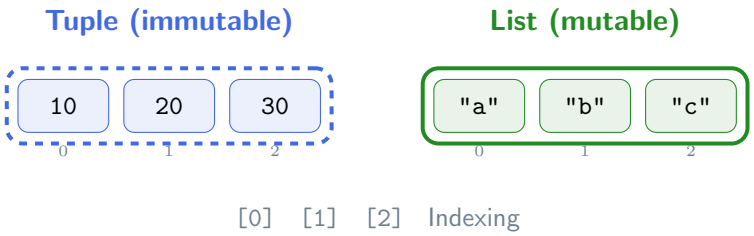
Programming Fundamentals
Lab 06

Tuples and Lists

Ordered Collections and Mutability



Comp 102 — Forman Christian University

Spring 2026

Estimated Time: ∼**3 hours** — Based on Lectures 11 & 12

How to Use This Lab

This lab is a **walkthrough tutorial**—it is designed to guide you through learning, not to test you. Work through each section at your own pace:

- **Predict** — Write down what you think will happen *before* touching the computer.
- **Type** — Enter the code in Thonny's **Code Editor** (top pane) and press **Run** (or **F5**).
- **Verify** — Compare your prediction with the actual result. If they differ, figure out *why*.
- Exercises are graded by difficulty:
  - ★ Easy — Immediate practice. Follow the pattern you just saw.
  - ★★ Medium — Requires some thinking. Combines concepts.
  - ★★★ Hard — Stretch goal. It is OK to need help!
- **Ask for help** whenever you are stuck for more than 5 minutes.

Prerequisites

This lab assumes you completed **Lab 05**. You should be comfortable with **for loops**, **while loops**, **range()**, and **string indexing**.

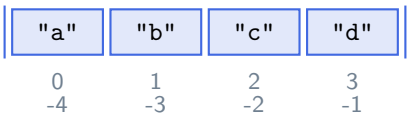
1 Tuples and Basic List Operations (~25 min)

Ref: Lecture 11 — Tuples and Lists

This section introduces ordered collections. Tuples are immutable (cannot change after creation), while lists are mutable. Both use indexing to access individual elements.

Indexing Reminder

Indices start at `0`. Positive indices count from the left; negative indices count from the right (`-1` is the last element).



1. ★ Easy Predict the output, then type it in the **Code Editor** and run:

```
coords = (5, 10)
print(coords[0])
print(coords[1])
```

Your prediction:

Tuples use parentheses `()`. What is at index `0`? At index `1`?

2. ★ Easy Predict the output for each print statement:

```
fruits = ["apple", "banana", "cherry", "date"]
print(fruits[0])
print(fruits[-1])
print(fruits[2])
```

`fruits[0]`: _____
`fruits[-1]`: _____
`fruits[2]`: _____

Negative indices count from the end. `-1` is always the last element.

3. ★★ Medium Predict the output, then verify:

```
numbers = [10, 20, 30, 40, 50]
print(numbers[1:4])
print(numbers[:3])
print(numbers[2:])
print(numbers[-3:-1])
```

`numbers[1:4]`: _____
`numbers[:3]`: _____
`numbers[2:]`: _____

`numbers[-3:-1]: _____`

*Slicing `[start:end]` includes **`start`** but excludes **`end`**.*

4. ★★ Medium The code below tries to modify a tuple. What happens?

```
point = (3, 7)
point[0] = 5
```

Your prediction:

Careful! *Tuples are immutable. This will raise a **`TypeError`**.*

5. ★★ Medium The same operation on a list works. Predict the output:

```
scores = [85, 90, 78]
scores[1] = 95
print(scores)
```

Your prediction:

*Lists use square brackets `[]` and **can** be modified after creation.*

6. ★★★ Hard **Write a program** that:

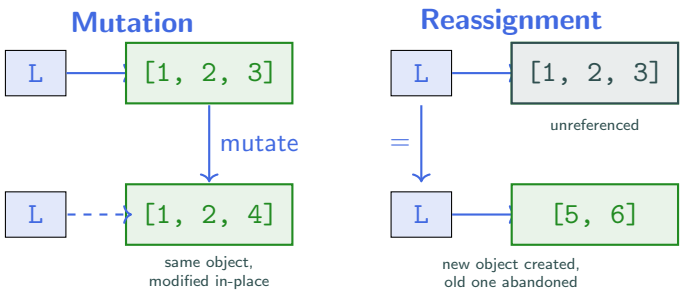
- Creates a tuple `rgb = (255, 128, 0)` representing a color
- Prints each channel on a separate line (Red, Green, Blue)
- Creates a list `grades = [75, 82, 91]`
- Changes the first grade to `80` and the last to `95`
- Prints the modified list

Write your code:

2 List Methods and Mutability (~30 min)

Ref: Lecture 11 — List Methods and Mutability

Lists have built-in methods that modify the list in place (mutating) versus creating a new list. Understanding this distinction is crucial for avoiding bugs.



⚠ Mutation vs. New List

`append()`, `extend()`, and `sort()` modify the original list. `+` and `sorted()` create new lists and leave the original unchanged.

7. ★ Easy Predict the output, then verify:

```
items = ["pen"]
items.append("pencil")
items.append("eraser")
print(items)
```

Your prediction:

`append()` adds a single element to the end of the list.

8. ★ Easy What is the difference between `append()` and `extend()`? Predict the output:

```
a = [1, 2]
b = [1, 2]
a.append([3, 4])
b.extend([3, 4])
print("a:", a)
print("b:", b)
```

a: _____

b: _____

*`append([3, 4])` adds the **entire list** as one element. `extend()` adds each element individually.*

9. ★★ Medium Predict the output for both lists, then verify:

```
x = [3, 1, 4]
y = x.sort()
print("x:", x)
print("y:", y)

a = [3, 1, 4]
b = sorted(a)
print("a:", a)
print("b:", b)
```

After `x.sort()`: x = _____, y = _____
After `sorted(a)`: a = _____, b = _____

`sort()` modifies in place and returns `None`. `sorted()` returns a new sorted list.

10. ★★ Medium `+` creates a new list, while `extend()` modifies in place. Predict the output:

```
nums1 = [1, 2]
nums2 = [3, 4]
result = nums1 + nums2
print("nums1:", nums1)
print("result:", result)

a = [1, 2]
b = [3, 4]
a.extend(b)
print("a:", a)
```

`nums1` after `+`: _____
`a` after `extend()`: _____

Notice that `nums1` is unchanged by `+`, but `a` is modified by `extend()`.

11. ★★ Medium Predict the output, then verify:

```
nums = [10, 20, 30, 40]
nums.insert(2, 99)
last = nums.pop()
print(nums)
print(last)
```

Your prediction:

`insert(i, x)` adds at index `i`. `pop()` removes and returns the last element.

12. ★★★ Hard **Write a program** that:

- Starts with `scores = [88, 70, 95, 82]`
- Creates a **new** list `sorted_scores` without changing `scores`
- Adds `100` to the end of `sorted_scores`
- Prints both lists

Write your code:

13. ★★★ Hard **Fill in the blanks** to build a list of even numbers from 2 to 10:

```
evens = []
for n in range(2, 11):
    if ____________ == 0:
        ____________
print(evens)
```

First blank: _____
Second blank: _____

14. ★★★ Hard **Write a program** that:

- Starts with an empty list called `shopping`
- Appends `"milk"`, `"eggs"`, and `"bread"`
- Extends it with `["butter", "jam"]`
- Sorts the list in place
- Creates a new list `reversed_shopping` that is the reverse of `shopping` (use slicing)
- Prints both lists

Write your code:

3 Strings to Lists and Back (~20 min)

Ref: Lecture 11 — String Split and Join

Strings and lists can be converted back and forth. `split()` breaks a string into a list; `join()` combines a list into a string.

Split and Join

`split()` with no argument splits on whitespace. `split(",")` splits on commas. `join()` is called on the separator string: `"-".join(words)`.

15. ★ Easy Predict the output, then verify:

```
sentence = "the quick brown fox"
words = sentence.split()
print(words)
print(len(words))
```

```
words: _____
len(words): _____
```

`split()` with no arguments splits on any whitespace (spaces, tabs).

16. ★ Easy Predict the output, then verify:

```
data = "apple,banana,cherry"
fruits = data.split(",")
print(fruits)

joined = "-".join(fruits)
print(joined)
```

```
fruits: _____
joined: _____
```

*The delimiter goes **before** `join()`: `separator.join(list)`.*

17. ★★ Medium What does this code print? Trace it on paper first:

```
text = "hello world from python"
parts = text.split(" ")
result = "|".join(parts)
print(result)
```

Your prediction:

First split creates a list of words, then join puts them back together with `|`.

18. ★★★ Hard **Write a program** that:

- Asks the user for a date in `YYYY-MM-DD` format using `input()`
- Splits the date into year, month, and day
- Prints them on separate lines with labels

Example:

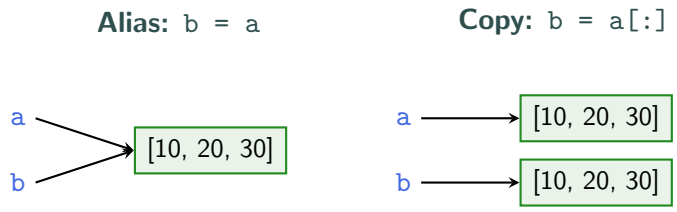
```
Enter date (YYYY-MM-DD): 2024-03-15
Year: 2024
Month: 03
Day: 15
```

Write your code:

4 Aliasing and Cloning (~25 min)

Ref: Lecture 12 — Aliasing and Cloning

When you assign `L2 = L1`, both variables point to the **same** list in memory. Changes through one variable affect the other. To make an independent copy, use cloning.



⚠ The Alias Trap

`L2 = L1` does NOT create a copy. It creates an alias. Use `L2 = L1[:]` or `L2 = list(L1)` or `L2 = L1.copy()` to clone.

19. ★ Easy Predict the output, then verify:

```
a = [1, 2, 3]
b = a
b[0] = 99
print("a:", a)
print("b:", b)
```

a: _____
b: _____

Tricky! *`b = a` makes `b` point to the same list. Changing `b` also changes `a`.*

20. ★ Easy Now with a proper clone. Predict the output:

```
x = [1, 2, 3]
y = x[:]
y[0] = 99
print("x:", x)
print("y:", y)
```

x: _____
y: _____

`x[:]` creates a new list with the same elements. Now `x` and `y` are independent.

21. ★★ Medium All these create clones. Predict the output (will all three be `True` or `False`?):

```
original = [1, 2, 3]

clone1 = original[:]
clone2 = list(original)
clone3 = original.copy()
```

```
6
7 print(clone1 == original)
8 print(clone1 is original)
```

clone1 == original: _____
clone1 is original: _____

== checks if values are equal. `is` checks if they are the same object in memory.

22. ★★ Medium Predict the output, then verify:

```
1 def add_end(lst):
2     lst = lst + [99]
3
4 nums = [1, 2]
5 add_end(nums)
6 print(nums)
```

Your prediction:

Inside the function, `lst` is reassigned to a new list. Does the original change?

23. ★★ Medium Predict the output, then verify:

```
1 grid = [[1, 2], [3, 4]]
2 copy_grid = grid[:]
3 copy_grid[0][0] = 99
4 print(grid)
```

Your prediction:

***Careful!** A shallow copy duplicates the outer list, but inner lists are still shared.*

24. ★★ Medium Predict the output, then verify:

```
1 items = ["a", "b"]
2 alias = items
3 clone = items[:]
4 alias.append("c")
5 clone.append("d")
6 print(items)
7 print(alias)
8 print(clone)
```

Your prediction:

`alias` shares the same list as `items`, but `clone` is separate.

25. ★★ Medium What happens here? Trace carefully:

```
def add_item(lst, item):
    lst.append(item)

groceries = ["milk"]
add_item(groceries, "eggs")
print(groceries)
```

> **Your prediction:**

When you pass a list to a function, you pass a reference. The function can modify the original!

26. ★★★ Hard **Write a function** `without_zeros(lst)` that:

- Returns a **new** list with all zeros removed
- Does **not** change the original list

Example:

```
Input:  [0, 2, 0, 3, 4, 0]
Output: [2, 3, 4]
Original still: [0, 2, 0, 3, 4, 0]
```

> **Write your code:**

27. ★★★ Hard **Bug hunt!** This student wanted to keep `original` unchanged but it gets modified. Why?

```
def remove_first(lst):
    lst.pop(0)
    return lst

original = [10, 20, 30]
new_list = remove_first(original)
print("original:", original)
print("new_list:", new_list)
```

> **The bug:**

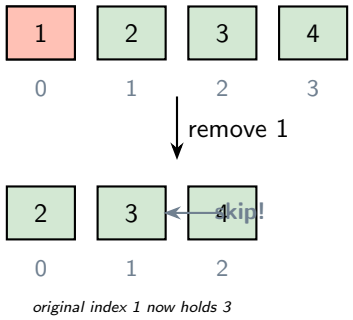
How to fix it:

Hint: Clone the list inside the function before modifying it.

5 The Iteration Mutation Trap (~25 min)

Ref: Lecture 12 — Safe Iteration

Removing items from a list while iterating over it causes elements to be skipped. The safe pattern is to iterate over a copy while modifying the original.



Never Modify While Iterating

`for x in L: L.remove(x)` will skip elements. Use `for x in L[:]` to iterate over a copy instead.

28. ★ Easy What does this code print? Trace through it:

```
numbers = [1, 2, 3, 4]
for n in numbers:
    print(n)
```

Output:

29. ★★ Medium This code has a bug! Predict what actually prints:

```
values = [1, 2, 3, 4]
for v in values:
    values.remove(v)
print(values)
```

Your prediction: _____

Careful! *When you remove 1, everything shifts left. 2 gets skipped!*

30. ★★ Medium The fix: iterate over a copy. Predict the output:

```
values = [1, 2, 3, 4]
for v in values[:]:
    values.remove(v)
print(values)
```

Your prediction: _____

`values[:]` creates a copy for iteration, so we safely modify the original.

31. ★★ Medium `del`, `pop()`, and `remove()` differ. Predict the output:

```
a = [10, 20, 30, 20]
del a[1]
print("after del:", a)

b = [10, 20, 30]
x = b.pop()
print("popped:", x, "list:", b)

c = [10, 20, 30]
c.remove(20)
print("after remove:", c)
```

After `del a[1]`: _____
After `pop()`: x = _____, b = _____
After `remove(20)`: _____

`del` removes by index. `pop()` removes and returns the last (or specified) element. `remove()` removes by value (first match only).

32. ★★★ Hard **Write a program** that:

- Starts with `items = ["keep", "remove", "keep", "remove", "keep"]`
- Removes all occurrences of `"remove"` safely (without skipping)
- Prints the cleaned list

Write your code:

6 Capstone Challenges (~20 min)

These problems integrate tuple packing, list mutation, cloning, and safe iteration. Write your solutions in Thonny's code editor.

33. ★★ Medium **Grade Statistics.** Write a program that:

1. Has a list of grades: `[85, 92, 78, 65, 88, 91, 73]`
2. Calculates and prints the average
3. Creates a new list containing only grades that are ≥ 80 (use a loop)
4. Prints the new list and its length

Write your code:

34. ★★★ Hard **Word Filter.** Write a program that:

1. Asks the user for a sentence
2. Splits it into words
3. Removes all words with 3 or fewer characters (safely!)
4. Joins the remaining words back into a sentence
5. Prints the result

Example:

```
Enter a sentence: The quick brown fox jumps
Result: quick brown jumps
```

Write your code:

35. ★★★ Hard **Palindrome Checker.** A palindrome reads the same forwards and backwards. Write a program that:

1. Asks the user for a word
2. Converts it to a list of characters (use `list(word)`)
3. Creates a reversed copy of that list (use slicing)
4. Compares the original and reversed lists
5. Prints whether it's a palindrome

Example:

```
Enter a word: radar
radar is a palindrome

Enter a word: hello
hello is not a palindrome
```

Write your code:

Self-Assessment Checklist

Before you leave, check off what you can do:

- ☐ I can create tuples and lists and access elements using positive and negative indices
- ☐ I can use slicing to extract portions of a list or tuple
- ☐ I understand that tuples are immutable while lists are mutable
- ☐ I can use `append()`, `extend()`, and `sort()` to modify lists in place
- ☐ I know the difference between `sort()` (mutating) and `sorted()` (returns new list)
- ☐ I can use `split()` to break strings into lists and `join()` to combine lists into strings
- ☐ I understand that `L2 = L1` creates an alias, not a copy
- ☐ I can create shallow copies using `[:]`, `list()`, or `.copy()`
- ☐ I know that `==` checks equality while `is` checks identity
- ☐ I understand why modifying a list while iterating over it causes skipped elements
- ☐ I can safely remove items during iteration by iterating over a copy (`L[:]`)
- ☐ I know the difference between `del`, `pop()`, and `remove()`
- ☐ I can combine all these skills to solve multi-step list problems