Fundamentos de Programação 2025-2026

Programming Fundamentals

Class #8 – Sets and Comprehensions

Problem

- **Who's Coming to the Party?**

- Scenario: You're organizing a party and invited two groups of friends:
  - Group A: your school friends
  - Group B: your soccer team

- Some people are in both groups. You want to:
  - Find out **who's coming** (all unique invitees).
  - See **who's in both groups** (overlap).
  - Identify **who's only in one group**.
  - Remove people who **can't come** from the list.

Possible solution (pseudo code)

```
# 1. All unique invitees
all_invitees = school_friends.union(soccer_team)

# 2. Friends in both groups
common_friends =
school_friends.intersection(soccer_team)

# 3. Friends only in one group
exclusive_friends =
school_friends.symmetric_difference(soccer_team)

# 4. Final guest list after removing those who can't come
final_guests = all_invitees.difference(cant_come)
```

What do we need?

- Sets
- Operations on sets
- Why Sets:
  - no duplicates,
  - fast operations,
  - and clear logic

Sets *(in Math)*

- A set is a well-defined collection of distinct objects.
- The objects in a set are called elements or members.

- Sets are usually written using curly braces:

  A = {1, 2, 3}
- The symbol ∈ means “is an element of”:

  2 ∈ A, but 5 ∉ A

- Examples:
  - Set of vowels: V = {a, e, i, o, u}.
  - Set of even numbers less than ten: E = {2, 4, 6, 8}.
  - Set of primary colors: P = {red, green, blue}.

Sets

- Sets are a fundamental data type in Math and computing.

- Key Properties:
  - No repeated elements
  - Order of elements doesn't matter
  - Can be finite or infinite

Methods of Defining Sets

There are two primary methods to define a set:

- **Enumeration Method**
  - Involves listing all the elements of the set explicitly, separated by commas and enclosed in curly braces.
  - **Example:**
    $A = \{2, 4, 6, 8\}$ — a set of even numbers less than 10.

- **Comprehension Method**
  - This method describes the properties or rules that determine the elements of the set
    - often using a variable and a condition.
  - **Example:**
    $B = \{ x \mid x \text{ is even and } x < 10\}$ — the same set as above, defined by a rule.

Common Set Operations

Operation	Symbol	Description
Union	$A \cup B$	All elements in A or B (or both)
Intersection	$A \cap B$	Elements common to both A and B
Difference	$A - B$	Elements in A but not in B
Complement	A'	Elements not in A (relative to a universal set)
Subset	$A \subseteq B$	All elements of A are also in B
Cardinality	$\lvert A \rvert$	Number of elements in set A

SETS IN PYTHON

Sets

A Set is an

Collection

- because it may contain zero or more items.

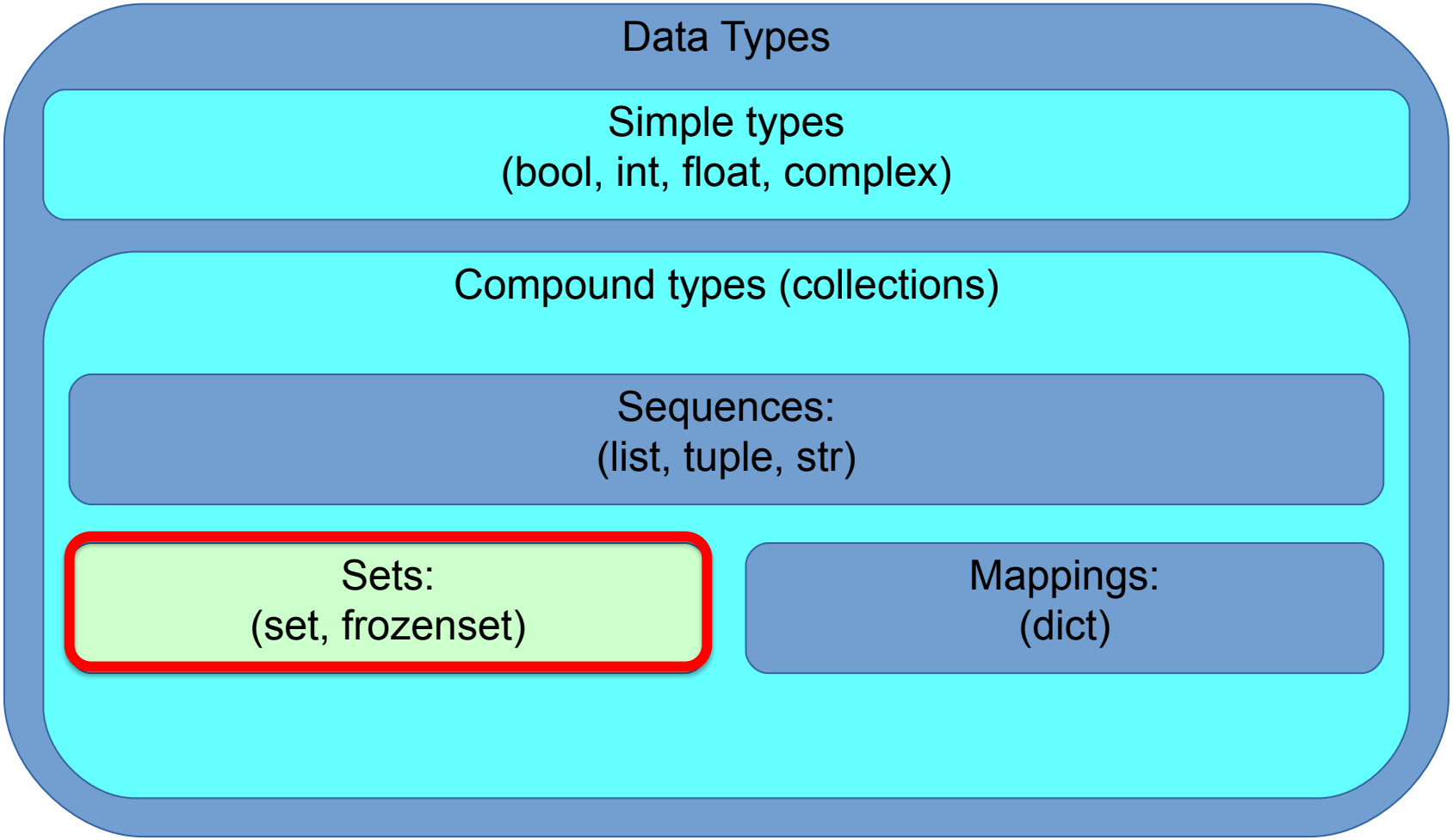
of Unordered

- elements are not in sequence.

Unique items.

- because elements cannot be repeated!.

Sets and Python data types



Official documentation on sets.

Creating Sets

- In Python, a set may be created using braces.

```
fruits = {'pear', 'apple', 'banana', 'orange'}
S = { x for x in fruits if x<'c' }  # (by comprehension)
```

- The set constructor converts from other types.

```
numbers = set([3, 1, 3])    #-> {1, 3}
```

- The empty set must be created with set(), because {} creates a dictionary.

```
empty = set()
```

Play ▸

Elements in a set are unique

- An object either is or is not in a set. It cannot be in the set more than once!

```
{1, 2, 1} == {1, 2}      #-> True
len({4, 5, 4, 5, 5})     #-> 2
```

Play ▸

  - Like keys in a dictionary.

- Unlike sequences.

```
[1, 2, 1] == [1, 2]      #-> False
```

- A common application of sets is for eliminating duplicate elements in sequences.

```
set([1, 2, 2, 2, 1])     #-> {1, 2}
set('banana')            #-> {'a', 'b', 'n'}
```

  - This eliminates order, too.

Elements in a set are unordered

- Sets don't recall the position or order of entry of elements.

Play ▸

```
s = {3, 1, 2}
print(s)  # {1, 2, 3}, {2, 3, 1} or ...
s == {2, 3, 1} == {1, 2, 3}  #-> True
```

- So, indexing, slicing, and concatenation are not allowed!

```
s[0]          # TypeError
s[0:2]        # TypeError
s + {4}       # TypeError
```

Elements in a set must be hashable

- A set may contain elements of various types, but only hashable types are allowed.
  - Just like dictionary **keys**.
- Simple immutable types (like numbers) are OK.
- Strings are OK.
- Tuples are OK, if their elements are hashable.

```
{ 23, 'eggs', (1997,10,23) }
```

- Lists, dictionaries, sets and other mutable types are not allowed!

```
{ [1,2] }       # TypeError
{ {1}, {1,2} }    # TypeError
```

Sets are mutable

- We can add or remove elements in sets.

```
S = {1,2,3}
S.add(4)          # S -> {1,2,3,4}
S.remove(2)       # S -> {1,3,4}
S.discard(7)      # No error!
```

Play ▸

- We can update the set by union, intersection or differences.

```
S |= {3,5,7}     # S.update({3,5,7})
S &= {1,2,3,4}   # S.intersection_update({1,2,3,4})
S -= {4,5,6}     # S.difference_update({4,5,6})
S ^= {1,2,4}     # S.symmetric_difference_update({1,2,4})
```

- Python also has immutable sets: the `frozenset` type.

```
T = frozenset({1,2,3})
```

Operations on sets

- Sets have a length and support the membership operator.

```
S = {23, 5, 12}
len(S)     #-> 3
5 in S     #-> True (Fast operation!)
```

- Sets support union, intersection, and differences.

```
{3,4,5} | {1,2,3} #-> {1,2,3,4,5} (set.union)
{3,4,5} & {1,2,3} #-> {3}         (set.intersection)
{3,4,5} - {1,2,3} #-> {4,5}       (set.difference)
{3,4,5} ^ {1,2,3} #-> {1,2,4,5}
  (set.symmetric_difference)
```

Play ▸

Operations on sets (2)

- Sets may be compared for equality.

```
S = {1,2}
{2,2,1} == S          # True
```

Play ▸

- We may test subset or superset relations.

```
S <= {1,2}        # True = S.issubset({1,2})
S < {1,2,3}       # True
S >= {1,2}        # True = S.issuperset({1,2})
S > {2}           # True
```

- But this is not a total ordering relation! You can have two sets A and B such that:

```
A < B,    A == B,    A > B     # All are False!
```

Sets vs lists

Feature	List	Set
Order	Ordered elements	Unordered elements
Duplicates	Allows duplicates	Removes duplicates automatically
Mutability	Can be changed	Can be changed
Indexing	Access by position (`lst[0]`)	No indexing
Use Case	Sequences where order matters	Unique items, fast membership tests
Syntax	`[1, 2, 2, 3]`	`{1, 2, 2, 3}` → `{1, 2, 3}`
Common Methods	`.append()`, `.remove()`, `.sort()`, `.reverse()`	`.add()`, `.remove()`, `.union()`, `.intersection()`

Sets vs dictionaries

Feature	🔢 Set	🗂️ Dictionary
Structure	Collection of unique values	Collection of key-value pairs
Order	Unordered	*Unordered* (insertion order preserved, since v. 3.7)
Duplicates	No duplicates allowed	Keys must be unique; values can repeat
Mutability	Mutable	Mutable
Indexing	No indexing	Access by key (`dict['key']`)
Use Case	Fast membership tests, uniqueness	Mapping relationships, structured data
Syntax	`{1, 2, 3}`	`{'a': 1, 'b': 2}`
Common Methods	`.add()`, `.remove()`, `.union()`, `.intersection()`	`.get()`, `.keys()`, `.values()`, `.items()`

SELECTING THE PROPER DATA TYPE

How to select the proper data type?

- Choosing the right type to store your data is very important.
- First, consider the different characteristics of the types.

Type	Type identifier	Collection?	Sequence?	Mutable?	Element type
Simple types	`bool, int, float, complex`	No (scalar)	---	No	---
String	`str`	Yes	Yes	No	Character
Tuple	`tuple`	Yes	Yes	No	Any type
List	`list`	Yes	Yes	Yes	Any type
Dictionary	`dict`	Yes	No (unorderable)	Yes	Key: Hashable Value: Any type
Set	`set`	Yes	No (unorderable)	Yes	Hashable
Immutable set	`frozenset`	Yes	No (unorderable)	No	Hashable

Some questions to help deciding

- Are the data simple (scalar) or compound (several elements)?
  - Compound => collection.
- Does element order/position matter?
  - Yes => sequence.
- Will the contents grow, shrink or change?
  - Yes => mutable.
- Need to quickly map a key to a value?
  - Yes => dictionary.

From Collections to Comprehensions

- What We've Learned So Far:
  - **Lists**: Ordered, mutable sequences
  - **Dictionaries**: Key-value mappings
  - **Sets**: Unordered, unique elements

- What's Next: Comprehensions
  - A **concise syntax** for creating new lists, dicts, and sets
  - Combines **iteration + condition + construction** in one line
  - Makes code **cleaner, faster, and more readable**

CONCISE SYNTAX FOR CREATING NEW LISTS, DICTS, AND SETS

Building lists

- Quite often, we need to **create collections** with elements **related to** those in **another collection**.
- For example: create a list of the squares of the values in lst.

```
lst = [1, -3, 2]
lst2 = []              # init result with empty list
for v in lst:          # loop over original list:
    v2 = v**2          # compute a new value
    lst2.append(v2) # append it to result
print( lst2 )          #-> [1, 9, 4]
```

Play ▸

- Another example: create a list of uppercase versions of the strings in lst.
  - What do you need to change in the code above?
- These programs always follow the same basic pattern.

The basic pattern

```
lst = [1, -3, 2]

lst2 = []          # 1. Initialize result
for v in lst:      # 2. Iterate
    v2 = v**2      # 3. Transform each item
    lst2.append(v2)
                   # 4. Collect into result

print(lst2)                  # -> [1, 9, 4]
```

List comprehensions

- Python provides a more concise way to produce such lists.

```
nums= [4, -5, 3, 7, 2, 3, 1]
nums2 = [ v**2 for v in nums ]
          #-> [16, 25, 9, 49, 4, 9, 1]
args = ['apple', 'dell', 'ibm', 'hp', 'sun']
args2 = [ s.upper() for s in args ]
          #-> ['APPLE', 'DELL', 'IBM', 'HP', 'SUN']
```

Play ▸

- These are **list comprehensions**:
  - **expressions** that generate lists by operating on the elements of other collections.

- Here the **`for..in`** clause is part of the expression.
  - Do not confuse it with the for statement.

Loops vs comprehensions

Classic Loop Pattern	List Comprehension
lst = [1, -3, 2]	lst = [1, -3, 2]
lst2 = []	lst2 = [v**2 for v in lst]
for v in lst:	
v2 = v**2	
lst2.append(v2)	
print(lst2)	print(lst2)
# → [1, 9, 4]	# → [1, 9, 4]

List comprehensions (2)

- List comprehensions may also include:
  - if clauses.

```
args = ['apple', 'dell', 'ibm', 'hp', 'sun’]

args3 = [ str.upper() for str in args if len(str)>3 ]
      #-> ['APPLE', 'DELL’]
```

  - multiple `for..in` and `if` clauses.

```
[(a,b) for a in [1,2] for b in nums if b>3*a]
     #-> [(1, 4), (1, 7), (2, 7)]
```

Play ▸

General structure of a comprehension

```
[expression for item in iterable
            if condition                    (optional)
            for item2 in iterable           (optional)
            ...                             (zero or more of above)
]
```

Comprehension for our example:

```
lst2 = [v**2 for v in lst]
```

Dictionary and set comprehensions

- Use of comprehensions is not limited to lists.
- We may also create dictionaries by comprehension.

```
args = ['apple', 'dell', 'ibm', 'hp', 'sun']
d = { a: len(a) for a in args }
#-> {'apple': 5, 'ibm': 3, 'hp': 2, ...}
```

Play ▸

- Sets may also be defined by comprehension.

```
s = { 2+x for x in [3, 4, 5, 4] }
```

Play ▸

- Other variations are possible too, of course.

Generator expressions

- **Generator expressions** are identical to the expressions used in list comprehensions
  - but enclosed in ( )
- They create an object that generates items only if and when needed
  - unlike list comprehensions.
- This strategy is called lazy evaluation and can save memory and time.
- They're convenient as arguments to some functions.

```
nums = [4, -5, 3, 7, 2, 3, 1]
sum( x/2 for x in nums if x%2==0 )  #-> 3.0
all( x>0 for x in nums )            #-> False
```

- We may use generator expressions to create other types of sequences.

```
tuple( v for v in nums if v<3 )  #-> (-5, 2, 1)
```

SOME COMMON MISTAKES BEGINNERS MAKE WITH SETS IN PYTHON

And How to Identify & Fix Them

Misunderstanding Set Syntax

- Mistake: Using {} thinking it's an empty set

```
empty_set = {} # Creates a dict!
```

- Correct:

```
empty_set = set()
```

Forgetting Set Uniqueness

- Mistake: Expecting duplicates in a set

```
s = {1, 2, 2, 3}
print(s)  # Output: {1, 2, 3}
```

- Tip: Sets automatically remove duplicates.

Misusing Set Operation

- Mistake: Using + to combine sets

```
a = {1, 2}
b = {3, 4}
c = a + b  # TypeError!
```

- Correct:

```
c = a.union(b)  # or a | b
```

Confusing Set Methods

- Mistake: Using the wrong method or symbol

Operation	Method	Symbol
Union	a.union(b)	a | b
Intersection	a.intersection(b)	a & b
Difference	a.difference(b)	a - b
Symmetric Difference	a.symmetric_difference(b)	a ^ b

Forgetting elements must be hashable

- Mistake: Using mutable elements in a set

```
s = {[1, 2], [3, 4]}
# TypeError: unhashable type: 'list'
```

- Tip: Only immutable (hashable) types like tuples can be set elements.

RETURNING TO OUR INITIAL PROBLEM …

Possible Solution in Python

```
# Define the two groups of friends
school_friends = {"Alice", "Bob", "Charlie", "Diana"}
soccer_team = {"Charlie", "Diana", "Ethan", "Fiona"}

# Define the list of people who can't come
cant_come = {"Bob", "Fiona"}

# 1. All unique invitees
all_invitees = school_friends.union(soccer_team)
print("All invitees:", all_invitees)

# 2. Friends in both groups
common_friends = school_friends.intersection(soccer_team)
print("Friends in both groups:", common_friends)

# 3. Friends only in one group
exclusive_friends = school_friends.symmetric_difference(soccer_team)
print("Friends only in one group:", exclusive_friends)

# 4. Final guest list after removing those who can't come
final_guests = all_invitees.difference(cant_come)
print("Final guest list:", final_guests)
```

```
All invitees: {'Fiona', 'Diana', 'Bob', 'Alice', 'Charlie', 'Ethan'}
Friends in both groups: {'Charlie', 'Diana'}
Friends only in one group: {'Fiona', 'Bob', 'Alice', 'Ethan'}
Final guest list: {'Alice', 'Charlie', 'Diana', 'Ethan'}
```