Finger Exercises 11

Tuples and Lists

≈ 45 min + stretch

Comp 102 — Programming Fundamentals

What this is. A workout for what we just covered in class. Most of the programs here are new, so you have to *trace* them, not remember them. The skills being drilled: *which position holds which element, does this line change the list or build a new one, and what does it hand back?*

How to do it. On **paper**, **no computer**, in one sitting. For every list, sketch it in the margin as a row of boxes. When a line changes the list, change your boxes; when a line builds a new list, draw a new row. If an item takes you more than a minute, skip it, finish the rest, then come back. Parts A–J are for *everyone*. Part K is a **stretch**: skip it freely, or attack it if A–J felt smooth. The Console Quests at the end are for tonight, *at* a computer. The answer key is on the last pages — do not peek until you are done.

Predict first, then check. Write down what you think *before* you look anything up or type anything in — even for the Console Quests. A guess that turns out wrong teaches you far more than an answer you only watched the computer produce.

Part A • Do You Remember?

RECALL ≈ 4 min

A1. Tuples are written with _____ brackets and are _____ (cannot be changed). Lists are written with _____ brackets and are _____.

A2. Which list operation adds an element at the end *and changes the list*? _____

And which returns a sorted *new* list, leaving the original alone? _____

A3. `s.split()` turns a _____ into a _____, and `' '.join(L)` turns a _____ into a _____.

A4. A sequence with 5 elements has its first element at index _____ and its last element at index _____. The last element can also be reached with the negative index _____.

A5. A function can hand back several values at once by returning one _____ that holds them all. The caller's line `s, c, a = get_stats(nums)` then splits it into three names; this is called _____ the tuple.

A6. In `L.append(5)` the dot means that `append` is an operation that belongs to the _____ `L`. It does its work by _____ `L`, and the value it returns is _____. A change like this is called a side _____.

A7. `id(L)` gives a number that identifies the _____ that `L` points at in memory. After `L.clear()` that number is _____; after `L = []` it is _____.

A8. Circle **True** or **False**:

(a) If `t` is a tuple, `t[0] = 5` works fine. True / False

(b) `L[1:3]` includes the element at index `3`. True / False

(c) `L1 + L2` makes a new list and leaves both originals unchanged. True / False

(d) `sorted(L)` changes `L` itself. True / False

(e) `list(range(3))` is `[0, 1, 2, 3]`. True / False

(f) One list may hold a number, a string and another list. True / False

(g) `L.extend([8, 9])` adds exactly one element to `L`. True / False

(h) `"a b c".split()` gives a list of three strings. True / False

(i) In `for i in range(len(L)):` the range is worked out only once. True / False

(j) A function can change a list that was passed to it, without returning anything. True / False

Part B • What Does It Print?

TRACE ≈ 8 min

Write the output exactly, one value per line, in order. Write a list with its square brackets and commas, the way Python prints it.

B1.

```
L = [2, 'a', 4, [1, 2]]
print(len(L))
print(L[3])
```

Output:

_____

_____

B2. Two slices of the same list. They differ by one character each.

(a)

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

Output: _____ , _____

(b)

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

Output: _____ , _____

B3. A tuple, and a function that returns two values at once.

```
t = (8, "ok", 2.5, (1, 2))
print(len(t))
print(t[1])
print(t[-1])
print(t[0] + t[2])
```

Output: _____ , _____ , _____ , _____

```
def low_high(a, b):
    if a < b:
        return (a, b)
    return (b, a)
lo, hi = low_high(9, 4)
print(hi - lo)
print(low_high(2, 7))
```

Output: _____ , _____

B4. A list inside a list.

```
A = [1]
B = [2]
C = A + B
B.append(C)
print(B)
print(len(B))
D = A.append(3)
print(A, D)
```

Output:

_____

_____

_____

B5. From a string to a list and back.

```
s = "to be or not"
w = s.split()
print(len(w), w[-1])
print('-'.join(w))
d = "2026-10-05".split('-')
print(d)
print(d[0] + d[1])
```

Output:

_____

_____

_____

_____

B6. Sorting and reversing; a function that builds a list.

```
L = [3, 1, 2]
M = sorted(L)
print(L, M)
L.reverse()
print(L)
N = L.sort()
print(L, N)
```

Output: _____

_____ , _____

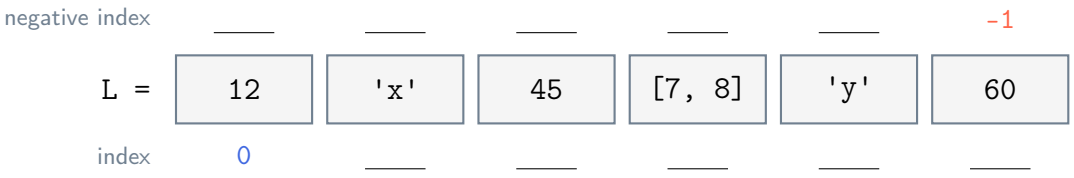
```
def plus_ten(L):
    new = []
    for e in L:
        new.append(e + 10)
    return new
A = [1, 2]
print(A, plus_ten(A))
```

Output: _____

Part C • The Index Ruler

TRACE ≈ 3 min

Positive indices go under the boxes, negative ones above. A tuple works the same way.



C1. Write the ten missing index numbers on the ruler.

C2. Read off the ruler:

`len(L)` → _____ `L[2]` → _____ `L[-2]` → _____ `L[-6]` → _____

`L[3]` → _____ `L[1:3]` → _____

C3. Now the other way round. Write an expression that gives:

(a) `60`, with a positive index: _____ and again with a negative index: _____

(b) the list `[45, [7, 8]]`: _____ the list `['y', 60]`: _____

C4. (a) One of `L[6]` and `L[-6]` is an error. Which one? _____ The other one gives: _____

(b) After the line `L[3] = 0` runs, `len(L)` is: _____

Part D • Does the List Change?

TRACE ≈ 3 min

D1. (a)

```
L1 = [2, 1, 3]
L2 = [4, 5]
L3 = L1 + L2
print(L1)
```

Output: _____

(b)

```
L1 = [2, 1, 3]
L2 = [4, 5]
L1.extend(L2)
print(L1)
```

Output: _____

D2.

```
L = [1, 2]
M = L.append(3)
print(L)
print(M)
```

Output:

_____

_____

D3. `append`, `extend` or `+`? Five separate experiments: **every row starts again from** `L = [1, 2]`. Write `L` after the line has run, and its length.

Line	`L` afterwards	`len(L)`
`L.append(3)`		
`L.append([3, 4])`		
`L.extend([3, 4])`		
`L.extend([[3, 4]])`		
`M = L + [3, 4]`		

Part E • Mutates, or Makes Something New?

CLASSIFY ≈ 5 min

E1. Tick whether each one **changes `L` itself** or **leaves `L` alone**. In the last column write what it hands back: `None`, a new list, or a number.

Expression	Changes `L`	Leaves `L` alone	It hands back
`L.append(7)`	☐	☐	
`sorted(L)`	☐	☐	
`L.sort()`	☐	☐	
`L + [7]`	☐	☐	
`L[0:2]`	☐	☐	
`L.reverse()`	☐	☐	
`L.clear()`	☐	☐	
`len(L)`	☐	☐	

E2. The life of one list. The program runs top to bottom, so each line starts from the list the line above left behind. Write `L` after each line, or *same* if the line did not change it.

	Line	`L` after this line
1	`L = [5, 2, 8]`	`[5, 2, 8]`
2	`L.append(1)`	
3	`L + [7]`	
4	`L.sort()`	
5	`L[0] = 9`	
6	`L.extend([3, 4])`	
7	`sorted(L)`	
8	`L.reverse()`	
9	`M = L[1:3]`	

At the end, `M` holds:

Two lines built a new list and then lost it, because nothing caught it. Their numbers:

E3. Same object, or a new one? Say `L = [3, 1, 2]` and `id(L)` is `5000`. Each line is tried on its own from that start. Tick **same** if `id(L)` is still `5000` afterwards, **new** if not.

Line	same	new
`L.append(4)`	☐	☐
`L = L + [4]`	☐	☐
`L.sort()`	☐	☐
`L = sorted(L)`	☐	☐

Line	same	new
`L[0] = 9`	☐	☐
`L = [3, 1, 2]`	☐	☐
`L.clear()`	☐	☐
`L = []`	☐	☐

Part F • Loops That Change the List

TRACE ≈ 4 min

F1. Fill in one row of the table for each round of the loop.

```
1 L = [3, 1, 4, 2]
2 for i in range(1, len(L)):
3     L[i] = L[i] + L[i - 1]
4 print(L)
```

Output: _____

Round	i	L after the round
1	____	_____
2	____	_____
3	____	_____

F2. Three loops, one line apart. Each one tries to add the double of every element to the end of the list.

(a)

```
1 L = [5, 6]
2 for i in range(len(L)):
3     L.append(L[i] * 2)
4 print(L)
```

Rounds: _____

Output: _____

(b)

```
1 L = [5, 6]
2 for e in L:
3     L = L + [e * 2]
4 print(L)
```

Rounds: _____

Output: _____

(c)

```
1 L = [5, 6]
2 for e in L:
3     L.append(e * 2)
4 print(L)
```

Rounds: _____

Output: _____

One of the three never stops. For that one, write `L` after each of its first three rounds:

_____ _____ _____

Part G • Lists and Frames

TRACE ≈ 5 min

Fill in every box the way the slides draw it: a frame row is `name : value`, and a function's value is `func`. **A list is never written inside a frame.** Write its elements in the row of boxes outside, and draw an arrow from the name to that row.

G1. The program pauses at the comment marked *: the loop has finished, but the call has not returned yet. Draw that moment.

```
1 def add_bonus(lst):
2     for i in range(len(lst)):
3         lst[i] = lst[i] + 5
4     # * pause here
5 quiz = [60, 75, 90]
6 r = add_bonus(quiz)
7 print(quiz, r)
```

Global Frame

_____ : _____

_____ : _____

_____ : _____

add_bonus frame

_____ : _____

_____ : _____

the list object



List objects at *: _____

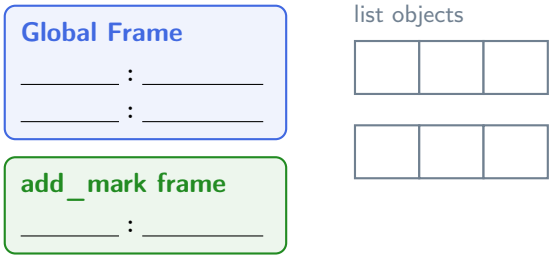
Line 7 prints:

_____

G2. One line apart. Both functions look as if they add a `5` to the caller's list; only one of them does. Draw each program at `*`. There are two rows of boxes for list objects: if a program has only one list object at `*`, cross the second row out. Leave spare boxes empty.

(a)

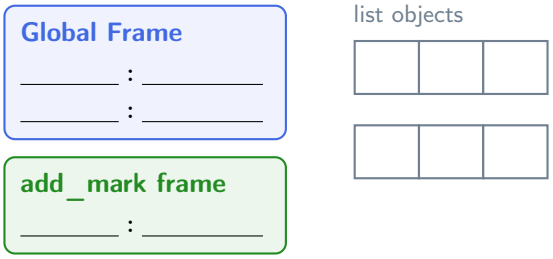
```
def add_mark(lst):
    lst.append(5)
    # * pause here
quiz = [60, 75]
add_mark(quiz)
print(quiz)
```



Output: _____

(b)

```
def add_mark(lst):
    lst = lst + [5]
    # * pause here
quiz = [60, 75]
add_mark(quiz)
print(quiz)
```



Output: _____

Part H • Strings to Lists and Back

TRANSLATE ≈ 4 min

H1. Write the value of each expression. Write a list with its brackets, and a string with its quotes.

(a) `"red green blue".split()` → _____

(b) `"red,green,blue".split(',')` → _____

(c) `"red,green,blue".split()` → _____

(d) `list("red")` → _____

(e) `'-'.join(['a', 'b', 'c'])` → _____

(f) `''.join(['a', 'b', 'c'])` → _____

(g) `' '.join("a-b-c".split('-'))` → _____

H2. Fill in the three blanks, so that the words come back in the opposite order: `reverse_words("you are how")` must give `'how are you'`.

```
def reverse_words(sen):
    words = sen._____()
    words._____()
    return ' '._____(words)
```

H3. Write **one expression** for each. `sen` is a sentence such as `"the quick brown fox"`, and `w` is a word such as `"cat"`.

(a) The number of words in `sen`. _____

(b) The last word of `sen`. _____

(c) The words of `sen` in alphabetical order, as a list. _____

(d) The letters of `w` with dashes between: `'c-a-t'`. _____

Part I • Spot the Bug

FIX ≈ 7 min

Each program has exactly one bug. Say what actually happens (wrong output, or the error name), then write the fix.

I1. Should return a new list with 1 added to each element. Instead `add_one([5])` returns `None`, and a longer list crashes.

```
1 def add_one(L):
2     new = []
3     for e in L:
4         new = new.append(e + 1)
5     return new
```

Fix: _____

I2. `t = (1, 2, 3)` followed by `t[0] = 99`. The second line is an error. Why, and what one-word change to line 1 would fix it?

_____

I3.

```
1 L = [3, 1, 2]
2 L = L.sort()
3 print(L[0])
```

Should print `1`.

Error: _____

Fix: _____

I4.

```
1 def double_all(L):
2     for e in L:
3         e = e * 2
4 nums = [1, 2, 3]
5 double_all(nums)
6 print(nums)
```

Should print `[2, 4, 6]`.

It prints: _____

Fix (the two lines of the loop):

_____

_____

I5.

```
1 def stats(L):
2     return sum(L)
3     return len(L)
4 r = stats([4, 6])
5 print(r)
```

Should print `(10, 2)`.

It prints: _____

Fix: _____

I6.

```
1 def count_words(sen):
2     return len(sen)
3 print(count_words("a cat sat"))
```

Should print `3`.

It prints: _____

Fix: _____

I7.

```
1 L = [1, 2]
2 L = L + 4
3 print(L)
```

Should print `[1, 2, 4]`.

Error: _____

Fix: _____

Part J • One Sentence Each

EXPLAIN ≈ 3 min

J1. In one sentence: how do `L.sort()` and `sorted(L)` differ?

J2. Why does `print(L.append(4))` show `None`, even though the `4` really was added?

J3. A `return` hands back one object. So how does `get_stats` from the slides give its caller a total, a count *and* an average?

J4. Tricky Example 1 stopped after four rounds; Tricky Example 2 never stopped. Both append to `L` inside the loop. What is the difference?

Part K • Write the Code

STRETCH ≈ 10 min, optional

Write each function in full. Work out the sample result on paper before you call it done.

K1. `smallest_largest(L)`: return a tuple that holds the smallest and the largest number in `L`, in that order, **without changing** `L`. `smallest_largest([4, 9, 1])` is `(1, 9)`. Then write one line that calls it and unpacks the result into `lo` and `hi`.

K2. `add_to_all(L, k)`: **mutate** `L` by adding `k` to every element, and return nothing. After `nums = [1, 2, 3]` and `add_to_all(nums, 10)`, `nums` is `[11, 12, 13]`.

K3. `initials(sen)`: return a string made of the first letter of every word of `sen`. `initials("Forman Christian University")` is `'FCU'`.

Console Quests

tonight · 5 min · at a computer

When unsure, just try it in the console. These are deliberately **not** answered anywhere on this sheet — the console is the answer key. Predict first, then type.

1. Type `t = (1, 2, 3)`, then `t[0] = 9`, and read the last line of the error. Then `a, b = t`, and read that one too. Now `type((5))` and `type((5,))`: what does the comma do?
2. Type `L = [4, 5, 6]` and `id(L)`. Then, with `id(L)` again after each one: `L.append(8)`, `L = L + [9]`, `L.clear()`, `L = []`. Which two steps changed the number?
3. Type `' '.join([1, 2, 3])` and read the error. Then `' '.join(['1', '2', '3'])`. What must every element be? Now `"a, b, c".split(',')`: look closely at the spaces.

Stuck on more than two items? Re-read the Lecture 11 slides on `append()`, on "List Mutation in Functions" and on the three Tricky Examples before the next class — then redo Parts D, E and F cold, drawing every list as a row of boxes. If you always know whether a line *changed* a list or *built a new one*, lists will stop surprising you.

Answer Key

Check your work *after* attempting everything. If an answer surprises you, that item is worth re-reading in the slides. Console Quests have no key — the console is the key.

A1. Tuples: round brackets `( )`, **immutable**. Lists: square brackets `[ ]`, **mutable**.

A2. `L.append(x)` mutates `L`; `sorted(L)` returns a new sorted list and leaves `L` exactly as it was.

A3. `split` turns a **string** into a **list** of words; `join` turns a **list of strings** back into a single **string**.

A4. `0`; `4`; `-1`. Indices start at 0, so the last one is always one less than the length. Negative indices count back from the end.

A5. one **tuple**; **unpacking**. The number of names on the left must match the number of elements in the tuple.

A6. the **object** `L` (everything in Python is an object, and an object has data and operations); by **mutating** (changing) `L`; `None`; a side **effect**.

A7. the **object**; **the same** (the same list, now empty); **different** (a brand-new empty list, and the name `L` now points at that one).

A8. (a) **False** — tuples are immutable, so that is an error. (b) **False** — a slice stops *just before* the second number. (c) **True** — `+` builds a brand-new list. (d) **False** — `sorted` returns a new list; `L.sort()` is the one that changes `L`. (e) **False** — it is `[0, 1, 2]`: `range(3)` stops before 3. (f) **True** — mixed types are allowed, though most lists hold one kind of thing. (g) **False** — two: `extend` adds each element of the list it is given. (h) **True** — `['a', 'b', 'c']`. (i) **True** — that is why Tricky Example 1 stopped. (j) **True** — `double_elements` on the slides did exactly that.

B1. `4` then `[1, 2]`. The length is **4, not 5** — the little list `[1, 2]` counts as *one* element, and `L[3]` hands that whole list back.

B2. (a) `[20, 30]` then `50`.
(b) `[20, 30, 40]` then `30`.
A slice `L[a:b]` stops *just before* index `b`, so `L[1:3]` gives two elements, not three. Negative indexing counts back from the end, with `-1` as the last element.

B3. Left: `4`, `ok`, `(1, 2)`, `10.5`. The inner tuple `(1, 2)` counts as one element, so the length is 4, and `t[-1]` hands that whole tuple back. `print` shows a string without its quotes. Right: `5`, then `(2, 7)`. `low_high(9, 4)` returns the single tuple `(4, 9)`; the line `lo, hi = ...` unpacks it, so `lo` is 4 and `hi` is 9. The last line does not unpack, so `print` shows the whole tuple, round brackets and all.

B4. `[2, [1, 2]]`, then `2`, then `[1, 3] None`. `A + B` builds a new list `[1, 2]`; `append` puts that whole list into `B` as *one* element, so `B` has length 2, not 3. The last `append` changes `A` and hands back nothing, so `D` is `None`.

B5. `4 not`, then `to-be-or-not`, then `['2026', '10', '05']`, then `202610`. `split` cuts at the spaces and gives four words; `join` glues them back with a dash between each pair. `split('-')` cuts at the dashes instead, and what it gives is always *strings*, even when they look like numbers: so `+` glues `'2026'` and `'10'` together instead of adding them.

B6. Left: `[3, 1, 2] [1, 2, 3]`, then `[2, 1, 3]`, then `[1, 2, 3] None`. `sorted` leaves `L` alone and hands back a new list; `reverse` and `sort` change `L` itself and hand back nothing, so `N` is `None`. Note that `reverse` turns the list around; it does not sort it backwards. Right: `[1, 2] [11, 12]`. The function starts from an empty list and appends once per round, then returns that new list. It never writes into its parameter, so the caller's `A` is untouched.

C1. Below the boxes: `0 1 2 3 4 5`. Above them: `-6 -5 -4 -3 -2 -1`. The inner list `[7, 8]` sits in one box, so it has one index.

C2. `6`; `45`; `'y'`; `12`; `[7, 8]`; `['x', 45]`. An index hands back one element, whatever it is. A slice always hands back a list, here the elements at indices 1 and 2.

C3. (a) `L[5]` and `L[-1]`. (b) `L[2:4]` and `L[4:6]`. The second number of a slice is one past the last index you want, so a slice that reaches the end of this list stops at 6.

C4. (a) `L[6]` is the error (`IndexError`: list index out of range), because six elements have the indices 0 to 5. `L[-6]` is fine: counting back six from the end lands on the first element, `12`. (b) Still `6`. The line puts a new value into one box; no box is added or removed. The inner list is replaced by the single number 0.

D1. (a) `[2, 1, 3]` — `+` builds a *new* list and puts it in `L3`; `L1` is untouched.
(b) `[2, 1, 3, 4, 5]` — `extend` mutates `L1` in place. Only one line changed.

D2. `[1, 2, 3]` then `None`. `append` does its job by *mutating* `L` and returns nothing at all, so `M` is `None`. Never write `M = L.append(...)`.

D3. `[1, 2, 3]`, length 3. `[1, 2, [3, 4]]`, length 3. `[1, 2, 3, 4]`, length 4. `[1, 2, [3, 4]]`, length 3. `[1, 2]`, length 2. `append` adds exactly *one* element, whatever you give it, even a whole list. `extend` adds each element of the list you give it; in row 4 that list has a single element, which is itself a list. `+` does not touch `L` at all: the new list goes to `M`.

E1. **Changes `L`:** `append`, `sort`, `reverse`, `clear`. All four hand back `None`. **Leaves `L` alone:** `sorted`, `L + [7]`, `L[0:2]` (each hands back a new list) and `len` (hands back a number).
Watch the pair `sorted(L)` and `L.sort()`: they read almost the same but do opposite things. A slice always hands back a new list. The pattern in the table: an operation that changes the list hands back `None`, and an operation that hands something back leaves the list alone.

E2. Line 2: `[5, 2, 8, 1]`. 3: same. 4: `[1, 2, 5, 8]`. 5: `[9, 2, 5, 8]`. 6: `[9, 2, 5, 8, 3, 4]`. 7: same. 8: `[4, 3, 8, 5, 2, 9]`. 9: same. `M` holds `[3, 8]`, the elements at indices 1 and 2 of the reversed list. Lines **3** and **7** each build a new list, but no name is given to it, so it is lost at once and the line has no effect. Line 9 also builds a new list and leaves `L` alone, but `M` catches it.

E3. **Same object:** `L.append(4)`, `L.sort()`, `L[0] = 9`, `L.clear()`. **New object:** `L = L + [4]`, `L = sorted(L)`, `L = [3, 1, 2]`, `L = []`. A dot operation, or `L[i] = ...`, reaches into the object that `L` already points at. A line that starts with `L =` points the name at whatever the right-hand side built, and each right-hand side here builds a new list. Even `L = [3, 1, 2]` gives a new object: the same contents, in a different place in memory.

F1. Round 1: `i` is 1, `[3, 4, 4, 2]`. Round 2: `i` is 2, `[3, 4, 8, 2]`. Round 3: `i` is 3, `[3, 4, 8, 10]`. Output: `[3, 4, 8, 10]`. Each round uses the element to its left *as it is now*, already changed by the round before, so every box ends up holding the total so far. The loop starts at 1 because the first element has nothing to its left.

F2. (a) **2** rounds, `[5, 6, 10, 12]`. `range(len(L))` is worked out once, while `L` has two elements, so `i` takes 0 and 1 and the growing list makes no difference. (b) **2** rounds, `[5, 6, 10, 12]`. The loop walks the original list object. Each round `L + [...]` builds a new list and points the name `L` at it, but the loop keeps walking the old object, which still holds only 5 and 6. (c) **Never stops**, so `print` never runs and there is no output. `append` makes the very list being walked longer: `[5, 6, 10]`, then `[5, 6, 10, 12]`, then `[5, 6, 10, 12, 20]`. Every round adds one more element to visit, so `e` never reaches the end.

G1. Global: `add_bonus : func`, and `quiz` with an arrow to the row of boxes. There is no `r` yet, because the call has not returned. `add_bonus` frame: `lst` with an arrow to the *same* row of boxes, and `i : 2`, its last value. The boxes hold `65`, `80`, `95`. There is only **1** list object: `quiz` and `lst` are two names for it, so `lst[i] = ...` changes the list that the caller sees. Line 7 prints `[65, 80, 95] None`: the function has no `return`, so `r` is `None`.

G2. (a) **One** list object, so the second row is crossed out. `quiz` (Global) and `lst` (`add_mark` frame) both point at it, and `append` has changed it to `60`, `75`, `5`. Output: `[60, 75, 5]`. (b) **Two** list objects. `lst + [5]` builds a new list `60`, `75`, `5`, and the assignment points the local name `lst` at it. `quiz` still points at the original `60`, `75`, which nothing has touched. Output: `[60, 75]`; the new list is lost when the frame is destroyed. Giving a parameter a new value never reaches the caller (`bump(n)` in Finger Exercises 10). Mutating the object it points at does.

H1. (a) `['red', 'green', 'blue']` (b) the same list again, because this time the cut is at each comma (c) `['red,green,blue']`, a list with *one* element: `split()` with empty brackets cuts at spaces, and this string has none (d) `['r', 'e', 'd']` (e) `'a-b-c'` (f) `'abc'`, with an empty string between the pieces (g) `'a b c'`: cut at the dashes, then glue with spaces.

H2. `split`, `reverse`, `join`: string to list, turn the list around, list back to string. The middle line stands alone because `reverse` mutates `words` and returns `None`.

H3. (a) `len(sen.split())` (b) `sen.split()[-1]` (c) `sorted(sen.split())` (d) `'-'.join(list(w))`; plain `'-'.join(w)` also works. In (a) to (c) the first step is the same: turn the sentence into a list of words, then use a list tool on it.

I1. The line should be just `new.append(e + 1)`. `append` returns `None`, so re-assigning `new` throws the list away on the very first pass. On a second pass `new` is `None`, and `None` has no `append`: an `AttributeError`.

I2. A tuple is **immutable**, so no element of it can ever be re-assigned — this is an error. If you need to change elements, use a list: `t = [1, 2, 3]`.

I3. `TypeError` ("'NoneType' object is not subscriptable"). `sort` rearranges `L` and returns `None`; line 2 then points `L` at that `None`, and `None[0]` is meaningless. Write `L.sort()` on its own, or `L = sorted(L)`.

I4. Prints `[1, 2, 3]`. `e` is a separate name that is given each element in turn; `e = e * 2` points that name at a new number and never writes into the list. To change the list, assign to a position in it: `for i in range(len(L)):` then `L[i] = L[i] * 2`.

I5. Prints `10`. The first `return` ends the call, so the second one never runs. To hand back two values, put both in one tuple: `return (sum(L), len(L))`.

I6. Prints `9`. `len` of a string counts its *characters*, and the two spaces count too. Turn the sentence into a list of words first: `return len(sen.split())`.

I7. `TypeError` ("can only concatenate list (not "int") to list"). `+` joins a list to another *list*, and `4` is a number. Wrap it in brackets, `L = L + [4]`, or write `L.append(4)`.

J1. `L.sort()` rearranges `L` itself and hands back nothing; `sorted(L)` leaves `L` alone and hands back a new sorted list.

J2. Because `append` works by *side effect*: it changes the list and returns nothing, so the value handed to `print` is `None`.

J3. It packs the three values into one tuple and returns that single object; the caller indexes into it or unpacks it into three names.

J4. Example 1 loops over `range(len(L))`, which is worked out once and never grows; Example 2 loops over `L` itself, and every `append` gives that loop one more element to visit.

K1. `s = sorted(L)` / `return (s[0], s[-1])`. Call: `lo, hi = smallest_largest([4, 9, 1])`. It must be `sorted`, not `L.sort()`: sorting in place would rearrange the caller's list. (Using `max(L)` for the second value is fine too.)

K2. `for i in range(len(L)):` / `L[i] = L[i] + k`. No `return` line at all. The loop must go over the *indices*: only `L[i] = ...` writes into the list (see I4).

K3. `result = ""` / `for w in sen.split():` / `result = result + w[0]` / then `return result` after the loop. `split` gives the words, and `w[0]` is the first character of each one.