Lecture 18: Dictionaries

Comp 102

Forman Christian University

Recap

Last Time...

- **Exceptions** — Python's error-reporting mechanism
  - `try/except` to catch; `raise` to throw
- **Assertions** — programmer's sanity check
  - `assert condition, "message"`
- **Class Grades** example with list-of-lists
  - `stu[0]`, `stu[1]` everywhere...

Last Time...

- **Exceptions** — Python's error-reporting mechanism
  - `try/except` to catch; `raise` to throw
- **Assertions** — programmer's sanity check
  - `assert condition, "message"`
- **Class Grades** example with list-of-lists
  - `stu[0]`, `stu[1]` everywhere...

Let's revisit that class grades example...

Remember This?

From Lecture 17 — the class grades data:

```
class_list = [
  [['Ali', 'Khan'],      [85, 90, 78]],
  [['Fatima', 'Noor'],   [92, 88, 95]],
  [['Deadpool'],         []]
]

for stu in class_list:
    print(stu[0], avg(stu[1]))
    # What does stu[0] mean?  stu[1]?
```

Remember This?

From Lecture 17 — the class grades data:

```
class_list = [
  [['Ali', 'Khan'],      [85, 90, 78]],
  [['Fatima', 'Noor'],   [92, 88, 95]],
  [['Deadpool'],         []]
]

for stu in class_list:
    print(stu[0], avg(stu[1]))
    # What does stu[0] mean?  stu[1]?
```

Code is **unreadable**. You have to *memorize* what each index means.

Parallel Lists Are Also Messy

Another approach — separate lists for each kind of info:

```
1  names  = ['ali', 'bilal', 'fahad', 'zain']
2  grades = ['A+',  'B',     'C',     'A'    ]
3
4  def get_grade(name, name_list, grade_list):
5      i = name_list.index(name)
6      return grade_list[i]
```

Parallel Lists Are Also Messy

Another approach — separate lists for each kind of info:

```
1  names  = ['ali', 'bilal', 'fahad', 'zain']
2  grades = ['A+',  'B',     'C',     'A'    ]
3
4  def get_grade(name, name_list, grade_list):
5      i = name_list.index(name)
6      return grade_list[i]
```

- Must maintain **multiple** lists in sync
- `.index()` is **slow** — scans the whole list
- What if someone adds to one list but not the other?

Real-World Hook

Your phone's **Contacts** app:

Real-World Hook

Your phone's **Contacts** app:

- You type a **name** → get a **number**

Real-World Hook

Your phone's **Contacts** app:

- You type a **name** → get a **number**
- No searching by index 0, 1, 2...

Real-World Hook

Your phone's **Contacts** app:

- You type a **name** → get a **number**
- No searching by index 0, 1, 2...
- No parallel lists of names and numbers

Real-World Hook

Your phone's **Contacts** app:

- You type a **name** → get a **number**
- No searching by index 0, 1, 2...
- No parallel lists of names and numbers

What if Python had **something like this**?

Dictionaries

Big Idea

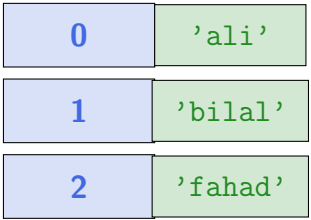
A **dictionary** maps **keys** to **values**.

Instead of integer indices,
use **custom labels**.

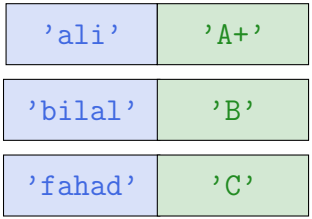
Like a real dictionary: word → definition.

List vs Dictionary

LIST

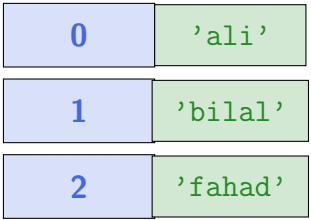


DICT

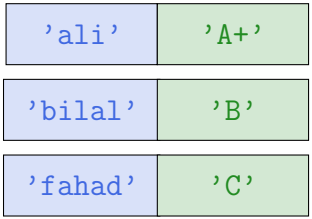


List vs Dictionary

LIST



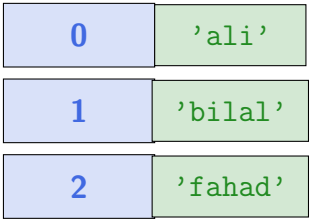
DICT



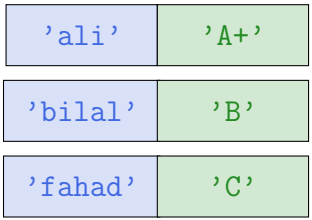
- Keys must be **immutable** (str, int, float, tuple) and **unique**

List vs Dictionary

LIST



DICT



- Keys must be **immutable** (str, int, float, tuple) and **unique**
- Values can be **anything**

Working with Dictionaries

Creating Dictionaries

Empty dictionary:

```
d = {}
```

Creating Dictionaries

Empty dictionary:

```
d = {}
```

Dictionary with data:

```
grades = {'ali': 'A+', 'bilal': 'B'}
```

Creating Dictionaries

Empty dictionary:

```
d = {}
```

Dictionary with data:

```
grades = {'ali': 'A+', 'bilal': 'B'}
```

key : value pairs, separated by commas

Accessing Values

Use the **key** as the index:

```
grades = {'ali': 'A+', 'bilal': 'B'}

grades['ali']

grades['fatima']
```

Accessing Values

Use the **key** as the index:

```
grades = {'ali': 'A+', 'bilal': 'B'}

grades['ali']      # 'A+'

grades['fatima']
```

Accessing Values

Use the **key** as the index:

```
grades = {'ali': 'A+', 'bilal': 'B'}

grades['ali']      # 'A+'

grades['fatima']   # KeyError!
```

Accessing Values

Use the **key** as the index:

```
grades = {'ali': 'A+', 'bilal': 'B'}

grades['ali']      # 'A+'

grades['fatima']   # KeyError!
```

KeyError — a new exception type!

Remember `try/except` from Lecture 17? We'll use it soon.

The Class Grades: Solved

Before (list-of-lists):

```
class_list = [
 ['Ali','Khan'], [85,90,78]],
 ['Fatima','Noor'],[92,88,95]]
]
# stu[0] = name, stu[1] = grades
```

The Class Grades: Solved

Before (list-of-lists):

```
class_list = [
 [['Ali','Khan'], [85,90,78]],
 [['Fatima','Noor'],[92,88,95]]
]
# stu[0] = name, stu[1] = grades
```

After (dict):

```
grades = {
 'Ali':    [85, 90, 78],
 'Fatima': [92, 88, 95]
}
# grades['Ali'] = [85, 90, 78]
```

The Class Grades: Solved

Before (list-of-lists):

```
class_list = [
 [['Ali','Khan'], [85,90,78]],
 [['Fatima','Noor'],[92,88,95]]
]
# stu[0] = name, stu[1] = grades
```

After (dict):

```
grades = {
 'Ali':    [85, 90, 78],
 'Fatima': [92, 88, 95]
}
# grades['Ali'] = [85, 90, 78]
```

One line replaces the entire search function:

```
grades['Ali']   # [85, 90, 78]
```

You Try!

Write `find_grades(grades, students)` — returns a list of grades for the given students.

```
def find_grades(grades, students):
    """ grades: dict mapping names (str) to grades (str)
        students: list of student names
        Returns list of grades for students (same order) """
    pass

d = {'Ali':'B', 'Bilal':'C', 'Hamza':'B', 'Sana':'A'}
find_grades(d, ['Bilal', 'Sana'])  # returns ['C', 'A']
```

You Try! — Solution

Solution:

```
def find_grades(grades, students):
    """ grades: dict mapping names (str) to grades (str)
        students: list of student names
        Returns list of grades for students (same order) """
    result = []
    for s in students:
        result.append(grades[s])
    return result
```

Example Usage:

```
scores = {'Zain':'A+', 'Fatima':'B+', 'Omar':'A', 'Hira':'C'}
find_grades(scores, ['Omar', 'Hira'])          # returns ['A', 'C']
find_grades(scores, ['Fatima'])                # returns ['B+']
find_grades(scores, ['Zain','Fatima','Omar'])  # returns ['A+','B+','A']
```

Dictionary Operations

```
grades = {'ali': 'A+', 'bilal': 'B'}
```

Dictionary Operations

```
grades = {'ali': 'A+', 'bilal': 'B'}

# Add a new entry:
grades['zain'] = 'A'
```

Dictionary Operations

```
grades = {'ali': 'A+', 'bilal': 'B'}

# Add a new entry:
grades['zain'] = 'A'

# Modify an existing entry:
grades['ali'] = 'B'
```

Dictionary Operations

```
grades = {'ali': 'A+', 'bilal': 'B'}

# Add a new entry:
grades['zain'] = 'A'

# Modify an existing entry:
grades['ali'] = 'B'

# Delete an entry:
del grades['bilal']
```

Dictionary Operations

```
grades = {'ali': 'A+', 'bilal': 'B'}

# Add a new entry:
grades['zain'] = 'A'

# Modify an existing entry:
grades['ali'] = 'B'

# Delete an entry:
del grades['bilal']
```

The dictionary is **mutated** — just like lists.

Membership Testing

```
grades = {'ali': 'A+', 'bilal': 'B'}

'ali' in grades

'zain' in grades

'A+' in grades
```

Membership Testing

```
grades = {'ali': 'A+', 'bilal': 'B'}

'ali' in grades    # True

'zain' in grades

'A+' in grades
```

Membership Testing

```
grades = {'ali': 'A+', 'bilal': 'B'}

'ali' in grades    # True

'zain' in grades   # False

'A+' in grades
```

Membership Testing

```
grades = {'ali': 'A+', 'bilal': 'B'}

'ali' in grades    # True

'zain' in grades   # False

'A+' in grades     # False!
```

Membership Testing

```
grades = {'ali': 'A+', 'bilal': 'B'}

'ali' in grades    # True

'zain' in grades   # False

'A+' in grades     # False!
```

`in` checks **keys** only, **not** values!

You Try!

Write `find_in_L(Ld, k)` — returns `True` if `k` is a key in **any** dict of list `Ld`.

```
1  def find_in_L(Ld, k):
2      """ Ld: list of dictionaries, k: a key
3          Returns True if k is a key in any dictionary of Ld """
4      pass
5
6  d1, d2 = {1:2, 3:4, 5:6}, {2:4, 4:6}
7  find_in_L([d1, d2], 2)    # True
8  find_in_L([d1, d2], 25)   # False
```

You Try!

Write `find_in_L(Ld, k)` — returns `True` if `k` is a key in **any** dict of list `Ld`.

```
def find_in_L(Ld, k):
    """ Ld: list of dictionaries, k: a key
        Returns True if k is a key in any dictionary of Ld """
    pass

d1, d2 = {1:2, 3:4, 5:6}, {2:4, 4:6}
find_in_L([d1, d2], 2)   # True
find_in_L([d1, d2], 25)  # False
```

Solution:

```
def find_in_L(Ld, k):
    for d in Ld:
        if k in d:
            return True
    return False
```

Handling KeyError

Remember `try/except` from Lecture 17?

```
1 def safe_lookup(d, key):
2     """ Return d[key], or None if
3         key is not found. """
4     try:
5         return d[key]
6     except KeyError:
7         return None
```

Handling KeyError

Remember try/except from Lecture 17?

```
def safe_lookup(d, key):
    """ Return d[key], or None if
        key is not found. """
    try:
        return d[key]
    except KeyError:
        return None

grades = {'ali': 'A+', 'bilal': 'B'}
safe_lookup(grades, 'ali')      # 'A+'
safe_lookup(grades, 'fatima')   # None (no crash!)
```

Aliasing

Alias (same object):

```
1 d1 = {'a': 1, 'b': 2}
2 d2 = d1
3
4 d2['a'] = 99
5 print(d1)  # {'a': 99, 'b': 2}
```

Aliasing

Alias (same object):

```
1 d1 = {'a': 1, 'b': 2}
2 d2 = d1
3
4 d2['a'] = 99
5 print(d1)  # {'a': 99, 'b': 2}
```

Copy (new object):

```
1 d1 = {'a': 1, 'b': 2}
2 d2 = d1.copy()
3
4 d2['a'] = 99
5 print(d1)  # {'a': 1, 'b': 2}
```

Aliasing

Alias (same object):

```
1 d1 = {'a': 1, 'b': 2}
2 d2 = d1
3
4 d2['a'] = 99
5 print(d1)  # {'a': 99, 'b': 2}
```

Copy (new object):

```
1 d1 = {'a': 1, 'b': 2}
2 d2 = d1.copy()
3
4 d2['a'] = 99
5 print(d1)  # {'a': 1, 'b': 2}
```

Same idea as lists — you already know this from Week 6!

Big Idea

Dictionaries are **mutable** — like lists.

Add, modify, delete entries.

Aliasing applies.

Iterating over Dictionaries

.keys(), .values(), .items()

```
grades = {'ali': 'A+', 'bilal': 'B',
          'fahad': 'C', 'zain': 'A'}
```

.keys(), .values(), .items()

```
grades = {'ali': 'A+', 'bilal': 'B',
          'fahad': 'C', 'zain': 'A'}

grades.keys()
# dict_keys(['ali','bilal','fahad','zain'])
```

.keys(), .values(), .items()

```
grades = {'ali': 'A+', 'bilal': 'B',
          'fahad': 'C', 'zain': 'A'}

grades.keys()
# dict_keys(['ali','bilal','fahad','zain'])

grades.values()
# dict_values(['A+','B','C','A'])
```

.keys(), .values(), .items()

```
grades = {'ali': 'A+', 'bilal': 'B',
          'fahad': 'C', 'zain': 'A'}

grades.keys()
# dict_keys(['ali','bilal','fahad','zain'])

grades.values()
# dict_values(['A+','B','C','A'])

grades.items()
# dict_items([('ali','A+'),('bilal','B'),...])
```

Looping Over a Dictionary

Loop over **keys** (default):

```
1 for k in grades:
2     print(k, grades[k])
```

Looping Over a Dictionary

Loop over **keys** (default):

```
for k in grades:
    print(k, grades[k])
```

Loop over **key-value pairs** (most common):

```
for k, v in grades.items():
    print(k, v)
```

Looping Over a Dictionary

Loop over **keys** (default):

```
for k in grades:
    print(k, grades[k])
```

Loop over **key-value pairs** (most common):

```
for k, v in grades.items():
    print(k, v)
```

Both produce:

```
ali A+
bilal B
fahad C
zain A
```

You Try!

Write `count_matches(d)` — returns how many entries have key **equal** to value.

```
1  def count_matches(d):
2      """ d: a dictionary
3      Returns count of entries where key == value """
4      pass
5
6  count_matches({1:2, 3:4, 5:6})        # 0
7  count_matches({1:2, 'a':'a', 5:5})    # 2
```

You Try!

Write `count_matches(d)` — returns how many entries have key **equal** to value.

```
def count_matches(d):
    """ d: a dictionary
    Returns count of entries where key == value """
    pass

count_matches({1:2, 3:4, 5:6})        # 0
count_matches({1:2, 'a':'a', 5:5})    # 2
```

Solution:

```
def count_matches(d):
    count = 0
    for k, v in d.items():
        if k == v:
            count += 1
    return count
```

Nested Dictionaries (Preview)

Values can be **dictionaries** themselves:

```
grades = {
  'ali':  {'quiz': [5,4,4], 'final': 'B'},
  'zaid': {'quiz': [6,7,8], 'final': 'A'}
}
```

Nested Dictionaries (Preview)

Values can be **dictionaries** themselves:

```
grades = {
  'ali':  {'quiz': [5,4,4], 'final': 'B'},
  'zaid': {'quiz': [6,7,8], 'final': 'A'}
}
```

```
grades['ali']                  # {'quiz':[5,4,4],'final':'B'}
grades['ali']['quiz']          # [5, 4, 4]
grades['ali']['quiz'][0]       # 5
```

Nested Dictionaries (Preview)

Values can be **dictionaries** themselves:

```
grades = {
  'ali':  {'quiz': [5,4,4], 'final': 'B'},
  'zaid': {'quiz': [6,7,8], 'final': 'A'}
}
```

```
grades['ali']                  # {'quiz':[5,4,4],'final':'B'}
grades['ali']['quiz']          # [5, 4, 4]
grades['ali']['quiz'][0]       # 5
```

Dictionary of dictionaries — more in **Lecture 19** (Data Abstraction).

List vs Dictionary

	List	Dictionary
Order	Ordered sequence	No guaranteed order
Index	Integer (0, 1, 2...)	Any immutable type
Access	`L[0]`	`d['key']`
`in`	Checks **values**	Checks **keys**
Mutable?	Yes	Yes
Use when	Order matters	Lookup by name

Putting It All Together

Extended Example: Word Frequency

Given song lyrics, **which words appear most often**?

Extended Example: Word Frequency

Given song lyrics, **which words appear most often**?

- Spell-checkers count word frequencies

Extended Example: Word Frequency

Given song lyrics, **which words appear most often**?

- Spell-checkers count word frequencies
- Search engines rank pages by word counts

Extended Example: Word Frequency

Given song lyrics, **which words appear most often**?

- Spell-checkers count word frequencies
- Search engines rank pages by word counts
- We'll build this in **two functions**:
  - `generate_word_dict` — count each word
  - `find_frequent_word` — find the most common

Step 1: generate_word_dict

Count how many times each word appears:

```
1  song = "RAH RAH DA DA DA ROM MAH RO MAH MAH"
2
3  def generate_word_dict(song):
4      words_list = song.lower().split()
5      word_dict = {}
6      for w in words_list:
7          if w in word_dict:
8              word_dict[w] += 1
9          else:
10             word_dict[w] = 1
11     return word_dict
```

Step 1: generate_word_dict

Count how many times each word appears:

```
1  song = "RAH RAH DA DA DA ROM MAH RO MAH MAH"
2
3  def generate_word_dict(song):
4      words_list = song.lower().split()
5      word_dict = {}
6      for w in words_list:
7          if w in word_dict:
8              word_dict[w] += 1
9          else:
10             word_dict[w] = 1
11     return word_dict
```

Trace through the song:

Word	word_dict
'rah'	{'rah':1}
'rah'	{'rah':2}
'da'	{'rah':2, 'da':1}
...	{'rah':2, 'da':3, 'rom':1, 'mah':3, 'ro':1}

Step 2: find_frequent_word

Find the word(s) with the highest count:

```
1  def find_frequent_word(word_dict):
2      highest = max(word_dict.values())
3      words = []
4      for k, v in word_dict.items():
5          if v == highest:
6              words.append(k)
7      return (words, highest)
```

Step 2: `find_frequent_word`

Find the word(s) with the highest count:

```
1  def find_frequent_word(word_dict):
2      highest = max(word_dict.values())
3      words = []
4      for k, v in word_dict.items():
5          if v == highest:
6              words.append(k)
7      return (words, highest)
```

```
word_dict = {'rah':2, 'da':3, 'rom':1,
             'mah':3, 'ro':1}
find_frequent_word(word_dict)
# (['da', 'mah'], 3)
```

Step 2: `find_frequent_word`

Find the word(s) with the highest count:

```
1  def find_frequent_word(word_dict):
2      highest = max(word_dict.values())
3      words = []
4      for k, v in word_dict.items():
5          if v == highest:
6              words.append(k)
7      return (words, highest)
```

```
word_dict = {'rah':2, 'da':3, 'rom':1,
             'mah':3, 'ro':1}
find_frequent_word(word_dict)
# (['da', 'mah'], 3)
```

Uses `.values()` for `max`, `.items()` to collect **ties**.

Putting It Together

```
1 song = "RAH RAH DA DA DA ROM MAH RO MAH MAH"
2
3 word_dict = generate_word_dict(song)
4 print(word_dict)
5 # {'rah':2, 'da':3, 'rom':1, 'mah':3, 'ro':1}
6
7 most_frequent = find_frequent_word(word_dict)
8 print(most_frequent)
9 # (['da', 'mah'], 3)
```

Putting It Together

```
song = "RAH RAH DA DA DA ROM MAH RO MAH MAH"

word_dict = generate_word_dict(song)
print(word_dict)
# {'rah':2, 'da':3, 'rom':1, 'mah':3, 'ro':1}

most_frequent = find_frequent_word(word_dict)
print(most_frequent)
# (['da', 'mah'], 3)
```

Pattern: Build a dictionary → query the dictionary.
This **counting pattern** is one of the most common uses of dictionaries.

You Try!

Write `keys_with_value(d, target)` — returns all keys that map to `target`.

```
1  def keys_with_value(d, target):
2      """ d: a dictionary, target: a value
3          Returns list of keys where d[key] == target """
4      pass
5
6  grades = {'ali': 'A', 'bilal': 'B', 'zain': 'A'}
7  keys_with_value(grades, 'A')   # ['ali', 'zain']
8  keys_with_value(grades, 'C')   # []
```

You Try!

Write `keys_with_value(d, target)` — returns all keys that map to `target`.

```
def keys_with_value(d, target):
    """ d: a dictionary, target: a value
        Returns list of keys where d[key] == target """
    pass

grades = {'ali': 'A', 'bilal': 'B', 'zain': 'A'}
keys_with_value(grades, 'A')   # ['ali', 'zain']
keys_with_value(grades, 'C')   # []
```

Solution:

```
def keys_with_value(d, target):
    result = []
    for k, v in d.items():
        if v == target:
            result.append(k)
    return result
```

Summary

Summary

- **Dictionaries** — map keys to values
  - Create: `{'key': 'value'}`
  - Access: `d['key']`; `KeyError` if missing
  - Modify: `d['key'] = val`; `del d['key']`
  - Membership: `'key' in d` (keys only!)
- **Iteration** — `.keys()`, `.values()`, `.items()`
  - `for k, v in d.items():` is most common
- **Counting pattern** — build a frequency dictionary
  - `if w in d: d[w] += 1; else: d[w] = 1`

Big Idea

Lists are **numbered** lockers.

Dictionaries are **labeled** lockers.

Use a dictionary when you need to look up by **name**, not by **position**.

Questions?