

# L-5

# Objects, Collections, and Files

ENGG1330  
Computer Programming I

# L4 Survey Results



421 submissions · 403–404 answers to these questions

# List Assignment

PY main.py

```
1 inventory = ['key']
2 backup = inventory
3 inventory[0] = 'map'
4 print(backup)
```

[Download: lec\\_05\\_inventory\\_prediction.py](#)

Will backup contain  
key or map?

Predict first, then follow the references

# List Assignment

PY main.py

```
1 inventory = ['key']
2 backup = inventory
3 inventory[0] = 'map'
4 print(backup)
```

Download: lec\_05\_inventory\_prediction.py

Names

Objects

inventory



list  
['key']

inventory refers to this list

# List Assignment

PY main.py

```
1 inventory = ['key']
2 backup = inventory
3 inventory[0] = 'map'
4 print(backup)
```

Download: lec\_05\_inventory\_prediction.py

Names

Objects

inventory

backup

list

['key']

Assignment adds a reference to the same list

# List Assignment

PY main.py

```
1 inventory = ['key']
2 backup = inventory
3 inventory[0] = 'map'
4 print(backup)
```

Download: lec\_05\_inventory\_prediction.py

Names

Objects

inventory

backup

list  
['map']

```
graph LR
    inventory --> list
    backup --> list
    subgraph Objects
        list["list ['map']"]
    end
```

The list changes; both names still refer to it

# Value and Identity

PY main.py

```
1 x = ['key']  
2 y = ['key']  
3 print(x == y)
```

[Download: lec\\_05\\_equality\\_intro.py](#)

`==` compares values

These two lists contain the same item

# Value and Identity

PY main.py

```
1 x = ['key']
2 y = ['key']
3 print(id(x))
4 print(id(y))
```

[Download: lec\\_05\\_id\\_intro.py](#)

`id(x)` identifies the object

Equal values, different objects: their IDs differ

# Value and Identity

PY main.py

```
1 x = ['key']  
2 y = x  
3 print(x is y)
```

[Download: lec\\_05\\_is\\_intro.py](#)

`is` checks for the same object

Both names refer to the same list

# Value and Identity

PY main.py

```
1 x = ['key']  
2 y = x  
3 print(x is y)
```

[Download: lec\\_05\\_is\\_intro.py](#)

`is` checks for the same object

Both names refer to the same list

Diagram labels 01, 02, ... stand for objects

They are not the numbers returned by `id()`

# Comparing Objects

PY main.py

```
1 first = [10, 20]
2 second = [10, 20]
3 alias = first
4 print(first == second)
5 print(first is second)
6 print(first is alias)
```

[Download: lec\\_05\\_equal\\_objects.py](#)

Names

Objects

first

alias

second

O1 · list  
[10, 20]

O2 · list  
[10, 20]

Two list objects; one additional name

O1, O2, ...: symbolic object IDs

# Comparing Objects

PY main.py

```
1 first = [10, 20]
2 second = [10, 20]
3 alias = first
4 print(first == second)
5 print(first is second)
6 print(first is alias)
```

[Download: lec\\_05\\_equal\\_objects.py](#)

Names

Objects

first

alias

second

O1 · list  
[10, 20]

O2 · list  
[10, 20]

`==` compares these lists by contents: True

O1, O2, ...: symbolic object IDs

# Comparing Objects

PY main.py

```
1 first = [10, 20]
2 second = [10, 20]
3 alias = first
4 print(first == second)
5 print(first is second)
6 print(first is alias)
```

[Download: lec\\_05\\_equal\\_objects.py](#)

Names

Objects

first

alias

second

O1 · list  
[10, 20]

O2 · list  
[10, 20]

first is second: False

O1, O2, ...: symbolic object IDs

# Comparing Objects

PY main.py

```
1 first = [10, 20]
2 second = [10, 20]
3 alias = first
4 print(first == second)
5 print(first is second)
6 print(first is alias)
```

[Download: lec\\_05\\_equal\\_objects.py](#)

Names

Objects

first

alias

second

O1 · list  
[10, 20]

O2 · list  
[10, 20]

first is alias: True

O1, O2, ...: symbolic object IDs

# Shared References

PY main.py

```
1 first = ['key']
2 alias = first
3 before = id(first)
4 alias[0] = 'map'
5 print(first, alias)
6 print(id(first) == before)
```

[Download: lec\\_05\\_shared\\_list.py](#)

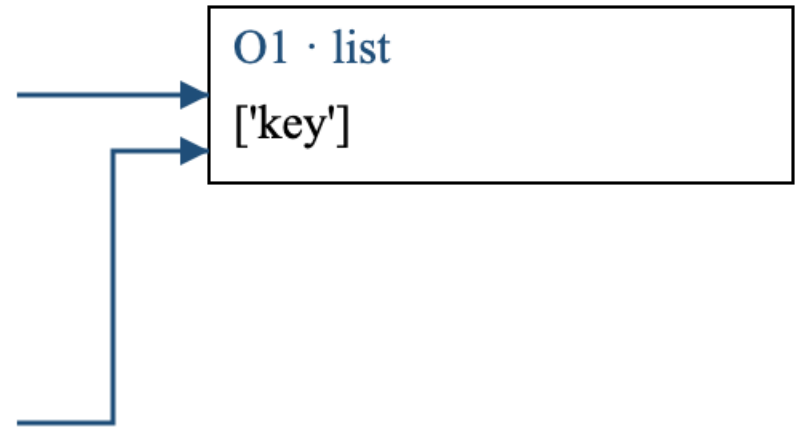
Names

Objects

first

alias

O1 · list  
['key']



```
graph LR
    first --> O1
    alias --> O1
    subgraph Objects
        O1["O1 · list<br/>['key']"]
    end
```

Mutation changes an existing object

O1, O2, ...: symbolic object IDs

# Shared References

PY main.py

```
1 first = ['key']
2 alias = first
3 before = id(first)
4 alias[0] = 'map'
5 print(first, alias)
6 print(id(first) == before)
```

[Download: lec\\_05\\_shared\\_list.py](#)

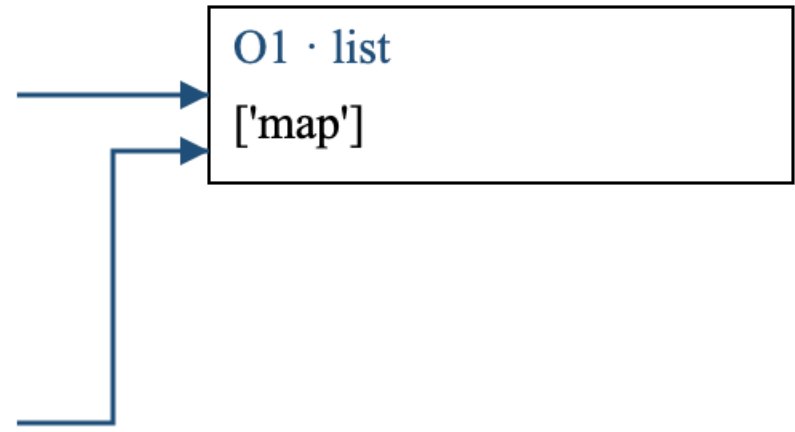
Names

Objects

first

alias

O1 · list  
['map']



```
graph LR
    first --> O1
    alias --> O1
    subgraph Objects
        O1["O1 · list<br/>['map']"]
    end
```

The same object remains: `id(first) == before`

O1, O2, ...: symbolic object IDs

# Rebinding

PY main.py

```
1 first = ['key']
2 alias = first
3 first = ['map']
4 print(first)
5 print(alias)
6 print(first is alias)
```

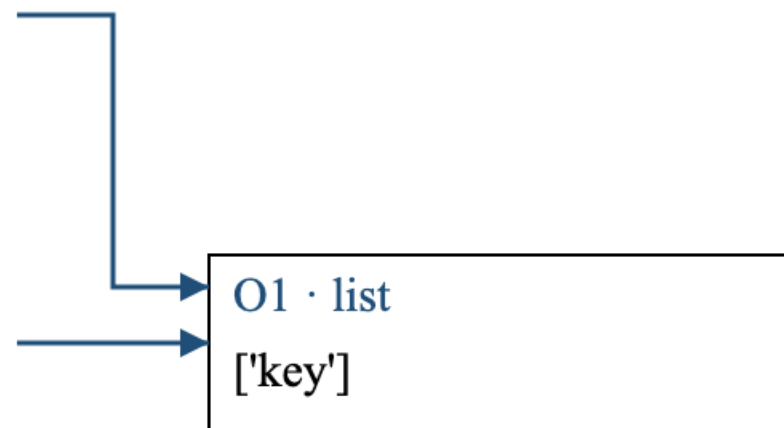
Download: [lec\\_05\\_rebind\\_list.py](#)

Names

Objects

first

alias



Rebinding changes which object a name refers to

O1, O2, ...: symbolic object IDs

# Rebinding

PY main.py

```
1 first = ['key']
2 alias = first
3 first = ['map']
4 print(first)
5 print(alias)
6 print(first is alias)
```

Download: [lec\\_05\\_rebind\\_list.py](#)

Names

Objects

first



O2 · list  
['map']

alias



O1 · list  
['key']

Only first moves to the new object O2

O1, O2, ...: symbolic object IDs

# String Immutability

PY main.py

```
1 word = 'door'
2 other = word
3 word.upper()
4 print(word, other)
5 word = word.upper()
6 print(word, other)
```

[Download: lec\\_05\\_string\\_return\\_check.py](#)

Predict both output lines  
Which statement changes  
what word refers to?

Explain your prediction before Run

O1, O2, ...: symbolic object IDs

# String Immutability

PY main.py

```
1 word = 'door'
2 other = word
3 word.upper()
4 print(word, other)
5 word = word.upper()
6 print(word, other)
```

[Download: lec\\_05\\_string\\_return\\_check.py](#)

O1, O2, ...: symbolic object IDs

# String Immutability

PY main.py

```
1 word = 'door'
2 other = word
3 word.upper()
4 print(word, other)
5 word = word.upper()
6 print(word, other)
```

[Download: lec\\_05\\_string\\_return\\_check.py](#)

O1, O2, ...: symbolic object IDs

# String Immutability

PY main.py

```
1 word = 'door'
2 other = word
3 word.upper()
4 print(word, other)
5 word = word.upper()
6 print(word, other)
```

[Download: lec\\_05\\_string\\_return\\_check.py](#)

O1, O2, ...: symbolic object IDs

# Tuples

ENGG1330  
Computer Programming I

# Single-Item Tuples

PY main.py

```
1 number = (0)
2 packed = (0,)
3 also_packed = 0,
4 print(type(number))
5 print(type(packed))
6 print(packed, also_packed)
```

Download: [lec\\_05\\_one\\_item\\_tuple.py](#)

# Tuple Concatenation

PY main.py

```
1 values = (1, 2)
2 previous = values
3 values += (3,)
4 print(previous, values)
5 print(values is previous)
```

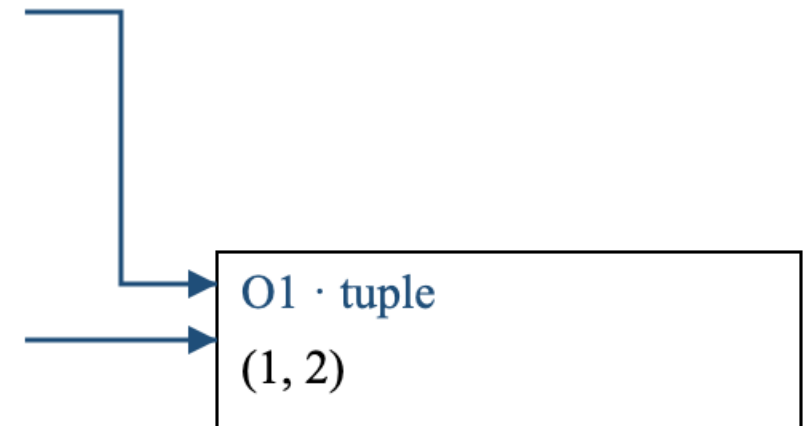
Download: lec\_05\_tuple\_rebinding.py

Names

Objects

values

previous



Two names refer to the same tuple

O1, O2, ...: symbolic object IDs

# Tuple Concatenation

PY main.py

```
1 values = (1, 2)
2 previous = values
3 values += (3,)
4 print(previous, values)
5 print(values is previous)
```

Download: [lec\\_05\\_tuple\\_rebinding.py](#)

Names

Objects

values



O2 · tuple  
(1, 2, 3)

previous



O1 · tuple  
(1, 2)

values moves to O2; previous keeps O1

O1, O2, ...: symbolic object IDs

# List Augmentation

PY main.py

```
1 values = [1, 2]
2 previous = values
3 values += [3]
4 print(previous, values)
5 print(values is previous)
```

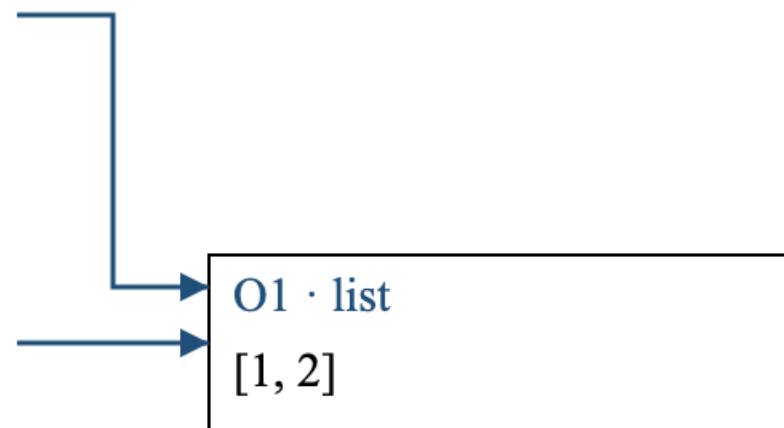
Download: [lec\\_05\\_list\\_augmentation.py](#)

Names

Objects

values

previous



Two names refer to the same list

O1, O2, ...: symbolic object IDs

# List Augmentation

PY main.py

```
1 values = [1, 2]
2 previous = values
3 values += [3]
4 print(previous, values)
5 print(values is previous)
```

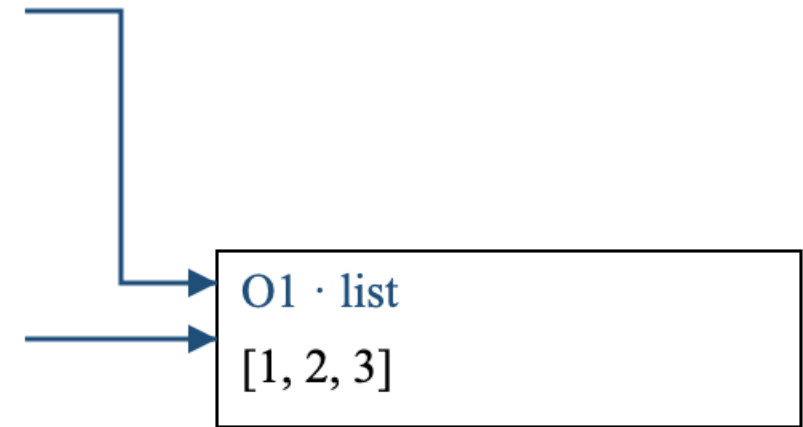
Download: [lec\\_05\\_list\\_augmentation.py](#)

Names

Objects

values

previous



O1 changes; both references stay

O1, O2, ...: symbolic object IDs

# Lists and Tuples: Check

PY main.py

```
1 a = [1]
2 b = a
3 x = (1,)
4 y = x
5 a += [2]
6 x += (2,)
7 print(b)
8 print(y)
```

Predict b and y  
Why does the same +=  
give different results?

Explain using objects and references

O1, O2, ...: symbolic object IDs

# Lists and Tuples: Check

PY main.py

```
1 a = [1]
2 b = a
3 x = (1,)
4 y = x
5 a += [2]
6 x += (2,)
7 print(b)
8 print(y)
```

# Lists and Tuples: Check

PY main.py

```
1 a = [1]
2 b = a
3 x = (1,)
4 y = x
5 a += [2]
6 x += (2,)
7 print(b)
8 print(y)
```

# Lists and Tuples: Check

PY main.py

```
1 a = [1]
2 b = a
3 x = (1,)
4 y = x
5 a += [2]
6 x += (2,)
7 print(b)
8 print(y)
```

# Function Returns

PY main.py

```
1 def show_product(x, y):  
2     print(x * y)  
3 def multiply(x, y):  
4     return x * y  
5 result = show_product(3, 4)  
6 print(result)  
7 result = multiply(3, 4)  
8 print(result)
```

print shows output

return sends a value to the caller

# Function Returns

PY main.py

```
1 def show_product(x, y):  
2     print(x * y)  
3 def multiply(x, y):  
4     return x * y  
5 result = show_product(3, 4)  
6 print(result)  
7 result = multiply(3, 4)  
8 print(result)
```

| After show_product(3, 4) | Value |
|--------------------------|-------|
| Output on screen         | 12    |
| result                   | None  |

There is no return in this function

# Function Returns

PY main.py

```
1 def show_product(x, y):  
2     print(x * y)  
3 def multiply(x, y):  
4     return x * y  
5 result = show_product(3, 4)  
6 print(result)  
7 result = multiply(3, 4)  
8 print(result)
```

print(result) displays None

The screen now shows 12, then None

# Function Returns

PY main.py

```
1 def show_product(x, y):  
2     print(x * y)  
3 def multiply(x, y):  
4     return x * y  
5 result = show_product(3, 4)  
6 print(result)  
7 result = multiply(3, 4)  
8 print(result)
```

| After multiply(3, 4) | Value |
|----------------------|-------|
| Returned value       | 12    |
| result               | 12    |

The final print displays 12

# Packing and Unpacking

PY main.py

```
1 point = (1, 2, 3)
2 x, y, z = point
3 print(x, y, z)
```

[Download: lec\\_05\\_packing\\_intro.py](#)

Packing groups values in one tuple

| Tuple Item 0 | Tuple Item 1 | Tuple Item 2 |
|--------------|--------------|--------------|
| 1            | 2            | 3            |

# Packing and Unpacking

PY main.py

```
1 point = (1, 2, 3)
2 x, y, z = point
3 print(x, y, z)
```

[Download: lec\\_05\\_packing\\_intro.py](#)

Unpacking assigns items to names, in order

| <b>x</b> | <b>y</b> | <b>z</b> |
|----------|----------|----------|
| 1        | 2        | 3        |

# Packing and Unpacking

PY main.py

```
1 point = (1, 2, 3)
2 x, y, z = point
3 print(x, y, z)
```

[Download: lec\\_05\\_packing\\_intro.py](#)

Use one target per item

() is an empty tuple

# Packing and Unpacking

PY main.py

```
1 point = (1, 2, 3)
2 x, y, z = point
3 print(x, y, z)
```

[Download: lec\\_05\\_packing\\_intro.py](#)

An iterable supplies items one at a time

Tuples, strings, and ranges are iterable

# Sequence Unpacking

PY main.py

```
1 x, y, z = (1, 2, 3)
2 a, b, c = range(3)
3 first, second = 'Hi'
4 print(x, y, z)
5 print(a, b, c)
6 print(first, second)
```

[Download: lec\\_05\\_unpack\\_sequences.py](#)

# Returning Multiple Values

PY main.py

```
1 def quotient_remainder(a, b):  
2     return a // b, a % b  
3 result = quotient_remainder(17, 5)  
4 print(result)  
5 quotient, remainder = result  
6 print(quotient, remainder)
```

[Download: lec\\_05\\_two\\_results.py](#)

# Unpacking Errors

PY main.py

```
1 x, y = (1, 2, 3)
2 print(x, y)
```

[Download: lec\\_05\\_unpack\\_error.py](#)

# Unpacking in Loops

PY main.py

```
1 rows = ()
2 for i in range(3):
3     rows += ((i, i * 2, i * 3),)
4 print(rows)
5 for x, y, z in rows:
6     print(x, y, z)
```

# Record Summary: Plan

PY main.py

```
1 values = (2, 5, 4)
2 names = ('Ada', 'Ben')
3 print(min(values))
4 print(max(values))
5 print(len(names))
```

Download: [lec\\_05\\_summary\\_plan.py](#)

min finds the smallest value; max finds the largest

| Value | Name |
|-------|------|
| 2     | Ada  |
| 5     | Ben  |
| 4     | Ada  |

# Record Summary: Plan

PY main.py

```
1 values = (2, 5, 4)
2 names = ('Ada', 'Ben')
3 print(min(values))
4 print(max(values))
5 print(len(names))
```

Download: [lec\\_05\\_summary\\_plan.py](#)

Collect every value, but each name only once

Three records give two distinct names

# Record Summary: Plan

PY main.py

```
1 values = (2, 5, 4)
2 names = ('Ada', 'Ben')
3 print(min(values))
4 print(max(values))
5 print(len(names))
```

Download: [lec\\_05\\_summary\\_plan.py](#)

The three summary results

| Minimum | Maximum | Names |
|---------|---------|-------|
| 2       | 5       | 2     |

Assume at least one record

# Record Summary: Code

PY main.py

```
1 def summarize(records):
2     values, names = (), ()
3     for value, name in records:
4         values += (value,)
5         if name not in names:
6             names += (name,)
7     return min(values), max(values), len(names)
8 records = ((2, 'Ada'), (5, 'Ben'), (4, 'Ada'))
9 print(summarize(records))
```

# Record Summary: Code

PY main.py

```
1 def summarize(records):
2     values, names = (), ()
3     for value, name in records:
4         values += (value,)
5         if name not in names:
6             names += (name,)
7     return min(values), max(values), len(names)
8 records = ((2, 'Ada'), (5, 'Ben'), (4, 'Ada'))
9 print(summarize(records))
```

# Record Summary: Code

PY main.py

```
1 def summarize(records):
2     values, names = (), ()
3     for value, name in records:
4         values += (value,)
5         if name not in names:
6             names += (name,)
7     return min(values), max(values), len(names)
8 records = ((2, 'Ada'), (5, 'Ben'), (4, 'Ada'))
9 print(summarize(records))
```

# Record Summary: Code

**PY** main.py

```
1 def summarize(records):
2     values, names = (), ()
3     for value, name in records:
4         values += (value,)
5         if name not in names:
6             names += (name,)
7     return min(values), max(values), len(names)
8 records = ((2, 'Ada'), (5, 'Ben'), (4, 'Ada'))
9 print(summarize(records))
```

# Unpacking Arguments

PY main.py

```
1 def show(x, y, z):  
2     print(x, y, z)  
3 point = (1, 2, 3)  
4 show(point)
```

[Download: lec\\_05\\_tuple\\_argument\\_error.py](#)

show(point) passes one tuple

The function needs three arguments

# Unpacking Arguments

PY main.py

```
1 def show(x, y, z):  
2     print(x, y, z)  
3 point = (1, 2, 3)  
4 show(1, 2, 3)  
5 show(point[0], point[1], point[2])  
6 show(*point)
```

\*point unpacks its items into separate arguments

All three calls supply the same values

# Point Displacement

PY main.py

```
1 def difference(x1, y1, z1, x2, y2, z2):  
2     return x2 - x1, y2 - y1, z2 - z1  
3 start = (1, 2, 3)  
4 end = (4, 6, 8)  
5 print(difference(*start, *end))
```

[Download: lec\\_05\\_vector\\_difference.py](#)

# Swapping Values

PY main.py

```
1 x, y = 3, 8
2 temporary = x
3 x = y
4 y = temporary
5 print(x, y)
6 x, y = y, x
7 print(x, y)
```

[Download: lec\\_05\\_swap\\_values.py](#)

# Tuple Immutability

PY main.py

```
1 pair = ([1, 2], [3, 4])  
2 pair[0] = [9, 9]  
3 print(pair)
```

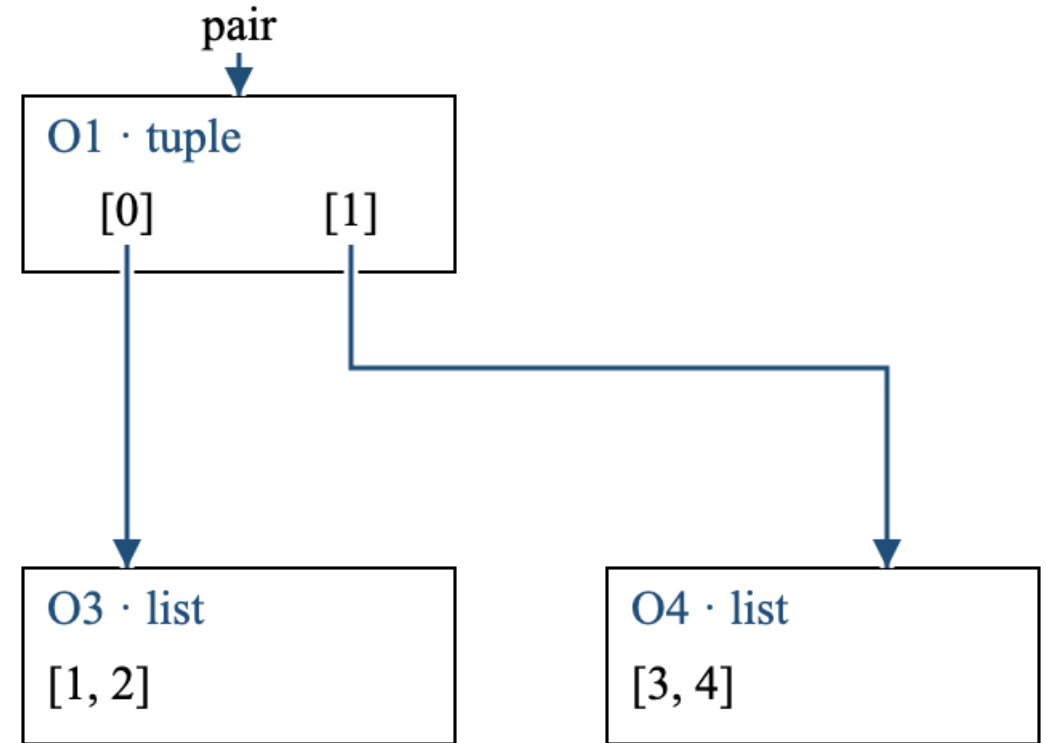
[Download: lec\\_05\\_tuple\\_assignment\\_error.py](#)

# Lists Inside Tuples

PY main.py

```
1 pair = ([1, 2], [3, 4])
2 pair[0][0] = 9
3 print(pair)
```

[Download: lec\\_05\\_tuple\\_contains\\_list.py](#)



Tuple slots refer to the inner lists

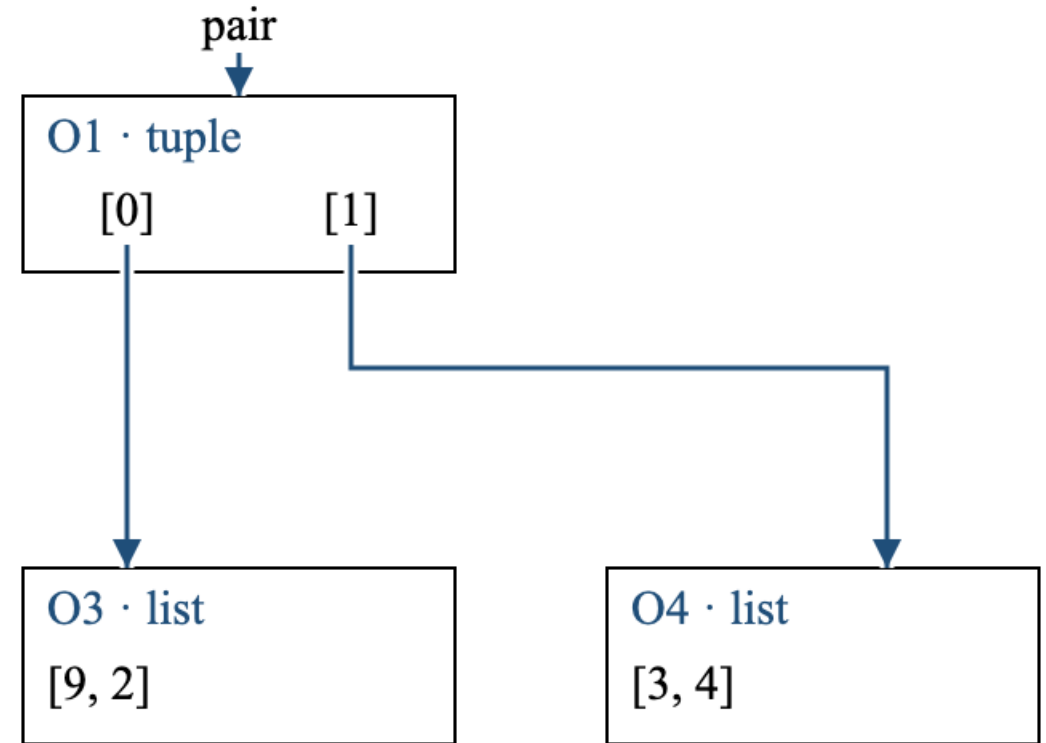
O1, O2, ...: symbolic object IDs

# Lists Inside Tuples

PY main.py

```
1 pair = ([1, 2], [3, 4])
2 pair[0][0] = 9
3 print(pair)
```

[Download: lec\\_05\\_tuple\\_contains\\_list.py](#)



The inner list changes; tuple slots stay

O1, O2, ...: symbolic object IDs

# List Operations

ENGG1330  
Computer Programming I

# Appending Items

PY main.py

```
1 values = [1, 2]
2 values.append([3, 4])
3 print(values)
4 print(len(values))
```

[Download: lec\\_05\\_append\\_item.py](#)

values · O1 · list

|   |   |
|---|---|
| 1 | 2 |
| 0 | 1 |

append adds one item

Here that item is a list

Start with two outer items

O1, O2, ...: symbolic object IDs

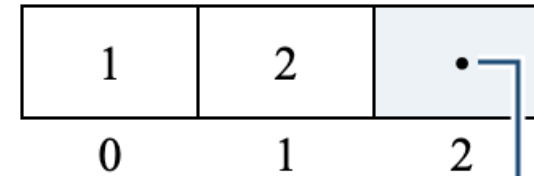
# Appending Items

PY main.py

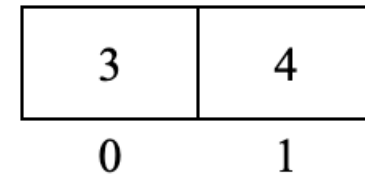
```
1 values = [1, 2]
2 values.append([3, 4])
3 print(values)
4 print(len(values))
```

[Download: lec\\_05\\_append\\_item.py](#)

values · O1 · list



O2 · list



One new outer slot: len(values) is 3

O1, O2, ...: symbolic object IDs

# Extending Lists

PY main.py

```
1 values = [1, 2]
2 values.extend([3, 4])
3 print(values)
4 print(len(values))
```

[Download: lec\\_05\\_extend\\_items.py](#)

values · O1 · list

|   |   |
|---|---|
| 1 | 2 |
| 0 | 1 |

extend adds each supplied item

Start with the same two outer items

O1, O2, ...: symbolic object IDs

# Extending Lists

PY main.py

```
1 values = [1, 2]
2 values.extend([3, 4])
3 print(values)
4 print(len(values))
```

[Download: lec\\_05\\_extend\\_items.py](#)

values · O1 · list

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 |

Two new outer slots

len(values) is 4; no inner list is added

O1, O2, ...: symbolic object IDs

# List Concatenation

PY main.py

```
1 values = [1, 2]
2 alias = values
3 values = values + [3]
4 print(values, alias)
5 print(values is alias)
```

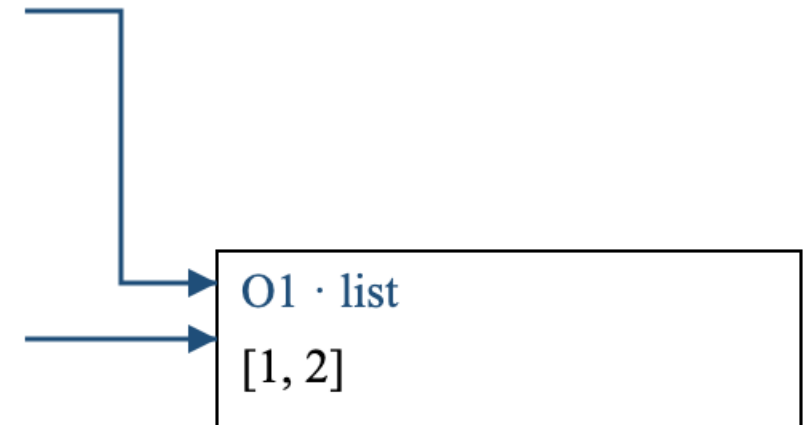
Download: [lec\\_05\\_list\\_addition.py](#)

Names

Objects

values

alias



Both names initially reach O1

O1, O2, ...: symbolic object IDs

# List Concatenation

PY main.py

```
1 values = [1, 2]
2 alias = values
3 values = values + [3]
4 print(values, alias)
5 print(values is alias)
```

Download: [lec\\_05\\_list\\_addition.py](#)

Names

Objects

values



O2 · list  
[1, 2, 3]

alias



O1 · list  
[1, 2]

+ creates O2; assignment redirects values

O1, O2, ...: symbolic object IDs

# Deleting and Removing

PY main.py

```
1 values = [4, 7, 4, 9]
2 del values[1]
3 print(values)
4 values.remove(4)
5 print(values)
```

Download: lec\_05\_remove\_items.py

values · O1 · list

|   |   |   |   |
|---|---|---|---|
| 4 | 7 | 4 | 9 |
| 0 | 1 | 2 | 3 |

del values[1] selects index 1

O1, O2, ...: symbolic object IDs

# Deleting and Removing

PY main.py

```
1 values = [4, 7, 4, 9]
2 del values[1]
3 print(values)
4 values.remove(4)
5 print(values)
```

[Download: lec\\_05\\_remove\\_items.py](#)

values · O1 · list

|   |   |   |
|---|---|---|
| 4 | 4 | 9 |
| 0 | 1 | 2 |

After deletion, later items shift left

O1, O2, ...: symbolic object IDs

# Deleting and Removing

PY main.py

```
1 values = [4, 7, 4, 9]
2 del values[1]
3 print(values)
4 values.remove(4)
5 print(values)
```

[Download: lec\\_05\\_remove\\_items.py](#)

values · O1 · list

|   |   |   |
|---|---|---|
| 4 | 4 | 9 |
| 0 | 1 | 2 |

remove(4) selects the first matching value

O1, O2, ...: symbolic object IDs

# Deleting and Removing

PY main.py

```
1 values = [4, 7, 4, 9]
2 del values[1]
3 print(values)
4 values.remove(4)
5 print(values)
```

[Download: lec\\_05\\_remove\\_items.py](#)

values · O1 · list

|   |   |
|---|---|
| 4 | 9 |
| 0 | 1 |

Only the first 4 is removed

O1, O2, ...: symbolic object IDs

# Sorting in Place

PY main.py

```
1 values = [3, 1, 2]
2 alias = values
3 result = values.sort()
4 print(values, alias)
5 print(result)
```

[Download: lec\\_05\\_sort\\_in\\_place.py](#)

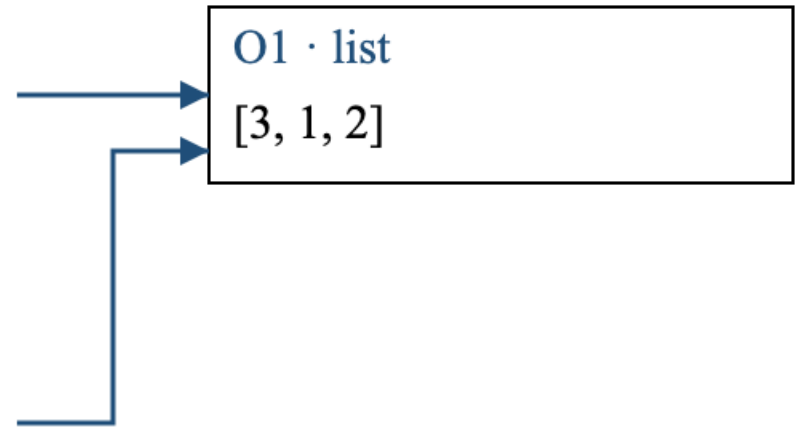
Names

Objects

values

alias

O1 · list  
[3, 1, 2]



```
graph LR
    values --> O1
    alias --> O1
    subgraph Objects
        O1["O1 · list<br/>[3, 1, 2]"]
    end
```

Two names share the unsorted list

O1, O2, ...: symbolic object IDs

# Sorting in Place

PY main.py

```
1 values = [3, 1, 2]
2 alias = values
3 result = values.sort()
4 print(values, alias)
5 print(result)
```

[Download: lec\\_05\\_sort\\_in\\_place.py](#)

Names

Objects

values

alias

O1 · list

[1, 2, 3]

sort() changes O1; both names see it

O1, O2, ...: symbolic object IDs

# Sorting in Place

PY main.py

```
1 values = [3, 1, 2]
2 alias = values
3 result = values.sort()
4 print(values, alias)
5 print(result)
```

[Download: lec\\_05\\_sort\\_in\\_place.py](#)

Names

Objects

values

alias

O1 · list

[1, 2, 3]

result = None

The returned value is None

O1, O2, ...: symbolic object IDs

# Creating a Sorted List

PY main.py

```
1 values = [3, 1, 2]
2 ordered = sorted(values)
3 print(values)
4 print(ordered)
5 print(values is ordered)
```

Download: lec\_05\_sorted\_copy.py

Names

Objects

values



O1 · list  
[3, 1, 2]

Start with the original ordering

O1, O2, ...: symbolic object IDs

# Creating a Sorted List

PY main.py

```
1 values = [3, 1, 2]
2 ordered = sorted(values)
3 print(values)
4 print(ordered)
5 print(values is ordered)
```

Download: lec\_05\_sorted\_copy.py

Names

Objects

values



O1 · list  
[3, 1, 2]

ordered



O2 · list  
[1, 2, 3]

sorted() returns O2; O1 keeps its order

O1, O2, ...: symbolic object IDs

# Copying with Slices

PY main.py

```
1 original = [1, 2]
2 copied = original[:]
3 copied[0] = 9
4 print(original, copied)
5 print(original is copied)
```

Download: [lec\\_05\\_slice\\_copy.py](#)

Names

Objects

original



O1 · list  
[1, 2]

copied



O2 · list  
[1, 2]

A slice creates a new outer list O2

O1, O2, ...: symbolic object IDs

# Copying with Slices

PY main.py

```
1 original = [1, 2]
2 copied = original[:]
3 copied[0] = 9
4 print(original, copied)
5 print(original is copied)
```

Download: [lec\\_05\\_slice\\_copy.py](#)

Names

Objects

original



O1 · list  
[1, 2]

copied



O2 · list  
[9, 2]

Replacing O2's item leaves O1 unchanged

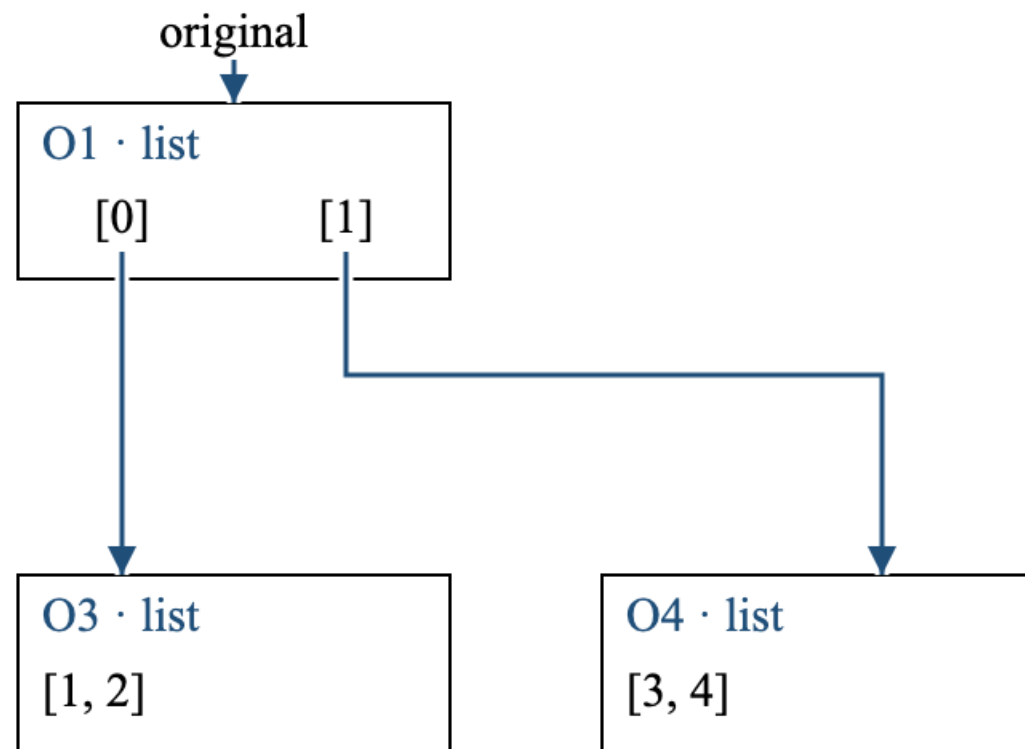
O1, O2, ...: symbolic object IDs

# Shallow Copies

PY main.py

```
1 original = [[1, 2], [3, 4]]
2 copied = original[:]
3 copied[0][1] = 9
4 print(original)
5 print(original is copied)
6 print(original[0] is copied[0])
```

[Download: lec\\_05\\_shallow\\_copy.py](#)



O1 contains references to O3 and O4

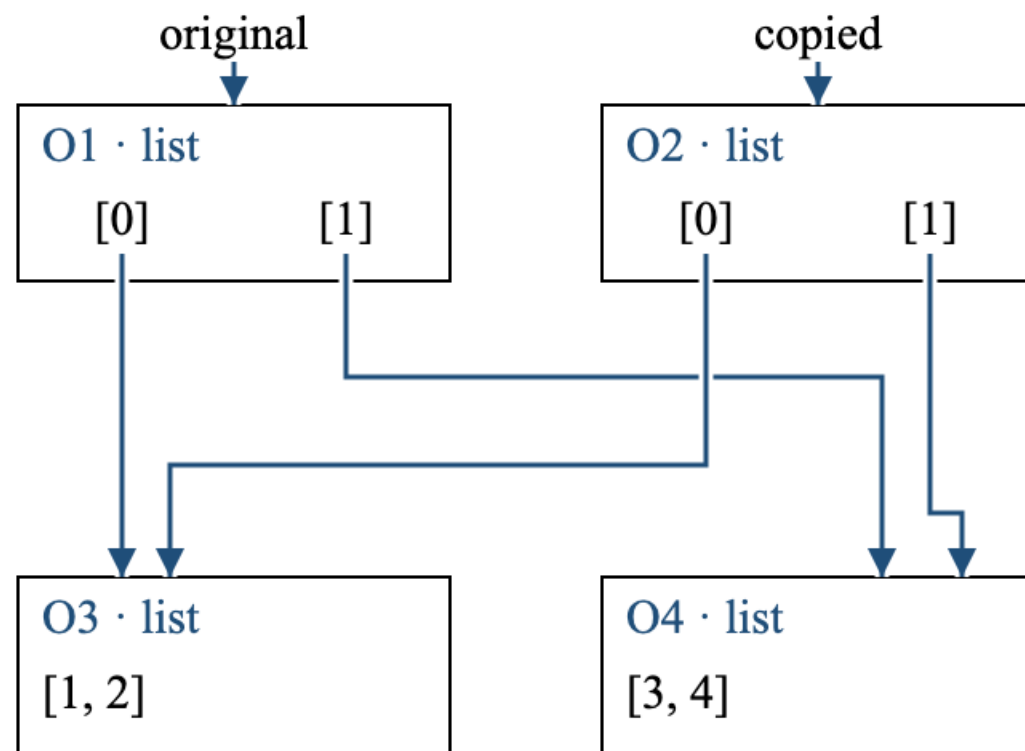
O1, O2, ...: symbolic object IDs

# Shallow Copies

PY main.py

```
1 original = [[1, 2], [3, 4]]
2 copied = original[:]
3 copied[0][1] = 9
4 print(original)
5 print(original is copied)
6 print(original[0] is copied[0])
```

[Download: lec\\_05\\_shallow\\_copy.py](#)



O2 gets copies of the same references

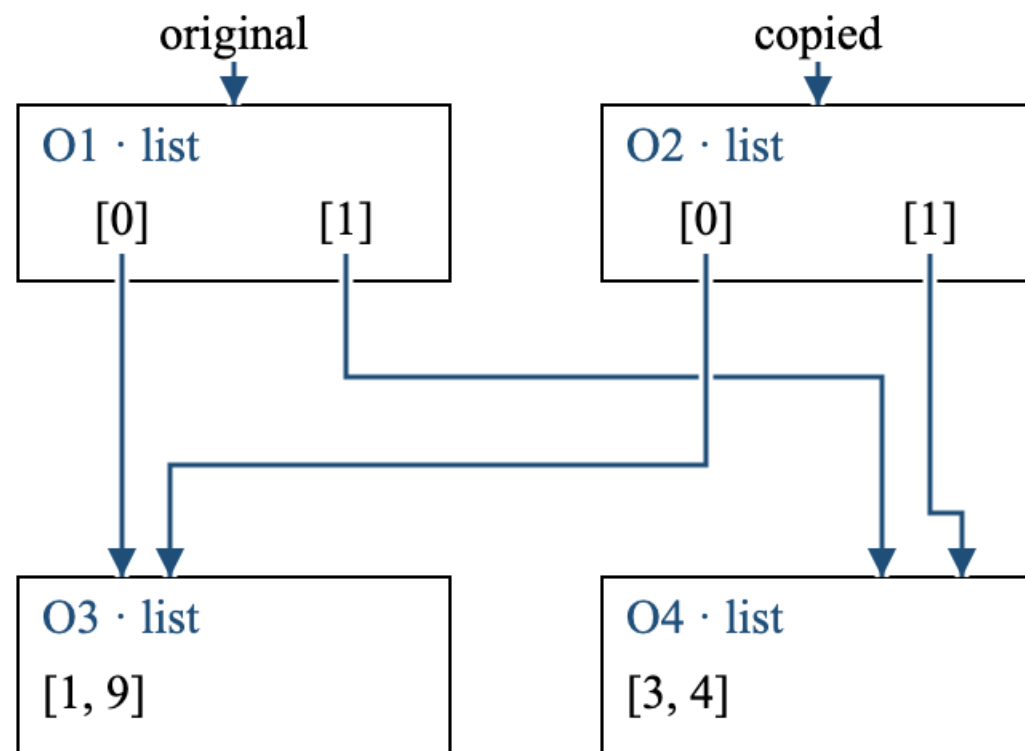
O1, O2, ...: symbolic object IDs

# Shallow Copies

PY main.py

```
1 original = [[1, 2], [3, 4]]
2 copied = original[:]
3 copied[0][1] = 9
4 print(original)
5 print(original is copied)
6 print(original[0] is copied[0])
```

[Download: lec\\_05\\_shallow\\_copy.py](#)



Both outer lists reach the changed O3

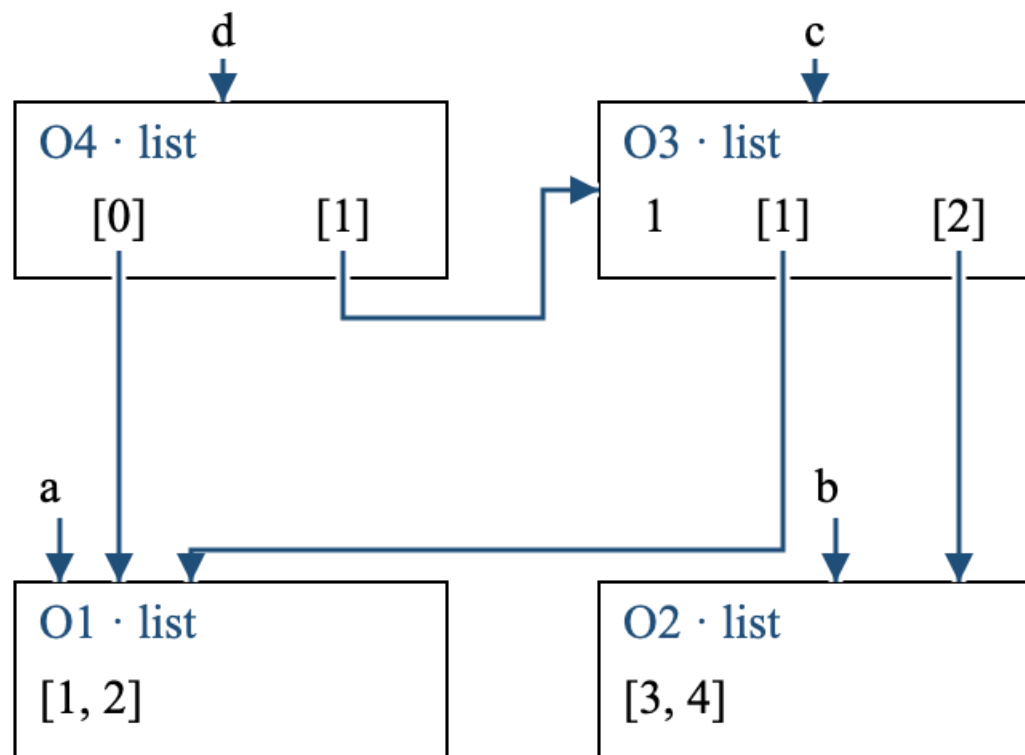
O1, O2, ...: symbolic object IDs

# Nested References

PY main.py

```
1 a = [1, 2]
2 b = [3, 4]
3 c = [1, a, b]
4 d = [a, c]
5 a[1] = 42
6 print(c)
7 print(d[0] is c[1])
```

Download: lec\_05\_nested\_references.py



$c[1]$  and  $d[0]$  both refer to O1

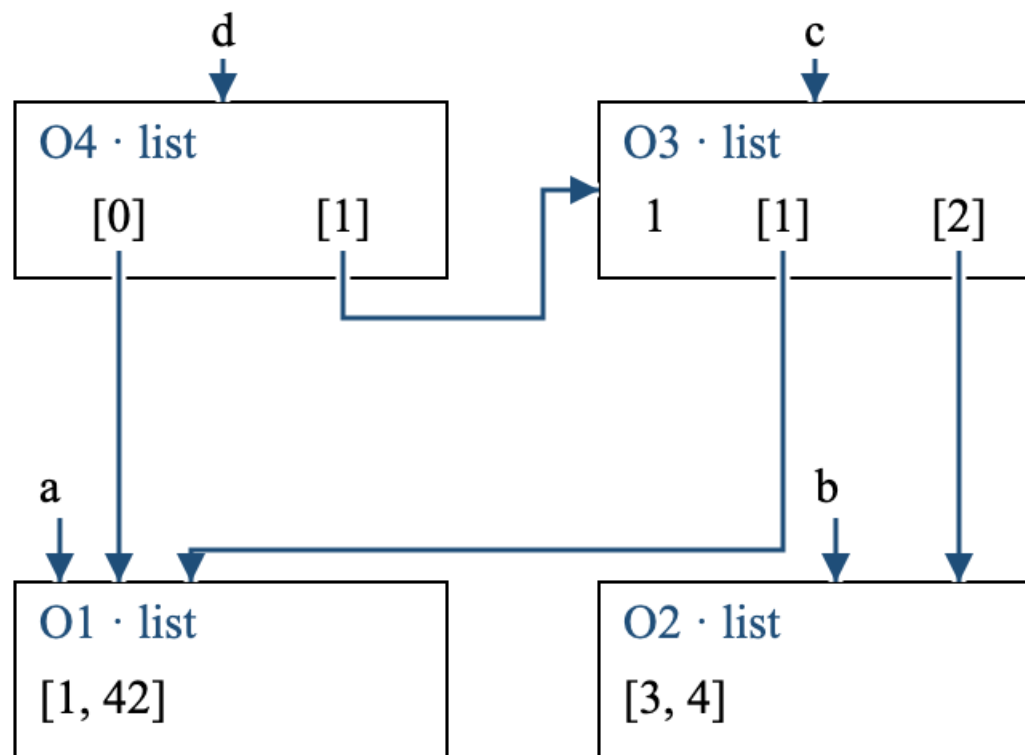
O1, O2, ...: symbolic object IDs

# Nested References

PY main.py

```
1 a = [1, 2]
2 b = [3, 4]
3 c = [1, a, b]
4 d = [a, c]
5 a[1] = 42
6 print(c)
7 print(d[0] is c[1])
```

Download: lec\_05\_nested\_references.py



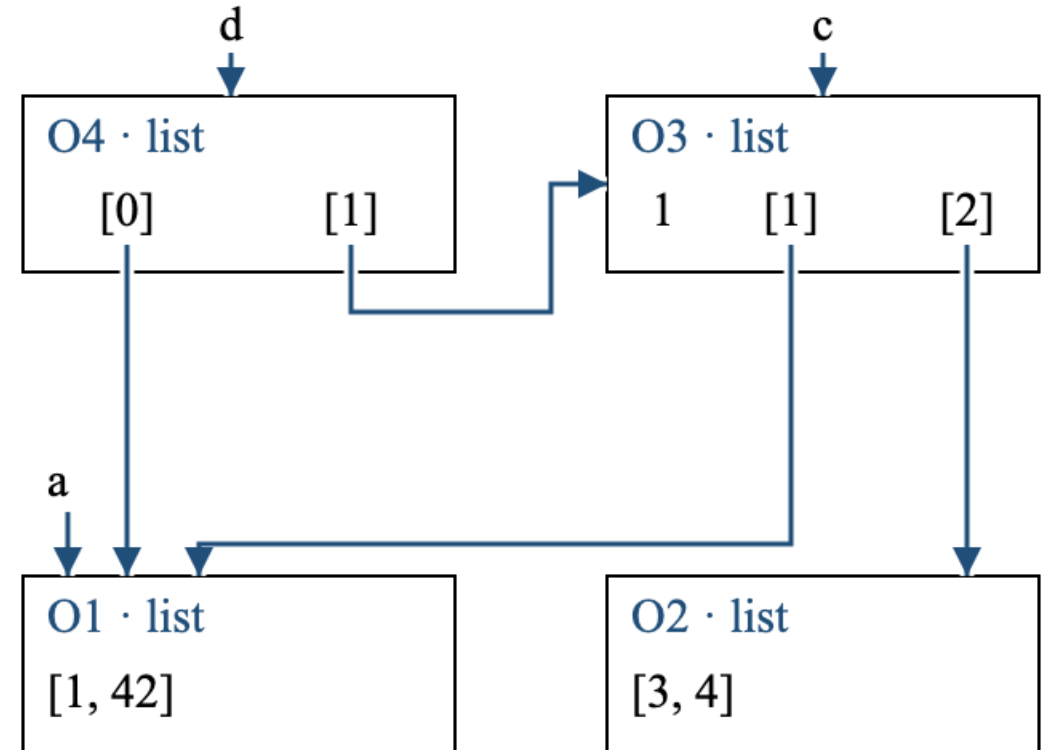
`a[1]` changes the shared O1

O1, O2, ...: symbolic object IDs

# Rebinding Nested Lists

PY main.py

```
1 a = [1, 42]
2 c = [1, a, [3, 4]]
3 d = [a, c]
4 a = [5, 6]
5 d[0] = 1
6 c[1][1] = 3
7 print(a)
8 print(d)
```



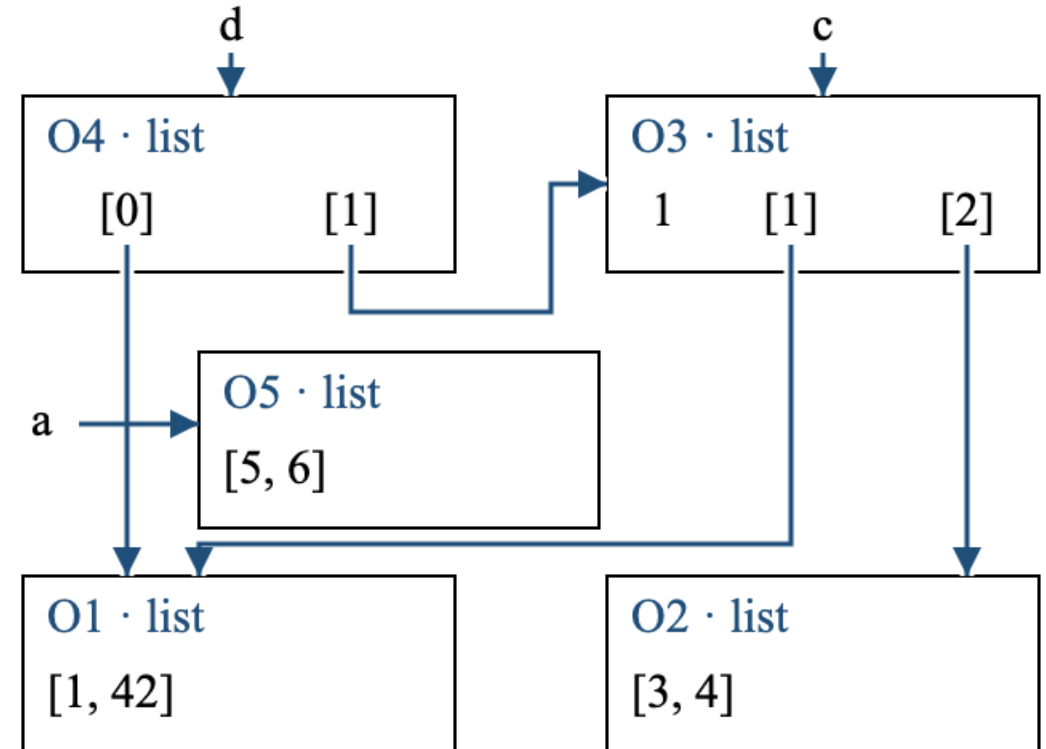
Start with the same shared inner list

O1, O2, ...: symbolic object IDs

# Rebinding Nested Lists

PY main.py

```
1 a = [1, 42]
2 c = [1, a, [3, 4]]
3 d = [a, c]
4 a = [5, 6]
5 d[0] = 1
6 c[1][1] = 3
7 print(a)
8 print(d)
```



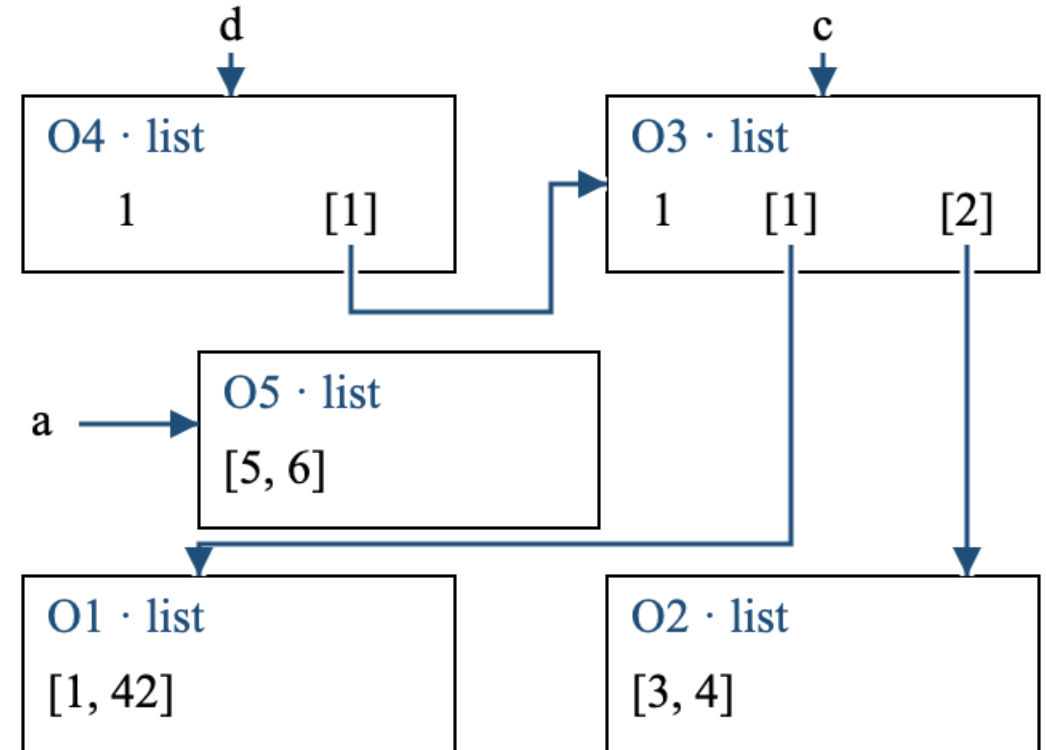
a moves to the new object O5

O1, O2, ...: symbolic object IDs

# Rebinding Nested Lists

PY main.py

```
1 a = [1, 42]
2 c = [1, a, [3, 4]]
3 d = [a, c]
4 a = [5, 6]
5 d[0] = 1
6 c[1][1] = 3
7 print(a)
8 print(d)
```



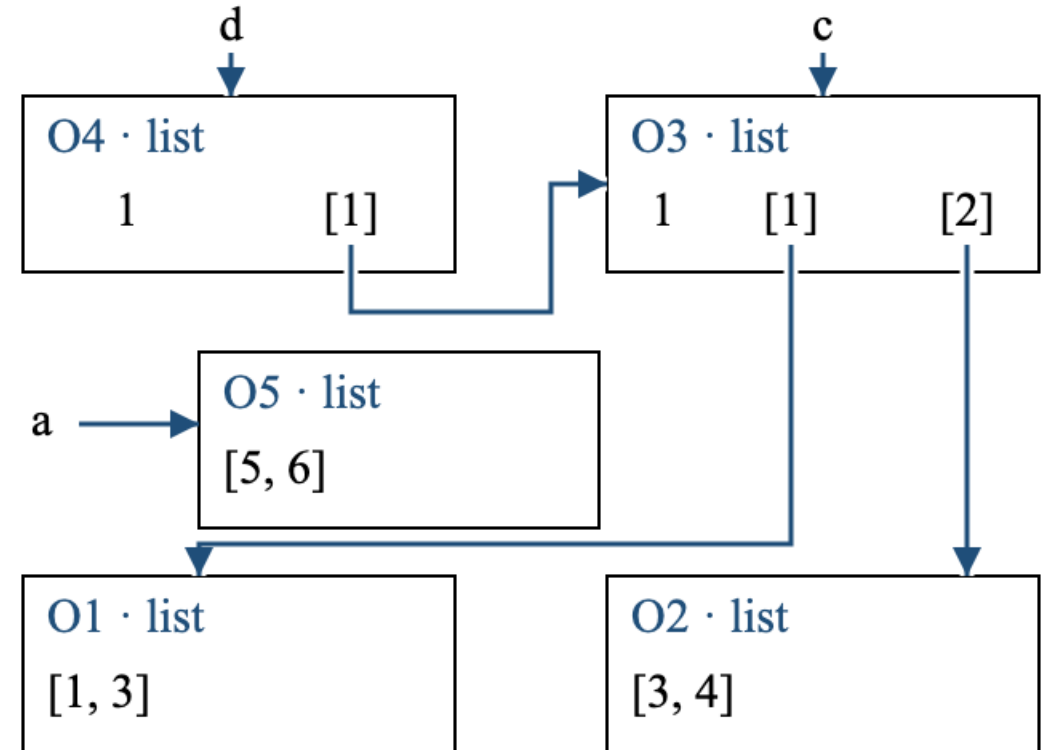
**d[0] becomes 1; c[1] still reaches O1**

O1, O2, ...: symbolic object IDs

# Rebinding Nested Lists

PY main.py

```
1 a = [1, 42]
2 c = [1, a, [3, 4]]
3 d = [a, c]
4 a = [5, 6]
5 d[0] = 1
6 c[1][1] = 3
7 print(a)
8 print(d)
```



$c[1][1]$  changes O1, not O5

O1, O2, ...: symbolic object IDs

# Shallow Copies: Check

PY main.py

```
1 a = [[1], [2]]
2 b = a[:]
3 b[0].append(9)
4 print(a)
5 b[0] = [7]
6 print(a, b)
```

Download: [lec\\_05\\_shallow\\_check.py](#)

Predict both output lines

Which statement changes the shared inner list?

# Shallow Copies: Check

PY main.py

```
1 a = [[1], [2]]
2 b = a[:]
3 b[0].append(9)
4 print(a)
5 b[0] = [7]
6 print(a, b)
```

[Download: lec\\_05\\_shallow\\_check.py](#)

# Shallow Copies: Check

PY main.py

```
1 a = [[1], [2]]
2 b = a[:]
3 b[0].append(9)
4 print(a)
5 b[0] = [7]
6 print(a, b)
```

[Download: lec\\_05\\_shallow\\_check.py](#)

# Shallow Copies: Check

PY main.py

```
1 a = [[1], [2]]
2 b = a[:]
3 b[0].append(9)
4 print(a)
5 b[0] = [7]
6 print(a, b)
```

[Download: lec\\_05\\_shallow\\_check.py](#)

# Removal During Iteration

PY main.py

```
1 def remove_blocked(values, blocked):
2     for value in values:
3         if value in blocked:
4             values.remove(value)
5 values = [1, 2, 3, 4]
6 blocked = [1, 2]
7 remove_blocked(values, blocked)
8 print(values)
```

[Download: lec\\_05\\_remove\\_while\\_iterating.py](#)

values · original list

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
|---|---|---|---|

0 1 2 3



Next

blocked = [1, 2]

Next visits index 0: value 1

# Removal During Iteration

PY main.py

```
1 def remove_blocked(values, blocked):
2     for value in values:
3         if value in blocked:
4             values.remove(value)
5 values = [1, 2, 3, 4]
6 blocked = [1, 2]
7 remove_blocked(values, blocked)
8 print(values)
```

[Download: lec\\_05\\_remove\\_while\\_iterating.py](#)

values · original list

|   |      |   |
|---|------|---|
| 2 | 3    | 4 |
| 0 | 1    | 2 |
|   | ↑    |   |
|   | Next |   |

2 has moved to index 0

After removing 1, Next advances to index 1

# Removal During Iteration

PY main.py

```
1 def remove_blocked(values, blocked):
2     for value in values:
3         if value in blocked:
4             values.remove(value)
5 values = [1, 2, 3, 4]
6 blocked = [1, 2]
7 remove_blocked(values, blocked)
8 print(values)
```

[Download: lec\\_05\\_remove\\_while\\_iterating.py](#)

values · original list

|   |   |   |
|---|---|---|
| 2 | 3 | 4 |
| 0 | 1 | 2 |

↑  
Next

3 is not blocked

Visit 3, then advance to index 2

# Removal During Iteration

PY main.py

```
1 def remove_blocked(values, blocked):
2     for value in values:
3         if value in blocked:
4             values.remove(value)
5 values = [1, 2, 3, 4]
6 blocked = [1, 2]
7 remove_blocked(values, blocked)
8 print(values)
```

[Download: lec\\_05\\_remove\\_while\\_iterating.py](#)

values · original list

|   |   |   |
|---|---|---|
| 2 | 3 | 4 |
| 0 | 1 | 2 |

4 is not blocked

Visit 4, then finish: 2 was never checked

# Iteration over a Copy

PY main.py

```
1 def remove_blocked(values, blocked):
2     for value in values[:]:
3         if value in blocked:
4             values.remove(value)
5 values = [1, 2, 3, 4]
6 blocked = [1, 2]
7 remove_blocked(values, blocked)
8 print(values)
```

[Download: lec\\_05\\_remove\\_from\\_copy.py](#)

Iteration copy

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 |

↑  
Next

values · original list

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 |

Iterate over a copy; remove from the original

# Iteration over a Copy

PY main.py

```
1 def remove_blocked(values, blocked):
2     for value in values[:]:
3         if value in blocked:
4             values.remove(value)
5 values = [1, 2, 3, 4]
6 blocked = [1, 2]
7 remove_blocked(values, blocked)
8 print(values)
```

[Download: lec\\_05\\_remove\\_from\\_copy.py](#)

Iteration copy

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 |

↑  
Next

values · original list

|   |   |   |
|---|---|---|
| 2 | 3 | 4 |
| 0 | 1 | 2 |

Remove 1; the copy still has 2 at index 1

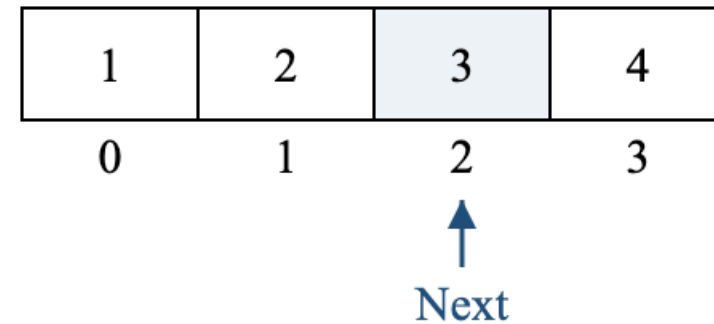
# Iteration over a Copy

PY main.py

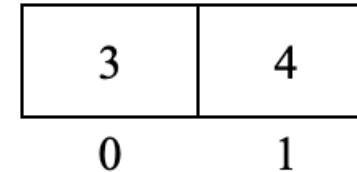
```
1 def remove_blocked(values, blocked):
2     for value in values[:]:
3         if value in blocked:
4             values.remove(value)
5 values = [1, 2, 3, 4]
6 blocked = [1, 2]
7 remove_blocked(values, blocked)
8 print(values)
```

[Download: lec\\_05\\_remove\\_from\\_copy.py](#)

Iteration copy



values · original list



Remove 2; the next copied item is 3

# Iteration over a Copy

PY main.py

```
1 def remove_blocked(values, blocked):
2     for value in values[:]:
3         if value in blocked:
4             values.remove(value)
5 values = [1, 2, 3, 4]
6 blocked = [1, 2]
7 remove_blocked(values, blocked)
8 print(values)
```

[Download: lec\\_05\\_remove\\_from\\_copy.py](#)

Iteration copy

|   |   |   |   |
|---|---|---|---|
| 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 |

values · original list

|   |   |
|---|---|
| 3 | 4 |
| 0 | 1 |

Visit 3 and 4; both remain in the original

# List Comprehensions

ENGG1330  
Computer Programming I

# Building a List

PY main.py

```
1 squares = []  
2 for number in range(5):  
3     squares.append(number * number)  
4 print(squares)
```

Download: [lec\\_05\\_comprehension\\_loop.py](#)

squares · result list

[]

range(5): 0, 1, 2, 3, 4

Start with an empty result list

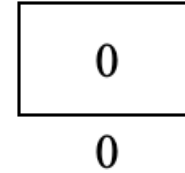
# Building a List

PY main.py

```
1 squares = []  
2 for number in range(5):  
3     squares.append(number * number)  
4 print(squares)
```

Download: lec\_05\_comprehension\_loop.py

squares · result list



number = 0

$0 \times 0 = 0$

Append 0; the list now has 1 item

# Building a List

PY main.py

```
1 squares = []
2 for number in range(5):
3     squares.append(number * number)
4 print(squares)
```

Download: lec\_05\_comprehension\_loop.py

squares · result list

|   |   |
|---|---|
| 0 | 1 |
| 0 | 1 |

number = 1

$1 \times 1 = 1$

Append 1; the list now has 2 items

# Building a List

PY main.py

```
1 squares = []
2 for number in range(5):
3     squares.append(number * number)
4 print(squares)
```

Download: [lec\\_05\\_comprehension\\_loop.py](#)

squares · result list

|   |   |   |
|---|---|---|
| 0 | 1 | 4 |
| 0 | 1 | 2 |

number = 2

$2 \times 2 = 4$

Append 4; the list now has 3 items

# Building a List

PY main.py

```
1 squares = []  
2 for number in range(5):  
3     squares.append(number * number)  
4 print(squares)
```

Download: lec\_05\_comprehension\_loop.py

squares · result list

|   |   |   |   |
|---|---|---|---|
| 0 | 1 | 4 | 9 |
| 0 | 1 | 2 | 3 |

number = 3

$3 \times 3 = 9$

Append 9; the list now has 4 items

# Building a List

PY main.py

```
1 squares = []  
2 for number in range(5):  
3     squares.append(number * number)  
4 print(squares)
```

Download: [lec\\_05\\_comprehension\\_loop.py](#)

squares · result list

|   |   |   |   |    |
|---|---|---|---|----|
| 0 | 1 | 4 | 9 | 16 |
| 0 | 1 | 2 | 3 | 4  |

number = 4

$4 \times 4 = 16$

Append 16; the list now has 5 items

# Comprehension Syntax

PY main.py

```
1 squares = [number * number for number in range(5)]  
2 print(squares)  
3 numbers = [number for number in range(5)]  
4 print(numbers)
```

Download: [lec\\_05\\_comprehension\\_basic.py](#)

# Filtering with Comprehensions

PY main.py

```
1 evens = [n for n in range(8) if n % 2 == 0]
2 inverses = [1 / n for n in range(-2, 3) if n != 0]
3 print(evens)
4 print(inverses)
```

Download: [lec\\_05\\_comprehension\\_filter.py](#)

# Building a Filtered List

PY main.py

```
1 values = [1, 2, 3, 4]
2 blocked = [1, 2]
3 kept = [value for value in values if value not in blocked]
4 print(kept)
5 print(values)
```

[Download: lec\\_05\\_filter\\_copy.py](#)

# Comprehensions: Check

```
result = [n * n for n in range(6) if n % 2 == 1]
```

Write the equivalent loop, then predict the result

# The random Module

PY main.py

```
1 import random
2 random.seed(7)
3 print(random.randint(1, 6))
4 print(random.choice('pass', 'retry'))
```

[Download: lec\\_05\\_random\\_intro.py](#)

A module groups reusable code

import random makes its functions  
available

# The random Module

PY main.py

```
1 import random
2 random.seed(7)
3 print(random.randint(1, 6))
4 print(random.choice(['pass', 'retry']))
```

[Download: lec\\_05\\_random\\_intro.py](#)

`seed(7)` sets a repeatable starting state

The dot selects a function from the module

# The random Module

PY main.py

```
1 import random
2 random.seed(7)
3 print(random.randint(1, 6))
4 print(random.choice(('pass', 'retry')))
```

[Download: lec\\_05\\_random\\_intro.py](#)

`randint(1, 6)` chooses an integer from  
1 to 6

Both limits are included

# The random Module

PY main.py

```
1 import random
2 random.seed(7)
3 print(random.randint(1, 6))
4 print(random.choice(('pass', 'retry')))
```

[Download: lec\\_05\\_random\\_intro.py](#)

choice selects one item from a sequence

Run twice and compare the outputs

# Random Samples

PY main.py

```
1 import random
2 random.seed(7)
3 rolls = [random.randint(1, 6) for roll in range(5)]
4 print(rolls)
```

[Download: lec\\_05\\_random\\_samples.py](#)

# Text Files

ENGG1330  
Computer Programming I

# Opening and Closing Files

PY main.py

```
1 with open('demo.txt', 'w') as file:  
2     print('File is open')  
3     print('The block has ended')
```

[Download: lec\\_05\\_file\\_block\\_intro.py](#)

open opens a file

'w' creates or replaces it; as file  
names the file object

# Opening and Closing Files

PY main.py

```
1 with open('demo.txt', 'w') as file:  
2     print('File is open')  
3 print('The block has ended')
```

[Download: lec\\_05\\_file\\_block\\_intro.py](#)

Indented statements run inside the with block

The file is open here

# Opening and Closing Files

PY main.py

```
1 with open('demo.txt', 'w') as file:  
2     print('File is open')  
3 print('The block has ended')
```

[Download: lec\\_05\\_file\\_block\\_intro.py](#)

Leaving the block closes the file

'r' opens an existing file for reading

# Opening and Closing Files

PY main.py

```
1 with open('demo.txt', 'w') as file:  
2     print('File is open')  
3     print('The block has ended')
```

[Download: lec\\_05\\_file\\_block\\_intro.py](#)

Later examples add `encoding='utf-8'`

This selects the text encoding

# File Methods

PY main.py

```
1- with open('demo.txt', 'w') as file:
2-     file.write('key\nmap\n')
3- with open('demo.txt', 'r') as file:
4-     print(repr(file.read()))
5- with open('demo.txt', 'r') as file:
6-     for line in file:
7-         print(repr(line))
```

write(text) writes a string

\n separates the two lines

# File Methods

PY main.py

```
1- with open('demo.txt', 'w') as file:
2-     file.write('key\nmap\n')
3- with open('demo.txt', 'r') as file:
4-     print(repr(file.read()))
5- with open('demo.txt', 'r') as file:
6-     for line in file:
7-         print(repr(line))
```

read() returns all remaining text

repr makes newline characters visible

# File Methods

PY main.py

```
1- with open('demo.txt', 'w') as file:
2-     file.write('key\nmap\n')
3- with open('demo.txt', 'r') as file:
4-     print(repr(file.read()))
5- with open('demo.txt', 'r') as file:
6-     for line in file:
7-         print(repr(line))
```

A file is iterable

for line in file: reads one line at a time

Each line can include its newline

# Writing and Reading Text

PY main.py

```
1- with open('demo_ids.txt', 'w', encoding='utf-8') as file:
2-     file.write('1001\n1002\n1003\n')
3- with open('demo_ids.txt', 'r', encoding='utf-8') as file:
4-     text = file.read()
5- print(repr(text))
```

[Download: lec\\_05\\_write\\_text.py](#)

# Reading Numbers

PY main.py

```
1- with open('demo_ids.txt', 'w', encoding='utf-8') as file:
2     file.write('1001\n1002\n1003\n')
3     ids = ()
4- with open('demo_ids.txt', 'r', encoding='utf-8') as file:
5     for line in file:
6         ids += (int(line.strip()),)
7     print(ids)
8     print(len(ids), ids[1])
```

# Saving Simulated Outcomes

PY main.py

```
1 import random
2 random.seed(7)
3 ids = (1001, 1002, 1003)
4 with open('demo_results.txt', 'w', encoding='utf-8') as file:
5     for identifier in ids:
6         outcome = random.choice(('pass', 'retry'))
7         file.write(f'{identifier},{outcome}\n')
8 with open('demo_results.txt', 'r', encoding='utf-8') as file:
9     print(file.read(), end='')
```

# L5 Survey

Complete the survey during the lecture

# L5 Coding Challenges

Work independently on the four challenges