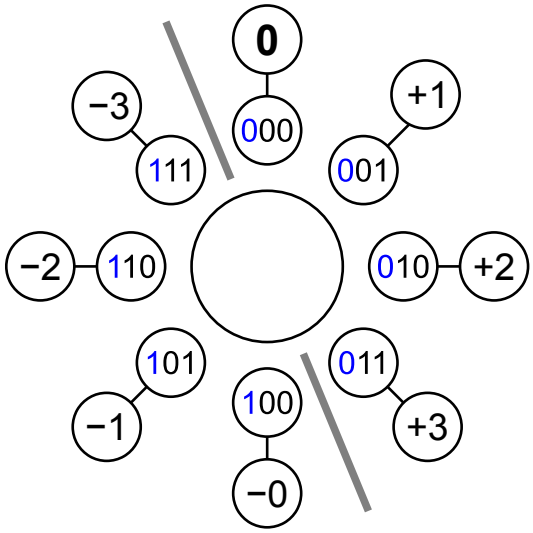
```
  110     6
+101    +5
 1011    3
```

Although the expected denary result would be eleven (11), the result with the limited binary digits is 011, which is the 3-bit unsigned representation of 3.

Sign And Magnitude Binary Representation of Integers

For sign and magnitude representation, the first bit determines whether the represented number is positive (0) or negative (1), while the remaining bits are interpreted as an unsigned number. The table and diagram below shows the denary values assigned to a 3-bit binary number using sign and magnitude representation. The sign bit has been coloured blue, while the number magnitude is black.

binary	*denary*
000	0
001	1
010	2
011	3
100	-0
101	-1
110	-2
111	-3



Sign and Magnitude Binary Assignment of Denary Values

The operation of addition and subtraction becomes cumbersome. If we wish to add a positive denary value to a positive denary number, in binary, it is normal addition.

```
  010     2
+001    +1
  011     3
```

However, if we wish to add a positive denary value to a negative denary number, we actually need to perform binary subtraction. Thus, the designer of the computer processor needs to use specialized circuitry to check the sign of each value before performing an addition or subtraction operation.

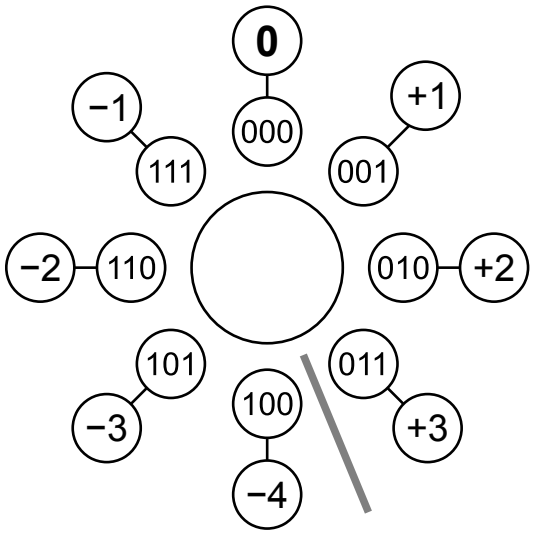
Binary Addition (incorrect result)		Binary Subtraction (correct result)	
110	$=-2$	110	$=-2$
+001	+1	− 001	+1
111	-3	101	-1

Although the sign and magnitude representation of denary numbers seems intuitive to humans, the additional complexity of implementing this representation in computer hardware makes it an extremely rare representation in practice. The next representation we will discuss, two's complement, has become the standard representation.

Two's Complement Binary Representation of Integers

The table below shows the two's complement binary representation of the denary values for 3-bit binary.

binary	***denary***
000	0
001	1
010	2
011	3
100	-4
101	-3
110	-2
111	-1



Two's Complement Assignment Denary of Values

For an integer represented in binary using three-bit two's complement, the maximum representable value is $+3$, and the minimum representable negative value is -4 . If we add 1 to the maximum positive integer that can be stored, the result will overflow and "wrap around" to become the minimum negative integer that can be stored. Similarly, if we subtract 1 from the minimum negative integer that can be stored, the result will overflow and "wrap around" to become the maximum positive integer that can be stored. Verify this for yourself using the circular diagram above right.

Integers in Java

Java has four different integer types: `byte`, `short`, `int`, and `long`. Each type uses a different number of bits, so is able to store a different range of integers, as shown in the table, below. Unlike some other languages, such as C, Java does not have any unsigned integer type. All Java integers are stored in two's complement form.

type	***bits of storage***	***range (exponential)***	***range***
`byte`	8	$-(2^7)$ to $+(2^7-1)$	-128 to $+127$
`short`	16	$-(2^{15})$ to $+(2^{15}-1)$	-65536 to $+65535$
`int`	32	$-(2^{31})$ to $+(2^{31}-1)$	-2147483648 to $+2147483647$
`long`	64	$-(2^{63})$ to $+(2^{63}-1)$	-9223372036854775808 to $+9223372036854775807$

Examine the short Java program and the resulting output that demonstrates the maximum and minimum values that may be stored for the `int` and `long` types, as well as overflow of the maximum and minimum `int` values.

```
public class TestingIntegers {
    public static void main(String[] args) {
        int num = 0x7FFF_FFFF;
        System.out.println("0x7FFF_FFFF           = " + num);
        System.out.println("Integer.MAX_VALUE     = " +
            Integer.MAX_VALUE);
        System.out.println("Integer.MAX_VALUE + 1 = " +
            (Integer.MAX_VALUE + 1) + "\n");

        int neg = 0x8000_0000;
        System.out.println("0x8000_0000           = " + neg);
        System.out.println("Integer.MIN_VALUE     = " +
                           Integer.MIN_VALUE);
        System.out.println("Integer.MIN_VALUE - 1 = " +
                           (Integer.MIN_VALUE - 1) + "\n");

        System.out.println("Long.MAX_VALUE in hex     = " +
                            Long.toHexString(Long.MAX_VALUE));
        System.out.println("Long.MAX_VALUE in decimal = " +
                            Long.MAX_VALUE);
        System.out.println("Long.MAX_VALUE in hex     = " +
                 Long.toHexString(Long.MIN_VALUE));
        System.out.println("Long.MIN_VALUE in decimal = " +
                            Long.MIN_VALUE);
    }
}
```

Resulting output:

```
0x7FFF_FFFF           = 2147483647
Integer.MAX_VALUE     = 2147483647
Integer.MAX_VALUE + 1 = -2147483648

0x8000_0000           = -2147483648
Integer.MIN_VALUE     = -2147483648
Integer.MIN_VALUE - 1 = 2147483647

Long.MAX_VALUE in hex     = 7fffffffffffffff
Long.MAX_VALUE in decimal = 9223372036854775807
Long.MAX_VALUE in hex     = 8000000000000000
Long.MIN_VALUE in decimal = -9223372036854775808
```

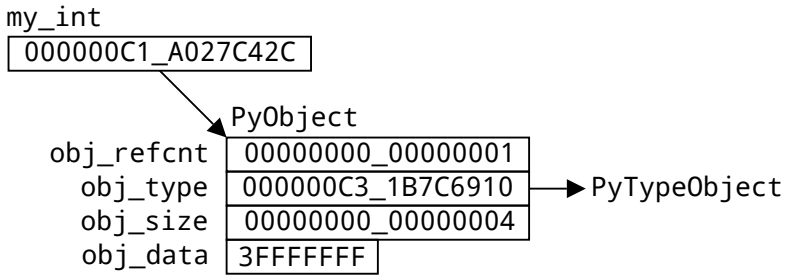
Python Integers

Integers are of Arbitrary Length

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

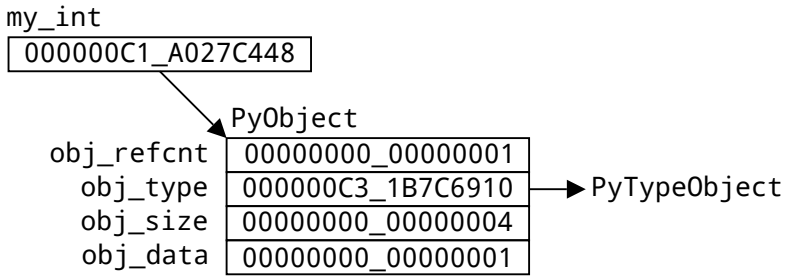
In Python, an `int` object is able to store an integer of variable size because, as mentioned previously, all variables in Python are objects, and we previously showed a diagrammatic representation of an `int` object that stores the value `42` in CPython.

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.



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).



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
>>> sys.getsizeof(2**100)
40
>>> 2**1000
10715086071862673209484250490600018105614048117055336074437503
88370351051124936122493198378815695858127594672917553146825187
14528569231404359845775746985748039345677748242309854210746050
62371141877954182153046474983581941267398767559165543946077062
914571196477686542167660429831652624386837205668069376
>>> sys.getsizeof(2**1000)
160
```

Integer Operations