Week 9
Sets and Dictionaries

1811ICT Programming Principles

This chapter explains the concepts and Python techniques covered in the Week 9 lecture materials, with examples designed to help you understand when and how to use sets and dictionaries.

This chapter is based on Week 9 lecture slides and the accompanying Sections 22, 23, and 24. The chapter follows the terminology and examples used in those materials while organising them into a student-oriented progression from concepts to Python operations and practical programming patterns.

Learning Objectives

- Explain what a set is and why its elements are unique.
- Create, modify, compare, and combine Python sets.
- Use union, intersection, difference, and symmetric difference.
- Use dictionaries to associate keys with values.
- Create, access, update, and remove dictionary entries.
- Use get(), keys(), values(), items(), and update().
- Use dictionaries for counting and frequency problems.
- Understand references, mutation, reassignment, and aliasing in collections.
- Use set comprehensions and dictionary comprehensions.
- Recognise whether a set or dictionary is the appropriate data structure for a problem.

Python Compound Types

Python has several built-in compound types that act as containers for other values. We focus on sets and dictionaries in this week.

Type	Example	Main characteristic	Typical purpose
list	[1, 2, 3]	Ordered collection	Store a sequence
tuple	(1, 2, 3)	Immutable ordered collection	Store a fixed sequence
dict	{'a': 1, 'b': 2}	Key-value mapping	Associate information
set	{1, 2, 3}	Unique collection	Membership and uniqueness

The Set Concept

What Is a Set?

A set is a collection of distinct objects. In other words, an element can occur in a set only once, and the order of elements is not the important property.

```
A = {1, 999, 42}

{1, 2, 2, 2, 3}    # duplicates are eliminated
```

Sets are useful when we care about whether an item is present and do not want duplicate items.

Membership

```
primes = {2, 3, 5, 7}

print(3 in primes)       # True
print(4 in primes)       # False
print(4 not in primes)   # True
```

Empty Sets

In mathematical notation an empty set may be written with empty braces. In Python, however, the Python-specific syntax for an empty set is set(). Empty braces {} create an empty dictionary.

```
empty_set = set()
empty_dict = {}

print(type(empty_set))    # <class 'set'>
print(type(empty_dict))   # <class 'dict'>
```

This distinction is important in Python because both sets and dictionaries use braces for non-empty literals.

Creating Sets

```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}

a = {1, 2}
a.add(5)

b = set([1, 2, 2, 2, 3])
print(b)                  # {1, 2, 3}
```

A set can also be created from another collection using set(). Repeated values disappear.

Set Elements

Note that set elements must be immutable. Thus numbers, strings, Boolean values, and tuples can be set elements, while lists, sets, and dictionaries cannot be set elements.

```
valid = {1, "hello", True, (2, 3)}

# Invalid examples:
# {[1, 2]}
# {{1, 2}}
# {{"a": 1}}
```

Set Operations

Suppose:

```
primes = {2, 3, 5, 7}
odds = {1, 3, 5, 7, 9}
```

Operation	Operator	Method	Meaning
Union	|	union()	Elements in either set
Intersection	&	intersection()	Elements in both sets
Difference	-	difference()	Elements in the first set but not the second
Symmetric difference	^	symmetric_difference()	Elements in exactly one set

Union

```
print(primes | odds)
# {1, 2, 3, 5, 7, 9}
```

Union combines the elements of both sets, keeping each element only once.

Intersection

```
print(primes & odds)
# {3, 5, 7}
```

Intersection contains only the elements common to both sets.

Difference

```
print(primes - odds)
# {2}
```

Difference is directional: A - B contains elements in A that are not in B. Therefore A - B and B - A can be different.

Symmetric Difference

```
print(primes ^ odds)
# {1, 2, 9}
```

Symmetric difference contains elements that occur in only one of the two sets. It excludes elements common to both.

```
s1 ^ s2    # equivalent to (s1 - s2) | (s2 - s1)
```

Set Methods

Method	Effect	Important point
s1.update(s2)	Union	Changes s1; returns None
s1.intersection_update(s2)	Intersection	Changes s1; returns None
s1.difference_update(s2)	Difference	Changes s1; returns None
s1.symmetric_difference_update(s2)	Symmetric difference	Changes s1; returns None
s.add(e)	Add e	Duplicates are not created
s.discard(e)	Remove e if present	No error if absent
s.remove(e)	Remove e	Error if absent
s.pop()	Remove and return an arbitrary element	Error if empty
s.clear()	Remove all elements	Leaves an empty set
del s	Delete the set variable	Variable is deleted

```
s = {1, 2, 3}
s.update({3, 4, 5})
print(s)                    # {1, 2, 3, 4, 5}
```

Comparing Sets

Expression	Meaning	Example
s1.issubset(s2) / s1 <= s2	s1 is a subset of s2	{1,2} <= {1,2,3}
s1 < s2	s1 is a proper subset of s2	{1,2} < {1,2,3}
s1.issuperset(s2) / s1 >= s2	s1 is a superset of s2	{1,2,3} >= {1,2}
s1 > s2	s1 is a proper superset of s2	{1,2,3} > {1,2}
s1.isdisjoint(s2)	No common elements	{1,2}.isdisjoint({3,4})

A set is a subset of itself, so C <= C is True. A proper subset must be smaller, so C < C is False.

Practical Set Example: Distinct Words

A classic set problem is to read a file and print all distinct words in ascending order. The set automatically removes duplicates.

```
filename = input("Enter file name: ")

with open(filename, "r") as file:
    words = set()

    for line in file:
        for word in line.split():
            words.add(word)

for word in sorted(words):
    print(word)
```

sorted(words) is used only when producing ordered output. The set itself is not an ordered sequence.

Dictionaries

A dictionary is a mapping from unique keys to associated values. This is similar to a real dictionary: a word is a key and its meaning is the associated value.

```
numbers = {"one": 1, "two": 2, "three": 3}

print(numbers["two"])
# 2
```

Dictionary keys are unique. Values do not have to be unique. Dictionary keys as needing to be immutable, while values may be mutable.

Creating Dictionaries

```
d1 = {"a": 1, "b": 2}
d2 = dict()

d3 = dict([
    ("Andrew", "x55016"),
    ("Jun", "x55017")
])

d1["c"] = 3
```

Using d[key] = value adds a new entry when the key is new. If the key already exists, the assignment replaces its associated value.

Empty Dictionaries

```
d1 = {}
d2 = dict()

print(d1 == d2)     # True
```

Dictionary Operations

Looking Up Values

```
phonebook = {"John": 28630, "Bela": 28761, "Alan": 28671}

print(phonebook["Alan"])     # 28671
```

Direct lookup with d[key] requires the key to exist. If the key does not exist, an exception is raised.

get()

```
print(phonebook.get("Alan"))
print(phonebook.get("Mary"))
print(phonebook.get("Mary", 0))
```

get(key, default) returns the value for the key. If the key is absent, the supplied default is returned.

Membership

```
print("Alan" in phonebook)       # True
print("Mary" not in phonebook)   # True
```

For a dictionary, in and not in test the keys.

Updating

```
phonebook["Mary"] = 29999
phonebook["Alan"] = 30000

d1 = {"a": 1, "b": 2}
d2 = {"b": 3, "c": 4}
d1.update(d2)
print(d1)
# {'a': 1, 'b': 3, 'c': 4}
```

Dictionary Methods

Method	Purpose
d.clear()	Remove all entries
d.get(key, default)	Get a value, with an optional fallback
d.setdefault(key, default)	Get a value; if absent, add the key with the default
d.keys()	Access the keys
d.values()	Access the values
d.items()	Access key-value pairs
d.update(d2)	Add/update entries from d2
d.pop(key [, default])	Return and remove an entry
d.popitem()	Remove and return a key-value pair
del d[key]	Delete an entry

```
scores = {"Alice": 85, "Bob": 91}

for name in scores.keys():
    print(name)

for score in scores.values():
    print(score)

for name, score in scores.items():
    print(name, score)
```

We recommend sorted (sequence) when ordered output is required.

```
for name in sorted(scores):
    print(name, scores[name])
```

Merging Dictionaries with |

Note that Python 3.9 and later support dictionary merging with |.

```
dict1 = {"a": 1, "b": 2}
dict2 = {"b": 3, "c": 4}

merged = dict1 | dict2
print(merged)
# {'a': 1, 'b': 3, 'c': 4}
```

If a key occurs in both dictionaries, the value from the right-hand dictionary is used.

Collections Store References

We introduce an important Python concept: variables are references to objects, and collections store references to objects.

```
xs = [42]
ys = [3, 4]
zss = [xs, ys]

print(zss)
# [[42], [3, 4]]
```

zss contains references to the existing list objects xs and ys. It does not contain independent copies.

Mutation

```
ys.append(5)
print(zss)
# [[42], [3, 4, 5]]
```

append() changes the list object. Since zss refers to the same list object, the change is visible through zss.

Reassignment and Aliasing

```
xs = ys
print(xs)
# [3, 4, 5]

print(zss)
# [[42], [3, 4, 5]]
```

xs = ys makes xs another reference to the same list object as ys. It does not modify the old object that zss was referring to through its first element.

Remember the distinction: mutation changes an object; reassignment changes what a variable refers to.

Dictionaries for Counting

Counting occurrences is one of the most useful dictionary patterns. The item is the key and its frequency is the value.

```
counts = {}
names = ["csev", "cwen", "csev", "zqian", "cwen"]
```

```
for name in names:
    if name not in counts:
        counts[name] = 1
    else:
        counts[name] = counts[name] + 1

print(counts)
# {'csev': 2, 'cwen': 2, 'zqian': 1}
```

Simplifying Counting with get()

```
counts = {}

for name in names:
    counts[name] = counts.get(name, 0) + 1
```

This is an important pattern. If the key exists, get() returns its current count. If it does not exist, get() returns 0. Adding 1 then creates or increments the count.

Finding the Most Used Word

We extend the counting pattern to a file-processing problem: count every word and find the word with the greatest frequency.

```
filename = input("Enter file: ")

with open(filename) as file:
    counts = {}

    for line in file:
        words = line.split()

        for word in words:
            counts[word] = counts.get(word, 0) + 1

bigcount = None
bigword = None

for word, count in counts.items():
    if bigcount is None or count > bigcount:
        bigword = word
        bigcount = count

print(bigword, bigcount)
```

The outer loop processes lines and the inner loop processes the words on each line. The second loop examines the completed frequency dictionary.

Set Comprehensions

A set comprehension constructs a set using a compact expression.

```
squares = {n ** 2 for n in range(12)}
print(squares)

remainders = {a % 3 for a in range(1000)}
```

```
print(remainders)
# {0, 1, 2}
```

A set comprehension follows the uniqueness rule of sets. If different inputs produce the same result, only one copy appears in the set.

Dictionary Comprehensions

A dictionary comprehension adds a key:value expression to the comprehension.

```
squares = {n: n ** 2 for n in range(6)}
print(squares)
# {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

The expression before the colon creates the key and the expression after the colon creates the value.

Choosing Between a Set and a Dictionary

If the problem asks you to...	Use	Why
Keep only distinct values	Set	Duplicates are removed automatically
Test whether an item is present	Set	Membership is a natural set operation
Associate one item with information	Dictionary	Keys map to values
Count how often items occur	Dictionary	Item → frequency
Look up information by a unique identifier	Dictionary	Key-based lookup

Common Mistakes

- Using {} when you need an empty set. Use set().
- Assuming the displayed order of a set is meaningful.
- Trying to use a list, set, or dictionary as a set element.
- Trying to use a list as a dictionary key.
- Forgetting that dictionary keys must be unique.
- Confusing d[key] with d.get(key): a missing key raises an exception with direct lookup.
- Thinking that in tests dictionary values. It tests keys.
- Forgetting that d[key] = value can replace an existing value.
- Forgetting that set update methods modify the set and return None.
- Confusing A - B with B - A.
- Confusing mutation of an object with reassignment of a variable.

Appendix

Example 1: Remove duplicates

```
numbers = [4, 2, 4, 7, 2, 9, 7]
unique = set(numbers)

print(sorted(unique))
# [2, 4, 7, 9]
```

Example 2: Count occurrences

```
words = ["red", "blue", "red", "green", "blue", "red"]
counts = {}

for word in words:
    counts[word] = counts.get(word, 0) + 1

print(counts["red"])      # 3
print(counts["blue"])     # 2
```

Example 3: Set operations

```
A = {1, 2, 3, 4}
B = {3, 4, 5}

print(A | B)      # {1, 2, 3, 4, 5}
print(A & B)      # {3, 4}
print(A - B)      # {1, 2}
print(A ^ B)      # {1, 2, 5}
```

Quick Checklist

- I can explain what makes a set different from a list.
- I know that set() creates an empty set and {} creates an empty dictionary.
- I can calculate union, intersection, difference, and symmetric difference.
- I know the difference between remove() and discard().
- I can recognise subset, proper subset, superset, and disjoint sets.
- I can create and access dictionary key-value pairs.
- I know how get() behaves when a key is missing.
- I can use keys(), values(), and items().
- I can add, replace, update, and remove dictionary entries.
- I can use a dictionary to count occurrences.
- I understand why get(key, 0) is useful for counting.
- I can explain mutation versus reassignment and aliasing.
- I can write simple set and dictionary comprehensions.
- I can decide whether a set or dictionary is more suitable for a programming problem.

Summary

A set is a collection of unique elements and is particularly useful for membership, uniqueness, and mathematical set operations.

A dictionary is a collection of key-value associations and is particularly useful for lookup, counting, and storing information associated with unique keys.

- Sets: unique elements, membership, union, intersection, difference, symmetric difference.
- Dictionaries: unique keys mapped to values.
- Use get() when a missing key should have a default value.
- Use sorted() when you need ordered output from a set or dictionary keys.
- Dictionaries are especially useful for counting and frequency analysis.
- Collections store references to objects, so mutation and aliasing matter.
- Set and dictionary comprehensions provide concise construction syntax.