Non-Primitive Data

Table of contents

Outline 1
Learning objectives 1
Lists 2
Dictionaries 3
Summary 6
Next lecture 6

See csse1001[1] for course logistics. This is Lecture 4B.

Outline

- Lists: mutable ordered collections
- Dictionaries: hash tables

Learning objectives

1. Lists are mutable ordered collections; tuples (python-primitive-data-types[2]) are immutable ordered collections. Both can be sliced and indexed.
2. A name bound to a list is a *reference*, so copying, appending and passing to functions do not always do what they look like they do.
3. A dictionary maps keys to values via a hash function, and is unordered and mutable.

[1] courses/csse1001/csse1001.pdf
[2] courses/csse1001/python-primitive-data-types.pdf

Lists

> **Definition (List).** A *list* is a *mutable* ordered collection of elements. These elements are *not necessarily* of the same type. Square brackets `[]` are used to create lists in Python.

See python-lists[3] for the full reference on lists: creation, mutability, comparison, membership, `append`/`extend`, aliasing, copying (shallow/deep), passing to functions, slicing, nested lists/matrices, type hints, and unpacking into function arguments. Two of those matter most and are drawn out there as figures — that `ys = xs` makes an *alias* rather than a copy, and that `.copy()` on a list-of-lists is only *shallow*.

Exercise: counting occurrences

```
def count(xs: list[int], ys: list[int]) -> int:
    """ Return the number of times members of ys are in xs.
    >>> count([1, 2], [1, 2, 3, 4])
    2
    >>> count([1, 2], [1, 2, 2, 2, 3, 1, 4])
    5
    """
```

A while-loop solution (python-while-loops[4]), walking `ys` by index:

```
def count(xs: list[int], ys: list[int]) -> int:
    ans, k = 0, 0
    while k < len(ys):
        y = ys[k]
        k += 1
        if y in xs:
            ans += 1
    return ans
```

It is certainly better to use for-loops in this situation:

```
def count(xs: list[int], ys: list[int]) -> int:
    ans = 0
    for y in ys:
        if y in xs:
```

[3] courses/csse1001/python-lists.pdf
[4] courses/csse1001/python-while-loops.pdf

```
            ans += 1
    return ans
```

There is also a one-line solution:

```
def count(xs: list[int], ys: list[int]) -> int:
    return sum(y in xs for y in ys)
```

(This last one relies on `True` counting as `1` under `sum`, and on the generator-expression syntax that 2025-08-25-comprehensions[5] introduces the week after.)

Exercise: rotating a list

We say the list `[0, 1, 2, 3, 4, 5]` *rotated* by 2 is `[2, 3, 4, 5, 0, 1]`. Write a function `def rotate(xs: list, k: int) -> list` that rotates a list by k.

This is left as an open exercise. Slicing (see python-lists[6]) is the hint about how it is meant to be solved.

Question: Caesar cipher

Write code for encrypting and decrypting messages using the Caesar cipher. In particular, write functions with the headers:

```
def encrypt_caeser(plaintext: str, shift: int) -> str:

def decrypt_caeser(ciphertext: str, shift: int) -> str:
```

This is left as an open exercise. A working `caesar_cipher` implementation (a single shift parameter instead of separate encrypt/decrypt functions) is in Lecture 5C — see week-five-exercises[7].

Dictionaries

Here is a deliberately vague question: write a function that returns the *character frequency* of a string, ignoring case. For example, the character frequency of `"hello world"` would encode that there is one "h", three " "'s and so on.

[5] courses/csse1001/2025-08-25-comprehensions.pdf
[6] courses/csse1001/python-lists.pdf
[7] courses/csse1001/week-five-exercises.pdf

Viable representations

Two lists (characters paired positionally with their counts):

```
[['h', 'e', 'l', 'o', ' ', 'w', 'r', 'd'],
 [ 1,   1,   3,   2,   1,   1,   1,   1 ]]
```

One-to-one encoding (a count for every letter of the alphabet, in order):

```
# a b c d e f g h i j k l m n o p q r s t u v w x y z
 [0,0,0,1,1,0,0,1,0,0,0,3,0,0,2,0,0,1,0,0,0,0,1,0,0,0]
```

Hash functions

In both cases we provided a way to map a **key** (character) to a **value** (number):

$$\texttt{str} \to \texttt{int}, \quad \texttt{h} \mapsto 1,\ \texttt{e} \mapsto 1,\ \texttt{l} \mapsto 3,\ \ldots$$

This mapping is called a **hash function**.

The hash function for the two-list encoding looks up the key's *position* in the first list, then indexes the second list at that position:

```
>>> def hash_1(key: str) -> int:
...     """Two list encoding."""
...     xs = ['h', 'e', 'l', 'o', ' ', 'w', 'r', 'd']
...     ys = [1, 1, 3, 2, 1, 1, 1, 1]
...     k = xs.index(key)   # position of key in xs
...     return ys[k]
>>> hash_1('o')
2
>>> hash_1('r')
1
```

The hash function for the one-to-one encoding computes the key's position in the alphabet directly, via `ord`:

```
>>> def hash_2(key: str) -> int:
...     """One-to-one encoding."""
...     xs = [0,0,0,1,1,0,0,1,0,0,0,3,0,0,2,0,0,1,0,0,0,0,1,0,0,0]
...     pos_in_alpha = ord(key) - ord('a')   # position of key in alphabet
...     return xs[pos_in_alpha]
>>> hash_2('o')
2
>>> hash_2('r')
1
```

Both return the same count for the letters of the example even though they store completely different things. That is the point of the exercise: the *encoding* is a free choice, and the hash function is whatever computation turns a key into an index into it.

They are not interchangeable in general. `hash_1` only knows the eight characters in its own `xs`, and `hash_2`'s `ord(key) - ord('a')` only lands inside its list for `'a'`–`'z'` — so the space character that `hash_1` counts is out of range for `hash_2`.

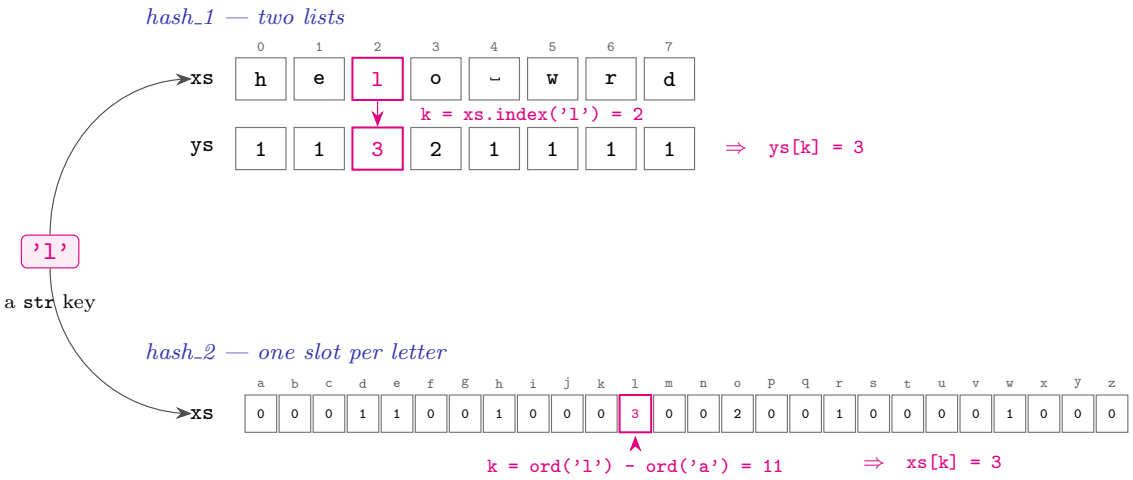

Figure 1: The same key routed through both encodings. `hash_1` finds the key's position in a list of characters and reads the count off a parallel list at that position; `hash_2` computes the position arithmetically from the character code and reads a single alphabet-sized list. Different storage, different index arithmetic, same answer — which is why Python's `dict` can hide the choice behind a *magic* hash function. Magenta marks the key and the value it leads to.

Generally speaking we can index a list by **any type of key** given some hash function which takes keys to values. Python has a *magic* hash function that will index anything appropriately. The result is implemented as the `dict` data-type.

Dictionaries have no ordering because the keys can be of mixed type.

The "hello world" example encoded using a `dictionary` is

```
>>> char_to_freq = {
...     'd': 1,
...     'o': 1,
...     ' ': 1,
...     'r': 1,
...     'w': 1,
...     'e': 1,
...     'l': 3,
...     'h': 0
...     }
```

> ⚠ Two of these counts are wrong
>
> Two entries disagree with the two-list encoding above (and with `"hello world"` itself): that encoding has `'o'` at **2** and `'h'` at **1**, which is what `hash_1('o')` returns above. As written, `char_to_freq` is wrong for the string it claims to encode.

See python-dictionaries[8] for the full reference on dictionaries: creation, indexing, `keys`/`values`/`items`, `clear`/`copy` (and its shallow-copy gotcha), `get`, and the requirement that keys be immutable.

Summary

Lists are mutable ordered collections; tuples are immutable ordered collections. Dictionaries are for storing *key-value* relationships — a key must be an *immutable* type and gets mapped to a value using a *hash function* that is magically fast.

Hash tables are a candidate for the most important/useful data-structure in computer science.

Next lecture

For-loops and comprehension — 2025-08-25-for-loops[9] and 2025-08-25-comprehensions[10].

[8] courses/csse1001/python-dictionaries.pdf
[9] courses/csse1001/2025-08-25-for-loops.pdf
[10] courses/csse1001/2025-08-25-comprehensions.pdf