Warmup (L9)

Add a line of code where specified to make this print 42, other than `x[1][1] = 42`

```
x = [[0, 0, 0], [0, 0, 0]]
# put a line here
x[0][1] = 42
print(x[1][1])
```

Pedantry corner

- This is a *mental model*, not a literal description of what’s going on in memory.
- If a type is immutable (think strings), there’s no way for you to tell if what’s in the slot is a reference or a value.
- Values are easier to reason about, so in our mental model, we think of all immutable things as values.

A note on equality

- Two lists are equal (== says "`True`") if they contain equal elements, even if they're not the same reference in memory
- If you want to know if they're the same reference in memory, there's another comparison for it, "**`is`**"

```
x = [2, 4, 6, 0, 1]
y = x
z = [2, 4, 6, 0, 1]
x == y and x == z # True
x is y # True
x is z # False
y[1] = 8
x == z # False, spooky action at a distance!
```

A note on equality

- Two lists are equal (== says “`True`”) if they contain equal elements, even if they’re not the same reference in memory
- If you want to know if they’re the same reference in memory, there’s another comparison for it, “**`is`**”
- It’s pretty rare to need **`is`**, and **`is`** can reveal surprising details about Python’s *real* memory model, so usually use ==.

Example break

- Let’s find the greatest value in a list

```
def greatest(lst: list[float]) -> float:
    assert len(lst) > 0, "No greatest in
                          an empty list"
    r = lst[0] # Need to start with
               # something!
    for val in lst:
        if val > r:
            r = val
    return r
```

Expanding lists

CS114 L9 (M4)

Insertion

- As well as *changing* values in the list, we can *insert* slots into the list (and put values there)

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Insertion

Where to insert the value.

Which value to insert.

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Insertion

Remember the dot (as in `math.sqrt`)? It’s also how you get special functions that act on lists (and other things we’ll see later).

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Insertion

These functions you get with dot ("on" the list) are called "methods".

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Insertion

Let’s add some prints to understand our lists as the code runs.

```
e = [2, 6, 8]
e.insert(1, 4)
e.insert(0, 0)
```

Appending

- There's a special version of `insert` for the common case of inserting at the end

```
e = [0, 2, 4, 6, 8]
e.append(10)
```

Using `append`

- Let’s collect all the common divisors of two integers into a list (sort of “all-cd” instead of gcd)

```
def divisors(x: int, y: int) -> list[int]:
    r = []
    i = 1
    while i <= x and i <= y:
        if x%i == 0 and y%i == 0:
            r.append(i)
        i = i + 1
    return r
```

Shrinking lists

- Just like we can add items, we can remove items by *pop*ping them out of the list:

```
e = [0, 2, 4, 6, 8]
e.pop(0) # Removes element at index 0
# e is now [2, 4, 6, 8]
e.pop() # By default, removes the last element
# e is now [2, 4, 6]
```

How does this affect loops?

- Nothing in Python stops you from changing the length of a list while you loop through it
- But, the behavior is *hugely* confusing, so best avoided

```
x = [2, 4, 6, 0, 1]
for v in x:
    print(v) # Which values will actually print here???
    x.pop(0)
```

Slicing and joining

CS114 L9 (M4)

References are Hell!

- Remember, lists are a reference type: if you pass a list to a function and it modifies it, you will see the changes!
- This was done because copying is slow
- But sometimes you *want* to copy!

```
x = [2, 4, 6, 0, 1]
y = x[:]
y[0] = 4
# x is still [2, 4, 6, 0, 1]
```

But what's this thing???

New syntax!

- `x[:]` was a *slice* of `x`
- Why did we call it a slice when it was a copy? Because that ~~hamburger~~ operator is so much more powerful!

```
x = [2, 4, 6, 0, 1]
x[1:3] == [4, 6]
```

New syntax!

- `(sequence)[from:to]`
- Makes a *copy* of the sequence, from `from` to `to`
- Like range, `from` is inclusive, `to` is exclusive
- The `from` and `to` are the indices, not values!

Basic slices

- Let’s split a list in half using slicing

```
midpoint = len(lst) // 2
left = lst[0:midpoint]
right = lst[midpoint:len(lst)]
```

More advanced slicing

- Where did `[:]` come from?
- Both from and to are optional!
  - By default, `from` = 0
  - By default, `to` = `len(`*the sequence*`)`
  - With both defaults, you copy the whole list
- There’s a third part, also optional: the step (just like `range`)
  - `lst[::2]` gets every second element
  - `lst[::-1]` reverses a sequence

More advanced indexing

- It’s common to want the last element in a list (or other sequence)
- Obvious way: `lst[len(lst)-1]`
- Python lets you shorthand by using a negative index: `lst[-1]`
- Same works with slicing, and insert, and pop, and everything else!

In-lecture quiz (L9)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q1: What does this code print?

```
nums = [0, 1, 2, 3, 4, 5]
print(nums[-4:-1])
```

A. [2, 3, 4]
B. [3, 4, 5]
C. [2, 3, 4, 5]
D. [1, 2, 3]

In-lecture quiz (L9)

- https://student.cs.uwaterloo.ca/~cs114/F25/quiz/
- Q2: What does this (awful) code print?

```
nums = [0, 1, 2, 3, 4, 5]
print(nums[-4:-1][::2][1])
```

A. Nothing or an error
B. 2
C. 3
D. 4

Joining

- You can also join (called *concatenate*) lists or other sequences with +

```
"Hello, " + "world!" == "Hello, world!"
[3, 1, 4] + [1, 5, 9] == [3, 1, 4, 1, 5, 9]
```

The reverse interleave

- Let’s write a function to *reverse interleave* a list
  - What this means is, for instance, turn `[1, 2, 3, 4, 5, 6]` into `[1, 3, 5, 2, 4, 6]`
  - Perfectly interleaving or reverse interleaving playing cards is a basic magicians’ trick

The reverse interleave

```
def reverseInterleave(deck: list) -> list:
    return deck[::2] + deck[1::2]
```

- (Yup, that’s it! Slicing is powerful, eh?)

Sequence conversions

- You can convert any other sequence into a list by using `list` as a function:

```
list("Hello!") ==
["H", "e", "l", "l", "o", "!"]
```

The second dimension

CS114 L9 (M4)

The second dimension

- To represent anything two-dimensional, it’s common to use lists within lists (nested lists)
- E.g., a tic-tac-toe board might look like this:

```
board = [[" ","x"," "],
         [" ","x","o"],
         [" "," "," "]]
```

The second dimension

- Think carefully about memory! Let’s draw our tic-tac-toe board on the blackboard.
- It’s easy to make surprising mistakes

```
board[2] = board[1]
board[2][1] = "o"
# Now board[1][1] is also "o"!
```

Tuples

CS114 L9 (M4)

Tuples are immutable lists

- One more sequence type: tuples
- Tuples are just immutable lists
- Created with parentheses:

```
x = (2, 4, 6, 0, 1)
```

Why tuples?

- Why would we want immutable lists when we already have lists?
- Usually to box together multiple values of disparate types that are related in meaning
  - The type for a list can only have one “what’s in the list” type, because it’s expandable
  - The type for a tuple can list every type in it, because it’s fixed

Typing tuples

```
x: tuple[int, str] = (24601, "Jean val Jean")
```

You can name each individual type in a tuple, but for a list, they all have to be the same!

Returning tuples

- Most common use is returning multiple things from a function
- We’ll do this with an example in a moment

Fun with lists

CS114 L9 (M4)

Longest string

- Find the longest string in a list
- Instead of returning a string, return a *list* of all strings of the same length, *and* the length

Longest string

```
def longest(strs: list[str]) -> list[str]:
    r = [strs[0]]
    for s of strs[1:]:
        if len(s) > len(r[0]):
            r = [s]
        elif len(s) == len(r[0]):
            r.append(s)
    return (r, len(r[0]))

[…]
(res, length) = longest(["a", "blue", "duck"])
```

You can assign to a tuple of variables to get the values out of a tuple. We could've also used indexing.

Another version of pi

- One way to calculate pi is the dartboard technique: throw darts at a square board with a quarter circle in it, and the proportion that land in the circle can tell us pi
- We'll need one new feature to do this: random numbers:

```
import random
random.random()  # A random number
                 # between 0.0 and
                 # 1.0
                 # (inclusive, exclusive)
```

Monte Carlo pi

```
import random
def monteCarloPi(rounds: int) -> float:
    xs = []
    ys = []
    for _ in range(rounds):
        xs.append(random.random())
        ys.append(random.random())

    # How many fall in the quarter circle?
    inside = 0
    for idx in range(rounds):
        x = xs[idx]
        y = ys[idx]
        if x*x + y*y <= 1:
            inside = inside + 1

    return 4 * inside / rounds
```

Module summary

CS114 L9 (M4)

Module summary

- Strings and ranges are *sequences*
- **`for`** loops loop over sequences
- Lists are *mutable* sequences
- Lists are *reference types*
- Memory model
- Expanding/contracting lists
- Slicing and joining lists
- Tuples are immutable lists