Python Variables and Memory Model

Table of contents

Memory 1
Interpreting memory: the integer type 1
Addresses and `id` 2
Variables 3

Memory

Computer memory is a very long series of on/off switches, called **bits** (short for *binary digit*). Eight consecutive bits form a **byte**; 4 or 8 bytes form a **word** (depending on machine architecture).

A bit is drawn as one square cell: **filled black = on**, **white with a black outline = off**. A bit string is then a horizontal row of such cells read left to right, and memory is such rows stacked vertically with an address in the left margin. Both figures below are drawn in that notation, which is collected in csse1001-diagram-notation[1].

We put something in memory by toggling the bits into some meaningful configuration — in Python, the "somethings" we put in memory are called **objects**.

Interpreting memory: the integer type

The bits alone do not say what they mean. A word of memory can be *interpreted* as an arithmetic expression in binary — the place value of each position is a power of two, and the number is the sum of the place values where the bit is on.

When Python executes `x = 3931377233` it toggles the bits at `x`'s memory location to the above and *remembers to interpret that region* as an unsigned 32-bit integer — its **type**. A value's type ultimately dictates which functions are compatible with the object.

[1] courses/csse1001/csse1001-diagram-notation.pdf

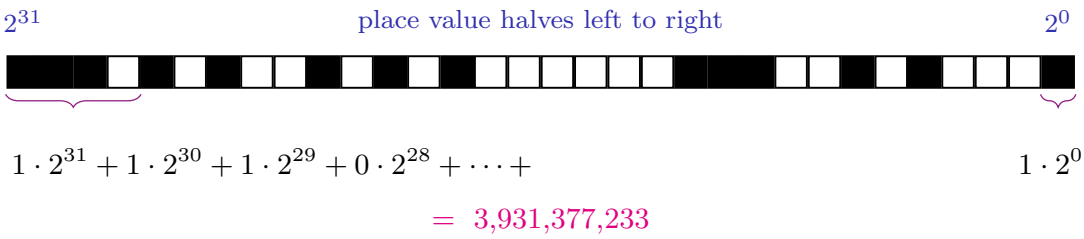

Figure 1: One word of memory read as an unsigned 32-bit integer (filled = on, empty = off). The 32 cells are exactly the binary expansion of 3,931,377,233 with the most significant bit on the left, so the positions run 2^{31} down to 2^0. The assignment `x = 3931377233` toggles exactly these bits and records that the word is to be read as an unsigned 32-bit integer. Nothing in the bits themselves says *integer* — that is what the type records.

Addresses and `id`

Memory is divided into words, and **each word gets an address**.

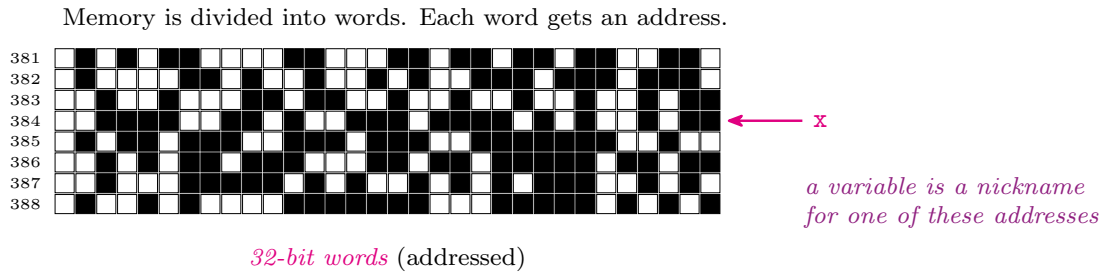

Figure 2: Memory as a stack of addressed 32-bit words, here at addresses 381–388. The bit contents are arbitrary. The magenta `x` and its arrow show that a variable is a nickname we give to an address in memory.

We can determine any object's address with `id`:

```
>>> 42
42
>>> id(42)
4384160760
```

Simply using a number gets it loaded into memory, and `4384160760` is the memory location where 42 is stored.

Variables

A **variable** is a nickname we give to an address in memory.

Assigning

```
>>> x = 3931377233
>>>
```

Assignment toggles the bits at the memory location nicknamed `x`. Nothing is printed, because the imperative here is to *perform an action* (a state change to memory), not to evaluate an expression.

Retrieving

Once `x` has been *assigned* a value, we *retrieve* it by evaluating it in the REPL — a variable evaluates to the value it was assigned, and can stand in for that value inside larger expressions:

```
>>> x
3931377233
>>> x // 100
39313772
```

Evaluating a name that has never been assigned raises a `NameError`:

```
>>> bear
NameError: name 'bear' is not defined
```

Meaningful names

Variable names should convey meaning. This is not meaningful:

```
>>> x = 2
>>> y = 3
>>> z = x*y
```

These names do convey meaning:

```
>>> width = 2
>>> height = 3
>>> area = width*height
```

`area` keeps working even after `width` and `height` change — another reason we use variables that can *vary*.

Naming rules

Variable names **must** start with a letter (`a`–`z`, `A`–`Z`) or underscore (`_`). Starting a name with a digit is a `SyntaxError`; digits are otherwise allowed inside a name.

```
>>> 1 = "one"
    1 = "one"
    ^
SyntaxError: cannot assign to literal
>>> x1 = 1
>>>
```

The course marks against a fuller naming style: the PEP8 rule for variables and functions is lowercase, words separated by underscores, and class names are `CamelCaps`. See python-pep8-style-guide[2].

[2] courses/csse1001/python-pep8-style-guide.pdf

Reassignment

= in Python means *assignment*, not mathematical equality. Read = as "is assigned" or "gets":

```
>>> x = 1         # x "gets" 1
>>> x = x + 1     # x "gets" x+1
>>> x
2
```

In mathematics $x = x+1 \implies 0 = 1$, a contradiction ($\Rightarrow\Leftarrow$) — so as a mathematical statement it is illegal. In Python it is perfectly ordinary, because = performs a one-time action: evaluate the right-hand side, then toggle the bits at the location x names to that result.

Because assignment is an action, the **order** of assignments matters — see 2025-08-04-python-memory-model[3] for five worked examples.

[3] courses/csse1001/2025-08-04-python-memory-model.pdf