Data Structure

Tuple, Set and Dictionary

Phạm Đình Huân

Ngày 16 tháng 6 năm 2025

Mục lục

Phần II: Mở rộng

1. Đặt vấn đề

1.1 Câu chuyện bắt đầu từ một dòng swap

Ví dụ

Hãy thử nhìn vào đoạn mã sau – hẳn chúng ta đã từng thấy hoặc viết nó rất nhiều lần:

```python
def swap(v1, v2):
    (v2, v1) = (v1, v2)
    return (v1, v2)

v1 = 2
v2 = 3
(v1, v2) = swap(v1, v2)
```

Nếu bạn chỉ chạy nó và thấy kết quả là `v1 = 3, v2 = 2` thì bạn có thể nghĩ: "Đơn giản mà, hoán đổi hai biến thôi." Nhưng nếu bạn là người đã học qua C/C++, bạn sẽ sớm đặt câu hỏi:

> *Tại sao hàm lại hoán đổi được giá trị **bên ngoài** trong Python, trong khi rõ ràng là các ngôn ngữ khác cần dùng `pointer`, `reference`, hoặc kiểu truyền địa chỉ?*

Còn nếu bạn là người mới học Python, bạn có thể hỏi:

> *Hàm này có thật sự "đổi chỗ" hai biến không? Hay là nó chỉ "trả về tuple rồi gán lại"? Nếu đúng là như vậy, thì tại sao lại cần dòng `(v2, v1) = (v1, v2)` bên trong hàm?*

1.2 Đằng sau sự đơn giản là cả một mô hình tư duy

Python được thiết kế dựa trên một triết lý rất riêng: **mọi thứ đều là object, và mọi biến chỉ là một tên ánh xạ đến object đó**. Điều này nghe có vẻ trừu tượng, nhưng lại ảnh hưởng đến **toàn bộ cách hoạt động của ngôn ngữ** – từ việc gán biến, truyền đối số, gọi hàm, đến việc swap hai giá trị như ví dụ ở trên.

Vấn đề nằm ở chỗ: những gì bạn thấy trên bề mặt (các dòng lệnh Python) không phản ánh trực tiếp những gì thực sự diễn ra trong bộ nhớ. Để hiểu đoạn code trên "tới tận gốc", bạn cần hiểu những khái niệm nền tảng như:

- Object được cấp phát ở đâu? (Heap hay Stack?)
- Biến có thật sự chứa giá trị không? Hay chỉ là một "tên gọi"?
- Khi truyền biến vào hàm, Python truyền gì? Giá trị, địa chỉ, hay... một thứ gì khác?
- Việc "hoán đổi" có đang diễn ra thật không, hay chỉ là một sự tái ánh xạ tên biến?
- Python dùng namespace như thế nào để quản lý các biến?

1.3 Khi hiểu sai bản chất dẫn tới hiểu sai ngôn ngữ

Rất nhiều lập trình viên – kể cả người đã viết Python lâu năm – vẫn mang theo tư duy của C/C++, Java,... khi nhìn vào đoạn code như trên. Họ lầm tưởng rằng Python "truyền tham chiếu", hoặc biến là "vùng nhớ chứa giá trị".

Những hiểu nhầm này không chỉ ảnh hưởng tới khả năng viết code đúng mà còn:

- Dẫn tới bug khó lường khi làm việc với object mutable như list, dict, numpy array.
- Làm cho người học không phân biệt được **binding** và **mutation**, từ đó sai lầm trong thiết kế logic.
- Khó hiểu được các khái niệm nâng cao như closure, decorator, scope, object identity, hoặc model copy trong AI.

1.4 Mục tiêu của phần mở rộng này

Phần này không chỉ nhằm giải thích "vì sao đoạn code swap hoạt động", mà sâu hơn: nhằm **"giải phẫu" từng bước chuyển động của object và reference trong bộ nhớ**, tái hiện rõ mô hình tư duy mà Python theo đuổi.

Chúng ta sẽ:

- Theo dõi quá trình binding biến → object từ lúc khởi tạo tới khi swap
- Phân tích cách stack frame được tạo, biến được ánh xạ, object được giữ nguyên hay thay đổi
- Vẽ biểu đồ vùng nhớ (heap, stack, namespace) trước – trong – sau khi swap
- So sánh cơ chế này với C/C++ để thấy sự khác biệt bản chất

Tóm lại: Đằng sau dòng lệnh ngắn gọn ấy là cả một cấu trúc triết lý về cách Python quản lý giá trị, tên biến và vùng nhớ. Muốn viết Python đúng, tối ưu, và không sai tư duy – bạn không thể chỉ học cú pháp. Bạn cần nhìn thấy những gì Python thật sự đang làm bên trong.

2. Một số kiến thức liên quan

2.1. Object

Trong Python, mọi thứ đều là Object

Dù là số nguyên, chuỗi, danh sách, hàm, class hay module... **tất cả đều là Object.**

```
a = 10        # int is an object
s = "hello"   # str is an object
l = [1, 2, 3] # list is an object
def f(): ... # function is also an object
```

Điều này có nghĩa là: mọi thứ chúng ta tạo ra, gán, truyền vào hàm hay thao tác trong Python **đều có bản chất là object**.

Một Object trong Python gồm những gì?

Mỗi object trong Python có 4 thành phần chính:

Thành phần	Ý nghĩa
Type	Kiểu của object: `int`, `str`, `list`, `dict`, `function`,...
Value	Giá trị của object (có thể là immutable hoặc mutable)
Reference Count	Số lượng tên (hoặc nơi) đang trỏ tới object này
Address	Địa chỉ bộ nhớ nơi object được lưu trữ (xem bằng `id(obj)`)

Giải thích từng phần:

- Type:
  Mỗi object đều có một kiểu, xác định bởi hàm `type()`:

```
type(42)          # <class 'int'>
type("hi")        # <class 'str'>
type([1,2,3])     # <class 'list'>
```

- Value:
  Giá trị chứa trong object:
  - Với `immutable` (như `int`, `str`) thì value không đổi được.
  - Với `mutable` (như `list`, `dict`) thì có thể thay đổi.

- Reference Count:
  Được quản lý tự động bởi trình thông dịch Python:

```
import sys
x = [1, 2, 3]
sys.getrefcount(x)  # Usually >1 because Python holds an internal temporary
    reference
```

- Address:
  Là địa chỉ vùng nhớ của object (RAM):

```
x = 99
print(id(x))  # Returns the memory address of object x (as an integer)
```

Mỗi giá trị chúng ta thấy trong Python chính là một object — và object đó lưu trữ value như một phần của nó. Tức là chúng ta không "nhìn thấy" giá trị, chúng ta chỉ "nhìn thấy" object đại diện cho giá trị đó. **Tóm lại: Tất cả value trong Python đều là object.**

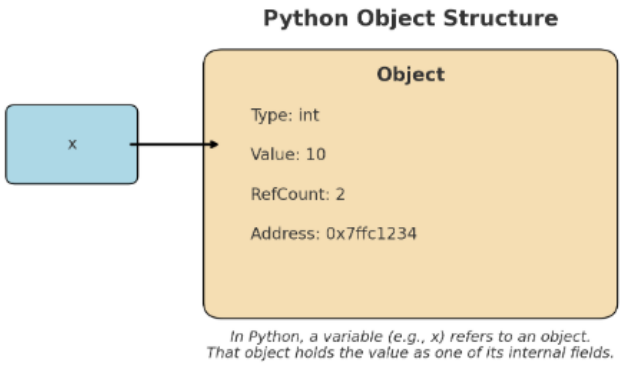
- Chúng ta không làm việc với giá trị trần như trong C/C++.
- Mà luôn làm việc với reference trỏ tới object thật sự.

Ví dụ:

```
x = 10
```

Ở đây:

- `10` là một **object** kiểu `int`
- `x` là một **tên (name)** được bind với object đó



Hình 1: Sơ đồ minh họa cấu trúc của một object trong Python

2.2. Cơ chế Binding và Re-binding trong Python

Giả sử bạn viết đoạn code sau:

```
x = [1, 2, 3]
```

Bạn có thể nghĩ rằng `x` chứa danh sách `[1, 2, 3]`, đúng không?
Sai rồi.
Thực tế, Python tạo một object list `[1, 2, 3]` trong vùng nhớ **heap**, sau đó liên kết tên `x` với object đó thông qua một quá trình gọi là **binding**. Nhưng để hiểu rõ bản chất, hãy nhìn nó dưới góc độ của **ánh xạ toán học**.

Khái niệm ánh xạ trong toán học

Ánh xạ

Một **ánh xạ (hàm số)** là quy tắc gán mỗi phần tử từ một tập hợp đầu vào với đúng một phần tử ở tập hợp đầu ra:

$$f : X \to Y, \quad f(x) = y$$

Trong Python, khái niệm này được áp dụng trực tiếp trong mô hình quản lý biến:

- Tập xác định X: các **tên biến** (dạng chuỗi như `'x'`, `'y'`)
- Tập giá trị Y: các **object** đang tồn tại trong bộ nhớ heap
- Hàm ánh xạ f: chính là **namespace** – chẳng hạn `globals()` hay `locals()`

Ví dụ:

```
x = [1, 2, 3]
y = x
print(globals())
```

Cho ra:

```
{'x': [1, 2, 3], 'y': [1, 2, 3]}
```

Tức là dưới góc nhìn ánh xạ:

$$f('x') = [1, 2, 3], \quad f('y') = [1, 2, 3]$$

Vậy Binding là gì?

Định nghĩa Binding

- **Binding** là quá trình liên kết (associate) một **tên** với một **object**.
- Tên không chứa giá trị – nó chỉ ánh xạ (map) đến object đó.
- Cơ chế ánh xạ này được tổ chức trong **namespace** – là tập hợp ánh xạ tên → object.

Theo tài liệu chính thức của Python:

> *"Binding is the association of a name with an object."*
> — Python Language Reference

Ví dụ minh họa

```
x = [10, 20]
y = x
```

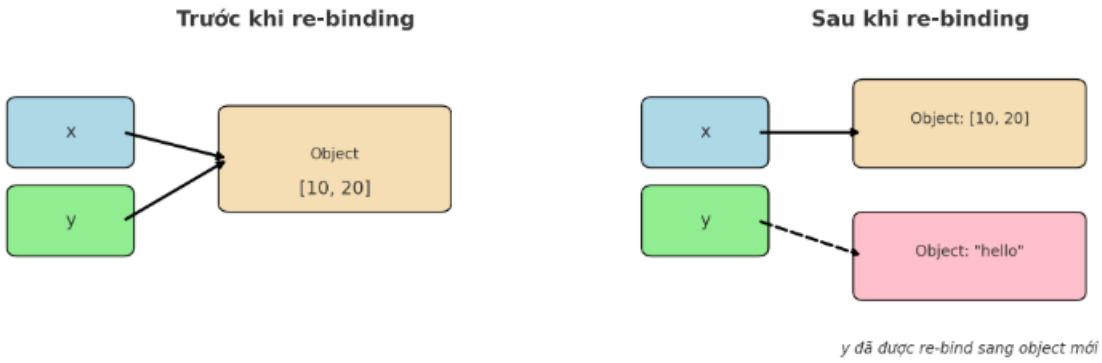
Khi thực hiện hai dòng lệnh trên:

- Chỉ có **một object list** được tạo trong Heap.
- Cả hai tên `x` và `y` đều ánh xạ đến cùng một object này.

```
y = "hello"
```

Khi gán lại như trên, ta thực hiện một **Re-binding**:

- Tên `y` bị gỡ khỏi object cũ, và ánh xạ sang object chuỗi mới `"hello"`.
- Tên `x` vẫn giữ nguyên ánh xạ đến object list ban đầu.



Hình 2: Minh họa cơ chế Binding và Re-binding trong Python

Tóm lại

- Tên biến trong Python là **ánh xạ** (binding) đến một object – nó không chứa giá trị.
- Việc gán lại chỉ là **thay đổi ánh xạ**, không phải sao chép dữ liệu.
- Tất cả ánh xạ tên → object được quản lý trong **namespace**, thường là một `dict`.

2.3 Heap vs Stack – Chứa cái gì ?

Một trong những hiểu nhầm kinh điển khi học về vùng nhớ trong Python là: "Biến nằm trong stack, còn dữ liệu nằm trong heap". Nhưng điều đó là chưa đủ – nếu không muốn nói là dễ gây hiểu sai khi tư duy đến hàm, object, và biến mutable.
Để hiểu đoạn code `swap` trong Python, ta phải phân biệt rõ vai trò của **Heap** và **Stack Frame**:

Stack Frame – nơi lưu trữ *binding tạm thời*

Mỗi khi một hàm được gọi, Python tạo ra một **stack frame mới** để xử lý hàm đó.
Vậy Stack Frame là gì ?
—

Stack Frame là gì?

Stack Frame là một **vùng nhớ tạm thời** được tạo mỗi khi hàm được gọi.

Đặc điểm:

- Dùng để lưu:
  - Biến cục bộ trong hàm (`a`, `b`, `temp`,...)
  - Thông tin trả về
  - Tên hàm đang chạy
- Khi hàm kết thúc, stack frame bị xóa.

Ví dụ: Stack Frame của hàm `swap`
Khi gọi hàm:

```
def swap(v1, v2):
    ...
swap(2, 3)
```

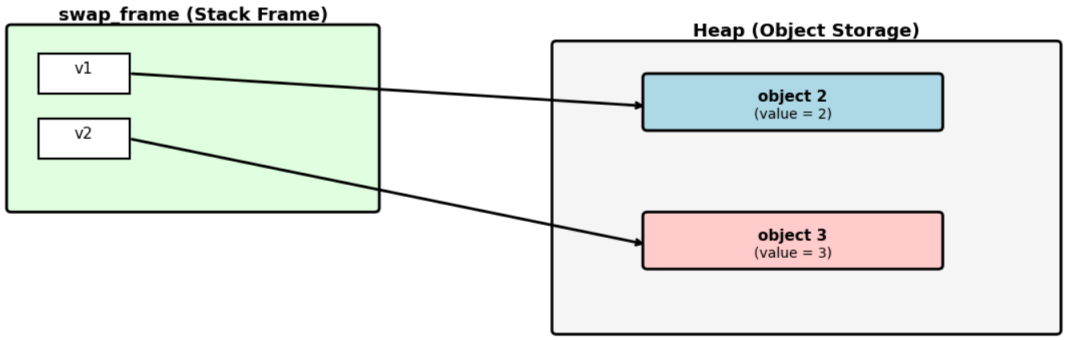
Python tạo một **Stack Frame** tạm thời cho hàm `swap`, gọi là `swap_frame`, có dạng:

```
swap_frame = {
    'v1': <object 2>,
    'v2': <object 3>
}
```

Với:

- `swap_frame` là một *namespace tạm thời* chứa các ánh xạ giữa tên (name) và object.
- ''v1'' được bind tới object có giá trị '2' (lưu trong Heap).
- ''v2'' được bind tới object có giá trị '3' (lưu trong Heap).

Sau khi `return`, `swap_frame` biến mất.



Hình 3: Minh họa ánh xạ của `v1`, `v2` trong Stack Frame tới các object trong Heap

Heap – nơi sống thật của các object

Tất cả object thật sự trong Python (số, chuỗi, list, tuple, dict, class, function...) đều được lưu trữ trong **Heap** – một vùng nhớ toàn cục, có tính chất "dài hạn" hơn Stack. **Vậy Heap là gì ?**

Heap là gì?

Heap là vùng nhớ **dài hạn** dùng để lưu trữ các **object** mà chương trình tạo ra trong quá trình chạy.

Đặc điểm:

- Các object trong Python (danh sách, số, chuỗi, class, dict,...) đều được cấp phát ở **heap**.
- Dữ liệu trong heap có thể tồn tại lâu, vượt qua phạm vi của một hàm.
- Python dùng **Garbage Collector** để xóa object trong heap khi không còn biến nào trỏ tới.

Ví dụ:

```
v1 = 2
```

Python không lưu giá trị `2` trong biến `v1`, mà:

1. Tạo object `2` trong Heap (gọi là `o_1`)
2. Ánh xạ tên `v1` trong Global Stack Frame tới `o_1`

Biến `v1` chỉ là một "nhãn dán" (reference), còn bản thân giá trị nằm trong Heap.

Quay trở lại đoạn code swap

Khi ta gọi `swap(v1, v2)`, Python tạo Stack Frame riêng cho `swap`, nhưng không tạo bản sao object. Các object `2`, `3` vẫn nằm nguyên trong Heap – chỉ là tên `v1`, `v2` trong hàm được gán lại.
Điều này lý giải vì sao:

- Việc gán `(v2, v1) = (v1, v2)` chỉ là re-binding tên trong Stack
- Không ảnh hưởng gì đến `v1`, `v2` ngoài Global
- Nếu không `return`, thì toàn bộ `swap` biến mất sau khi Stack Frame bị xoá

Phân biệt Stack Frame & Heap

Thành phần	Stack Frame	Heap
Chứa gì?	Tên biến → object	Object thật (int, list, tuple, function...)
Tồn tại bao lâu?	Trong suốt thời gian gọi hàm	Cho tới khi không còn biến nào trỏ đến
Biến mất khi nào?	Khi thoát khỏi hàm	Khi reference count về 0 (garbage collection)
Thay đổi có ảnh hưởng gì?	Thay đổi tên ánh xạ	Thay đổi nội dung object (nếu mutable)

Kết luận

Trong Python, **Stack Frame là nơi quyết định "ai trỏ đến cái gì"**, còn **Heap là nơi "cái gì thật sự tồn tại"**.

2.4. Namespace trong Python: Ai quản lý tên biến?

"Tên" trong Python được quản lý ở đâu?

Khi bạn viết:

```
x = 10
def greet():
    name = "Alice"
    print("Hello", name)
```

Có bao giờ bạn tự hỏi:

- Biến `x` nằm ở đâu?
- Biến `name` trong hàm `greet` tồn tại đến bao giờ?
- Nếu bạn viết thêm một hàm khác, liệu nó có thấy được `name`?

Đây chính là lúc khái niệm **Namespace** (không gian tên) xuất hiện.

Namespace là gì?

Namespace là gì?

Namespace là một bảng ánh xạ (dictionary) được tự động tạo ra trong quá trình thực thi, dùng để lưu trữ **mối liên kết (binding)** giữa **tên biến (key)** và **object (value)** tương ứng trong một **phạm vi cụ thể (scope)**.
Nói cách khác:

> *Mỗi khi tạo ra một tên – dù là biến, hàm, lớp, hay module – Python sẽ lưu tên đó trong một namespace.*

Khi nào namespace được tạo ra?

Python tạo namespace tại các thời điểm sau:

Loại namespace	Được tạo khi nào?
Global Namespace	Khi một module được chạy
Local Namespace	Khi một hàm được gọi
Built-in Namespace	Khi trình thông dịch Python khởi động
Class Namespace	Khi định nghĩa class

Một số ví dụ đơn giản

Ví dụ 1: Global namespace

```
x = 10
def abc():
    pass
```

Ở cấp độ này:

```
globals() = {
    'x': 10,
    'abc': <function object>
}
```

Ví dụ 2: Local namespace

```
def greet():
    name = "Alice"
    print(name)
```

Khi gọi `greet()`, Python tạo một local namespace cho hàm:

```
locals() = {
    'name': 'Alice'
}
```

Sau khi hàm kết thúc, namespace này bị huỷ.

Ví dụ 3: Built-in namespace

Các hàm như `len()`, `print()`, `sum()`... đến từ built-in namespace:

```
print(len.__name__)  # 'len'
print(__builtins__.__dict__.keys())  # danh sach ham built-in
```

Tầng namespace: Khi Python tìm tên biến

Khi bạn gọi một tên biến, Python sẽ tìm trong các namespace theo thứ tự từ trong ra ngoài – gọi là **LEGB Rule**:

Tầng	Namespace	Ví dụ về tên biến
L	Local	Biến trong hàm hiện tại
E	Enclosing	Biến trong hàm bao ngoài (closure)
G	Global	Biến cấp module (file)
B	Built-in	`len`, `print`, `sum`

Namespace không phải là vùng nhớ chứa object

Namespace bản chất chỉ là dictionary nên chỉ quản lý **tên biến**, không chứa object. Các object thật được lưu trong heap. Nghĩ namespace như một bảng tra cứu: "Tên nào trỏ đến object nào?"

Ví dụ tổng hợp: theo dõi namespace qua globals() và locals()

```
x = 42
def show():
    y = "hello"
    print("Locals:", locals())

show()
print("Globals:", globals())
```

Kết quả:

```
Locals: {'y': 'hello'}
Globals: {'x': 42, 'show': <function object>, ...}
```

Tóm lại:

- Namespace là nơi Python ghi nhớ: "Tên nào trỏ đến object nào?"
- Có nhiều namespace tồn tại song song: local, global, built-in, class.
- Khi gọi biến, Python tìm theo thứ tự LEGB (Local → Enclosing → Global → Built-in).
- Namespace không chứa dữ liệu – chỉ ánh xạ tên → object (trong heap).

"Namespace là danh bạ điện thoại. Object là con người. Biến là tên. Còn heap là nơi các object thực sự sống."

Sau khi hiểu được Namespace, chúng ta sẽ có được cách hiểu thực sự bản chất về biến.

2.5. Biến trong Python: Liệu có chứa giá trị không?

Biến trong Python KHÔNG chứa giá trị – chúng chỉ là tên gắn vào object

Trong Python, khái niệm "biến" thường bị hiểu nhầm theo kiểu truyền thống: *biến là một hộp chứa giá trị*. Nhưng thật ra, Python không vận hành theo cách đó.
Trong Python:

- Biến là một **tên** (string) sống trong namespace
- Biến được **ánh xạ (bind)** tới một object nào đó đang sống trong Heap
- Bản thân tên biến KHÔNG chứa giá trị – nó chỉ trỏ tới object có chứa giá trị

Ta hãy xem xét lại ví dụ ban đầu:

```
def swap(v1, v2):
    (v2, v1) = (v1, v2)
    return (v1, v2)

v1 = 2
v2 = 3
(v1, v2) = swap(v1, v2)
```

—

Chuyện gì đang thực sự xảy ra?

Khi chạy đoạn code trên, Python không tạo bản sao của giá trị 2 và 3 để hoán đổi trực tiếp. Thay vào đó, Python chỉ thực hiện việc:

- Tạo object số `2`, `3` trong Heap
- Gắn tên `v1`, `v2` vào các object đó
- Khi gọi hàm `swap`, Python tạo Stack Frame riêng với hai tên `v1`, `v2` mới — và *gắn lại* các tên này với cùng object 2 và 3
- Việc gán `(v2, v1) = (v1, v2)` chỉ là đổi lại ánh xạ tên → object trong Stack Frame
- Cuối cùng, `return` trả về tuple chứa hai object, và ngoài phạm vi hàm, hai tên `v1`, `v2` ngoài Global được gắn lại theo kết quả đó

$$\texttt{Trước swap:} \quad \texttt{v1} \mapsto 2, \quad \texttt{v2} \mapsto 3$$
$$\texttt{Sau swap:} \quad \texttt{v1} \mapsto 3, \quad \texttt{v2} \mapsto 2$$

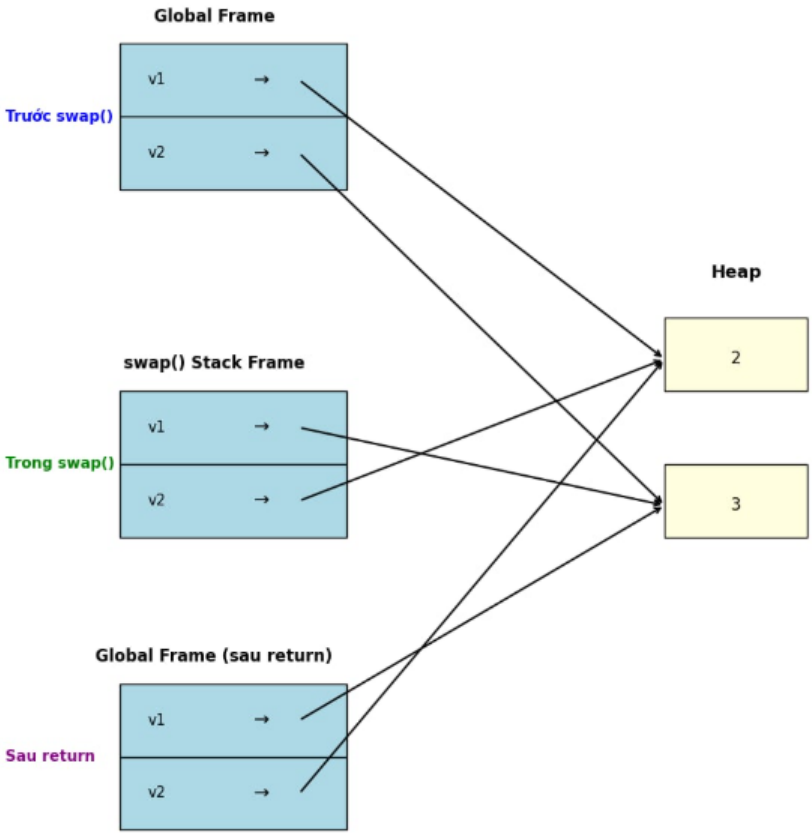
Không hề có sự "di chuyển giá trị" như trong tư duy "biến là hộp chứa số". Tất cả chỉ là thay đổi ánh xạ.

—

Vậy "biến" là gì trong Python?

Biến là gì?

- Biến trong Python là một **tên (name)** được lưu trong **namespace**, ánh xạ đến một object đang tồn tại trong vùng nhớ Heap.
- Biến không phải là vùng nhớ, cũng không "chứa" giá trị. Nó chỉ "trỏ đến" object có chứa giá trị.



Hình 4: Minh họa binding và stack frame trong lời gọi swap(v1, v2)

Một object – nhiều tên cùng trỏ

Điều này dẫn tới một hệ quả thú vị: **nhiều biến có thể cùng trỏ đến một object**. Ví dụ:

```
x = [1, 2, 3]
y = x
```

Giờ thì cả `x` và `y` đều trỏ tới cùng một object list trong Heap. Nếu bạn thay đổi nội dung object qua một tên, tên còn lại cũng sẽ thấy thay đổi đó.

—

Tóm lại: Đừng nghĩ biến là hộp

- Biến không chứa giá trị – nó chỉ là tên gắn vào object
- Object sống trong Heap, còn biến sống trong namespace
- "Gán biến" trong Python thật ra là "gắn lại tên vào object khác" (re-binding)

"Biến trong Python là nhãn dán, không phải hộp đựng."

2.6. Cơ chế Packing và Unpacking trong Python

Python cho phép viết các dòng gán "lạ lùng" mà người mới học thường không hiểu:

```
a, b = 1, 2
a, b = b, a
```

Điều gì đang xảy ra ở đây? Tại sao có thể gán nhiều biến cùng lúc? Lý do là vì Python hỗ trợ một cơ chế rất đặc trưng gọi là: **packing và unpacking**.

Định nghĩa

- **Packing**: Gom nhiều giá trị lại thành một tuple (hoặc list).
- **Unpacking**: Phân rã một iterable (tuple, list, dict,...) thành các phần tử riêng lẻ và gán cho nhiều tên.

Cơ chế hoạt động

Ví dụ 1: Tuple Packing + Unpacking

```
a = 1
b = 2
pair = (a, b)            # Packing     tuple
x, y = pair              # Unpacking
```

Phân tích:

- Bên phải: gom thành tuple '(1, 2)'
- Bên trái: phân rã tuple và binding 'x = 1', 'y = 2'

Multiple Assignment trong Python

```
x, y, z = 1, 2, 3
```

Thực chất:

```
temp = (1, 2, 3)     # Packing
x = temp[0]
y = temp[1]
z = temp[2]          # Unpacking
```

Python thực hiện **packing bên phải → unpacking bên trái** hoàn toàn tự động.

Tóm lại

- **Packing** là "gom lại thành 1 tuple"
- **Unpacking** là "rút từng phần tử ra để gán"
- Python dùng cơ chế này ở mọi nơi: gán biến, gọi hàm, for loop, return,...
- Việc hoán đổi biến, gán nhiều biến cùng lúc đều chỉ là biểu hiện của unpacking

3. Phân tích

Đoạn code ban đầu

```
def swap(v1, v2):
    (v2, v1) = (v1, v2)
    return (v1, v2)

v1 = 2
v2 = 3
(v1, v2) = swap(v1, v2)
```

Giai đoạn 1: Biên dịch

- Khi chương trình bắt đầu, Python sẽ tự động tạo ra một **Global Namespace**, thực chất là một bảng ánh xạ (dictionary) dùng để lưu trữ các tên định nghĩa ở cấp độ module.
- Namespace này được gắn vào **Global Stack Frame**, chứ không phải lưu trong Heap như các object thông thường.
- Sau khi biên dịch định nghĩa hàm `swap`, Python tạo một `function object` trong Heap và ánh xạ tên `"swap"` tới object này trong Global Namespace:

```
globals() = {
    'swap': <function swap at 0x...>
}
```

Giai đoạn 2: Gán ban đầu

```
v1 = 2
v2 = 3
```

Khi thực thi hai dòng trên:

- Python tạo ra hai **object immutable kiểu int** với giá trị `2` và `3` (thường nằm trong vùng nhớ **heap** hoặc **Integer cache** tùy implementation).
- Tên `v1` và `v2` được tạo ra trong **Global Namespace** – nằm trong Global Stack Frame.
- Quá trình này là một **binding**: mỗi tên được ánh xạ tới địa chỉ của object tương ứng.

Lúc này, trạng thái của Global Namespace là:

```
globals() = {
    'swap': <function swap at 0x...>,
    'v1': 2,
    'v2': 3
}
```

Về mặt bản chất (logic ánh xạ):

- ‘’v1‘‘ → Object ‘int(2)‘ trong Heap
- ‘’v2‘‘ → Object ‘int(3)‘ trong Heap
- ‘’swap‘‘ → Function Object ‘swap()‘ trong Heap

Ghi chú:

- Dù object `2` và `3` là immutable và có thể tái sử dụng (caching), nhưng về mặt ngữ nghĩa, ta vẫn hiểu là Python tạo binding từ tên đến object, không phải gán giá trị vào tên.
- Biến trong Python chỉ là **tên** – không chứa giá trị – mà đơn thuần là ánh xạ đến object có thật nằm trong heap.

Giai đoạn 3: Gọi hàm `swap(v1, v2)`

Khi Python gặp lời gọi hàm:

```
(v1, v2) = swap(v1, v2)
```

Quá trình thực hiện gồm các bước sau:

1. Python tra cứu `v1` và `v2` trong **Global Namespace**, lấy ra hai object: `int(2)` và `int(3)` từ heap.
2. Một **stack frame mới** được tạo trên **Call Stack** để thực hiện hàm `swap`.
3. Trong stack frame này, một **Local Namespace** mới được khởi tạo và binding các tên tham số `v1` và `v2` vào hai object vừa nêu.

Stack frame mới (Local Namespace) có dạng:

```
swap_frame = {
    'v1': 2,    # tro toi object int(2) trong Heap
    'v2': 3     # tro toi object int(3) trong Heap
}
```

Giải thích rõ hơn:

- Đây không phải là tạo biến mới giống biến toàn cục, mà là binding tên mới `v1`, `v2` trong **local namespace**.
- Do kiểu `int` là immutable, nên việc hoán đổi sẽ không làm thay đổi object, mà chỉ thay đổi binding giữa tên và object.
- Cơ chế gọi hàm này trong Python được gọi là: **pass-by-object-reference** (hoặc gọi đơn giản hơn là **pass-by-assignment**).

Lúc này, hệ thống bộ nhớ:

- **Heap:** chứa các object `2`, `3`, `function swap`.
- **Global Namespace:**

```
globals() = {
    'swap': <function swap at 0x...>,
    'v1': 2,
    'v2': 3
}
```

- **Stack:**
  - Global Frame (chứa namespace toàn cục)
  - Stack Frame của `swap`, chứa local namespace:

```
swap_frame = {
    'v1': 2,
    'v2': 3
}
```

Giai đoạn 4: Thực thi dòng `(v2, v1) = (v1, v2)` trong hàm

```
(v2, v1) = (v1, v2)
```

Diễn giải chi tiết quá trình:

1. **Bên phải – Tuple Packing:**
   - Python đánh giá biểu thức `(v1, v2)` trong **local namespace** của hàm.
   - Tại thời điểm này:
     ```
     v1 -> int(2)
     v2 -> int(3)
     ```
   - Do đó, biểu thức tạo ra một **tuple mới** `(2, 3)` được lưu trong **heap**.
2. **Bên trái – Tuple Unpacking:**
   - Tuple vừa tạo có hai phần tử: `2` và `3`.
   - Python lần lượt **unbind và rebind** các tên ở vế trái:
     - `v2` (local) được bind lại với `int(2)`
     - `v1` (local) được bind lại với `int(3)`
   - **Lưu ý:** Không có đối tượng nào bị thay đổi – chỉ là các binding bị cập nhật lại.

Trạng thái Local Namespace sau khi unpacking:

```
swap_frame = {
    'v1': <int object 3>,
    'v2': <int object 2>
}
```

Heap lúc này chứa:

- `int(2)`, `int(3)` – vẫn không đổi.
- Tuple tạm thời: `(2, 3)` – sẽ được xoá khi không còn binding nào.

Tóm lại:

- Đây là bước **hoán đổi binding trong local namespace**, không ảnh hưởng đến các biến bên ngoài hàm.
- Cơ chế này hoạt động do Python hỗ trợ unpacking rất mượt cho các đối tượng iterable như tuple.

Giai đoạn 5: Trả về tuple mới

```
return (v1, v2)  # -> (3, 2)
```

Chi tiết quá trình:

- Python đánh giá biểu thức `(v1, v2)` trong **local namespace** của hàm.
- Tại thời điểm này:
  ```
  v1 -> int(3)
  v2 -> int(2)
  ```

- Do đó, một **tuple mới** chứa hai object hiện có là `(3, 2)` được tạo trên **heap**.
- Tuple này sẽ là giá trị trả về của hàm `swap`.

Sau dòng `return`, các bước ngầm diễn ra:

- **Stack frame của hàm `swap`** bị xoá khỏi stack.
- **Local namespace** tương ứng cũng bị xoá.
- Các binding `v1, v2` cục bộ biến mất – nhưng object `int(2)`, `int(3)` vẫn tồn tại trong heap do vẫn được tham chiếu từ global namespace và tuple mới tạo.

Trạng thái sau khi hàm kết thúc:

- **Heap:** có thêm tuple mới `(3, 2)`.
- **Global namespace:** vẫn giữ các binding cũ cho đến bước unpack tiếp theo.
- **Stack:** chỉ còn lại global frame.

Ghi chú:

- Tuple trả về được giữ lại tạm thời trong bộ nhớ để chuẩn bị sử dụng cho dòng unpack tiếp theo: `(v1, v2) = swap(...)`.
- Python quản lý bộ nhớ bằng cơ chế **reference counting**, nên miễn là tuple đó còn được dùng, nó chưa bị xoá.

Giai đoạn 6: Unpacking kết quả trả về

```
(v1, v2) = swap(v1, v2)
```

Chi tiết quá trình:

- Gọi hàm `swap(v1, v2)` đã trả về một **tuple mới** chứa: `(3, 2)`.
- Dòng lệnh này sử dụng **tuple unpacking** để gán từng phần tử của tuple vào hai tên bên trái.
- Cụ thể:
  - Phần tử đầu tiên (3) → bind vào tên `v1` trong **global namespace**.
  - Phần tử thứ hai (2) → bind vào tên `v2` trong **global namespace**.
- Điều này dẫn đến **re-binding** (gán lại) cho `v1` và `v2`:

```
globals() = {
    'swap': <function object>,
    'v1': <int object 3>,
    'v2': <int object 2>
}
```

- Các binding cũ của `v1 = 2` và `v2 = 3` bị ghi đè — nếu không còn binding nào khác trỏ đến các object đó, chúng sẽ được gom rác sau này.

Trạng thái cuối cùng:

- **Global namespace:**

```
'swap' -> <function object>
'v1'   -> int(3)
'v2'   -> int(2)
```

- **Heap:** Vẫn giữ các object: `int(2)`, `int(3)`, `function swap`, và `tuple (3, 2)` nếu còn tham chiếu.
- **Stack:** Chỉ còn lại global stack frame.

Tổng kết bước này:

- Đây là bước then chốt hoàn tất việc hoán đổi biến.
- Tuy không có bất kỳ object nào bị thay đổi nội tại (**no mutation**), nhưng binding được hoán đổi – nghĩa là **tên biến** trỏ sang object khác.
- Cơ chế này thể hiện rõ tính **pass-by-object-reference** (truyền tham chiếu đối tượng) của Python.

Giai đoạn 7: Dọn dẹp (Garbage Collection và kết thúc lời gọi hàm)

- Khi hàm `swap()` kết thúc, **stack frame** của nó (gồm local namespace và địa chỉ trả về) bị **pop khỏi call stack**.
- Các tên `v1`, `v2` trong local namespace của hàm sẽ bị **huỷ binding**, vì namespace này không còn tồn tại nữa.
- Nếu không còn bất kỳ tham chiếu nào khác trỏ đến các object trong stack frame đó (như tuple tạm thời), thì các object đó sẽ bị đánh dấu để thu gom bởi **Garbage Collector (GC)** của Python.
- Trong ví dụ này:
  - `tuple (2, 3)` – được tạo tạm thời trong dòng hoán đổi bên trong hàm – đã hoàn thành nhiệm vụ và không còn tên nào bind tới nó → **có thể bị GC thu gom**.
  - Các object `int(2)` và `int(3)` vẫn còn được tham chiếu bởi `v1` và `v2` trong global namespace → **không bị huỷ**.
  - `tuple (3, 2)` – kết quả trả về – hiện vẫn còn được unpack và bind bởi `v1`, `v2` → **vẫn tồn tại**.
- Python có thể thu gom các object không còn sử dụng nữa sau khi đảm bảo chúng không còn reference nào trỏ đến.

Tóm lại:

- Stack frame `swap()` bị loại bỏ khỏi call stack.
- Local namespace đi kèm bị hủy hoàn toàn.
- Các object tạm thời trong hàm có thể bị thu gom nếu không còn reference nào.
- Global namespace vẫn giữ các binding mới của `v1`, `v2`.

Tổng kết quy trình

Bước	**Mô tả**
Biên dịch	Ghi nhớ các tên và hàm vào namespace
Gán ban đầu	Tạo object 2, 3 và gán tên trong global namespace
Gọi hàm	Tạo stack frame riêng, truyền binding object vào
Unpacking	Dùng tuple để hoán đổi bằng rebinding
Trả về	Tạo tuple mới trong heap và trả ra ngoài
Gán lại biến	Global `v1`, `v2` rebind theo unpacking
Dọn bộ nhớ	Xoá frame, huỷ object không dùng