Python Memory Model

Table of contents

Outline 1
Learning objectives 1
How much can a byte hold? 2
Variables and memory 3
Sequencing: order matters 3
Summary 4
Next lecture 4

Lecture 2A. See csse1001[1] for course logistics.

Outline

- Memory as a sequence of bits
- Variables as named memory locations
- Sequencing: why the order of instructions matters

Learning objectives

1. Memory is a sequence of bits that we can toggle.
2. We can (and should) store values in named memory locations called variables.
3. The order of instructions matters.

[1] courses/csse1001/csse1001.pdf

How much can a byte hold?

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

To see how much can be stored in a byte, build up: one bit, two bits, three bits, four bits.

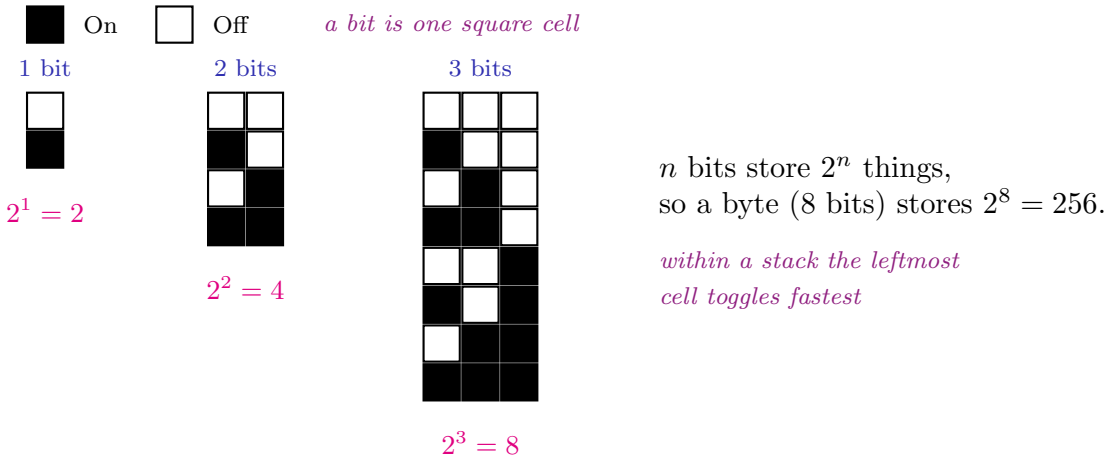

Figure 1: Adding a bit doubles the number of things you can code. The three stacks enumerate every pattern of one, two and three bits (filled = on, empty = off). Within each stack the **leftmost** cell toggles fastest (`off-off`, `on-off`, `off-on`, `on-on`), which reads the leftmost cell as the *least* significant; a word read as an integer is read the other way round. A byte is 8 bits, so it stores $2^8 = 256$ things.

In general, n bits store 2^n distinct things:

- 1 bit: $2^1 = 2$ things (`off`, `on`)
- 2 bits: $2^2 = 4$ things (`off-off`, `on-off`, `off-on`, `on-on`)
- 3 bits: $2^3 = 8$ things
- 4 bits: $2^4 = 16$ things

A byte is eight bits, so it codes $2^8 = 256$ things.

A very long series of such bits is computer memory. We *put something in memory* by toggling the on/off bits into some meaningful configuration; in Python the "somethings" we put in memory are called **objects**.

Variables and memory

See python-variables-and-memory-model[2] for how Python interprets a word of memory as a typed object, how `id()` reveals an object's address, and the rules for assigning, retrieving, naming, and reassigning variables. In brief:

- A word of bits means nothing until a **type** says how to read it — `x = 3931377233` both toggles the bits and records "read this as an unsigned 32-bit integer". The type dictates which functions are compatible with the object.
- Memory is divided into addressed words; `id(42)` returns the address where `42` is stored.
- A **variable** is a nickname for one of those addresses. Assignment prints nothing, because its imperative is to perform an action (a state change to memory).
- `=` is *assignment*, not equality: read `x = x + 1` as "`x` gets `x+1`".

Sequencing: order matters

The same instructions executed in a different order will (usually) produce a different outcome. Five short programs make the point, each one to step through in Python Tutor.

Python Tutor 1 — the two lines cannot be swapped:

```
>>> x = 1
>>> x = x + 1
```

`x = x + 1` cannot come first: the name `x` has no associated value on line 1, so evaluating the right-hand side throws a `NameError`.

Python Tutor 2 and **3** — the same three assignments, the last two swapped, giving 4 and 3:

```
>>> x = 1
>>> x = x + 1
>>> x = 2*x
>>> x
4
```

```
>>> x = 1
>>> x = 2*x      # swapped
>>> x = x + 1    # swapped
>>> x
3
```

[2] courses/csse1001/python-variables-and-memory-model.pdf

Python Tutor 4 and **5** — two variables, and the assignment that gets clobbered decides which value survives:

```
>>> x = 2
>>> y = 3
>>> y = x
>>> x = y
>>> x
2
>>> y
2
```

```
>>> x = 2
>>> y = 3
>>> x = y
>>> y = x
>>> x
3
>>> y
3
```

The last two lines are the ones that change between these two examples — the first two assignments are identical.

Neither ordering swaps `x` and `y`, which is what these four lines look like they should do. Swapping two names needs a third name to hold one value while the other is overwritten (or the tuple packing of 2025-08-04-primitive-data[3]).

Summary

We can manipulate memory using Python and refer to that memory with variables. The order in which we do this manipulation matters — the same sequence of instructions done in a different order will (usually) result in a different outcome.

Next lecture

2025-08-04-primitive-data[4] — immutable objects (the things we put in memory that cannot be changed).

[3] courses/csse1001/2025-08-04-primitive-data.pdf
[4] courses/csse1001/2025-08-04-primitive-data.pdf

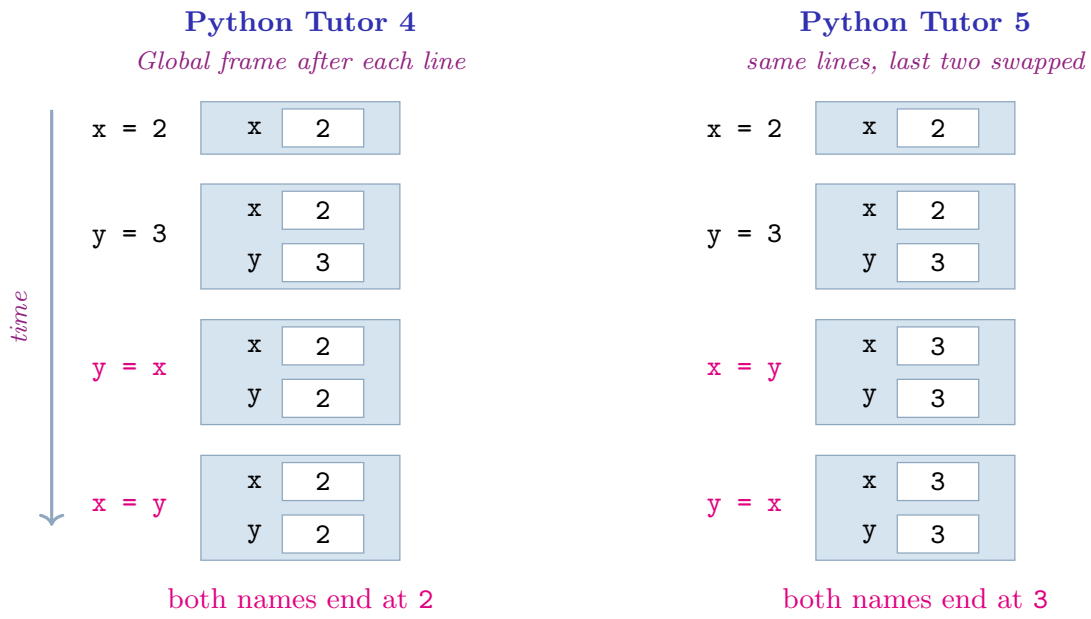

Figure 2: Why examples 4 and 5 disagree: the first assignment of the swapped pair overwrites one name, and the second then copies the *overwritten* value back — so both names always end up equal, but at 2 when `y = x` goes first and at 3 when `x = y` does. The two reordered lines are magenta. Each pale blue box is the global frame after that line, drawn the way Python Tutor shows it: one row per name, value in the slot. `y` does not exist until line 2.