AI VIET NAM
@aivietnam.edu.vn

AI VIET NAM – AI COURSE 2025

Variables and Addresses in Python

Nguyễn Thế Hào và Đinh Quang Vinh

I. Giới thiệu

Python là một ngôn ngữ hướng đối tượng, nơi mọi giá trị đều là một object, và các biến chỉ đơn thuần là tên dùng để tham chiếu đến object đó. Chính vì vậy, hiểu được sự khác biệt giữa các loại object như mutable (có thể thay đổi), immutable (không thể thay đổi), cũng như cách object được lưu trữ và tham chiếu trong bộ nhớ.

Trong bài viết này, chúng ta sẽ cùng tìm hiểu sâu hơn về cách Python xử lý biến và địa chỉ, phân tích sự khác biệt giữa kiểu dữ liệu mutable và immutable, cách hoạt động của hàm id(), cơ chế sao chép list, cũng như một tối ưu thú vị của Python: Small integer caching.

II. Variable và Address

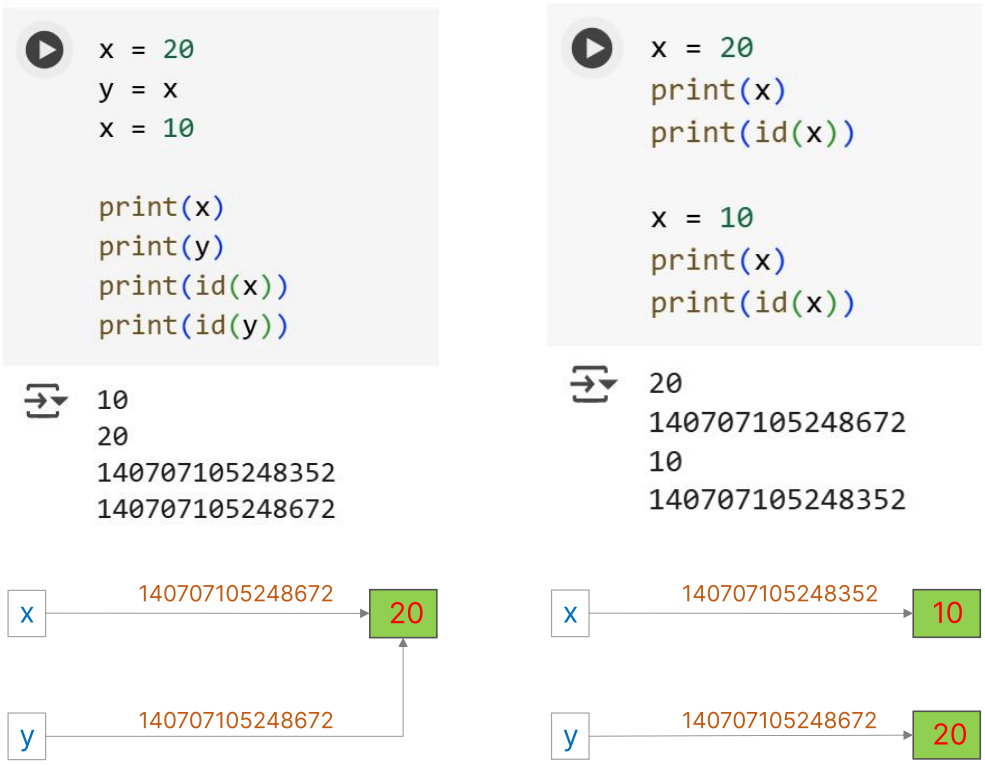
II.1. Tổng quan

Biến (Variable): Trong Python không thực sự "chứa" giá trị như trong một số ngôn ngữ khác. Thay vào đó, biến là một tên dùng để tham chiếu đến một object trong bộ nhớ. Ví dụ: Khi bạn viết x = 10, bạn không tạo ra một "hộp" tên x chứa số 10, mà bạn tạo ra object 10 trong bộ nhớ, rồi gán tên x để trỏ đến nó.

Địa chỉ (Address): Là vị trí trong bộ nhớ nơi object được lưu trữ. Python cung cấp hàm id() để kiểm tra địa chỉ (hoặc chính xác hơn là "identity") của object. Khi hai biến có cùng id(), tức là chúng trỏ đến cùng một object.

Hãy xét ví dụ dưới đây.

- Với đoạn code bên trái khi chúng ta gán cho y = x là chúng ta đang gán cho y địa chỉ mà object x trỏ tới cụ thể ở đây là 20, nên khi in ra id của x và y chúng ta thấy nó bằng nhau.
- Còn với đoạn code bên phải ta thấy khi chúng ta gán cho x = 10 sau khi x = 20. Lúc này thay đổi ở đây chính là address của x từ address của 20 thành address của 10. Ta có thể thấy x không còn đường nối tới địa chỉ của object 20.



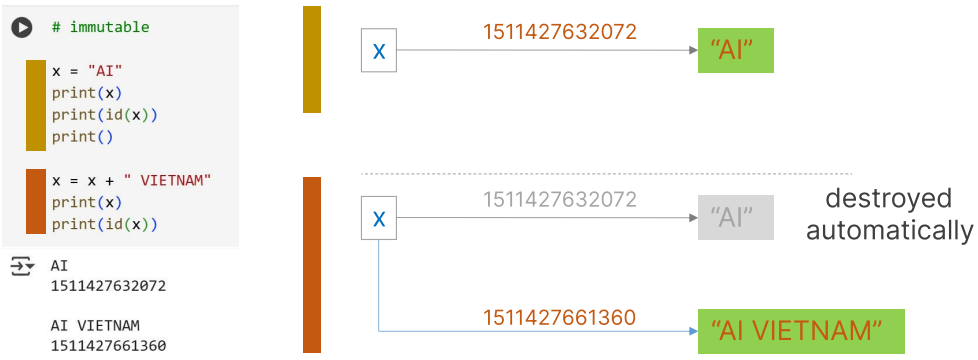
Hình 1: Minh họa về variables and address

II.2. Mutable và Immutable

Trong Python, tất cả mọi thứ đều là object, nhưng không phải object nào cũng cư xử giống nhau khi được gán hoặc thay đổi. Sự khác biệt lớn nhất nằm ở đặc tính mutable và immutable của từng kiểu dữ liệu.

Immutable objects là các object không thể bị thay đổi tại chỗ (inplace). Một khi được tạo ra, giá trị của chúng không thể chỉnh sửa trực tiếp. Các kiểu dữ liệu phổ biến thuộc nhóm này là: int, float, str, bool, và tuple.

Hãy xem ví dụ với kiểu dữ liệu string ở trên đây.



Hình 2: Minh họa về immutable object

```python
a = [1, 2, 3]
print('Before:', a, '| id:', id(a))

a.append(4)
print('After: ', a, '| id:', id(a))

a = a + [4, 5]
print('a_list: ', a)
print('a_list address: ', id(a))
```

```
Before: [1, 2, 3] | id: 132522325615168
After:  [1, 2, 3, 4] | id: 132522325615168
a_list:  [1, 2, 3, 4, 4, 5]
a_list address:  132522325615680
```

Hình 3: Minh họa về mutable objects

- So với ví dụ ban đầu thay vì gán giá trị mới chúng ta sẽ thay đổi x bằng cách thêm vào một giá trị để xem giá trị object của x có được thay đổi hay không.
- Và có thể thấy rằng thay vì thêm vào chuỗi AI thành chuỗi AI VIETNAM, lúc này Python tạo ra một object AI VIETNAM và gán lại địa chỉ cho X. Và vì y trỏ tới object AI nên giá trị y không thay đổi khi x thay đổi địa chỉ.

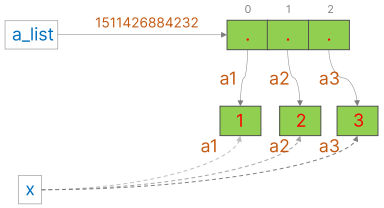
Mutable objects thì ngược lại chúng ta có thể thay đổi nội dung của object tại chỗ (inplace) mà không tạo ra object mới. Các kiểu phổ biến bao gồm list, dict, set. Chúng ta sẽ đi đến ví dụ đầu tiên về các hành vi thay đổi giá trị của mutable objects giữa in-place operation và những phép thay đổi giá trị biến thường gặp. Với ví dụ này

- Chúng ta có thể thấy trong đoạn code trên với phép append() thì address của biến a không hề thay đổi vì append() là phương thức thay đổi nội dung tại chỗ (in-place operation).
- Nhưng đối với phép cộng thì tương tự với các ví dụ về immutable Python tạo ra object mới là [1, 2, 3, 4, 4, 5] và gán address mới về lại cho biến a.

Tiếp theo chúng ta xét tới các phần tử con của các tập hợp thông qua ví dụ bên dưới. Ta thấy rằng các phần tử con a[i] đều trỏ tới các object do Python tạo ra ví dụ a[0] trỏ tới object 1. Và chúng ta có gán chúng là x đi chăng nữa thì x vẫn trỏ tới các object đó.

```python
# aivietnam

a_list = [1, 2, 3]
for i in range(3):
    print(f'a_list[{i}] address is {id(a_list[i])}')

print()
for x in a_list:
    print(f'x address is {id(x)}')
```

```
a_list[0] address is 140707105248064
a_list[1] address is 140707105248096
a_list[2] address is 140707105248128

x address is 140707105248064
x address is 140707105248096
x address is 140707105248128
```



a1 = 140707105248064
a2 = 140707105248096
a3 = 140707105248128

Hình 4: Minh họa về address của mutable

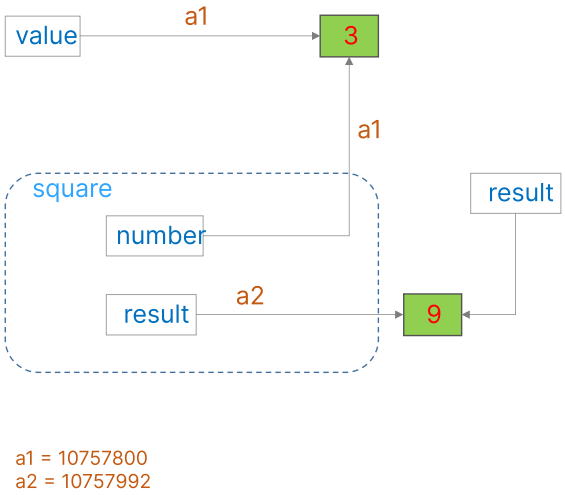
III. Variable assignment

Tại phần này chúng ta sẽ tìm hiểu Python gán biến theo cơ chế tham chiếu (reference) hay sao chép (copy). Chúng ta cùng phân tích ví dụ đầu tiên (Hình 5) trong phần này.

- Chúng ta có thể thấy number được tham chiếu qua value nên nó trở tới 3.
- Tuy nhiên với result ở bên trong hàm square và result nằm ngoài hàm square chúng là 2 biến khác nhau nhưng trỏ tới cùng một object 9.
- Vậy Python truyền tham chiếu đối tượng. Giả sử ta gán thêm 1 biến x = 9 thì lúc này thay vì tạo ra một object 9 mới Python sẽ truyền tham chiếu của object 9 đã có vào cho x.

```
# aivietnam

def square(number):
    result = number*number

    print(f'number address is {id(number)}')
    print(f'result address is {id(result)}')

    return result

# test the function square
value = 3
print(f'value address is {id(value)}')

print(square(value))
```

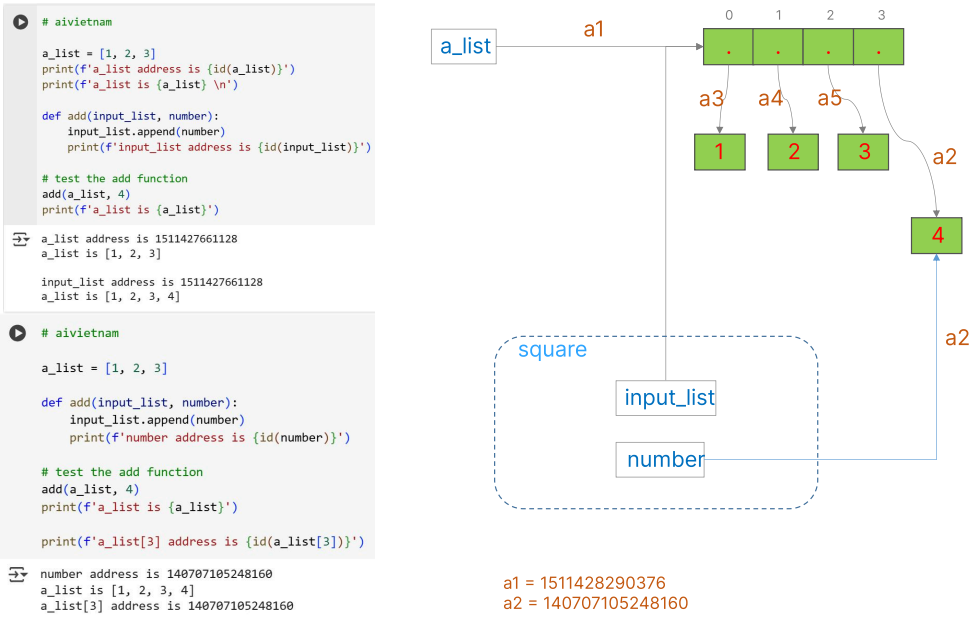
```
value address is 10757800
number address is 10757800
result address is 10757992
9
```



Hình 5: Minh họa cách python truyền tham chiếu với immutable

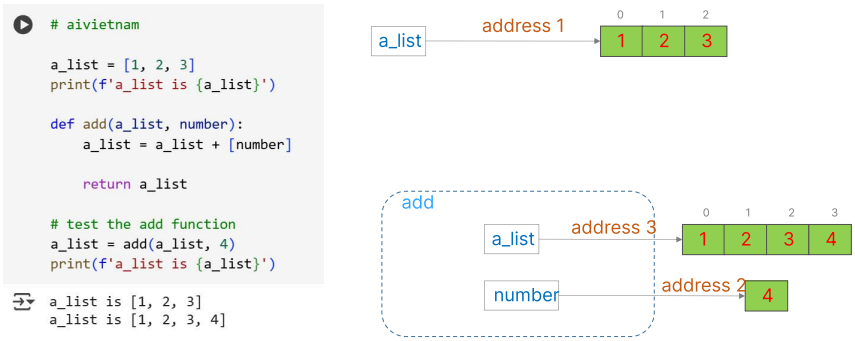
Vừa rồi chúng ta truyền tham chiếu với một biến immutable tiếp theo cùng quan sát và phân tích thực hành tham chiếu với một biến dạng mutable(list)
Cùng quan sát ví dụ trong hình 6:

- Đối với biến a-list Python cũng truyền tham chiếu tương tự như ví dụ trên. Và a-list lẫn input-list được tạo ra đề trỏ về cùng address đầu tiên là [1, 2, 3]
- Đối với các biến con của a-list Python cũng truyền tham chiếu giống hết ví dụ trước. Thể hiện ở object 4. Dù biến number chỉ tồn tại trong hàm square và khi kết thúc hàm number



Hình 6: Minh họa cách python truyền tham chiếu với mutable

Tiếp theo cũng từ ví dụ trên thay vì sử dụng append trong hàm add chúng ta sử dụng phép + và quan sát address của biến a-list



Hình 7: Mutable object với non-inplace operation

Phân tích 1 chút ví dụ này như sau:

- Biến a-list trong hàm add đầu tiên tham chiếu tới address1. Sau được gán lại bằng chính nó cộng thêm một [number] bằng phép + một phép non-inplace vì vậy lúc này Python tạo ra một object mới là [1, 2, 3, 4] và gán address3 lại cho a-list
- Khi biến a-list ở ngoài gọi hàm add lúc này python tham chiếu giá trị address3 cho a-list. Và sau cùng a-list sẽ có address3.

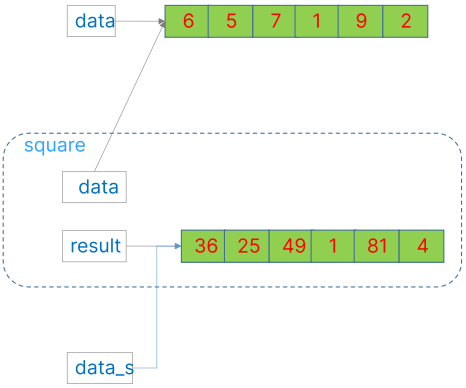
Chúng ta sẽ đi sâu hơn một chút về cách xử lý list thường gặp. Đầu tiên là làm việc với list mới nhưng không thay đổi list gốc.

Với cách làm này chúng ta sẽ có một list độc lập và có thể có data từ list đã có trước đó giúp chúng ta quản lý riêng biệt các list khác nhau. Trong ví dụ chúng ta:

- Tạo một mảng rỗng sau đó dùng hàm inplace append() lấy các phần tử từ list đã có.
- Lúc này addres của mảng rỗng vừa tạo khác hoàn toàn so với mảng ban đầu đồng thời mang các giá trị của mảng ban đầu.

```
[ ]  def square(data):
         result = []
         for value in data:
             result.append(value * value)
         return result

     data = [6, 5, 7, 1, 9, 2]
     print("Original data:", data)

     data_s = square(data)
     print("Squared data:", data_s)
```

```
Original data: [6, 5, 7, 1, 9, 2]
Squared data: [36, 25, 49, 1, 81, 4]
```

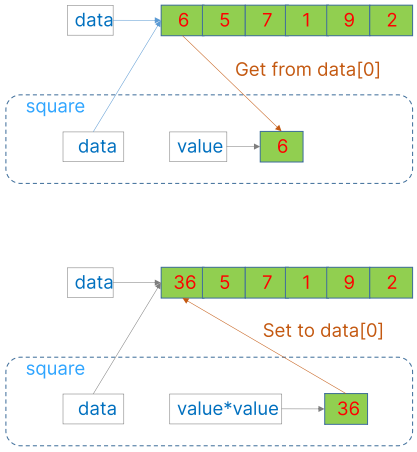


Hình 8: Cách tạo list mới từ list đã có

Cách tiếp theo là chúng ta mong muốn cập nhật các phần tử con của list. Vì list là mutable nên chỉ cần cập nhật thẳng các phần tử của chúng là được.

```
[ ]  # Hàm tính bình phương các phần tử trong danh sách
     def square(data):
         length = len(data)
         for i in range(length):
             value = data[i]
             data[i] = value * value
         return data

     # Dữ liệu thử nghiệm
     data = [6, 5, 7, 1, 9, 2]
     print("Original data:", data)

     # Tính bình phương
     squared_data = square(data)
     print("Squared data:", squared_data)
```

```
Original data: [6, 5, 7, 1, 9, 2]
Squared data: [36, 25, 49, 1, 81, 4]
```



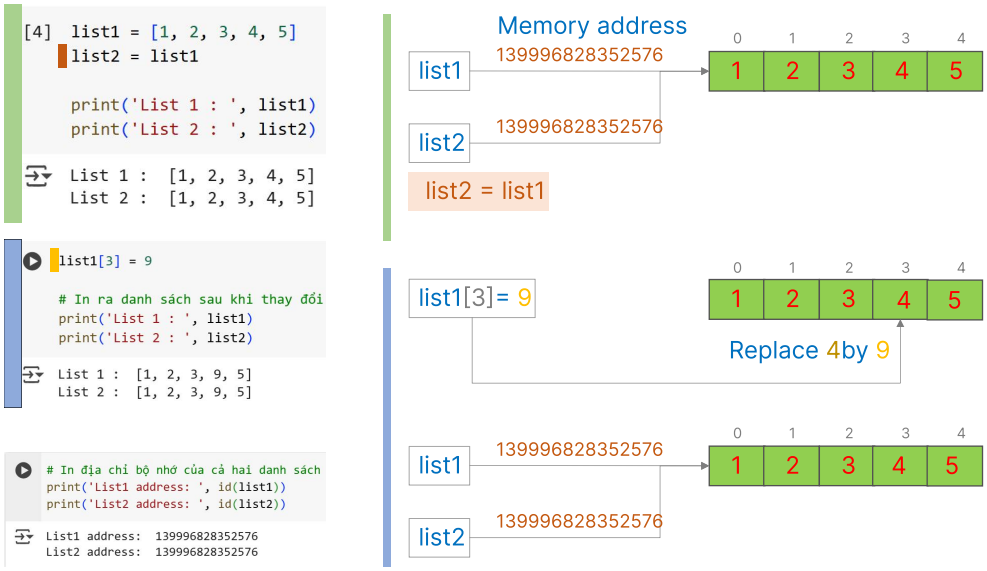
Hình 9: Cách cập nhật các phần tử của list

IV. Sao chép biến, address và List

Sau khi đã hiểu bản chất của việc Python truyền tham chiếu như thế nào chúng ta sẽ tìm hiểu tiếp theo về cách Python gán list và thực hiện copy một list.

Cùng quan sát ví dụ (Hình 10) dưới đây:

- Đoạn code đầu tiên khi tạo list2 = list1 hoàn toàn giống với kiến thức đã giảng ở phần trước lúc này list 2 sẽ trỏ tới cùng address của list1
- Tiếp theo ta thay đổi phần tử tại list1[3] = 9. Tuy nhiên lúc này vì list2 cũng trỏ tới address của list1 nên tại index 3 của list 2 cũng thay đổi thành list2[3] = 9.
- Tiếp theo ta cùng in ra address của 2 list để kiểm tra và đúng là cả 2 list đều trỏ tới cùng 1 address.

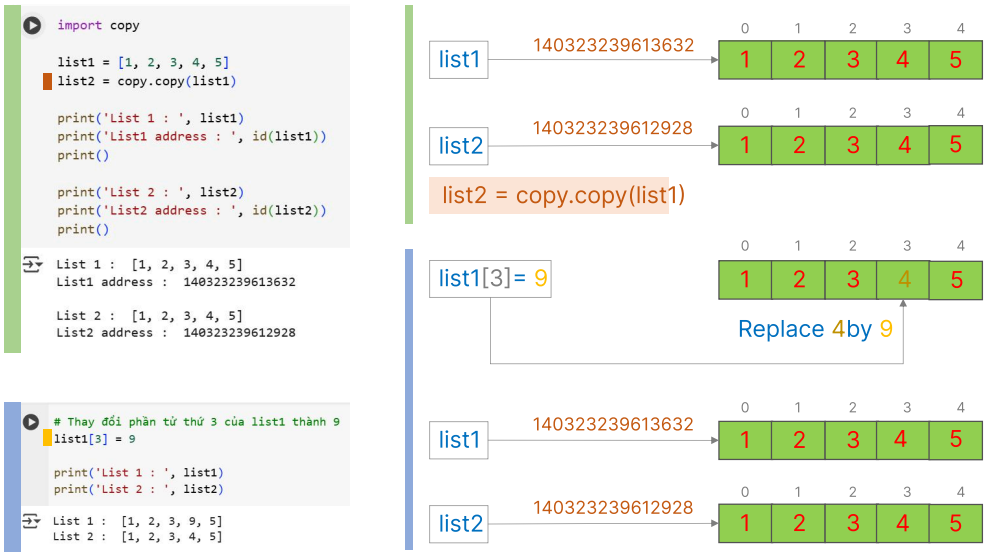


Hình 10: Minh họa về cách python gán list

Với cách gán này khi thay đổi giá trị list gốc các list tham chiếu tới cũng thay đổi giá trị. Trong lập trình việc này đôi khi gây ra bug nếu chúng ta không quản lý cẩn thận các list cùng tham chiếu tới một địa chỉ. Vậy có cách nào để tạo ra một list mới tách biệt với list cũ một cách nhanh gọn mà không cần phải viết hàm xử lý như ví dụ hình 8 hay không ?

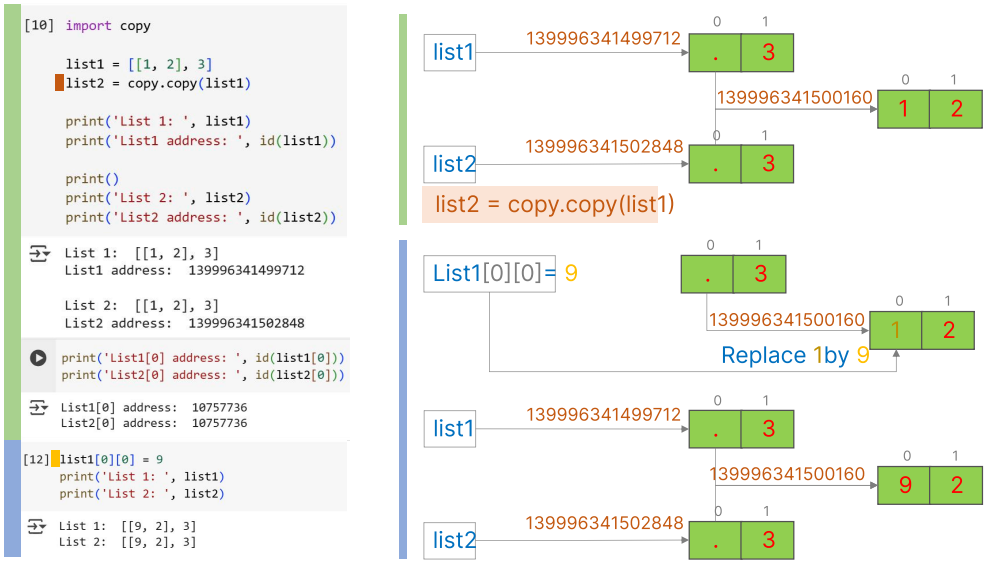
Ta sẽ dùng hàm copy có sẵn trong python và cùng quan sát các hành vi của hàm này thông qua ví dụ sau:

- Với ví dụ này hoàn toàn giống với ví dụ trước khác ở chỗ thay vì gán trực tiếp ta gán list2 = copy.copy(list1).
- Quan sát thấy khi in ra chúng ta có 2 address khác nhau và để kiểm tra list2 có còn tham chiếu tới address của list1 không ta thay đổi phần tử con của list1 như ví dụ trước list1[3] = 9. Và lần này sau ta thấy rõ list2 không bị thay đổi theo chứng tỏ list2 không còn tham chiếu tới list1.



Hình 11: Minh họa về hàm copy

Qua ví dụ trên ta thấy khi sử dụng copy() chúng ta tạo ra một list có address khác hoàn toàn với list gốc và có dữ liệu của phần từ gốc. Quan sát kỹ hơn một tí các phần tử trong list hiện đang là immutable. Vậy nếu phần tử trong list là mutable thì sẽ thế nào? Chúng ta cùng quan sát trong ví dụ sau:



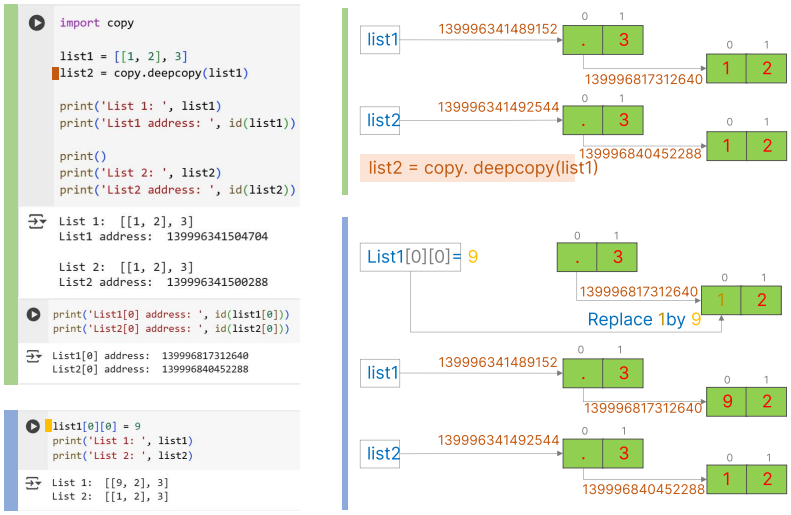
Hình 12: Minh họa về shallow copy

- Như ví dụ trước, ta thấy rõ list1 và list2 đã khác address. Tuy nhiên lần này phần tử index 0 của cả 2 là một list[1,2].
- Quan sát thấy cả list1[0] và list2[0] đều trỏ tới cùng một address. Ta thử thay đổi giá trị của list[1,2] lúc này cả list1[0] và list2[0] đều bị thay đổi.

Với lệnh copy() này chúng ta gọi là **shallow copy** là một bản sao nông, tức là list mới được tạo ra có địa chỉ mới nhưng các phần tử con vẫn cùng tham chiếu với list gốc. Vậy lệnh copy() chúng ta nên dùng khi các phần tử con là immutable. Tiếp theo chúng ta cùng thực hiện tạo ra một bản sao với các phần tử con là mutable nhưng chúng có address khác với address của phần tử con gốc.

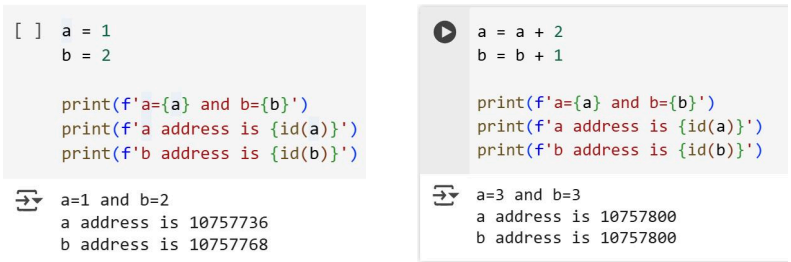
Quan sát ví dụ dưới ta thấy:

- Lần này chúng ta sử dụng lệnh deepcopy() và thấy rằng list1[0] và list2[0] đã khác address. Thử thay đổi một phần tử của list1[0] để kiểm chứng và lúc này ta thấy rõ là list2[0] hoàn toàn không bị ảnh hưởng. Vậy là ta đã thực hiện deepcopy().
- Vậy lệnh deepcopy() sẽ dùng trong các trường hợp nested list (các phần tử con trong list lại là một list)



Hình 13: Minh họa về deepcopy

Vậy các phần tử con mà là immutable khi sao chép thì chúng có địa chỉ như thế nào nhỉ? Thật ra chúng ta có thể dễ dàng trả lời với kiến thức từ phần variables assignment đối với các immutalbe int Python sẽ tận dụng những object đã tạo để gán tham chiếu cho chúng mà thôi. Nhìn ví dụ hình 14 ta sẽ thấy a = b = 3 đều có chung một address.



Hình 14: Minh họa về small number caching

V. Kết Luận

Qua các ví dụ và phân tích ở trên, chúng ta có thể rút ra một số điểm then chốt trong cách Python quản lý biến, bộ nhớ và kiểu dữ liệu:

- Biến trong Python chỉ là tên tham chiếu đến object, không phải "hộp chứa giá trị"như trong một số ngôn ngữ khác.
- Kiểu dữ liệu mutable và immutable có hành vi rất khác biệt trong thao tác và truyền tham chiếu. Việc hiểu rõ điều này giúp tránh được nhiều lỗi khó phát hiện.
- Các phép toán in-place và non-inplace ảnh hưởng trực tiếp đến việc có tạo ra object mới hay không.
- Gán list = list khác không phải là copy, mà là alias – hai biến trỏ về cùng một vùng nhớ.
- Để sao chép thật sự, cần dùng copy() hoặc deepcopy() tùy theo độ phức tạp (shallow vs. nested list).
- Python có cơ chế tối ưu bộ nhớ như small integer caching, giúp tái sử dụng object với các giá trị immutable nhỏ.

Việc nắm vững các cơ chế trên không chỉ giúp viết code đúng đắn, hiệu quả mà còn nâng cao khả năng tối ưu bộ nhớ và tránh các bug logic khó phát hiện trong lập trình Python.

- Hết -