Non-Primitive Data

Table of contents

See csse1001[1] for course logistics — this note covers Lecture 4B's technical content.

Today's 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.
2. A dictionary maps keys to values via a hash function, and is unordered and mutable.

Lists

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.

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

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:

```
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
```

A for-loop is more natural for this problem:

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

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`.

> No worked solution is given in the source for this exercise — left as an open exercise rather than invented here.

Question: Caesar cipher

Write code for encrypting and decrypting messages using the Caesar cipher, with function headers:

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

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

> Left unsolved in the source — the lecture moves directly on to dictionaries afterwards. A working `caesar_cipher` implementation (a single shift parameter instead of separate encrypt/decrypt functions) appears later, in Lecture 5C — see week-five-exercises[4].

Dictionaries

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

What are some 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]
```

[4] courses/csse1001/week-five-exercises.pdf

Hash functions

In both representations we provided a way to map a **key** (a character) to a **value** (a count) — this mapping is called a **hash function**:

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

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]
```

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]
```

Generally, we can index a collection by *any* type of key given some hash function that maps keys to values. Python has a "magic" hash function that indexes anything appropriately — implemented as the `dict` type. See python-dictionaries[5] 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.

The "hello world" character frequency, encoded as a dictionary:

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

[5] courses/csse1001/python-dictionaries.pdf

Summary

Lists are mutable ordered collections; dictionaries store key-value pairings, found quickly via a "magically" fast hash function.

Next lecture

For-loops and list comprehensions.