Fundamentos de Programação 2025-2026

Programming Fundamentals

Class #7 - Dictionaries

Overview

- Motivation problem
- Useful information
  - Key-value pairs
  - Hash functions
  - Hash tables
  - Hashable

- Dictionaries
  - Dictionaries in python
    - Creation, access, methods, …
  - Application examples of dictionaries
  - Dictionaries, tuples and lists
  - Some common mistakes

Problem

Counting Word Frequencies in a Sentence

- **Scenario:**

  You’re given a sentence as a string.

  Your task is to count how many times each word appears.

- **Example Input:**

  sentence = "apple banana apple orange banana apple"

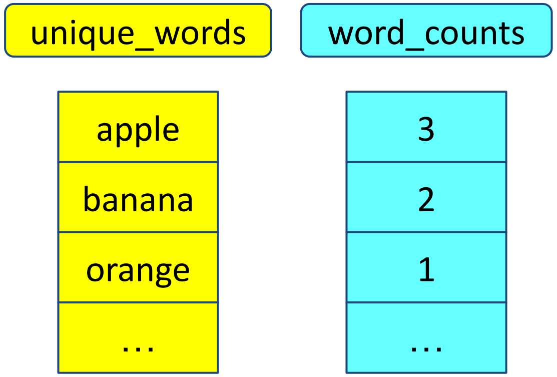
Pseudo code of solution with lists

```
1. Split the sentence into a list of words
 words_list ← split(sentence)

2. Initialize two empty lists:
  unique_words ← []
  word_counts ← []

3. FOR each word IN words_list DO
    IF word is in unique_words THEN
      index ← position of word in unique_words
      word_counts[index] ← word_counts[index] + 1
    ELSE
      append word to unique_words
      append 1 to word_counts
  END FOR

4. OUTPUT the word frequencies:
  FOR i FROM 0 TO length of unique_words - 1 DO
    PRINT unique_words[i], "→", word_counts[i]
  END FOR
```



unique_words	word_counts
apple	3
banana	2
orange	1
…	…

Why Lists/Arrays Make This Hard

- Using only lists, we need:
  - A list to store unique words (unique_words)
  - A parallel list to store counts (word_counts)
  - To check to see if a word is already in the list

    index ← position of word in unique_words
  - To handle index tracking and updates.

- This leads to complex, inefficient and error-prone code.

What do we need?

- Direct mapping from word → count.
- Constant-time lookup and update.
- Clean, readable code.

Possible solution (pseudo code)

```
Function CountWordFrequencies(sentence):

  Initialize empty dictionary called word_counts

  Split sentence into list of words, space as
  delimiter

  For each word in the list of words:
     If word is already in word_counts:
        Increment word_counts[word] by 1
        Else:
         Set word_counts[word] = 1

  Return word_counts
```

Before we start our topic …

SOME USEFUL INFORMATION

🔑 Key-Value pairs

- In many real-life situations we deal with **pairs of related information**, such as:
  - A **phone number** and its **owner**
  - A **person** and their **address**
  - A **book** and its **price**
- Often, we want to **find one piece of information using the other**.
- For example:
  ➡️ Given a person’s name, find their phone number.

- To do this, we use a structure called a **key-value pair**:
  - The **key** is the piece of information we use to look something up (e.g., the person’s name).
  - The **value** is the information we want to retrieve (e.g., the phone number).
- Together, they form a **key-value pair**
  - commonly written as: (Key, Value) or Key ⟶ Value
- This concept is fundamental in programming and is used in **dictionaries**, maps, and hash tables.

Key-Value pairs

- A **key-value pair** is a basic data representation format.

- Each **key** acts as a unique identifier.
- Each **value** is the data associated with that key.

- Analogy:
  - A dictionary: the word is the key, and its definition is the value.

- Example:

  'name’ □ 'Alice’
  'age’ □ 25
  'city’ □ 'Aveiro'

- Keys: 'name', 'age', 'city'
- Values: 'Alice', 25, 'Aveiro'

Why use Key-Value pairs?

- Fast data retrieval using a key.
- Simple and flexible data organization.

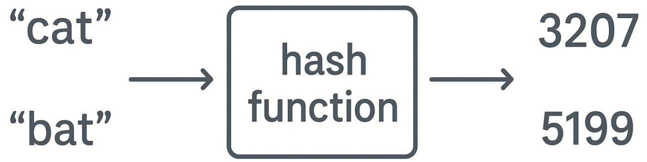
- Used in many programming contexts:
  - Dictionaries (Python)
  - Objects (JavaScript)
  - Hash maps (Java, C++)
  - Key-value databases (Redis, DynamoDB)

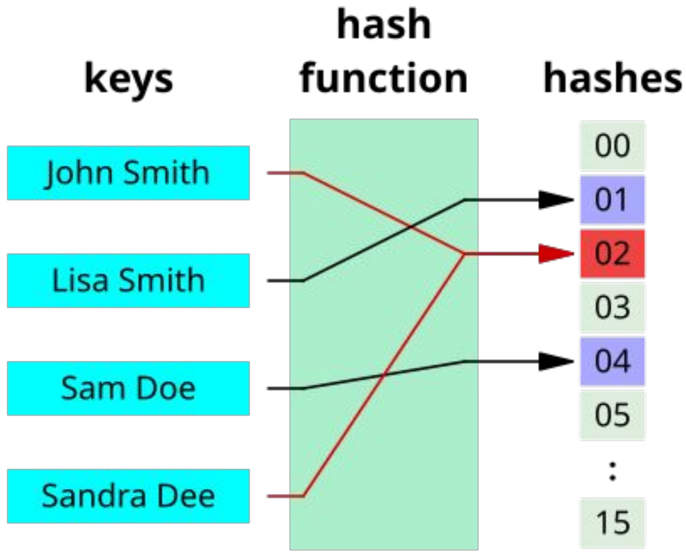
Hash Function

- Transforms data of any size ⟶ fixed size “hash code”
- Key Idea:

  Input ⟶ hash function ⟶ short numerical code (hash)

- Same input ⟶ same hash code
- Small change ⟶ big difference
- Examples:





- Not perfect !
  - Different inputs can have the same hash code

Why do we need them?

- Used to quickly find or store data
  - e.g., in dictionaries, hash tables

- Helps in:
  - Fast lookups in databases or associative arrays
  - Checking data integrity (detecting changes)
  - Password storage (only hash is saved)

How they work? (Simplified)

- Basic Idea (for strings):
  1. Convert each character into its ASCII value
  2. Combine those values (add or multiply)
  3. Reduce the result to a fixed range (using modulus)

- Example:

```
                 ord('C')   ord(‘A')   ord(‘A')
hash('CAT') = (67 + 65 + 84) mod 10
              = 216 mod 10 = 6
```

Hash functions properties

- Deterministic
  - same input → same output
- Uniform
  - spread outputs evenly
- Irreversible
  - cannot recover input
- Fast to compute

Hash Table

- Is a data structure that stores key-value pairs.
- Like an indexed set of mailboxes

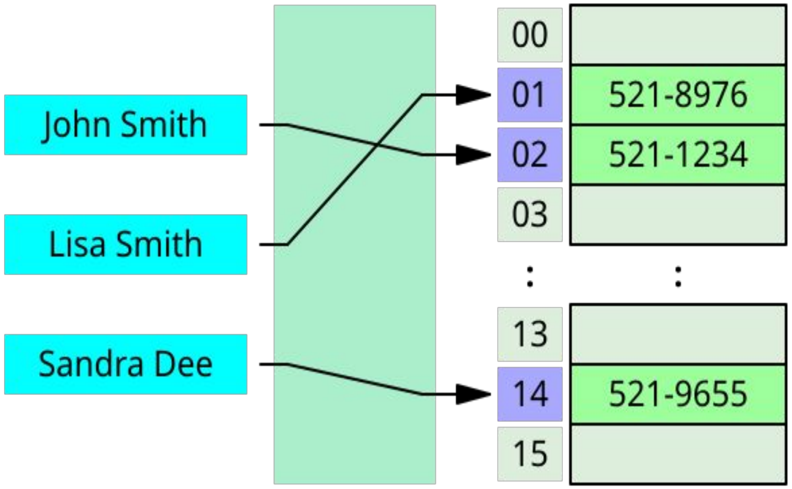
- Purpose: Enable very fast lookups, insertions, and deletions.

- Uses a hash function to compute an index for each key
  - called a hash code

keys

hash function

Table w values



Hashable

- An object is *hashable* if it has a hash value that never changes during its lifetime
  - i.e., it’s immutable

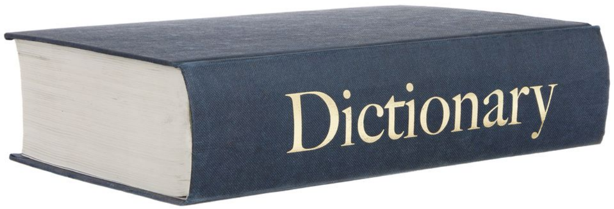
- ✅ Hashable Objects (Immutable):
  - int, float, str, tuple (if all elements are hashable), frozenset

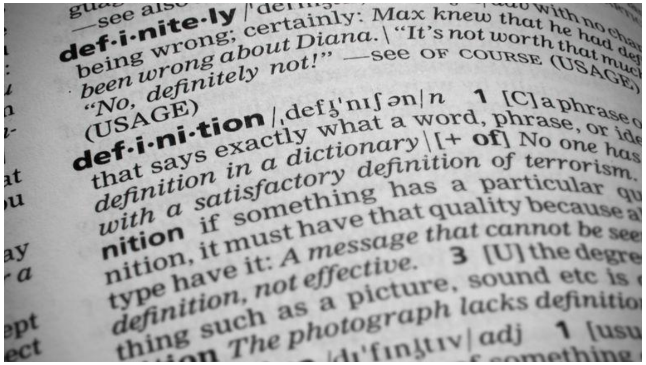
- ❌ Not Hashable (Mutable):
  - list, dict, set, bytearray

CLASS TOPIC

Dictionaries (in real life)

- **A dictionary** is a reference book or digital resource that lists words and explains their meanings.
- Each word is followed by its definition, pronunciation, and sometimes examples or translations.
- Words are arranged alphabetically.





—see also
def·i·nite·ly /ˈde… with no chan
being wrong; certainly: *Max knew that he had def*
been wrong about Diana.\ "*It's not worth that muc*
"*No, definitely not!*" —see OF COURSE (USAGE)
(USAGE)
def·i·ni·tion /ˌdefɪ̈ˈnɪʃən/ *n* **1** [C] a phrase o
that says exactly what a word, phrase, or ide
definition in a dictionary\[+ **of**] No one has
with a satisfactory definition of terrorism.
nition if something has a particular qu
nition, it must have that quality because a
type have it: *A message that cannot be see*
definition, not effective. **3** [U] the degre
thing such as a picture, sound etc is
The photograph lacks definitio
/dɪˈfɪnɪ̈tɪv/ *adj* **1** [usu

Dictionaries (in programming)

A **dictionary** is an

Associative

- because each item associates a key to a value.

Collection

- because it may contain zero or more items.

of Unordered

- because the order of the items does not matter for equality.
  - However, items are kept in insertion order (guaranteed since Python 3.7).

Unique items.

- because no two items can have the same key.

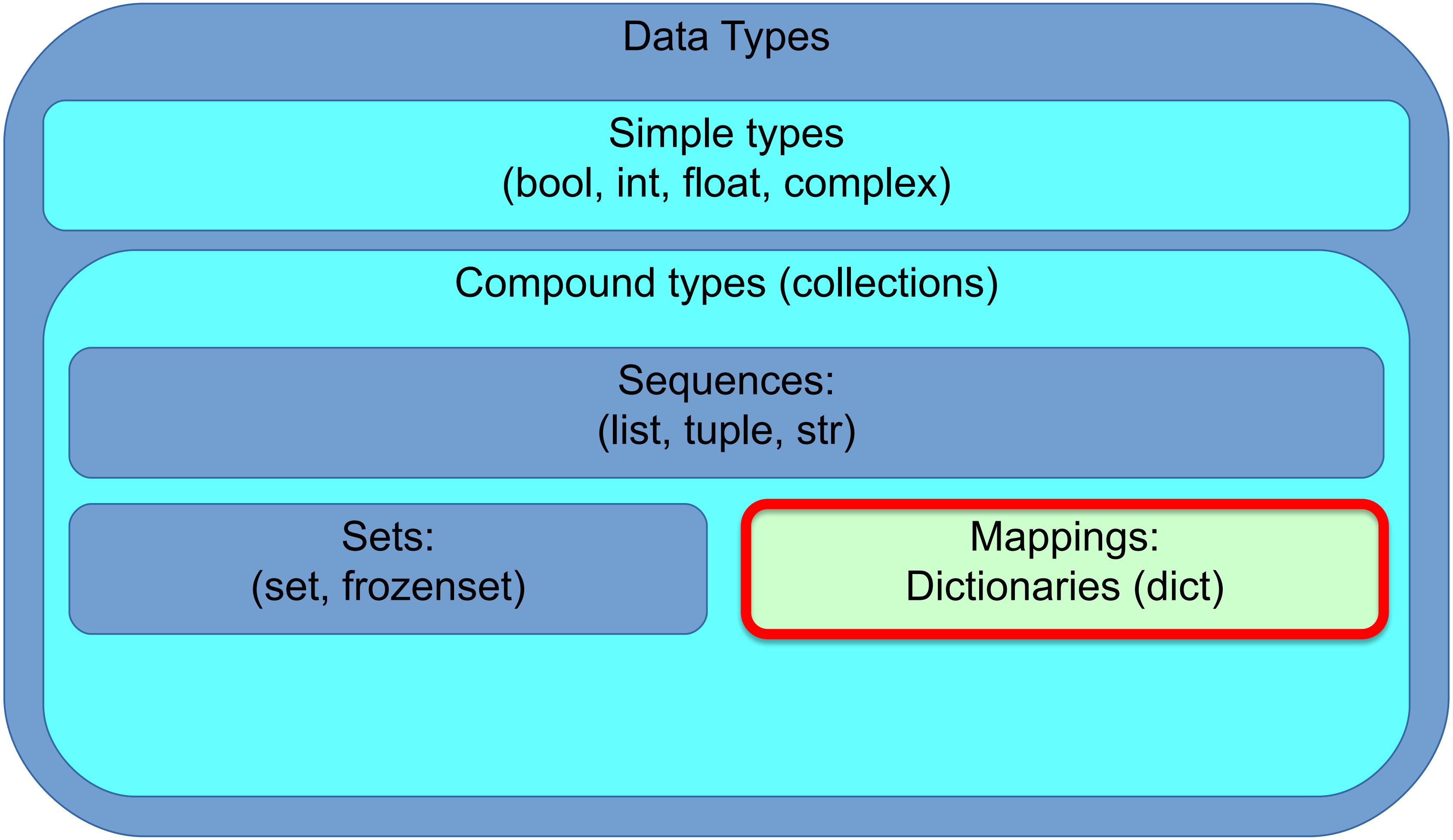
Dictionaries

- Dictionaries are also called **associative arrays** or **maps**.
  - Because they establish a mapping (or association) between keys and values.
- Dictionary items are also called key-value pairs.

- More on dictionaries in the official Python tutorial.

DICTIONARIES IN PYTHON

Python Data Types



Data Types

Simple types
(bool, int, float, complex)

Compound types (collections)

Sequences:
(list, tuple, str)

Sets:
(set, frozenset)

Mappings:
Dictionaries (dict)

Creating Dictionaries

- A dictionary may be created using braces (curly brackets).
  - With key-value pairs inside, separated by commas
    - key is followed by “:” and value

```
eng2sp = {'one': 'uno', 'two': 'dos', 'three': 'tres’}

shop = {'eggs': 12, 'sugar': 1.0, 'coffee': 3}
```

Play ▶

- An empty dictionary may be created with {} or dict().

Creating dictionaries

https://youtu.be/4t10v2QmTHU?si=wwMztgvGEgQjFVLy&t=21

Accessing dictionary items

```
eng2sp={'one': 'uno', 'two': 'dos', 'three': 'tres'}
shop = {'eggs': 12, 'sugar': 1.0, 'coffee': 3}
```

- To access the value for a given key, use square brackets.

```
shop['sugar']      #-> 1.0
eng2sp['two']      #-> 'dos’
```

Play ▶

- Trying to access an inexistent key is an error.

```
shop[‘sugar’]     #-> 1.0
shop[‘tea’]       #-> KeyError
```

Play ▶

Accessing dictionary items

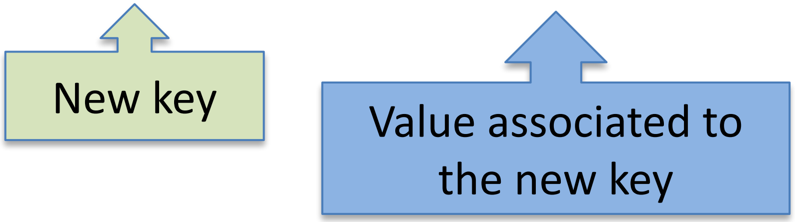
https://youtu.be/4t10v2QmTHU?si=FqsXywQ1fYj-3_TL&t=84

Changing dictionaries

- Dictionaries are mutable

- We can add new a new key-value pair

```
shop['bread’]  =   6 # Add a new key-value association
```



New key

Value associated to the new key

- We can also change the value associated to a key

```
shop['eggs']=24 # Change the value for an existing key

#-> shop = {'eggs':24, 'sugar':1.0, 'coffee':3, 'bread':6}
```

Changing dictionaries - removing

- Use pop(key) to remove the item with the given key and return its value.

```
d = {10:'dez', 20:'vinte', 1000:'mil’}

x = d.pop(10)    #-> x == 'dez’

print(d)         # {20:'vinte', 1000: 'mil'}
```

Play ▶

Changing dictionaries - removing

- We can also delete an item with the del operator.

```
del d[20]
print(d)              # {1000:'mil’}
```

- The popitem() method removes the last inserted item from the dictionary
  - and returns its (key, value) pair as a tuple.

```
d = {10:'dez', 20:'vinte', 1000:'mil'}
t = d.popitem()   #-> (1000,'mil')
print(d)              # {10:’dez’, 20:'vinte’}
```

Play ▶

Note: In versions before 3.7, the popitem() method removed a random item.

Changing dictionaries

https://youtu.be/4t10v2QmTHU?si=FqYWPTvFouX0jbbb&t=361

Value and key types

- **Values** in a dictionary can be of any type.

```
shop['eggs'] = [1, 'a']
shop['eggs'] = {'brown': 6, 'white': [2, 3]}
```

- Keys must be hashable.
- In practice, this means:
  - keys must be immutable scalars (ints, floats)
  - Or immutable collections containing only hashable elements (strings, tuples).
- So, lists are not valid keys!

- Examples:

```
eng2sp[4] = 'quatro'     # integer key is fine
d[(12,25)] = 'Christmas' # tuple key is fine
d[[1,2]] = 'A'           #-> TypeError: unhashable type
```

Working with Entire Dictionaries

- So far, we've learned how to:
  - Create dictionaries
  - Add or update entries
  - Access individual values using keys
- But in many real-world scenarios, we need to perform **operations on the entire dictionary**
  - not just individual entries.

- Common Dictionary-Wide Actions:
  - Get the total number of entries
  - Accessing keys, values, key-value pairs
  - Check if a key exists
  - Loop through keys, values, key-value pairs

Number of entries of a dictionary

- The `len()` function returns the number of key-value pairs.

```
d = {10:'dez', 20:'vinte', 1000:'mil'}
print(len(d))      #->  3
```

Accessing keys, values and items

- Three methods return sequences of keys, values and items.

- Method keys() returns object with all keys

```
d.keys()    #-> 10, 20, 1000
```

- Method values() returns object with all values

```
d.values() #-> 'dez', 'vinte', 'mil’
```

- Method items() returns tuples containing all key-value pairs

```
d.items()  #-> (10, 'dez'), (20, 'vinte'), (1000, 'mil')
```

Checking if a key exists

- The in operator tells you whether something appears as a key in the dictionary.
  - It is fast!

```
'two' in eng2sp#-> True ('two' is a key)
'uno' in eng2sp   #-> False ('uno' is not a key)
```

- To see whether something is a value in the dictionary, you could use (but this is slow):

```
'uno' in eng2sp.values()  #-> True
```

Accessing by key

- As mentioned before, trying to access an inexistent key is an error.
- You can avoid this error using the **get()** method, that returns a default value for inexistent keys.

  [Note: it doesn’t create the pair for the inexistent key]

```
d = {10:'dez', 20:'vinte', 1000:'mil’}

d.get(10)           #-> 'dez'  (same as d[10])
d.get(0)            #-> None   (no error!)
d.get(0, 'nada')    #-> 'nada' (no error)
0 in d              #-> False  (.get() did not change d)
print(d)   # {10:'dez', 20:'vinte', 1000:'mil’}
```

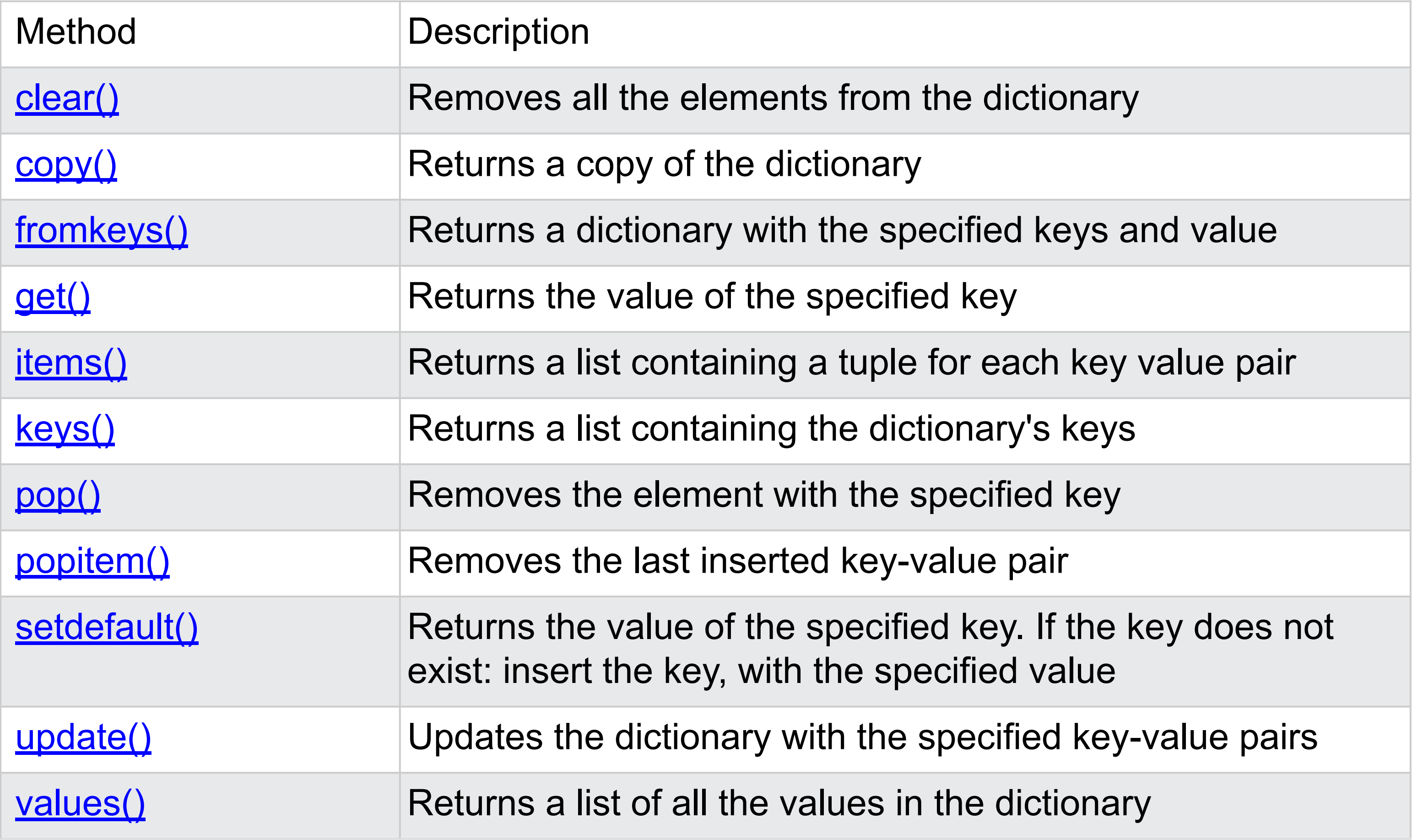
- The **setdefault()** method is similar, but it also creates a new item if it was missing!

```
d.setdefault(0, 'nada')  #-> 'nada'
0 in d                   #-> True
print(d)   # {10:'dez', 20:'vinte', 1000:'mil' 0:'nada'}
```

Dictionary methods

https://youtu.be/4t10v2QmTHU?si=ovgmtI0zxdAezxgk&t=415

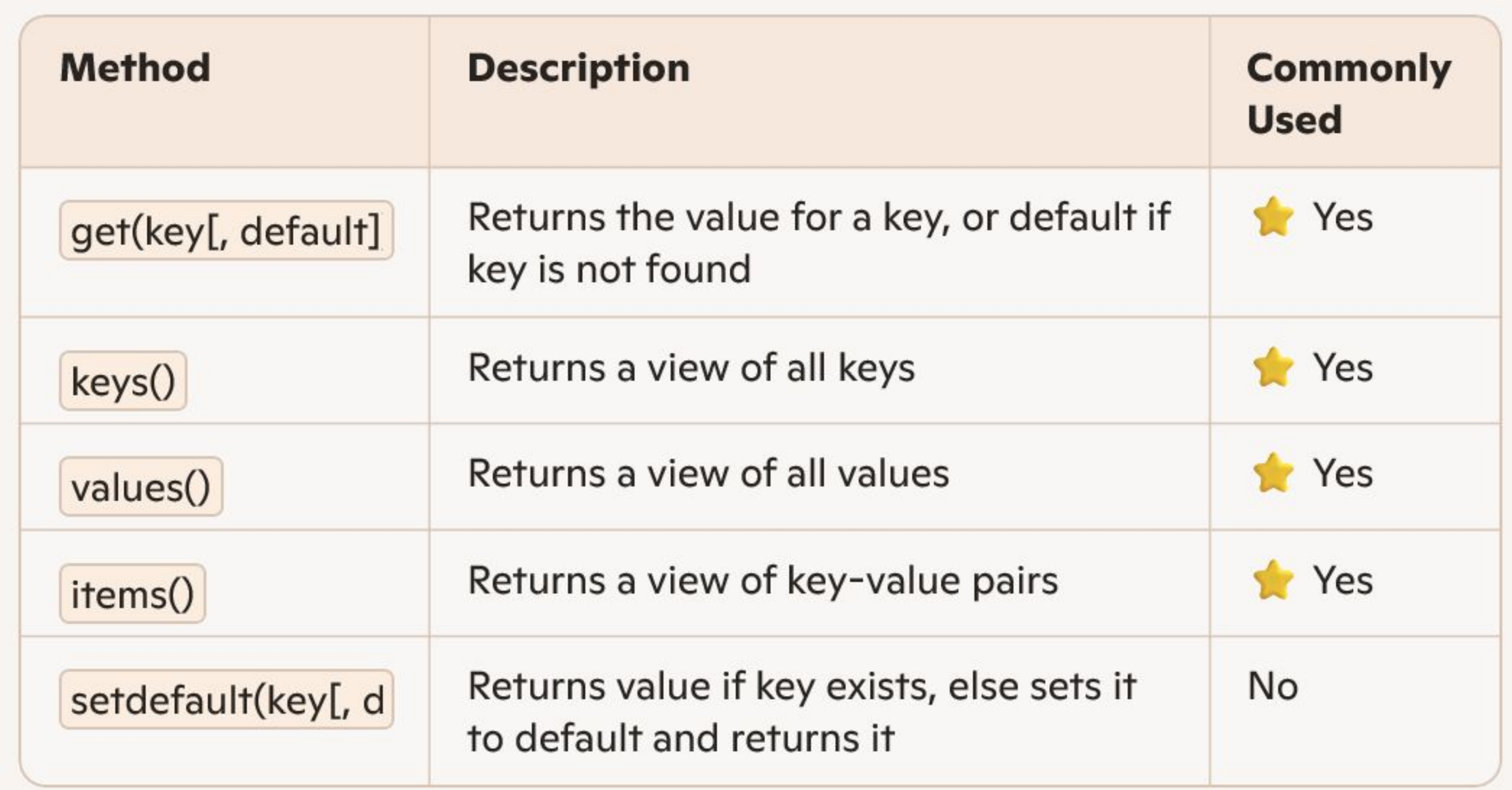
Dictionary methods - summary



Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

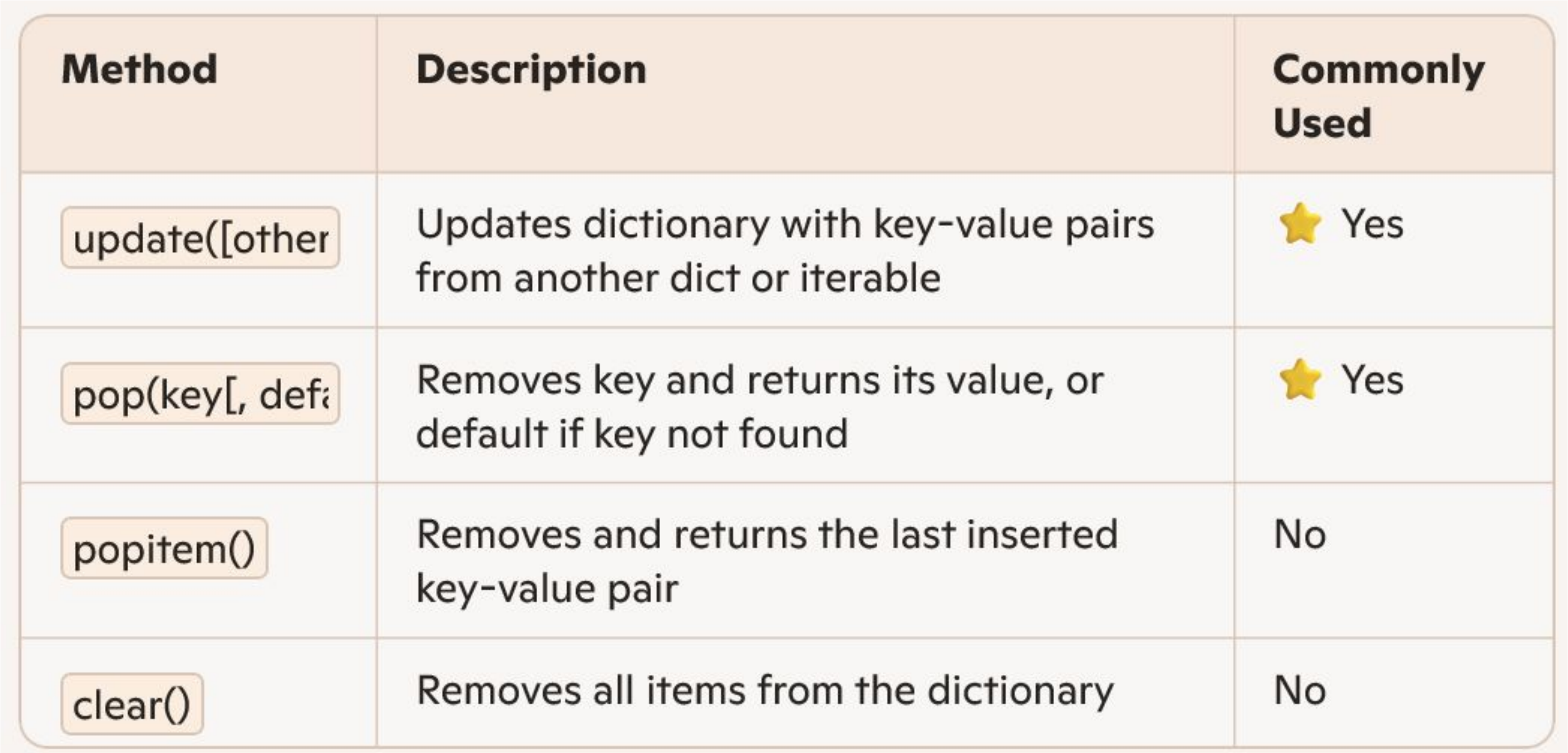
From: Python Dictionary Methods

Access & Inspection Methods



Method	Description	Commonly Used
get(key[, default]	Returns the value for a key, or default if key is not found	⭐ Yes
keys()	Returns a view of all keys	⭐ Yes
values()	Returns a view of all values	⭐ Yes
items()	Returns a view of key-value pairs	⭐ Yes
setdefault(key[, d	Returns value if key exists, else sets it to default and returns it	No

Modification Methods



Method	Description	Commonly Used
update([other	Updates dictionary with key-value pairs from another dict or iterable	⭐ Yes
pop(key[, def	Removes key and returns its value, or default if key not found	⭐ Yes
popitem()	Removes and returns the last inserted key-value pair	No
clear()	Removes all items from the dictionary	No

FIQUEI AQUI AULA 1

Dictionary traversal

- The **for** instruction is commonly used to traverse dictionary **keys**.

```
shop = {'eggs':24, 'bread':6,
        'coffee':3, 'sugar':1.0}
for key in shop:
    print(key, shop[key])
```

```
eggs 24
bread 6
sugar 1.0
coffee 3
```

Play ▸

- This is equivalent to:

```
for key in shop.keys():
    print(key, shop[key])
```

- We may also traverse (key, value) pairs directly:

```
for key, value in shop.items():
    print(key, value)
```

Application Examples (1)

Problem: Count how many times each letter appears in a message (string)

Possible solution:

Part 1- count number of occurrences of each letter

```
def count_occurences(text):
     # create empty dictionary
    counts = dict()

    # for all letters ...
    for letter in text:
        # if letter not yet in dictionary
        if letter not in counts:
            # create dict item, count = 1
            counts[letter] = 1
        else:
            # increase count by 1
            counts[letter] += 1
    return counts
```

Part 2: Show the results, by traversing the keys

```
message = 'parrot’
result = count_occurences(message)
for key in result:
    print(key, result[key])
```

Play ►

Application Examples (2)

Problem: Map from frequencies to letters

Possible solution:

Part 1- Get values for each key and use them as keys

```
def freq2letters(occurences):
    inverse = dict() # create dictionary
    # for all keys (letters)
    for key in occurences:
        # get value (count)
        value = occurences[key]
        # if value not yet in dictionary
        if value not in inverse:
            # create a new list with the key
            inverse[value] = [key]
        else:
            # append the key (a letter) to the
    list for that value
            inverse[value].append(key)
    return inverse
```

Part 2: Apply to the results of the function of the first example and show result

```
message = 'programming'
counts=
    count_occurences(message)
result = freq2letters(counts)
print(result)
```

```
{1: ['p', 'o', 'a', 'i', 'n'], 2: ['r', 'g', 'm']}
```

Play ▶

Updating dictionaries

- Solution for both examples include typical code for updating a dictionary

```
if letter not in counts:
    # create dict item, count = 1
    counts[letter] = 1
else:
    # increase count by 1
    counts[letter] += 1
```

```
if value not in inverse:
    # create a new list with the key
    inverse[value] = [key]
else:
    # append key to the list for the value
    inverse[value].append(key)
```

- Many other algorithms require updating a dictionary one item at a time.
- There are a few equivalent alternatives …

Updating dictionaries (continuation)

	Code

Variant A – Used in application examples and commonly used	`d = {}`
	`for c in message:`
	`    if c not in d:`
	`        d[c] = 1`
	`    else:`
	`        d[c] += 1`

Variant B – Similar to A but avoiding the else	`d = {}`
	`for c in message:`
	`    if c not in d:`
	`        d[c] = 0`
	`    d[c] += 1`

Variant C – Using method `get()`
Note the use of default value (zero) for non-existing keys.
	`d = {}`
	`for c in message:`
	`    d[c] = d.get(c, 0) + 1`

Variant D – Using method `setdefault()`	`d = {}`
	`for c in message:`
	`    d.setdefault(c, 0)`
	`    d[c] += 1`

Updating dictionaries (continuation)

- Example: Grouping words in lists according to word length.

```
d = {}
for w in wordlist:
    k = len(w)
    if k not in d:
        d[k] = [w]
    else:
        d[k].append(w)
```

```
d = {}
for w in wordlist:
    k = len(w)
    if k not in d:
        d[k] = []
    d[k].append(w)
```

```
d = {}
for w in wordlist:
    k = len(w)
    d[k] = d.get(k, [])
    d[k].append(w)
```

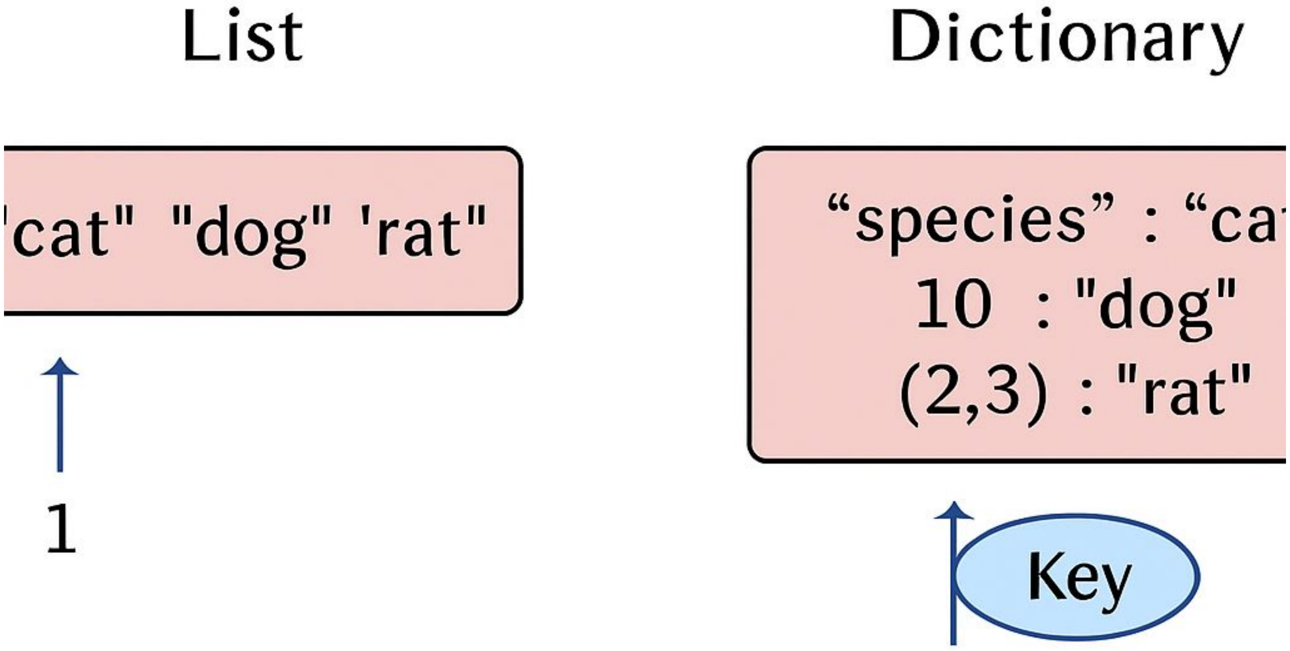
```
d = {}
for w in wordlist:
    k = len(w)
    d.setdefault(k, []).append(w)
```

```
wordlist=['to','be','or','not','to','be','that','is','the','question']
# All variants produce…
d -> {2: ['to', 'be', 'or', 'to', 'be', 'is'], 3: ['not', 'the'],
4: ['that'], 8: ['question']}
```

Play ▶

DICTIONARIES, LISTS AND TUPLES

Accessing lists and dictionaries



Mandatory integer indices for lists

Keys of various types for dictionaries

Dictionaries *versus* lists

- When accessing items, a dictionary is a kind of generalized list.
- Unlike lists, which require integer indices, dictionaries allow access using keys of various types
  - such as strings, numbers, or tuples.

```
lst = [50, 51, 52]
dic = {'um':1, 'vinte':20, 'mil':1000}
lst[1]        #-> 51
dic['mil']    #-> 1000
```

- Unlike lists, the order of items in a dictionary is irrelevant.

```
{'a':1, 'b':2} == {'b':2, 'a':1}    #-> True
[1, 2] == [2, 1]                    #-> False
```

- And we cannot take slices from dictionaries!

```
d = {10:'dez', 20:'vinte', 1000:'mil'}
d[10:20]     # NONSENSE! -> TypeError
```

Dictionaries and lists of tuples

- Method items() returns a sequence of tuples
  - where each tuple is a key-value pair.

```
d = {'a':0, 'b':1, 'c':2}
t = d.items()
#-> dict_items(('a', 0), ('c', 2), ('b', 1))
```

- We can use a list of tuples to initialize a new dictionary:

```
t = [('a', 0), ('c', 2), ('b', 1)]
d = dict(t) #-> {'a': 0, 'c': 2, 'b': 1}
```

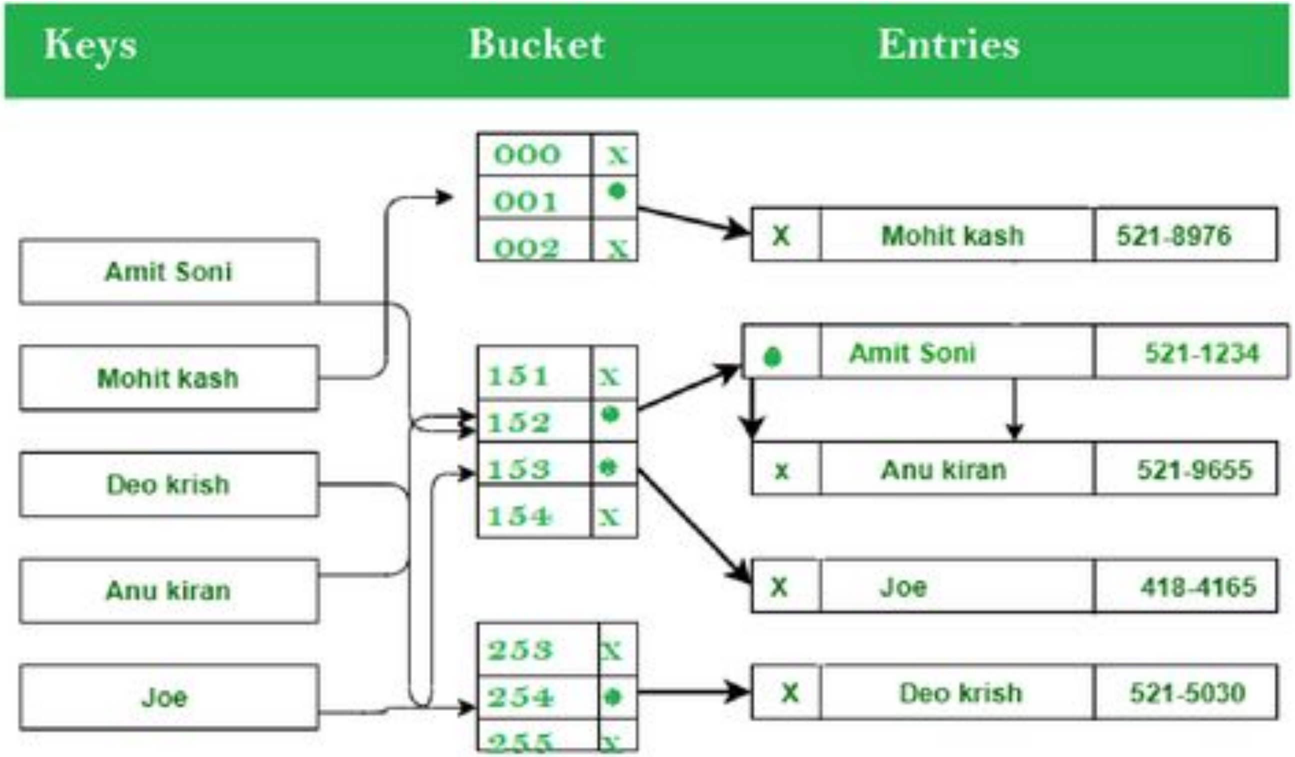
- Combining items, tuple assignment and for:

```
for key, val in d.items():
    print(val, key)
```

MORE ADVANCED

Implementation details

- Python uses hash tables to implement dictionaries.



- More information at [Internal Structure of Python Dictionary - GeeksforGeeks]()

Nested Dictionaries

- A dictionary can contain dictionaries;
  - this is called nested dictionaries.

```
child1 = {
  "name" : "Emil",
  "year" : 2004
}
child2 = {
  "name" : "Tobias",
  "year" : 2007
}


myfamily = {
  "child1" : child1,
  "child2" : child2,
  }
```

Nested Dictionaries

- To access values in a nested dictionary, start by referencing the outer dictionary, then progressively access each inner dictionary using its corresponding key

- Example: Print the name of child2

```
print(myfamily["child2"]["name"])
```

- `More info at` Python - Nested Dictionaries

SOME COMMON MISTAKES BEGINNERS MAKE WITH DICTIONARIES IN PYTHON

And How to Identify & Fix Them

Using Unhashable Types as Keys

- Mistake: Trying to use mutable types (like lists or dictionaries) as dictionary keys.
- Why it fails: Dictionary keys must be hashable (immutable).
- Example:

```
my_dict = {[1, 2]: "value"}
# ❌ TypeError: unhashable type: 'list'
```

Accessing Missing Keys Without Handling Errors

- Mistake: Accessing a key that doesn’t exist, without checking or using .get().
- Why it fails: Raises a KeyError.
- Example:

```
my_dict = {"name": "Alice"}
print(my_dict["age"])  # ❌ KeyError

# Better
print(my_dict.get("age", "Not found")) # ✅
```

Confusing Keys and Values

- Mistake: Trying to access a value using the wrong syntax or assuming values are keys.
- Why it fails: Only keys can be used to retrieve values.
- Example:

```
my_dict = {"name": "Alice"}
print(my_dict["Alice"])  # ❌ KeyError
```

Overwrite Values

- Mistake: Assuming adding a key again will create a new entry.
- Why it fails: Keys must be unique; assigning a value to an existing key overwrites it.
- Example:

```
my_dict = {"name": "Alice"}
my_dict["name"] = "Bob"
                     # Overwrites "Alice"
```

Misusing Dictionary Methods

- Mistake: Misunderstanding .keys(), .values(), or .items() as lists.
- Why it fails: These return *view objects*, not lists.

```
my_dict = {"a": 1, "b": 2}
print(type(my_dict.keys())) #-> <class 'dict_keys'>
```

- Example:

```
my_dict = {"a": 1, "b": 2}
print(my_dict.keys()[0])  # ❌ TypeError

# Correct:
print(list(my_dict.keys())[0])  # ✅
```

Assuming Dictionaries Are Ordered (Pre-Python 3.7)

- Mistake: Expecting insertion order in older Python versions.
- Why it fails: Dictionaries were unordered before Python 3.7.

- Note: From Python 3.7+, dictionaries preserve insertion order.

RETURNING TO OUR INITIAL PROBLEM …

Possible Solution in Python

```python
sentence = "apple banana apple orange banana apple"

# Split the sentence into words
words = sentence.split()

# Initialize an empty dictionary
freq = {}

# Count each word
for word in words:
    freq[word] = freq.get(word, 0) + 1

# Print the result
for word, count in freq.items():
    print(f"{word} → {count}")
```

Exercises

- Do these CodeCheck exercises.
- Do the tutorial at Python Dictionaries
- Solve the questions at Dictionaries - Practice.ipynb – Colab