

Let's analyze this classic Python mutable-default-argument question carefully 🧐

🌿 Code:

python

📄 Copy code

```
def f(x, l=[1, 2]):  
    l.append(x)  
    return l  
  
print(f(3))  
print(f(4))
```

Step 1 — Understanding default argument behavior

- In Python, **default mutable arguments** (like lists or dicts) are created **only once** when the function is defined — not each time it's called.
- So, both function calls share the **same list object**.

Step 2 — Trace execution



Step 2 — Trace execution

First call: `f(3)`

- Default `l = [1, 2]`
- Append `3` → `l = [1, 2, 3]`
- ✓ Returns `[1, 2, 3]`

Second call: `f(4)`

- Same list reused → current `l = [1, 2, 3]`
- Append `4` → `l = [1, 2, 3, 4]`
- ✓ Returns `[1, 2, 3, 4]`

✓ Final Output:

```
csharp
```

[Copy code](#)

```
[1, 2, 3]  
[1, 2, 3, 4]
```

✓ Correct Option:

B → `[1, 2, 3]` `[1, 2, 3, 4]`

💡 Shortcut Tip for Exams:

Whenever you see a mutable default argument (like `list`, `dict`, or `set`):

→ Changes persist across function calls unless you explicitly reset it.

📄 👍 🗨️ 🔗 ↺ ⋮

#Q.

```
t = (1, [2, 3], (4, 5))
t[1].append(99)
print(t)
```

- A (1, [2, 3], (4, 5))
- B (1, [2, 3, 99], (4, 5))
- C Error
- D (1, 2, 3, 99, (4, 5))

Let's analyze this carefully 🧐

🌿 Code:

python

📄 Copy code

```
t = (1, [2, 3], (4, 5))
t[1].append(99)
print(t)
```

Step 1 — Key concept: tuples are immutable, but they can contain mutable objects (like lists).

- You cannot modify the tuple structure itself (e.g., `t[1] = something` → ❌ error).
- But you can modify the mutable object inside the tuple (like the list `[2, 3]`). ✅

Step 2 — `t[1]` refers to `[2, 3]`

Then:

python

📄 Copy code


```
t[1].append(99)
```



→ modifies that list to `[2, 3, 99]`.

Step 3 — Final tuple:

ini

 Copy code

```
t = (1, [2, 3, 99], (4, 5))
```

✓ Output:

CSS

 Copy code

```
(1, [2, 3, 99], (4, 5))
```

✓ Correct Option:


B → (1, [2, 3, 99], (4, 5))

💡 Quick Tip:

Tuples are immutable, but their **elements can be mutable**, so nested lists *can* be changed.

✓ Answer:

python

 Copy code

```
(1, [2, 3, 99], (4, 5))
```

Explanation:

Tuples are immutable, but they can contain mutable objects like lists.

Here, `t[1]` refers to the list `[2, 3]`, so `t[1].append(99)` modifies that list in place.


#Q.

```
d={True: "Yes", 1: "No", 1.0: "Maybe"}  
print(len(d), d[True])
```

- A 3 Yes
- B 2 No
- C 1 Maybe
- D Error

Let's analyze the question carefully 🧐

python

 Copy code


```
d = {True: "Yes", 1: "No", 1.0: "Maybe"}  
print(len(d), d[True])
```

Step 1: Understanding keys in Python dictionary

In Python, dictionary keys must be unique.

Also note that:


python

 Copy code

```
True == 1 == 1.0 # ✅ This is True
```

and their hash values are equal:

python

 Copy code

```
hash(True) == hash(1) == hash(1.0) # ✅ Same hash
```

That means all three keys (`True`, `1`, and `1.0`) are considered the same key.


Step 2: How Python handles duplicate keys



When a duplicate key appears, the last assigned value overwrites previous ones.

python

```
d = {True: "Yes", 1: "No", 1.0: "Maybe"}
```

 Copy code

After execution, the dictionary becomes:

python


```
d = {True: "Maybe"}
```

 Copy code

Step 3: Check length and value


python

```
len(d) # 1  
d[True] # "Maybe"
```

 Copy code

✅ **Output:**

1 Maybe

 Copy code

✅ **Correct Answer: C. 1 Maybe**

     ...

#Q.

```
s = set()
s.add((1,2))
s.add([3,4])
print(s)
```

- A `{(1,2), [3,4]}`
- B `{(1,2)}`
- C Error
- D `{(1,2), (3,4)}`

Let's analyze this Python question carefully 📌

python

 Copy code

```
s = set()
s.add([1, 2])
s.add([3, 4])
print(s)
```

Step 1: What is allowed inside a `set` ?


A Python `set` can only contain **hashable** (immutable) elements —
for example: `int`, `str`, `tuple`, etc.

But **lists are mutable and unhashable**, meaning you cannot add a list to a set.

Step 2: What happens when you run this?

When Python executes:

python

 Copy code

```
s.add([1, 2])
```



it raises:

```
bash
```

[Copy code](#)

```
TypeError: unhashable type: 'list'
```

So, the code will **not** run successfully at all.

✅ Correct Answer: C. Error

📄 👍 💬 ⬆️ ↺ ⋮

```
#Q. python
x = 1000
y = 1000
print(x == y, x is y)
```

- A True True
- B True False
- C False True
- D False False

Let's analyze this question carefully 🖱️

```
python
```

[Copy code](#)

```
x = 1000
y = 1000
print(x == y, x is y)
```

Step 1: `==` vs `is`




- `==` → compares **values** (are they equal?)
 - `is` → compares **identities** (are they the same object in memory?)
-

Step 2: Integer caching in Python

Python caches small integers between -5 and 256, meaning:

```
python



a = 100
b = 100
a is b  # True
```


 Copy code

But integers **outside this range** (like `1000`) are **not cached** by default —
each assignment like `x = 1000` and `y = 1000` creates **separate objects** in memory.

Step 3: Evaluate

```
python


x == y  #  True (values are equal)
x is y  #  False (different memory locations)
```

 Copy code

Output:

```
graphql

True False
```

 Copy code

Correct Answer: B. True False

     ...

□ ◀ ▶ ↺ ↻ ⋮

```
#Q. python
a = {1, 2, 3}
b = {2, 3, 4}
print(a - b, b - a)
```

- A {1} {4}
- B {2,3} {4}
- C {1,4} {}
- D Error

Let's analyze this Python question 📌

python

Copy code

```
a = {1, 2, 3}
b = {2, 3, 4}
print(a - b, b - a)
```

Step 1: Understand `set` difference (-)

- `a - b` means → elements in `a` not in `b`
- `b - a` means → elements in `b` not in `a`

Step 2: Compute manually

makefile

Copy code

```
a = {1, 2, 3}
b = {2, 3, 4}

a - b = {1}      # (2,3 removed)
```



```
a - b = {1}      # (2,3 removed)
b - a = {4}      # (2,3 removed)
```

[Copy code](#)

✓ Output:

```
{1} {4}
```

[Copy code](#)

✓ Correct Answer: A. {1} {4}

[Copy](#) [Like](#) [Dislike](#) [Share](#) [Refresh](#) [More](#)

#Q. Which of these cannot be a dict key?

- A int
- B tuple of ints
- C list of ints
- D str

Let's analyze the question 📌

Question:

Which of these cannot be a dict key?

Options:

- A. int
- B. tuple of ints
- C. list of ints
- D. str



✅ Rule:

In Python, dictionary keys must be immutable and hashable.

Type	Mutable?	Hashable?	Can be dict key?
int	❌ No	✅ Yes	✅ Yes
tuple	❌ No (if contains only immutable elements)	✅ Yes	✅ Yes
list	✅ Yes	❌ No	❌ No
str	❌ No	✅ Yes	✅ Yes

❌ Hence, list of ints cannot be a dict key.


✅ Correct Answer: C. list of ints

📄 👍 💬 ↗️ ↺️ ⋮

```
#Q. python
import copy
a=[[1,2], [3,4]]
b=copy.deepcopy(a)
b[0][0]=99
print(a[0][0],b[0][0])
```

Let's analyze the code carefully 📌

python

 Copy code


```
import copy
a = [[1, 2], [3, 4]]
b = copy.deepcopy(a)
b[0][0] = 99
print(a[0][0], b[0][0])
```

Step 1: `copy.deepcopy()`

`deepcopy()` creates a **completely independent clone** of the original object, including all nested lists.

So after copying:


lua

 Copy code

```
a = [[1, 2], [3, 4]]
b = [[1, 2], [3, 4]] # a completely separate copy
```

Step 2: Modify `b`


python

 Copy code

```
b[0][0] = 99
```

Now:


lua

 Copy code

```
b = [[99, 2], [3, 4]]
a = [[1, 2], [3, 4]]
```


Step 3: Print values

python

 Copy code

```
print(a[0][0], b[0][0])
```

Output:

 Copy code

```
1 99
```

✅ Final Answer: 1 99

     ...

#Q. Find the output of the following program:

```
li = ['a', 'b', 'c', 'd', 'e']  
print(li[10:])
```

A ['a', 'b', 'c', 'd', 'e']

B ['c', 'd', 'e']


C []

D ['a', 'b']



Let's analyze the question 📌

python


 Copy code

```
li = ['a', 'b', 'c', 'd', 'e']  
print(li[10:])
```

Step 1: Understanding list slicing

In Python, slicing syntax is:

python

 Copy code

```
list[start:end]
```

- `start` = index to begin (inclusive)
- `end` = index to stop (exclusive)
- If an index is **out of range**, **no error occurs** — it simply returns an **empty list**.

Step 2: Apply `li[10:]`

Here, the list `li` has length 5, and index `10` is **beyond the end**.

So the slice starts from an index that doesn't exist.

Result → an empty list `[]`

✓ Output: `[]`

✓ Correct Answer: C. `[]`

     ...

#Q. Find the output of the following program:


```
a = [1, 2, 3, 4]
b = a
c = a.copy()
d = a
a[0] = [5]
print(a, b, c, d)
```

- ☒ A. `[5, 2, 3, 4] [5, 2, 3, 4] [1, 2, 3, 4] [1, 2, 3, 4]`
- ☐ B. `[[5], 2, 3, 4] [[5], 2, 3, 4] [[5], 2, 3, 4] [1, 2, 3, 4]`
- ☐ C. `[5, 2, 3, 4] [5, 2, 3, 4] [5, 2, 3, 4] [1, 2, 3, 4]`
- ☐ D. `[[5], 2, 3, 4] [[5], 2, 3, 4] [1, 2, 3, 4] [[5], 2, 3, 4]`



Let's analyze the code carefully 📌

python

 Copy code


```
a = [1, 2, 3, 4]
b = a
c = a.copy()
d = a
a[0] = [5]
print(a, b, c, d)
```

Step 1: Understand the assignments

- `b = a` → `b` references the same list as `a`.
- `c = a.copy()` → `c` gets a **shallow copy** (a new independent list).
- `d = a` → again, `d` references the same list as `a`.

So before modification:


css

 Copy code

```
a → [1, 2, 3, 4]
b → same as a
c → separate copy [1, 2, 3, 4]
d → same as a
```

Step 2: Modify `a[0]`

python

 Copy code


```
a[0] = [5]
```

This changes only the first element of list `a` (and anything referencing it).

Now:


ini

```
a = [[5], 2, 3, 4]
b = [[5], 2, 3, 4] # same object as a
c = [1, 2, 3, 4]   # independent copy, unchanged
d = [[5], 2, 3, 4] # same object as a
```

 Copy code

✓ Final Output:

CSS

 Copy code

```
[[5], 2, 3, 4] [[5], 2, 3, 4] [1, 2, 3, 4] [[5], 2, 3, 4]
```

✓ Correct Answer: B


     ...

```
#Q. Find the output of the following program:
li = [1, 1.33, 'GFG', 0, 'NO', None, 'G', True]
val1, val2 = 0, ""
for x in li:
    if(type(x) == int or type(x) == float):
        val1 += x
    elif(type(x) == str):
        val2 += x
    else:
        break
print(val1, val2)
```

☒ A 2 GFGNO
☐ B 2.33 GFGNOG
☐ C 2.33 GFGNONoneGTrue
☐ D 2.33 GFGNO

Let's analyze the question carefully 🐼


python

 Copy code

```
li = [1, 1.33, 'GFG', 0, 'NO', None, 'G', True]
val1, val2 = 0, ""
for x in li:
    if (type(x) == int or type(x) == float):
        val1 += x
    elif (type(x) == str):
        val2 += x
    else:
        break
print(val1, val2)
```

Step 1: Initial values

ini


 Copy code

```
val1 = 0
val2 = ""
```

Step 2: Iterate through `li`

x	type(x)	Action	val1	val2
1	int	val1 += 1	1	""
1.33	float	val1 += 1.33	2.33	""
'GFG'	str	val2 += 'GFG'	2.33	"GFG"
0	int	val1 += 0	2.33	"GFG"
'NO'	str	val2 += 'NO'	2.33	"GFGNO"
None	NoneType	triggers else: break → loop stops here	2.33	"GFGNO"

✓ Final Output:

 Copy code

2.33 GFGNO

✓ Correct Answer: D. 2.33 GFGNO

     ...

#Q. Find the output of the following program:

```
a = []  
a.append([1, [2, 3], 4])  
a.extend([7, 8, 9])  
print(a[0][1][1] + a[2])
```

A Type Error


B 12

C 11

D 38

Let's analyze the given Python code step-by-step 📌


python

 Copy code

```
a = []  
a.append([1, [2, 3], 4])  
a.extend([7, 8, 9])  
print(a[0][1][1] + a[2])
```

Step 1: After `append`

python

 Copy code

```
a.append([1, [2, 3], 4])
```

Now `a` becomes:

lua


 Copy code

```
a = [[1, [2, 3], 4]]
```

👉 The entire list `[1, [2, 3], 4]` is added as a **single element**.

Step 2: After `extend`


python

 Copy code

```
a.extend([7, 8, 9])
```

The `extend()` method adds each element individually:


ini

 Copy code

```
a = [[1, [2, 3], 4], 7, 8, 9]
```

Step 3: Evaluate the expression


python

 Copy code

```
a[0][1][1] + a[2]
```

Step 3: Evaluate the expression

python


 Copy code

```
a[0][1][1] + a[2]
```

Let's break it down:


- `a[0]` → `[1, [2, 3], 4]`
- `a[0][1]` → `[2, 3]`
- `a[0][1][1]` → `3`
- `a[2]` → `8`

So:

 Copy code

```
a[0][1][1] + a[2] = 3 + 8 = 11
```

✅ Final Output:

 Copy code

```
11
```

✅ Correct Answer: C. 11


     ...

```
#Q. What is the output of the following program?
li = [3, 1, 2, 4]
tup = ('A', 'b', 'c', 'd')
li.sort()
counter = 0
for x in tup:
    li[counter] += int(x)
    counter += 1
print(li)
```

☐ A [66, 97, 99, 101]
☐ B [66, 100, 102, 104]
☐ C [66, 67, 68, 69]
☐ D Value Error

Let's analyze this step by step 🖱️


python

 Copy code

```
li = [3, 1, 2, 4]
tup = ('A', 'b', 'c', 'd')
li.sort()
counter = 0
for x in tup:
    li[counter] += int(x)
    counter += 1
print(li)
```

Step 1: Sort the list


python

 Copy code

```
li.sort()
```

Now:

ini

 Copy code

```
li = [1, 2, 3, 4]
```

Step 2: Loop through `tup`

```
tup = ('A', 'b', 'c', 'd')
```


Now here's the key:

`int(x)` will throw an error — because `'A'`, `'b'`, `'c'`, `'d'` are not numeric strings.

So Python cannot convert `'A'` to an integer.

If this code actually runs, you'll get:


csharp

 Copy code

```
ValueError: invalid literal for int() with base 10: 'A'
```

✅ Final Output:

javascript

 Copy code

```
Value Error
```

✅ Correct Answer: D. Value Error


     ...

D. ValueError

After `li.sort()`, `li = [1, 2, 3, 4]`.

In the loop, the first iteration tries `int('A')`, which is invalid (non-numeric string), so Python raises:

csharp

 Copy code

```
ValueError: invalid literal for int() with base 10: 'A'
```

< 2/2 >      ...

#Q. Find the output of the following program:

```

a = ['Learn', 'Quiz', 'Practice', 'Contribute']
b = a + ['X', 'b', 'c', 'd']
c = a[:3]
b[0] = 'Code'
c[1] = 'Mcq'
count = 0
for c in (a, b, c):
    if c[0] == 'Code':
        count += 1
    if c[1] == 'Mcq':
        count += 10
print(count)

```

Options: A: 4, B: 5, C: 11, D: 12

Let's analyze this Python code step by step 🧐

Code:

```

python
Copy code

a = ['Learn', 'Quiz', 'Practice', 'Contribute']
b = a
c = a[:3]
b[0] = 'Code'
c[1] = 'Mcq'
count = 0

for c in (a, b, c):
    if c[0] == 'Code':
        count += 1
    if c[1] == 'Mcq':
        count += 10


print(count)

```

Step 1: Assignments

Step 1: Assignments

python


 Copy code

```
a = ['Learn', 'Quiz', 'Practice', 'Contribute']  
b = a  
c = a[:]
```

- `b = a` → both `a` and `b` refer to the same list.
- `c = a[:]` → `c` is a shallow copy (a new list).

Step 2: Mutations

python


 Copy code

```
b[0] = 'Code'
```

This changes `a`'s first element because `a` and `b` point to the same list.

Now:


ini

 Copy code

```
a = ['Code', 'Quiz', 'Practice', 'Contribute']  
b = ['Code', 'Quiz', 'Practice', 'Contribute']  
c = ['Learn', 'Quiz', 'Practice', 'Contribute']
```

Then:

python


 Copy code

```
c[1] = 'Mcq'
```

This affects only `c`, not `a` or `b`.

Final state:

ini

 Copy code

```
a = ['Code', 'Quiz', 'Practice', 'Contribute']  
b = ['Code', 'Quiz', 'Practice', 'Contribute']  
c = ['Learn', 'Mcq', 'Practice', 'Contribute']
```

Step 3: Loop

The loop iterates over (a, b, c) — i.e., 3 lists.

Iteration 1 → list a

- `c[0] == 'Code' → ✓ → count = 1`
- `c[1] == 'Mcq' → ✗ → no change`
(count = 1)

Iteration 2 → list b

- `c[0] == 'Code' → ✓ → count = 2`
- `c[1] == 'Mcq' → ✗ → no change`
(count = 2)

Iteration 3 → list c

- `c[0] == 'Code' → ✗`
- `c[1] == 'Mcq' → ✓ → count += 10 → count = 12`

✓ Final Output:

Copy code

12

Correct Option: D (12)

✓ The output of the given code is 12

Here's why step by step:

1. Initial list

python


Copy code

```
a = ['Learn', 'Quiz', 'Practice', 'Contribute']  
b = a  
c = a[:]
```

b refers to the same list as a, while c is a copy.

2. Mutations


python

 Copy code

```
b[0] = 'Code'      # changes a[0] also
c[1] = 'Mcq'
```

Now:


ini

 Copy code

```
a = ['Code', 'Quiz', 'Practice', 'Contribute']
b = ['Code', 'Quiz', 'Practice', 'Contribute']
c = ['Learn', 'Mcq', 'Practice', 'Contribute']
```

3. Loop

python

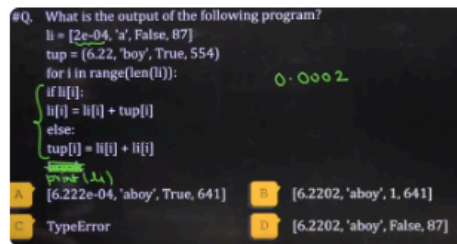
 Copy code

```
for c in (a, b, c):
    if c[0] == 'Code': count += 1
    if c[1] == 'Mcq':  count += 10
```

- For **a**: first condition true → +1
 - For **b**: first condition true → +1
 - For **c**: second condition true → +10
- Total = 12

 Final Answer: D) 12

     ...



Let's go step by step through this program 📌

Code:

python

📄 Copy code

```
li = [2e-04, 'a', False, 87]
tup = (6.22, 'boy', True, 554)

for i in range(len(li)):
    if li[i]:
        li[i] = li[i] + tup[i]
    else:
        tup[i] = li[i] + li[i]

print(li)
```

Step 1: Initial values

ini

📄 Copy code

```
li = [0.0002, 'a', False, 87]
tup = (6.22, 'boy', True, 554)
```

Step 2: Loop iteration by index

i = 0

li[0] = 0.0002 → truthy ✓

→ li[0] = li[0] + tup[0] = 0.0002 + 6.22 = 6.2202

Now:

ini

Copy code

```
li = [6.2202, 'a', False, 87]
```

i = 1

li[1] = 'a' → truthy ✓

→ li[1] = li[1] + tup[1] = 'a' + 'boy' = 'aboy'

Now:

ini

Copy code

```
li = [6.2202, 'aboy', False, 87]
```

i = 2

li[2] = False → falsy ✗

→ tup[i] = li[i] + li[i] = False + False = 0

⚠ But tuples are immutable, so this assignment

tup[i] = ... will cause a **TypeError** ✗

✓ Final Output:

javascript

Copy code

```
TypeError
```

Correct Option → C) TypeError

S

#Q. Find the output of the following program:

```
a = {}
a[1] = 1
a['1'] = 2
a[1] = a[1] + 1
count = 0
for i in a:
    count += a[i]
    print(count)
```

A. 2

B. 4


C. 1

D. Error

Let's go through this Python question step by step 🐡

Code:

python

 Copy code

```
a = {}
a[1] = 1
a['1'] = 2
a[1] = a[1] + 1
count = 0
for i in a:
    count += a[i]
print(count)
```

Step 1: Dictionary assignments

Initially:

ini

 Copy code

```
a = {}
```

Then line by line:

1 a[1] = 1

→ a = {1: 1}

2 a['1'] = 2



2 `a['1'] = 2`

→ `a = {1: 1, '1': 2}`


(Note: `'1'` (string) and `1` (integer) are different keys in Python)

3 `a[1] = a[1] + 1`

→ `a[1] = 1 + 1 = 2`

Now:


```
ini
```

 Copy code

```
a = {1: 2, '1': 2}
```

Step 2: Loop over dictionary

```
python
```


 Copy code

```
count = 0
for i in a:
    count += a[i]
```

- `i = 1` → `count += 2` → `count = 2`
- `i = '1'` → `count += 2` → `count = 4`

✓ Final Output:

```
4
```

 Copy code

Correct answer → B) 4

     ...