CS 105: Introduction to Computer Science

Prof. Thao Nguyen

Spring 2025

Materials adapted from Dave Wonnacott

Python Lists, Part 1: they're like strings

In Python (or almost any other language), you can create a *list*

- A collection of any number of elements, typically of the same type
- Entered using [ ] around values *(demo)*
- Can identify each individual element using the [ ] subscript operation
  - similar to subscripting a *string*, but get 1 element, not a 1-element list, with 1 index *(demo)*
- Also share other string operations *(demo)*
  - \+ *concatenates* lists (or strings)
  - len gives the number of elements in the list (or string)
  - in checks whether an item is contained anywhere in a list (or string)
- (*Note: other languages may require all to be the same type, use term "array"*)

Practice with Lists, Part 1: "Pure" Operations

Recall that the following operations work with list objects *but don't change them*:

- len (to find the length) and + (to concatenate) work as they do for strings
- "subscripting", i.e. [ ], like for strings except when only 1 subscript is given
- "in" looks for a *value*, not a sub-list, in a list

How can we extract/create the string "mba" from X, for each definition?

- `X='mba'`
- `X='wombat'`
- `X=['phd', 'mba', 'md']`
- `X=['catapult', 'roomba', 'hat']`
- `X=[['bag'],['x','y'],['a','m','d']]`

Python Lists, Part 2: unlike strings, they can *change*

Python lists differ from strings in several ways:

1. Describing the type is more challenging: `from typing import List`
2. You can change elements of a list

```
bills: List[int] = [1, 5, 10, 20, 50, 100]   # a list of numbers
print(bills)       # what does this do?
print(bills[2])    # what does this do?
bills[2] = 7       # We've replaced the $10 bill?
print(bills)       # now what happens?
```

it's fine to use the list "bills" in your Lab 4 (and 5), if you like :-)

Changing lists complicates the imperative approach!

Based on what we've seen so far, there are *two* sensible guesses to this: Can you figure them both out?

```
bills: List[int] = [1, 5, 10, 20, 50, 100]   # a list of numbers
print(bills[2:4])  # what does this do? contrast with print(bills)
print(bills[2])    # how about this?    contrast with print(bills[2:3])

new_bills: List[int] = bills                 # change the currency
new_bills[2] = 7
print(new_bills)   # no surprise if we see $7 but not $10 here
print(bills)       # What could happen here? Two good guesses…
```

Together, we'll figure out each, then check what Python does and why

Changing lists in the pure-functional approach

If the previous remains confusing, you can often just *not* change lists:

```
bills: List[int] = [1, 5, 10, 20, 50, 100]   # a list of numbers

new_bills: List[int] = bills[:1]+[7]+bills[2:] # different list, with $7
print(new_bills)   # no surprise if we see $7 but not $10 here
print(bills)       # What could happen here? Only one thing makes sense
new_bills[3]=17    # what happens here? what to do instead in pure func.?
```

Python “id” & “is”

```
bills: List[int] = [1, 5, 10, 20, 50, 100]
print(id(bills))                # gives the unique id of an object
new_bills: List[int] = bills
print(id(new_bills))            # can we guess the output?
print(new_bills is bills)       # checks if two objects are the same object


bills2: List[int] = [1, 5, 10, 20, 50, 100]
print(bills2 == bills)          # can we guess the output?
print(bills2 is bills)          # what about for this line?
```