Dictionaries, Tuples, Hashing

6.1000 LECTURE 7

FALL 2025

Storing associated data

- E.g., registrar stores term records of each student's living group and course registration
- One strategy: parallel lists
  - ```
    names = [name1, name2, name3]
    addresses = [address1, address2, address3]
    classes = [classes1, classes2, classes3]
    ```
- Another strategy: nested lists
  - ```
    records = [
        [name1, address1, classes1],
        [name2, address2, classes2],
        [name3, address3, classes3],
    ]
    ```
- Disadvantages
  - access by index is hard to read
  - indexing is error-prone, need to keep lists synchronized

Another way: Python dictionaries

- Use student names as direct “indicies”/selectors into data
- Parallel dict strategy
  - ```
    addresses = {
        name1: address1, name2: address2, name3: address3
    }
    classes = {
        name1: classes1, name2: classes2, name3: classes3
    }
    ```
  - `addresses[name1] → address1`
- Nested dict strategy
  - ```
    records = {
        name1: {"address": address1, "classes": classes1},
        name2: {"address": address2, "classes": classes2},
        name3: {"address": address3, "classes": classes3},
    }
    ```
  - `records[name1]["address"] → address1`

Python `dict` overview

- A mapping type from keys to values
  - https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

- Square bracket syntax connotes analogy to list indexing
  - **`list`**s have implicit sequential indicies
  - **`dict`**s have explicit key labels

- Keys can be (almost) any Python object
  - so ordering no longer makes sense

- **`dict`** objects in memory are conceptually two-column tables
  - left column contains references to keys
  - right column contains references to values associated with those keys
  - **like `lists`, no objects stored in `dicts`**

`dict` mutating operations

- add or update key-value pair
  - `dict[key] = value`
- delete key-value pair
  - `del dict[key]`
  - `dict.pop(key, default)`
- merge with other dictionaries
  - `dict | other`
  - `dict |= other`
  - `dict.update(other)`

`dict` operations

- object creation
  - `{}` is empty **`dict`**
  - **`{key1: val1, key2: val2}`** is **`dict`** literal
  - **`dict()`** constructor copies any **`dict`** passed in
    - or creates new **`dict`** from sequence of **`[key, value]`** pairs
  - **`len(`*`dict`*`)`**
  - ***`dict`*`.copy()`**
  - ***`dict`*`.clear()`**

- key and value retrieval
  - **`key in`** ***`dict`***
  - ***`dict`*`[key]` → `value`**
  - ***`dict`*`.get(key, default)` → `value`**
    - if **`key`** not in ***`dict`***, returns **`default`** instead of raising **`KeyError`**

Immutability of `dict` keys

- **`dict` keys must be hashable**
  - for Python's built-in types, hashable basically means **immutable**
- When you associate a key with a value:
  - expect to be able to retrieve the value by looking up with an equivalent (**==**) key, no matter how it was constructed
  - if keys are mutable (e.g., lists), code that runs after a key mutation may be unaware that the key has changed

Common types for `dict` keys

- `int`s
  - `bool`s are really `int`s underneath the hood, so avoid those
- `float`s are a bad idea
  - mathematically equivalent expressions may not yield equivalent `float`s
- `str`s are a great idea
  - natural labels
  - this is one reason why Python's `str`s are immutable
- `tuple`s are also good
  - `tuple`s are just like `list`s, but immutable
  - for a `tuple` to be hashable, all its nested contents must also be immutable

Iterating over dicts

- No inherent ordering of **`dict`** keys, unlike **`list`** indicies
  - but still wish to retrieve all elements
  - nature of code/time means have to do so sequentially
- Python's **`for`** directly iterates over **`dict`** keys
  - ```
    for key in dict:
        value = dict[key]
    ```
  - **`list(`*`dict`*`)`** will product a **`list`** of ***`dict`***'s keys
- Can also iterate over dictionary views
  - ***`dict`*`.keys()`**
  - ***`dict`*`.values()`**
  - ***`dict`*`.items()`** → produces **`(key, value)`** tuples
    ```

Example: word frequencies in song lyrics

- Study code on your own
- Note how **`dict`**s are being created, updated, iterated over, deleted from

List indexing and direct addressing

- List elements are contiguous in memory
  - each cell, being a reference to another location in memory, is a fixed size
  - so if you know where index **`0`** is, can immediately calculate where to look up index **`i`** very
    - don't need to traverse list sequentially from start to index **`i`**
- However, membership test requires sequential traversal
- Another storage scheme: **direct addressing**
  - let the list index represent the actual data
    - because any object's bit representation can be interpreted as an **`int`**
  - when putting object in list, store a **`True`** flag at the index that represents the object
  - theoretically possible, impractical due to insane memory space

Hashing and dictionaries

- **Idea:** translate each objects into a smaller address space
  - magic translation through a mathematical **hash function**
- As long as number of objects is still smaller than number of addresses, underlying list has enough spaces to store pointers to all objects
  - **Note: order is unlikely to be preserved**
- **Issue:** sometimes, hash function will map two objects to the same index
  - called a **collision**
  - **Fix:** store objects in **secondary lists called chains** at each index in the underlying list
- **Result: hash tables** have the performance of direct addressing when determining membership
  - Python **`dict`**s are just hash tables for keys, with values stored next to each key