Finger Exercises 12

Aliasing and Cloning

≈ 55 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. Almost every item is one of three questions in disguise: *how many list objects are there, which names point at each one, and which object does this line change?*

How to do it. On **paper**, **no computer**, in one sitting. Draw as you go: a box for every list object, and an arrow from every name to its box. `M = L` adds an arrow and no box; `M = L[:]` adds a box. If an item takes you more than a minute, skip it and come back. Parts A–K are for *everyone*; Part L is a **stretch**. The Console Quests 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. `Lnew = Loriginal` does *not* make a copy. The two names now point at the same object — we say `Lnew` is an _____ of `Loriginal`.

A2. Three ways to actually copy a list `L`: 1. `L_____` 2. `_____(L)` 3. `L._____`

A3. A **shallow** copy of a nested list makes a new _____ list, but the _____ lists are still shared. For full independence use _____.

A4. `==` asks whether two lists have the same _____. `is` asks whether two names point at the same _____. For a list and its alias, `is` gives _____. For a list and its clone, `is` gives _____ and `==` gives _____.

A5. `del L[i]` and `L.pop(i)` both remove by _____. `L.remove(x)` removes by _____, and only the _____ match it finds. Of the three, the only one that hands the removed item back is _____.

A6. Circle **True** or **False**:

(a) `del L[0]` hands back the item it removed. True / False

(b) `L.remove(20)` removes every `20` in `L`. True / False

(c) It is safe to delete items from a list while looping over it. True / False

(d) `L.pop()` with no index removes and returns the *last* item. True / False

(e) `a == b` is `True` only when `a` and `b` are the same object. True / False

(f) After `M = L[:]`, the test `M is L` gives `False`. True / False

(g) `L.remove(7)` is an error when `L` holds no `7`. True / False

(h) `copy.deepcopy` copies the inner lists as well as the outer list. True / False

Part B • One Character Apart

TRACE ≈ 4 min

Write the output exactly.

B1. Versions (a) and (b) differ by two characters on line 2.

(a)

```
1 L = [4, 5, 6]
2 M = L
3 M[0] = 10
4 print(L)
```

Output: _____

(b)

```
1 L = [4, 5, 6]
2 M = L[:]
3 M[0] = 10
4 print(L)
```

Output: _____

B2.

```
1 nested = [[1, 2], [3, 4]]
2 L = nested[:]
3 L[0][1] = 5
4 print(nested)
```

Output: _____

B3. Versions (a) and (b) differ on line 3 only. Both lines look as if they sort `M`.

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

(a) Output: _____

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

(b) Output: _____

Part C • Draw the Arrows

DRAW ≈ 9 min

C1. Draw an arrow from each **name** to the object it points at after line 2.

```
1 A = [1, 2, 3]
2 B = A
3 B.append(4)
```

A B [1, 2, 3]

Objects: ____ print(A): _____

```
1 A = [1, 2, 3]
2 B = list(A)
3 B.append(4)
```

A B [1, 2, 3] [1, 2, 3]

Objects: ____ print(A): _____

C2. Count the boxes. Each row is a small program of its own, run from left to right. How many **list objects** are there when it ends? An inner list is an object too.

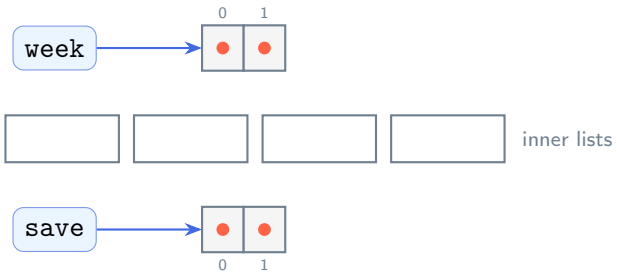
	Line 1	Line 2	Line 3	List objects
1	`A = [1, 2]`	`B = A`	`C = B`	_____
2	`A = [1, 2]`	`B = A[:]`	`C = B[:]`	_____
3	`A = [[1], [2]]`	`B = A`		_____
4	`A = [[1], [2]]`	`B = A[:]`		_____
5	`A = [[1], [2]]`	`B = copy.deepcopy(A)`		_____
6	`A = [1, 2]`	`B = [A, A]`		_____
7	`A = [1, 2]`	`B = [A[:], A[:]]`		_____

C3. A nested list as the slides draw it: an outer list with one dot for each position, and from each dot an arrow to an inner list. Finish the picture as it is **after line 4**: fill in the inner boxes and draw an arrow from each of the four dots. One box is spare: cross it out.

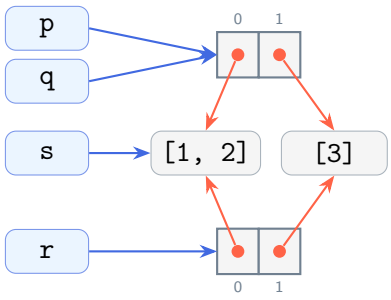
```
1 week = [[8, 9], [6, 7]]
2 save = week[:]
3 save[0][0] = 5
4 save[1] = [1, 1]
```

print(week): _____

print(save): _____



C4. Now the other way round: the picture is finished, and you read it.



(a) `True` or `False`? `p is q` _____ `p is r` _____ `p == r` _____ `r[0] is s` _____

(b) List objects in the picture: _____

(c) The picture was built by `p = [[1, 2], [3]]` and three more lines. Finish them.

`q =` _____ `r =` _____ `s =` _____

(d) The line `s.append(0)` runs. Circle each `print` that now shows a `0`:

`print(p)` `print(q)` `print(r)`

(e) Start again from the picture. The line `q[1] = [9]` runs.

`print(p)`: _____ `print(r)`: _____

Part D • Does the Original Change?

CLASSIFY ≈ 5 min

D1. Each row starts `A = [[1, 2], [3, 4]]`, makes `B` as shown, then runs `B[0][1] = 99`.

How `B` was made	`A` changes	`A` is safe
`B = A`	☐	☐
`B = A[:]`	☐	☐
`B = list(A)`	☐	☐
`B = A.copy()`	☐	☐
`B = copy.deepcopy(A)`	☐	☐

D2. Now it is the other way round. `B` is made the same way in every row, as the shallow copy `B = A[:]` of `A = [[1, 2], [3, 4]]`, and the line that follows changes. Every row starts again. Tick one box. If `A` changes, write `A` afterwards.

Line after `B = A[:]`	`A` changes	`A` is safe	`A` afterwards, if it changed
`B[0] = [9, 9]`	☐	☐	_____
`B[0][0] = 9`	☐	☐	_____
`B.append([5, 6])`	☐	☐	_____
`B[1].append(5)`	☐	☐	_____
`B[1] = B[1] + [5]`	☐	☐	_____
`B[1].reverse()`	☐	☐	_____
`del B[0]`	☐	☐	_____
`B[0].clear()`	☐	☐	_____

D3. One word apart, and this time it is the *original* that changes after the copy was made. `import copy` has run; `copy.copy(n)` is the shallow copy, the same as `n[:]`.

```
1 n = [[1], [2]]
2 s = copy.copy(n)
3 n[0].append(9)
4 n.append([3])
5 print(s)
```

(a) Output: _____

```
1 n = [[1], [2]]
2 d = copy.deepcopy(n)
3 n[0].append(9)
4 n.append([3])
5 print(d)
```

(b) Output: _____

Part E • == or is?

CLASSIFY ≈ 4 min

E1. The eight lines on the left have run. Write `T` (True) or `F` (False) for each test. Draw the picture first: it answers every row.

```
1 import copy
2 a = [1, 2]
3 b = a
4 c = a[:]
5 d = [1, 2]
6 n = [[1], [2]]
7 s = n[:]
8 t = copy.deepcopy(n)
```

Test	T or F	Test	T or F
`a == b`	_____	`s == n`	_____
`a is b`	_____	`s is n`	_____
`a == c`	_____	`s[0] is n[0]`	_____
`a is c`	_____	`t == n`	_____
`a == d`	_____	`t is n`	_____
`a is d`	_____	`t[0] is n[0]`	_____
`c is d`	_____	`t[0] == n[0]`	_____

E2. (a) Now `b.append(3)` runs. Which answers in the **left** table flip? _____

(b) Then `n[0].append(9)` runs. Which answers in the **right** table flip? _____

Part F • By Position or by Value?

TRACE ≈ 5 min

F1. Every row starts again from `L = [2, 0, 1, 3]`. Each value in this list could also be an index, so decide first: is the number in the line a position or a value? Write `L` afterwards, and `x` where the line has one. One row is an error: write *error* for it.

Line	L afterwards	x	Line	L afterwards	x
`del L[1]`	_____	—	`x = L.pop(0)`	_____	_____
`x = L.pop(1)`	_____	_____	`L.remove(0)`	_____	—
`x = L.remove(1)`	_____	_____	`del L[3]`	_____	—
`x = L.pop()`	_____	_____	`L.remove(4)`	_____	—

F2. Say it in Python. Write **one line** for each job. `L` is a list of at least three numbers.

(a) Take the last item off `L` and keep it in `last`. _____

(b) Remove the first `0` that `L` holds. _____

(c) Remove the item at index `1`. You will not need it again. _____

(d) Empty the list, so that every alias of `L` sees it empty too. _____

(e) Make `M` a clone of `L`. _____

(f) Ask whether `M` and `L` are one object. _____

F3. All three ways of deleting, used through an alias and through a clone. Write the output exactly.

```
1 L = [5, 6, 7, 8]
2 M = L
3 N = L[:]
4 x = M.pop()
5 del N[0]
6 L.remove(6)
7 print(L, N, x)
```

Which two names share one list?

_____

Output:

_____

Part G • The Skipping Pointer

TRACE ≈ 8 min

A `for` loop over a list keeps a pointer. It looks at index 0, then 1, then 2, and it stops when the pointer is past the end of the list *as the list is at that moment*. When an item is removed, everything to its right slides one place left, and the pointer still moves on by one.

G1. Fill in one row for each round of the loop. Round 1 is done for you.

```
1 def remove_all(L, e):
2     for item in L:
3         if item == e:
4             L.remove(item)
5
6 L = [1, 2, 2, 3, 2, 2]
7 remove_all(L, 2)
8 print(L)
```

Rounds: _____

Output: _____



Round	index	item	removed?	L afterwards
1	0	1	no	[1, 2, 2, 3, 2, 2]
2	____	____	_____	_____
3	____	____	_____	_____
4	____	____	_____	_____

G2. The same `remove_all(L, 2)` on six more lists. No table this time, use what you just saw: after a removal, the item that slides into the gap is skipped. Write what is left of `L`, and tick the row if every 2 is gone.

L at the start	What is left	all gone?
[2, 2]	_____	☐
[2, 1, 2]	_____	☐
[2, 2, 2, 2]	_____	☐

L at the start	What is left	all gone?
[1, 2, 3, 2]	_____	☐
[2, 2, 1]	_____	☐
[2, 3, 2, 2]	_____	☐

The function gives the right answer only when the list has no _____.

G3. Four more attempts at `remove_all`. Each is run the same way: `L = [1, 2, 2, 2]`, then `remove_all(L, 2)`, then `print(L)`. The right answer is `[1]`. Write what each one prints, or *error*, and tick it if it is right.

(a)

```
1 def remove_all(L, e):
2     while e in L:
3         L.remove(e)
```

Prints: _____ right? ☐

(b)

```
1 def remove_all(L, e):
2     for i in range(len(L)):
3         if L[i] == e:
4             del L[i]
```

Prints: _____ right? ☐

(c)

```
1 def remove_all(L, e):
2     keep = L
3     L.clear()
4     for item in keep:
5         if item != e:
6             L.append(item)
```

Prints: _____ right? ☐

(d)

```
1 def remove_all(L, e):
2     new = []
3     for item in L:
4         if item != e:
5             new.append(item)
6     L = new
```

Prints: _____ right? ☐

Part H • Is the Caller's List Safe?

CLASSIFY ≈ 5 min

H1. Six versions of a function `f`. Each is called the same way: `nums = [3, 1, 2]` and then `r = f(nums)`. In the table write `nums` and `r` after the call, and `True` or `False` for `r is nums`.

(a)

```
def f(L):
    L.sort()
    return L
```

(b)

```
def f(L):
    M = L[:]
    M.sort()
    return M
```

(c)

```
def f(L):
    M = L[:]
    L.sort()
    return M
```

(d)

```
def f(L):
    return sorted(L)
```

(e)

```
def f(L):
    L = L[:]
    L.pop()
    return L
```

(f)

```
def f(L):
    L.pop()
    return L[:]
```

	(a)	(b)	(c)	(d)	(e)	(f)
`nums`	_____	_____	_____	_____	_____	_____
`r`	_____	_____	_____	_____	_____	_____
`r is nums`	_____	_____	_____	_____	_____	_____

H2. After version (a), the caller runs `r.append(0)`. What is `nums` now? _____

Why? _____

Part I • Follow Every Name

TRACE ≈ 3 min

Write the output exactly. Keep a picture going beside each program: one box for each list, and move an arrow only when a line says `name = ...`

I1. In (a) one name is given a new list halfway down. In (b) a shallow copy is followed by three assignments, and their order matters.

(a)

```
a = [1, 2]
b = a
c = b
c.append(3)
b = [7]
b.append(8)
print(a, b, c)
```

Output: _____

(b)

```
n = [[1, 2], [3, 4]]
L = n[:]
L[1] = [0, 0]
L[1][0] = 5
L[0][0] = 6
print(n)
print(L)
```

Output: _____

_____

Part J • Spot the Bug

FIX ≈ 7 min

J1 is the pair from the slides. From J2 on, each program has exactly one bug: say what actually happens (the wrong output, or the name of the error), then write the fix.

J1. Both should have left `[1]` and `[30, 40]`. Circle the offending line and write the fix.

(a) gives `[1, 2]`

```
def remove_all(L, e):
    for item in L:
        if item == e:
            L.remove(item)
```

Fix: _____

(b) gives `[20, 30, 40]`

```
def remove_dups(L1, L2):
    L1_copy = L1
    for e in L1_copy:
        if e in L2:
            L1.remove(e)
```

Fix: _____

J2.

```
def last_two_sum(L):
    a = L.pop()
    b = L.pop()
    return a + b
bill = [40, 25, 10]
print(last_two_sum(bill))
print(bill)
```

Should print `35`, then `[40, 25, 10]`.

It prints: _____ , then _____

Fix: _____

J3.

```
scores = [[5, 8], [9, 4]]
before = scores.copy()
scores[1][1] = 0
print(before)
```

Should print the scores as they were: `[[5, 8], [9, 4]]`.

It prints: _____

Fix: _____

J4. Two short ones.

(a) should print `3`

```
L = [5, 3, 8]
gone = L.remove(3)
print(gone)
```

It prints: _____

Fix: _____

(b) should print `[10, 30]`

```
L = [10, 20, 30]
L.pop(20)
print(L)
```

Error: _____

Fix: _____

J5.

```
a = [1, 2, 3]
b = [1, 2, 3]
if a is b:
    print("same marks")
else:
    print("different")
```

Should print `same marks`.

It prints: _____

Fix: _____

Part K • One Sentence Each

EXPLAIN ≈ 2 min

K1. Why is `L1_copy = L1` not good enough when you plan to remove items from `L1`?

K2. You shallow-copied a nested list, then changed an inner list. Why did the original change too?

K3. A function `def f(L):` is called as `f(marks)`. Why can a line inside `f` change `marks`, when `f` never uses the name `marks`?

Part L • Write the Code

STRETCH ≈ 10 min, optional

Write each function in full. Run the sample call on paper before you call it done.

L1. `drop_lowest(marks)`: return a **new** list that holds every mark except the lowest one. If the lowest mark is there twice, drop it once. **The caller's list must not change.** `drop_lowest([7, 4, 9, 4])` is `[7, 9, 4]`.

L2. `remove_negatives(L)`: **mutate** `L` so that no negative number is left in it, and return nothing. After `nums = [3, -1, -2, 5]` and `remove_negatives(nums)`, `nums` is `[3, 5]`. Trace your loop on this very list: its two negatives stand next to each other.

L3. `deep_clone(nested)`: `nested` is a list of lists of numbers. Return a copy that shares *nothing* with it, written by hand, **without** the `copy` module. After `d = deep_clone(n)`, no change made through `d` may show in `n`.

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. Open `pythontutor.com`, paste the four lines of C3 and press *Next* one line at a time. Does the last picture match the one you drew? Then change line 2 to `save = week` and run it again.
2. Type `import copy`, `A = [[1], [2]]`, `B = A[:]` and `C = copy.deepcopy(A)`. Now `id(A[0])`, `id(B[0])` and `id(C[0])`: which two numbers match? Check with `A[0] is B[0]` and `A[0] is C[0]`.
3. Type `L = [1, 2, 3, 4, 5, 6]`, then the loop `for x in L:` with the one line `L.remove(x)` inside it, then `L`. Every item was meant to go. Which ones are left, and why those? Then read the last line of the error from each of `L.pop(9)`, `L.remove(9)` and `del L[9]`.

Stuck on more than two items? Re-read the Lecture 12 slides "One List or Two?", "Tricky: Mutating While Iterating" and the two pictures of a nested list before the next class — then redo Parts C, D and G cold, with a pencil picture beside every program. If you always know *how many list objects there are* and *which names point at each*, aliasing 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. an **alias**. Assignment (=) never copies a list; it only gives the same object a second name.

A2. `L[:]` (slicing), `list(L)`, and `L.copy()`.

A3. a new **outer** list, but the **inner** lists are still shared; use `copy.deepcopy(L)`.

A4. the same **contents**; the same **object**. Alias: `True`. Clone: `is` gives `False` and == gives `True`. So == cannot tell an alias from a clone, and `is` can.

A5. by **position** (index); by **value**, and only the **first** match; `pop`.

A6. (a) **False** — `del` just deletes; `L.pop(0)` is the one that hands the item back. (b) **False** — only the *first* 20. (c) **False** — never add to or delete from a list while iterating over it. (d) **True**. (e) **False**: == compares contents, so two separate lists with equal elements also give `True`. `is` is the test for the same object. (f) **True**: a slice builds a second list. (g) **True**. (h) **True**: that is the whole difference from a shallow copy.

B1. (a) `[10, 5, 6]` — `M` and `L` are two names for *one* list, so changing one changes both.
(b) `[4, 5, 6]` — `L[:]` builds a second list, so `M` can be changed on its own. Those two characters are the whole lesson of this lecture.

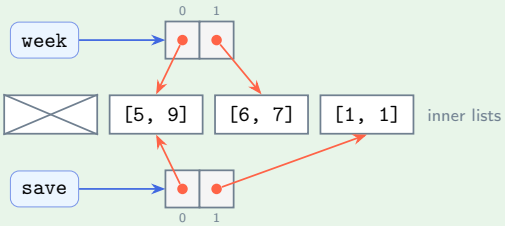
B2. `[[1, 5], [3, 4]]`. This is the trap: `nested[:]` is a *shallow* copy, so the new outer list still points at the very same two inner lists. Reaching in with `L[0][1]` therefore changes `nested` too. Only `copy.deepcopy` would have protected it.

B3. (a) `[1, 2, 3] [1, 2, 3]`. `sort` rearranges the one list that both names point at.
(b) `[3, 1, 2] [1, 2, 3]`. `sorted` builds a new list, and `M =` points the name `M` at it. From that line on `M` is no longer an alias of `L`: an alias lasts only until the name is given something else.

C1. **Left:** one arrow from `A` and one from `B`, both to the *same* box — **1** list object, and `print(A)` shows `[1, 2, 3, 4]`. `B` is an alias, so appending through `B` is appending to `A`.
Right: `A` to the top box, `B` to the bottom box — **2** list objects, and `print(A)` shows `[1, 2, 3]`. `list(A)` built a real copy.

C2. **1, 3, 3, 4, 6, 2, 4.** Count boxes, not names. A plain name on the right of = adds no box (rows 1 and 3; row 3 has one outer list and two inner lists). Each `[:]` adds exactly one box, an outer list (rows 2, 4 and 7). `deepcopy` adds a box for the outer list and one more for every inner list (row 5). In row 6, `B` is one new list whose two positions both point at `A`: 2 boxes. In row 7 each position gets a clone of its own: 4 boxes.

C3. `print(week)`: `[[5, 9], [6, 7]]`. `print(save)`: `[[5, 9], [1, 1]]`. The picture has five list objects: two outer and three inner.



`save[0][0] = 5` goes through `save`'s position 0 to an inner list that `week` shares, so `week` shows the 5 too. `save[1] = [1, 1]` puts a new list into `save`'s *own* position 1; position 1 of `week` still points at `[6, 7]`. (Which of your boxes holds which list does not matter. The arrows do.)

C4. (a) `True`, `False`, `True`, `True`. `p` and `q` have arrows to one outer list; `r` has its own, with equal contents. `r[0]` and `s` both lead to the one `[1, 2]`. (b) **4**: two outer, two inner. (c) `q = p`, `r = p[:]`, `s = p[0]` (`q[0]` or `r[0]` is the same object). (d) All three. There is only one list `[1, 2, 0]`, and every outer list has an arrow to it. (e) `[[1, 2], [9]]` and `[[1, 2], [3]]`. `q` is `p`, so `p` shows the new list. `r` has its own outer list, and its position 1 still points at `[3]`.

D1. `A` changes in the **first four** rows; only `copy.deepcopy(A)` keeps `A` safe.
Rows 2–4 are the discriminator: `[:]`, `list()` and `.copy()` are all real copies, but only of the *outer* list — the inner lists are still shared, and `B[0][1]` reaches an inner list.

D2. `A` changes in rows 2, 4, 6 and 8: `[[9, 2], [3, 4]]`, `[[1, 2], [3, 4, 5]]`, `[[1, 2], [4, 3]]` and `[[], [3, 4]]`. It is safe in rows 1, 3, 5 and 7.

The rule: `B[i] = ...`, `del B[i]` and `B.something()` work on `B`'s own outer list, which `A` does not share. `B[i][j] = ...` and `B[i].something()` first follow position `i` to an inner list, and the inner lists are shared. Row 5 is the one to study: `+` builds a new list, and that list is stored in `B`'s own position 1, so `A` never sees the 5.

D3. (a) `[[1, 9], [2]]`. The shallow copy shares the inner lists, so it sees the `9`. It has its own outer list, so it does not see the `[3]` that was appended to `n`'s outer list. A shallow copy is independent one level deep and no further.
(b) `[[1], [2]]`. The deep copy shares nothing with `n`, so nothing done to `n` shows in it.

E1. Left table: `True`, `True`, `True`, `False`, `True`, `False`, `False`. Right table: `True`, `False`, `True`, `True`, `False`, `False`, `True`.
Every `==` is `True`, because all these lists were built with equal contents. `is` is `True` only where two arrows end at one box: `a` and `b`, and the inner list that `s` and `n` share. `d` was typed out again, so it is a separate object that happens to look the same. `s[0] is n[0]` against `t[0] is n[0]` is exactly shallow against deep.

E2. (a) `a == c` and `a == d` flip to `False`: `a` is now `[1, 2, 3]`, and `c` and `d` still hold `[1, 2]`. `a == b` stays `True`, because they are one object. (b) `t == n` and `t[0] == n[0]` flip to `False`. `s == n` stays `True`: `s` shares that inner list, so it changed for both. No `is` answer flips, here or ever: changing what is inside an object does not change which object a name points at.

F1. Left table: `[2, 1, 3]`. `[2, 1, 3]`, `x` is `0`. `[2, 0, 3]`, `x` is `None`. `[2, 0, 1]`, `x` is `3`. Right table: `[0, 1, 3]`, `x` is `2`. `[2, 1, 3]`. `[2, 0, 1]`. *error* (`ValueError`: there is no `4` in the list).
`del L[1]` and `L.pop(1)` take out whatever sits at index 1, which is the `0`. `L.remove(1)` looks for the *value* 1 and finds it at index 2. Only `pop` hands the item back; `remove` returns `None`. Three different lines (`del L[1]`, `L.pop(1)` and `L.remove(0)`) leave the same list.

F2. (a) `last = L.pop()` (b) `L.remove(0)` (c) `del L[1]` (`L.pop(1)` also works; it hands back an item that you then ignore) (d) `L.clear()`. Not `L = []`: that points the name `L` at a new empty list, and the aliases still see the old full one. (e) `M = L[:]`, or `M = list(L)`, or `M = L.copy()` (f) `M is L`

F3. `L` and `M` share one list; `N` is a clone. Output: `[5, 7] [6, 7, 8] 8`. `M.pop()` takes the `8` off the shared list and hands it to `x`, and `L.remove(6)` takes the `6` out of that same list. `N` was cloned while the list still had four items, and it loses only its own first item.

G1. Round 2: index 1, `item` is 2, removed, `[1, 2, 3, 2, 2]`. Round 3: index 2, `item` is 3, not removed, same list. Round 4: index 3, `item` is 2, removed, `[1, 3, 2, 2]`. **4** rounds, output `[1, 3, 2, 2]`. After round 2 the second 2 slid into index 1, but the pointer went on to index 2, so that `2` was never looked at. After round 4 the pointer moves to index 4, and the list now has only four items (indices 0 to 3), so the loop ends with two 2s still in the list. A list of six items got only four rounds.

G2. Left table: `[2]`; `[1]`, all gone; `[2, 2]`. Right table: `[1, 3]`, all gone; `[2, 1]`; `[3, 2]`. It is right only when the list has no **two 2s next to each other**: the second of such a pair slides into the gap and is skipped. So `[2, 1, 2]` and `[1, 2, 3, 2]` come out right, by luck. A bug that hides on some inputs is the dangerous kind: two tests passed, and the function is still wrong.

G3. (a) `[1]`, **right**. No `for` loop walks the list, so there is no pointer to skip anything; `e in L` looks at the whole list afresh each time. (b) *error*: `IndexError`. `range(len(L))` is worked out once, as 0 to 3 (Lecture 11), but two deletions leave `[1, 2]`, and then `L[3]` does not exist. (c) `[]`. `keep = L` is an alias, not a clone, so `L.clear()` empties the one list and the loop has nothing to walk. It needs `keep = L[:]`. (d) `[1, 2, 2, 2]`. The new list is built correctly, but `L = new` only points the local name `L` at it. The caller's list was never touched, and the new one is lost when the call ends (`add_mark` in Finger Exercises 11).

H1. (a) `nums` is `[1, 2, 3]`, `r` is `[1, 2, 3]`, `True`. (b) `[3, 1, 2]`, `[1, 2, 3]`, `False`. (c) `[1, 2, 3]`, `[3, 1, 2]`, `False`. (d) `[3, 1, 2]`, `[1, 2, 3]`, `False`. (e) `[3, 1, 2]`, `[3, 1]`, `False`. (f) `[3, 1]`, `[3, 1]`, `False`.
The caller's list is safe in (b), (d) and (e): each of them does its work on a clone. (e) clones first and gives the clone the name `L`, so from line 2 on `L` is the function's own list. (c) makes the clone and then sorts the wrong one: the caller's list comes back sorted, and the unsorted clone is returned. (f) pops from the caller's list before it clones.

H2. `[1, 2, 3, 0]`. Version (a) returns the very list it was given, so `r` is an alias of `nums`: one object with two names in the caller's own code. A function that returns its parameter hands back an alias, not a copy.

I1. (a) `[1, 2, 3] [7, 8] [1, 2, 3]`. All three names start on one list, so the `3` appended through `c` shows through `a`. `b = [7]` moves only `b`'s arrow to a new list; `a` and `c` still share the old one, and the `8` goes into `b`'s new list.

(b) `[[6, 2], [3, 4]]`, then `[[6, 2], [5, 0]]`. Line 3 gives `L` a list of its own at position 1, so the `5` of line 4 lands there and `n` never sees it. Position 0 was never replaced: it is still the shared `[1, 2]`, so the `6` of line 5 shows in both.

J1. (a) Loop over a *copy*: `for item in L[:]`. Removing an item shifts everything left, so the loop pointer skips the element that slid into the vacated slot.
(b) `L1_copy = L1[:]`. Despite its name, `L1_copy = L1` is an *alias*, so the loop is still walking the very list it is deleting from — a helpful variable name proves nothing.

J2. Prints `35`, then `[40]`. The sum is right, but `L` is an alias of `bill`, so the two `pop`s took two items out of the caller's list. Fix: add `L = L[:]` as the first line of the function, so that it pops from a clone. (Not popping at all, `return L[-1] + L[-2]`, is fine too.)

J3. Prints `[[5, 8], [9, 0]]`. `.copy()` is a shallow copy: `before` has its own outer list, but both outer lists point at the same two inner lists. Fix: `before = copy.deepcopy(scores)`, with `import copy` at the top.

J4. (a) Prints `None`. `remove` takes the `3` out but hands nothing back. To remove an item and keep it, use `pop` with its position: `gone = L.pop(1)`.
(b) `IndexError` ("pop index out of range"). `pop` takes a *position*, and a list of three items has no index 20. To remove by value write `L.remove(20)`; by position, `L.pop(1)`.

J5. Prints `different`. `a` and `b` are two separate lists that hold equal marks, and `is` asks whether they are one object. To compare contents write `if a == b:`.

K1. Because `=` makes an alias, not a copy — both names point at one list, so you are still deleting from the list you are looping over.

K2. A shallow copy duplicates only the outer list; both outer lists still point at the very same inner lists, so a change to an inner list is visible from both.

K3. During the call the parameter `L` is an alias of `marks`: both names point at one list object, so mutating it through `L` changes the list that `marks` names.

L1. `rest = marks[:]` / `rest.remove(min(rest))` / `return rest`. Clone first, then remove from the clone. `remove` takes out only the first match, which is exactly "drop it once".

L2. `for x in L[:]:` / `if x < 0:` / `L.remove(x)`. The loop walks a clone while the real list shrinks. With `for x in L:` the `-2` slides into the gap and is skipped, and `nums` ends as `[3, -2, 5]`.

L3. `new = []` / `for inner in nested:` / `new.append(inner[:])` / then `return new` after the loop. A new outer list, and into it a clone of every inner list. With `new.append(inner)`, without the `[:]`, it would be a shallow copy again.