AI VIET NAM
@aivietnam.edu.vn

AI VIET NAM – WARM-UP COURSE 2026

Python - Advanced Variables, Numeric Types and Operators

Nguyễn Quốc Thái Xỉn Quý Hùng Đinh Quang Vinh

I. Một số toán tử trong Python

Operator là gì?

Trong Python, toán tử (operator) là các ký hiệu đặc biệt để thực hiện các phép tính số học hoặc logic trên các giá trị hoặc biến.

Minh họa về operator

```
a = 5
b = 3
print(a + b) # Output: 8
```

Khi bắt đầu làm quen với Python, các toán tử đầu tiên mà chúng ta được học là các phép tính cộng trừ nhân chia cơ bản $(+, -, *, /)$. Các toán tử này còn được gọi là Toán tử số học (Arithmetic Operator), và là một trong những loại toán tử cơ bản và phổ biến nhất. Bên cạnh đó, Python còn hỗ trợ rất nhiều loại toán tử khác.

Dựa vào chức năng, người ta phân loại các toán tử thành các nhóm như: **Arithmetic Operator** (đã được học từ trước), **Comparison Operator**, **Logical Operator**, **Assignment Operator** và **Bitwise Operator**

Comparison Operator

Comparison Operator

Đây là các toán tử dùng để so sánh giá trị của các biến với nhau. Giá trị trả về của các toán tử này là `True` hoặc `False`

Các toán tử này trong Python tương tự như các phép so sánh thông thường trong toán học (`>`, `>=`, `<`, `<=`). Tuy nhiên, phép so sánh bằng là `==` và so sánh khác là `!=`

Ví dụ về comparison operator

```
a = 10
b = 2
print(a > b) # Output: True
```

Logical Operator

Logical Operator

Đây là các toán tử dùng để thực hiện các phép tính logic trong toán học. Ba loại chính bao gồm: **and, or**, và **not**. Giá trị trả về của các toán tử này là `True` hoặc `False`

Ví dụ về logical operator

```
a = True
b = False
print(a and b) # Output: False
```

Assignment Operator

Assignment Operator

Đây là các toán tử dùng để gán giá trị cho biến. Có thể thấy, phép `=` thuộc loại này. Bên cạnh đó, Python hỗ trợ thêm các toán tử thực hiện tính và gán cùng một lúc. Đó là các phép `+=, -=, *=, /=`.

Các toán tử này giúp quá trình code nhanh hơn, đặc biệt khi chúng ta có tên biến quá dài.

Ví dụ về assignment operator

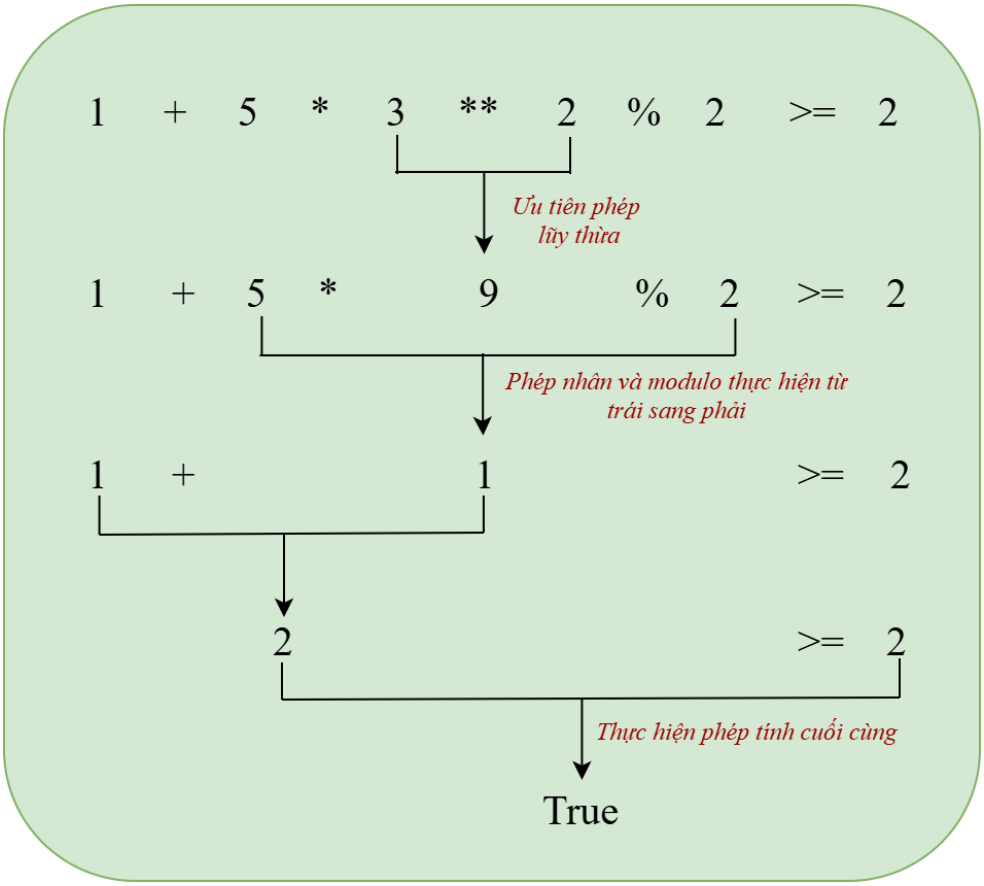
```
a = 5
a += 3
print(a) # Output: 8
```

Thứ tự các phép tính

Một 'câu thần chú' quen thuộc trong toán học là 'Nhân chia trước, cộng trừ sau'. Đối với Python, các toán tử không chỉ dừng lại ở các phép tính cộng trừ nhân chia thông thường mà còn các phép như **Comparison**, **Assignment**,... Vậy, trong Python, quy luật các phép tính này như thế nào?

Trong Python, người ta có quy định toán tử nào sẽ được ưu tiên thực hiện trước. Các phép tính cùng thứ tự sẽ được thực hiện từ trái qua phải. Thứ tự như sau:

1. Phép mũ: `**`
2. Phép nhân, chia, phần dư và phép chia lấy phần nguyên: `*, /, //`
3. Phép cộng, trừ: `+, -`
4. Các toán tử so sánh: `>, >=, <, <=`
5. Các toán tử so sánh bằng: `==, !=`
6. Các toán tử gán: `=, %=, /=, //=, -=, +=, *=, **=`



Hình 1: Ví dụ về thứ tự tính toán trong Python.

Ví dụ về thứ tự tính toán

Giải thích: Phép tính thực hiện đầu tiên là phép lũy thừa (**), rồi đến phép nhân (*), phép cộng (+) và cuối cùng là phép gán (+=)

Ví dụ về thứ tự tính toán

```
a = 5
a += 5 + 3 ** 2  * 5
print(a) # Output: 55
```

II. Mutable và Immutable trong Python

Mutable objects trong Python là những đối tượng **có thể** thay đổi giá trị sau khi đã được khởi tạo. Ngược lại, Immutable objects trong Python là những đối tượng **không thể** thay đổi giá trị sau khi khởi tạo

Immutable Object

Ví dụ cho Immutable object trong Python là các kiểu dữ liệu **int, Float, bool, string**. Ắt hẳn chúng ta sẽ thắc mắc rằng, trong quá trình lập trình, rõ ràng chúng ta vẫn có thể 'thay đổi' giá trị của các biến dữ liệu trên, nhưng tại sao chúng lại là Immutable? Thật ra, về bản chất, thứ chúng ta làm là tạo ra một object mới với giá trị mới, chứ không thực sự thay đổi giá trị

Ví dụ về immutable trong Python

Để hiểu rõ hơn về khái niệm 'tạo ra giá trị mới' đối với các kiểu dữ liệu *immutable*, ta sử dụng hàm `id` trong Python. Hàm này trả về một con số đại diện cho object trong bộ nhớ. Hai object có `id` khác nhau nghĩa là chúng hoàn toàn độc lập với nhau.

Ở ví dụ bên, ta thấy rằng sau khi gán `a = 6`, giá trị `id(a)` đã thay đổi. Điều này cho thấy biến `a` không còn trỏ đến object cũ nữa: Python đã tạo ra một object mới rồi gán giá trị `6`.

Trong trường hợp của `string`, tính chất *immutable* còn rõ hơn. Khi ta cố gắng thay đổi trực tiếp một ký tự trong chuỗi, Python sẽ báo lỗi vì chuỗi không thể bị chỉnh sửa tại chỗ.

Ví dụ về immutable

```
a = 5
print(id(a)) # Output: 11654504
a = 6
print(id(a)) # Output: 11654536

s = "AI"
s[0] = "I"   # Báo lỗi: string là
                    immutable
```

Mutable Object

Ngược lại, các Mutable object trong Python bao gồm **list, dict, set**. Các kiểu dữ liệu này có điểm chung là đều dùng để chứa một lượng phần tử. Chúng đều cho phép thêm và xóa phần tử, nên chúng được coi là Mutable object

Ví dụ về Mutable object

Trái với các kiểu dữ liệu Immutable, các Mutable object như **list** cho phép chúng ta thay đổi nội dung của object mà không tạo ra object mới. Điều này có thể được quan sát thông qua hàm `id` trong Python.

Trong ví dụ bên, ta có thể thấy rằng mặc dù chúng ta thêm phần tử vào list bằng `append`, giá trị `id` vẫn giữ nguyên. Điều đó chứng tỏ rằng object `lst` không hề được tạo mới, mà được chỉnh sửa trực tiếp.

Ví dụ về Mutable object

```
lst = [1, 2, 3]
print(id(lst))    # Output:
                    135930949555200

lst.append(4)     # Thay đổi nội dung
                    list
print(id(lst))    # Output không đổi
```

III. Số thực trong Python

Giá trị inf

Giá trị inf có thể được coi là một số thực trong Python, và số này có giá trị rất lớn, lớn hơn mọi số khác. Chú ý rằng **inf không phải là một con số cụ thể**, chỉ đơn giản là một số lớn vô tận.

Ta thường gặp giá trị inf khi xảy ra hiện tượng tràn số (overflow) trong quá trình tính toán số thực. Vì vậy, trong thực tế, giá trị này được xem như một dạng lỗi. Trong AI, inf thường xuất hiện khi xảy ra hiện tượng Gradient Exploding — một vấn đề khiến các giá trị và gradient trở nên quá lớn, vượt giới hạn biểu diễn và dẫn đến overflow

Khi nào xuất hiện giá trị "inf"?

Khác với các giá trị nguyên (int), các số thực trong python có giới hạn về giá trị. Trong quá trình tính toán, nếu một số vượt quá giá trị trên của nó, sẽ xảy ra hiện tượng tràn số (overflow). Lúc này, giá trị trả về sẽ được gán thành inf.

Tràn số trong Python

```
a = 1e400 # 10 mũ 400
print(a) # inf
print(type(a)) # <class "float">
```

Vì là số thực, chúng ta có thể thực hiện các toán tử với nó (`+, -, *, >, <, ==`,...). Tuy nhiên, các kết quả trả về sẽ có chút 'khác biệt' so với thông thường

Cách Python xử lí inf

Do inf là một số 'vô cùng lớn', nên nếu lấy inf trừ đi bất kì số nào thì giá trị trả về inf. Tương tự, nếu ta tiếp tục cộng inf với bất kì số nào (kể cả inf) cũng sẽ nhận được giá trị inf

Toán tử với inf

```
print(float("inf") - 1e100) # inf
print(float("inf") + float("inf"))
# inf
```

Giá trị nan

Nan là viết tắt của "Not a number", nghĩa là con số này không có ý nghĩa gì cả. Trong DS, ta thường bắt gặp nó trong bộ dữ liệu thô, với nhiều giá trị không đầy đủ. Lúc này, các giá trị bị thiếu đó thường là NaN.

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S
5	6	0	3	Moran, Mr. James	male	NaN	0	0	330877	8.4583	NaN	Q
7	8	0	3	Palsson, Master. Gosta Leonard	male	2.0	3	1	349909	21.0750	NaN	S

Hình 2: Ví dụ về giá trị NaN trong dữ liệu Titanic.

Trong tính toán, ta cũng có thể bắt gặp nó thông qua các phép tính vô nghĩa, gây ra các trường hợp vô định. Trong toán học, người ta định nghĩa các dạng vô định gồm:

$$\frac{0}{0},\ \frac{\infty}{\infty},\ \infty - \infty,\ 0 \cdot \infty,\ 0^0,\ 1^\infty,\ \infty^0.$$

Trong Python, khi ta thực hiện các phép toán dạng vô định, ta sẽ nhận được giá trị nan. Ngoài ra, các phép tính với nan cũng sẽ trả về nan. Tuy nhiên, một số phép tính sẽ gây lỗi thay vì trả về nan vì Python thực hiện kiểm tra điều kiện trước khi tính toán, chẳng hạn như các phép chia cho 0. Hoặc, phép `pow(1,float("inf"))` sẽ ra kết quả là 1.

Cách Python xử lí nan

Việc trừ 2 số inf với nhau sẽ trả về giá trị nan thay vì 0. Ta có thể hiểu là inf chỉ có thể coi là một con số vô cùng lớn chứ không phải là một con số cụ thể. Do đó, đây là một dạng vô định

Tương tự, đối với phép nhân inf với 0, kết quả trả về nên là inf hay 0? Với tư duy trên, ta cũng nhận định được đây là dạng vô định

Ở phép tính cuối cùng, toán tử giữa giá trị nan và bất kì giá trị nào cũng sẽ dẫn đến nan

Minh họa một số phép tính với nan

```python
print(float("inf") - float("inf"))
# nan
print(float("inf") * 0)
# nan
print(float("nan") + float("inf"))
# nan
```

Sai số trong số thực

Các giá trị thực trong Python không được lưu chính xác mà được lưu bằng giá trị gần đúng, do cơ chế dấu phẩy động (Floating point). Điều này khá phổ biến trong các ngôn ngữ lập trình khác, không chỉ riêng Python

0.1 + 0.2 != 0.3?

Mặc dù biểu thức $0.1 + 0.2$ về mặt toán học đúng bằng 0.3, Python (và hầu hết các ngôn ngữ lập trình) không thể biểu diễn chính xác các số như 0.1 trong hệ nhị phân. Do đó, mỗi giá trị được lưu dưới dạng một số xấp xỉ, dẫn đến tổng của chúng cũng bị sai số. Khi so sánh trực tiếp hai số dấu phẩy động, sự khác biệt rất nhỏ này khiến biểu thức trả về `False`.

Minh họa về dấu phẩy động

```python
print(0.1 + 0.2 == 0.3)
# False
```

Khi thực hiện nhiều phép tính với số thực, các số ngày càng được làm tròn, khiến cho kết quả sau cùng có nhiều sai lệch