Python Variables and Memory Model

Table of contents

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).

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

A word of memory can be interpreted as an arithmetic expression in binary — e.g. $1 \cdot 2^{32} + 1 \cdot 2^{31} + 1 \cdot 2^{30} + 0 \cdot 2^{29} + \cdots + 1 \cdot 2^0 = 3{,}931{,}377{,}233$.

When Python executes `x = 3931377233` it toggles the bits at `x`'s memory location to this value 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 it.

Addresses and `id`

Memory is divided into addressed words. We can determine any object's address with `id`:

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

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. `x = 2; y = 3; z = x*y` is not meaningful, but `width = 2; height = 3; area = width*height` is — and `area` keeps working even after `width`/`height` change, which is 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.

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

Reassignment

`=` in Python means *assignment* ("gets"), not mathematical equality:

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

In mathematics, $x = x+1 \implies 0 = 1$, a contradiction — so this would be an "illegal" statement. In Python it's perfectly valid, because `=` is a one-time action performed left-to-right, not a persistent equality claim.