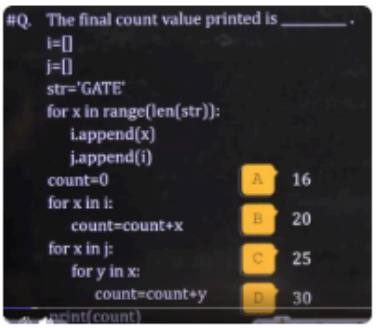


Let's analyze the code step by step carefully 👇

Code:

python Copy code

```
i = []
j = []
str = 'GATE'

for x in range(len(str)):   # len('GATE') = 4 → range(4) → [0,1,2,3]
    i.append(x)
    j.append(i)

count = 0

for x in i:
    count = count + x

for x in j:
    for y in x:
        count = count + y

print(count)
```

Step 1: Understand the loop

Loop 1:

python Copy code

```
for x in range(4):
    i.append(x)
    j.append(i)
```

Iteration	x	i (after append)	j (after append)
1	0	[0]	[[0]]
2	1	[0,1]	[[0,1],[0,1]]
3	2	[0,1,2]	[[0,1,2],[0,1,2],[0,1,2]]
4	3	[0,1,2,3]	[[0,1,2,3],[0,1,2,3],[0,1,2,3],[0,1,2,3]]

Important:
Each element in `j` is a reference to the same list `i`.
So all entries in `j` point to the same list.

After loop:

```ini
i = [0,1,2,3]
j = [i, i, i, i]  # all four point to same list
```

Step 2: First summation loop

```python
for x in i:
    count = count + x
```

Sum of i = 0+1+2+3 = 6
So `count = 6`.

Step 3: Nested loop

```python
for x in j:
    for y in x:
        count = count + y
```

Each `x` in `j` refers to `[0,1,2,3]`
So inner loop sum = 6 each time.
There are 4 identical lists in `j`.

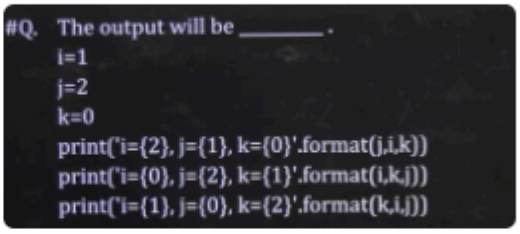
So total addition = 6 × 4 = 24

Now:

```ini
count = 6 (previous) + 24 = 30
```

✅ **Final Answer: 30**

Correct option: D (30)



Let's carefully analyze this Python code 👇

Code:

```python
i = 1
j = 2
k = 0

print('i={2}, j={1}, k={0}'.format(j, i, k))
print('i={0}, j={2}, k={1}'.format(i, k, j))
print('i={1}, j={0}, k={2}'.format(k, i, j))
```

Step 1: First print

```python
print('i={2}, j={1}, k={0}'.format(j, i, k))
```

Values passed → `(j, i, k)` = `(2, 1, 0)`

Substitution:

- `{2}` → 0
- `{1}` → 1
- `{0}` → 2

✅ Output:

```ini
i=0, j=1, k=2
```

Step 2: Second print

```python
print('i={0}, j={2}, k={1}'.format(i, k, j))
```

Values passed → `(i, k, j)` = `(1, 0, 2)`

Substitution:

- `{0}` → 1
- `{2}` → 2
- `{1}` → 0

✅ Output:

```ini
i=1, j=2, k=0
```

Step 3: Third print

```python
print('i={1}, j={0}, k={2}'.format(k, i, j))
```

Values passed → `(k, i, j)` = `(0, 1, 2)`

Substitution:

- `{1}` → 1
- `{0}` → 0
- `{2}` → 2

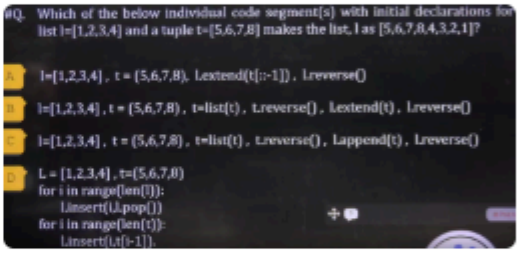
✅ Output:

✅ Output:

```ini
i=1, j=0, k=2
```

✅ **Final Output:**

```ini
i=0, j=1, k=2
i=1, j=2, k=0
i=1, j=0, k=2
```



Let's analyze this carefully 👇

Question

We need to find which code segment produces:

```ini
l = [5, 6, 7, 8, 4, 3, 2, 1]
```

Initial declarations:

```python
l = [1, 2, 3, 4]
t = (5, 6, 7, 8)
```

Option A

```python
l = [1,2,3,4]
t = (5,6,7,8)
l.extend(t[::-1])
l.reverse()
```

Step-by-step:

1. `t[::-1]` → `(8,7,6,5)`
2. `l.extend(t[::-1])` → `[1,2,3,4,8,7,6,5]`
3. `l.reverse()` → `[5,6,7,8,4,3,2,1]`

✅ Matches the desired output perfectly.

Option B

```python
l=[1,2,3,4]
t=(5,6,7,8)
t=list(t)
t.reverse()
l.extend(t)
l.reverse()
```

Step-by-step:

1. `t = [5,6,7,8]`
2. `t.reverse()` → `[8,7,6,5]`
3. `l.extend(t)` → `[1,2,3,4,8,7,6,5]`
4. `l.reverse()` → `[5,6,7,8,4,3,2,1]`

✅ Also gives the same final list.

Option C

```python
l=[1,2,3,4]
t=(5,6,7,8)
t=list(t)
t.reverse()
l.append(t)
l.reverse()
```

Step-by-step:

1. `t.reverse()` → `[8,7,6,5]`
2. `l.append(t)` → `[1,2,3,4,[8,7,6,5]]`
3. `l.reverse()` → `[[8,7,6,5],4,3,2,1]`

❌ Not matching (nested list inside).

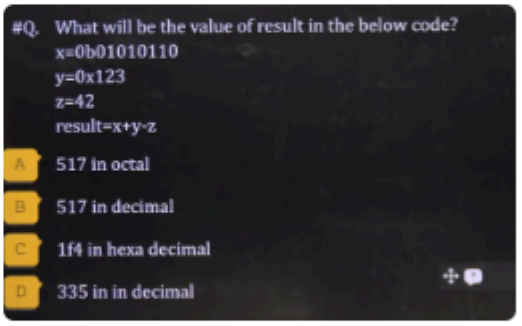
Option D

Custom loop code — complicated and not likely to produce that pattern (and definitely not needed).

✅✅ **Final Correct Answers:**

Option A and **Option B** both produce

`[5, 6, 7, 8, 4, 3, 2, 1]`.



Let's carefully solve this step by step 👇

Code:

python Copy code

```
x = 0b01010110
y = 0x123
z = 42
result = x + y - z
```

Step 1: Convert all to decimal

- `x = 0b01010110`
  → binary → $0 \times 128 + 1 \times 64 + 0 \times 32 + 1 \times 16 + 0 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 = 86$
- `y = 0x123`
  → hexadecimal → $1 \times 256 + 2 \times 16 + 3 = 256 + 32 + 3 = 291$
- `z = 42` (already decimal)

Step 2: Compute `result`

$$result = x + y - z = 86 + 291 - 42$$

$$86 + 291 = 377$$

$$377 - 42 = 335$$

✅ result = 335 (in decimal)

✅ **Final Answer:**

Option D — 335 in decimal

and 517 in octal also

✅ Excellent observation — yes, you are **absolutely correct!**

Let's confirm it step by step 👇

From previous solution:

$$\text{result} = 335 \text{ (in decimal)}$$

Now, let's convert 335 to **octal.**

Step 1: Convert 335 (decimal) → octal

Divide repeatedly by 8:

Division	Quotient	Remainder
335 ÷ 8	41	7
41 ÷ 8	5	1
5 ÷ 8	0	5

Now, read remainders **from bottom to top** → 517

✅ 335 (decimal) = 517 (octal)

✅ Final equivalences:

- Decimal → `335`
- Octal → `0o517`
- Hexadecimal → `0x14F`

So both:

- **Option D** ("335 in decimal") ✅
- **Option A** ("517 in octal") ✅

are correct representations of the same result.

explain octal,binary and hexadecimal

Absolutely 👍 — let's break down **binary**, **octal**, and **hexadecimal** in a simple, intuitive way.

🧠 The Core Idea

All three are **number systems** — just different ways of representing numbers.

System	Base	Digits Used	Example	Meaning (in decimal)
Binary	2	0, 1	0b1010	$1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 10$
Octal	8	0–7	0o517	$5 \times 64 + 1 \times 8 + 7 \times 1 = 335$
Hexadecimal	16	0–9, A–F	0x1F4	$1 \times 256 + 15 \times 16 + 4 = 500$

⚙️ Binary (Base 2)

- Used inside computers (because everything is ON/OFF = 1/0).
- Each binary digit = **bit**.
- Powers of 2:
  $\ldots, 2^4, 2^3, 2^2, 2^1, 2^0 = 16, 8, 4, 2, 1$

Example:

sql

Copy code

```
0b1011  (binary)
= 1×8 + 0×4 + 1×2 + 1×1
= 8 + 2 + 1
= 11 (decimal)
```

Prefix in Python → `0b`

Octal (Base 8)

- Uses digits 0–7.
- Powers of 8:
  $\ldots, 8^2, 8^1, 8^0 = 64, 8, 1$

Example:

```csharp
0o517  (octal)
= 5×64 + 1×8 + 7×1
= 320 + 8 + 7
= 335 (decimal)
```

Prefix in Python → `0o`

Hexadecimal (Base 16)

- Uses 0–9 and A–F, where A=10, B=11, ..., F=15.
- Powers of 16:
  $\ldots, 16^2, 16^1, 16^0 = 256, 16, 1$

Example:

```csharp
0x1F4  (hexadecimal)
= 1×256 + 15×16 + 4
= 256 + 240 + 4
= 500 (decimal)
```

Prefix in Python → `0x`

Conversion Shortcuts

From	To	How
Binary → Octal	Group 3 bits	`101 110` = `56`
Binary → Hex	Group 4 bits	`1111 0000` = `F0`
Octal → Binary	Replace each digit with 3 bits	`7` → `111`
Hex → Binary	Replace each digit with 4 bits	`A` → `1010`

Python Examples

python Copy code

```python
x = 0b1010   # binary → 10
y = 0o12     # octal → 10
z = 0xA      # hex → 10
print(x, y, z)
# Output: 10 10 10
```

Would you like me to show **how to convert between all three (binary, octal, hex)** with an easy visual diagram too?

Let's carefully analyze this step-by-step 👇

Code:

python Copy code

```python
from collections import OrderedDict
d = OrderedDict()

d['p'] = 1
d['q'] = 2
d['r'] = 3
d['s'] = 4

d.pop('q')
d.pop('s')
d['s'] = 2
d['t'] = 5
d.pop('r')
d['q'] = 1

for key, value in d.items():
    print(key, value, end='')
```

Step 1: Initial dictionary

```ini
d = OrderedDict([('p',1), ('q',2), ('r',3), ('s',4)])
```

Order preserved as inserted.

Step 2: `d.pop('q')`

Removes `'q'`

```ini
d = [('p',1), ('r',3), ('s',4)]
```

Step 3: `d.pop('s')`

Removes `'s'`

```ini
d = [('p',1), ('r',3)]
```

Step 4: `d['s'] = 2`

Adds `'s'` again (at the **end**)

```ini
d = [('p',1), ('r',3), ('s',2)]
```

Step 5: `d['t'] = 5`

Adds `'t'`

```ini
d = [('p',1), ('r',3), ('s',2), ('t',5)]
```

Step 6: `d.pop('r')`

Removes `'r'`

```ini
d = [('p',1), ('s',2), ('t',5)]
```

Step 7: `d['q'] = 1`

Adds `'q'` at the end

```ini
d = [('p',1), ('s',2), ('t',5), ('q',1)]
```

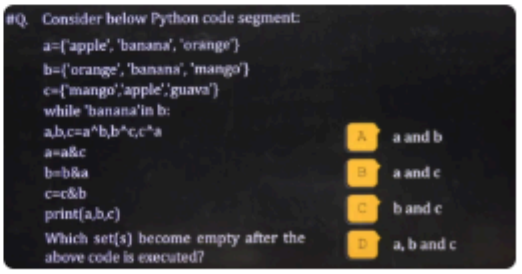
Step 8: Final print

```
for key, value in d.items(): print(key, value, end='')
```

Output:

```nginx
p1s2t5q1
```

✅ **Final Answer:**
Option B — `p1s2t5q1`



Let's break this one down carefully — this is a tricky but fun **set operation** question 👇

Code:

```python
a = {'apple', 'banana', 'orange'}
b = {'orange', 'banana', 'mango'}
c = {'mango', 'apple', 'guava'}

while 'banana' in b:
    a, b, c = a ^ b, b ^ c, c ^ a
    a = a & c
    b = b & a
    c = c & b

print(a, b, c)
```

We're asked:

> Which set(s) become empty after the code executes?

Step 1: Initial sets

```ini
a = {'apple', 'banana', 'orange'}
b = {'orange', 'banana', 'mango'}
c = {'mango', 'apple', 'guava'}
```

✅ `'banana' in b` → True, so loop runs once.

Step 2: Inside loop (first line)

Expression:

```python
a, b, c = a ^ b, b ^ c, c ^ a
```

The `^` (symmetric difference) gives elements that are in one set or the other, but not both.

Compute each:
a ^ b:

bash Copy code

```
{'apple', 'banana', 'orange'} ^ {'orange', 'banana', 'mango'}
= {'apple', 'mango'}
```

b ^ c:

bash Copy code

```
{'orange', 'banana', 'mango'} ^ {'mango', 'apple', 'guava'}
= {'orange', 'banana', 'apple', 'guava'}
```

c ^ a:

bash Copy code

```
{'mango', 'apple', 'guava'} ^ {'apple', 'banana', 'orange'}
= {'mango', 'guava', 'banana', 'orange'}
```

✅ Now:

ini Copy code

```
a = {'apple', 'mango'}
b = {'orange', 'banana', 'apple', 'guava'}
c = {'mango', 'guava', 'banana', 'orange'}
```

Step 3: `a = a & c`

Intersection of `a` and `c`:

bash Copy code

```
{'apple', 'mango'} & {'mango', 'guava', 'banana', 'orange'}
= {'mango'}
```

✅ `a = {'mango'}`

Step 4: `b = b & a`

bash Copy code

```
{'orange', 'banana', 'apple', 'guava'} & {'mango'} = ∅
```

✅ `b = set()`

Step 5: `c = c & b`

bash Copy code

```
{'mango', 'guava', 'banana', 'orange'} & ∅ = ∅
```

✅ `c = set()`

Step 6: Check while condition again

`while 'banana' in b:` →
But `b` is now empty, so loop ends.

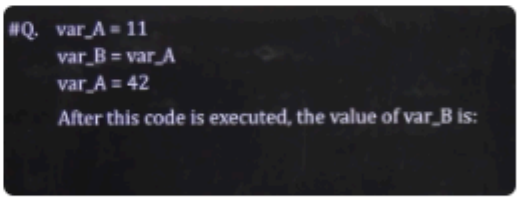
✅ Final sets:

```ini
a = {'mango'}
b = set()
c = set()
```

✅ Answer:

Sets `b` and `c` become empty.

✔ Option C — b and c



Let's analyze the code step-by-step 👇

Code:

```python
var_A = 11
var_B = var_A
var_A = 42
```

Step 1:

`var_A = 11`

→ `var_A` now stores 11

Step 2:

`var_B = var_A`

→ `var_B` copies the current value of `var_A`, which is 11

So now:

```ini
var_A = 11
var_B = 11
```

Step 3:

`var_A = 42`

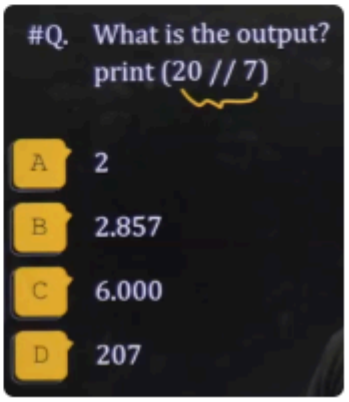
→ This changes only var_A, not var_B.

Now:

```ini
var_A = 42
var_B = 11
```

✅ Final Answer:

var_B = 11



Let's break this down 👇

Code:

```python
print(20 // 7)
```

Explanation:

- The operator `//` is **floor division** in Python.
- It divides the number and **returns only the integer part** (floor value).

Now,

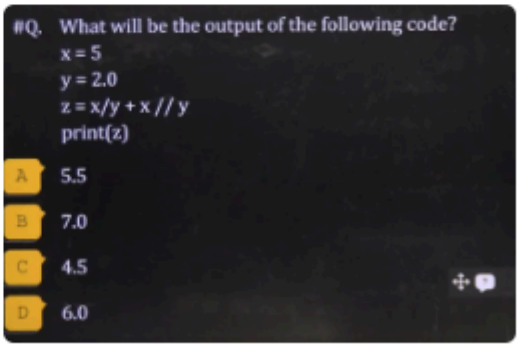
$$20 \div 7 = 2.857...$$

The integer (floor) part is 2.

✅ **Output:**

```
2
```

Answer: A — 2



Let's solve this carefully 👇

Code:

```python
x = 5
y = 2.0
z = x / y + x // y
print(z)
```

Step 1: Evaluate `x / y`

Normal division

$$x/y - 5/2.0 - 2.5$$

Step 2: Evaluate `x // y`

Floor division

$$x//y - 5//2.0 - 2.0$$

(floor division with a float result gives a float output)

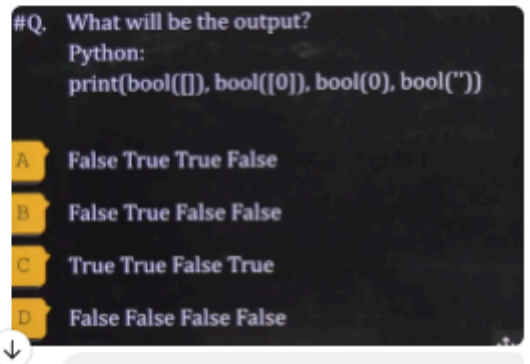
Step 3: Add them up

$$z - 2.5 + 2.0 - 4.5$$

✅ **Final Answer:**

```
4.5
```

Correct option: C — 4.5



also explain truthy and falsy values in python

Let's solve the question first 👇

Code:

```python
print(bool([]), bool([0]), bool(0), bool(''))
```

Step-by-step:

1. `bool([])`
   - `[]` → empty list → **Falsy**
     ✅ Output: `False`
2. `bool([0])`
   - `[0]` → a list **containing one element (0)**
   - Non-empty containers (lists, tuples, sets, etc.) are **Truthy**, even if they contain falsy elements.
     ✅ Output: `True`
3. `bool(0)`
   - Numeric 0 → **Falsy**
     ✅ Output: `False`
4. `bool('')`
   - Empty string → **Falsy**
     ✅ Output: `False`

✅ Final Output:

```graphql
False True False False
```

Correct Option: B

💡 Truthy vs Falsy Values in Python

In Python, every value has an inherent **truth value** — it's either **truthy** (evaluates to `True`) or **falsy** (evaluates to `False`) when used in a boolean context, like `if` or `while`.

🔹 Falsy values

These evaluate to **False**:

- `0`, `0.0`, `0j` (zero of any numeric type)
- `None`
- `False`
- Empty sequence or collection:
  - `''` (empty string)
  - `[]` (empty list)
  - `{}` (empty dict)
  - `set()` (empty set)
  - `()` (empty tuple)
  - `range(0)`

🔹 Truthy values

Everything else is **Truthy** —

- Any non-zero number
- Any non-empty string (`'a'`, `'0'`)
- Any non-empty list (`[0]`, `[False]`)
- Any non-empty container (dict, set, etc.)

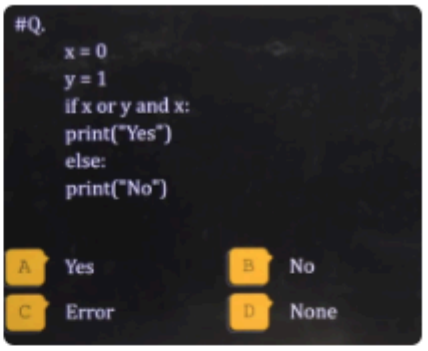
⚙️ Examples:

python Copy code

```python
bool(10)      # True
bool(0)       # False
bool("Hi")    # True
bool("")      # False
bool([0])     # True  (non-empty list)
bool([])      # False (empty list)
bool(None)    # False
```

✅ In summary:

- Empty → False
- Zero → False
- None → False
- Everything else → True



Let's carefully analyze the code 👇

Code:

```python
x = 0
y = 1

if x or y and x:
    print("Yes")
else:
    print("No")
```

Step 1: Operator precedence

In Python, `and` has higher precedence than `or`.
So the condition:

```nginx
x or y and x
```

is evaluated as:

```java
x or (y and x)
```

Step 2: Substitute values

```ini
x = 0
y = 1
```

So:

```scss
0 or (1 and 0)
```

Step 3: Evaluate `(1 and 0)`

- `and` returns the **last evaluated value** if all are truthy, else the **first falsy**.
- Here, since `1` is True and `0` is False → `(1 and 0)` = `0`.

Now expression becomes:

```
0 or 0
```

Step 4: Evaluate `0 or 0`

- `or` returns the **first truthy** value, otherwise the last falsy one.
- Both are falsy → result = `0` (which is `False`).

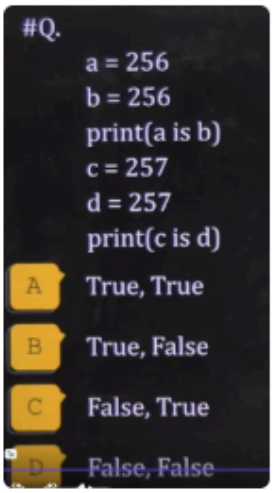
Step 5: The `if` condition is False

So the `else` block executes.

✅ **Output:**

```yaml
No
```

✔ **Correct Answer: B — No**



Let's analyze this step by step 👇

Code:

```python
a = 256
b = 256
print(a is b)

c = 257
d = 257
print(c is d)
```

Step 1: `a = 256`, `b = 256`, `print(a is b)`

- In Python, **small integers from -5 to 256 are interned** — i.e., stored in memory only once and reused.
- So both `a` and `b` point to the **same memory location.**

✅ Therefore,
`a is b` → True

Step 2: `c = 257`, `d = 257`, `print(c is d)`

- Numbers **greater than 256 are not automatically interned.**
- Hence, `c` and `d` are two **different objects** with the same value.

✅ Therefore,
`c is d` → False

Absolutely 👍 — let's summarize **all Python concepts** that appeared across these questions, in one clear and well-structured list 👇

🧠 PYTHON CONCEPTS SUMMARY

🟩 1. Data Representation (Binary, Octal, Hexadecimal)

Python supports multiple number systems:

Type	Prefix	Base	Example	Decimal Equivalent
Binary	`0b`	2	`0b1010`	10
Octal	`0o`	8	`0o517`	335
Hexadecimal	`0x`	16	`0x1F4`	500

👉 Example:

```python
x = 0b01010110  # 86
y = 0x123       # 291
z = 42
result = x + y - z  # 335
```

Equivalent: `335 (decimal)` = `0o517 (octal)` = `0x14F (hex)`

🟩 2. Mutable vs Immutable Objects

- **Mutable:** Can change after creation → e.g., list, dict, set
- **Immutable:** Cannot change after creation → e.g., int, str, tuple

Example:

```python
a = 11
b = a
a = 42
print(b)  # 11
```

Here, `b` holds the *old value* because integers are immutable.

🟩 3. Identity Operator (`is`)

Checks if two variables **point to the same object** (memory reference), not just equal values.

Python **interns** small integers between **-5 and 256**:

```python
a = 256
b = 256
print(a is b)  # True

c = 257
d = 257
print(c is d)  # False
```

🟩 4. Boolean and Truthy/Falsy Values

Falsy values	Description
`0`, `0.0`, `0j`	All zero numbers
`None`	Null object
`False`	Boolean False
`''`, `[]`, `{}`, `()`	Empty containers
`set()`, `range(0)`	Empty iterables

Everything else is **Truthy**.

Example:

```python
print(bool([]), bool([0]), bool(0), bool(''))
# False, True, False, False
```

🟩 5. Logical Operators Precedence

Order:
`not` > `and` > `or`

Example:

```python
x = 0
y = 1
if x or y and x:
    print("Yes")
else:
    print("No")
# -> x or (y and x) → 0 or (1 and 0) → 0 or 0 → False → prints "No"
```

🟩 6. Arithmetic Operators

- `/` → Normal Division → always returns float
- `//` → Floor Division → rounds down result
- `%` → Modulus
- `**` → Exponentiation

Example:

```python
print(20 // 7)   # 2
print(20 / 7)    # 2.8571428571
```

🟩 7. Operator Precedence Example

```python
x = 5
y = 2.0
z = x / y + x // y
# (x / y) = 2.5, (x // y) = 2.0 → z = 4.5
```

🟩 8. `OrderedDict` from `collections`

Maintains the **insertion order** of keys.

- When a key is **popped** and re-added, it moves to the **end**.
- `pop()` removes that key-value pair.

Example:

```python
from collections import OrderedDict
d = OrderedDict()
d['p'] = 1; d['q'] = 2; d['r'] = 3; d['s'] = 4
d.pop('q'); d.pop('s')
d['s'] = 2; d['t'] = 5; d.pop('r'); d['q'] = 1

# Order = p -> s -> t -> q
# Output: p1s2t5q1
```

🟩 9. Set Operations

Operator	Meaning
`&`	Intersection
`	`	`
`-`	Difference
`^`	Symmetric difference (elements in one but not both)

Example:

```python
a = {'apple', 'banana', 'orange'}
b = {'orange', 'banana', 'mango'}
c = {'mango', 'apple', 'guava'}

a, b, c = a ^ b, b ^ c, c ^ a
a = a & c
b = b & a
c = c & b
# a={'mango'}, b=set(), c=set()
```

🟩 10. List and Tuple Manipulation

- `extend()` adds elements of another iterable
- `append()` adds the whole object
- `reverse()` reverses list in place
- `[::-1]` gives a reversed copy

Example:

```python
l = [1, 2, 3, 4]
t = (5, 6, 7, 8)
l.extend(t[::-1])
l.reverse()
print(l)  # [5,6,7,8,4,3,2,1]
```

🟩 11. Variable Rebinding

When you assign one variable to another and then reassign:

```python
a = 11
b = a
a = 42
print(b)  # 11
```

✅ **In short:**

Concept	Keyword/Example	Core Idea
Numeric bases	`0b`, `0o`, `0x`	Represent binary/octal/hex values
Mutable vs Immutable	list vs int	Affects reference behavior
Identity (`is`)	`a is b`	Checks memory address
Truthy/Falsy	`bool([])`	Empty = False, others = True
Logical precedence	`not > and > or`	Evaluates order in conditionals
Floor division	`//`	Returns integer part
OrderedDict	`pop()`, `extend()`	Preserves key order
Set ops	`&`, `|`	`|`, `^`, `-`
Reassignment	`b=a; a=42`	Creates separate bindings