COMP101 — Week 4
Functions & Containers

functions, lists, sets, dictionaries

UM6P — SASE

November 12, 2025

Legend for Terms in Code

- **Must know**: use today, required moving forward.
- **Need to learn**: useful soon or slightly advanced variants.
- **Optional**: enrichment, stretch, or alternative approaches.

Today: Four simple ideas you'll use everywhere

- **Functions** — a *named recipe*. Write the steps once, give it a name, and *call* it by writing its name with parentheses. Example: `clean(text)`, `distance(a,b)`. It can hand back a result (`return`).

Today: Four simple ideas you'll use everywhere

- **Functions** — a *named recipe*. Write the steps once, give it a name, and *call* it by writing its name with parentheses. Example: `clean(text)`, `distance(a,b)`. It can hand back a result (`return`).
- **Lists** — a *line* of things: first → next → next. Use it when order matters and you want to loop through items.

Today: Four simple ideas you'll use everywhere

- **Functions** — a *named recipe*. Write the steps once, give it a name, and *call* it by writing its name with parentheses. Example: `clean(text)`, `distance(a,b)`. It can hand back a result (`return`).
- **Lists** — a *line* of things: first → next → next. Use it when order matters and you want to loop through items.
- **Sets** — a *no-duplicates bag*. If it's already inside, it won't go in twice. Great for fast "have we seen this?" checks.

Today: Four simple ideas you'll use everywhere

- **Functions** — a *named recipe*. Write the steps once, give it a name, and *call* it by writing its name with parentheses. Example: `clean(text)`, `distance(a,b)`. It can hand back a result (`return`).
- **Lists** — a *line* of things: first → next → next. Use it when order matters and you want to loop through items.
- **Sets** — a *no-duplicates bag*. If it's already inside, it won't go in twice. Great for fast "have we seen this?" checks.
- **Dictionaries (`dict`)** — *labeled boxes*. Look up by name, get the thing. Like contacts: `name` → phone.

Today: Four simple ideas you'll use everywhere

- By the end, you should be able to:
  Define a named recipe with `def` and call it like `name(...)`
  Walk through many items in order (list)
  Keep only uniques and test membership (set)
  Store and fetch values by name (dict)
  later: docstrings, sorting, set/dict comprehensions

Algorithm

Challenge: Given a, b, c (with $a \neq 0$), decide quickly: Are there **no**, **one**, or **two** real roots?

- There's a well-known *recipe*: compute the discriminant $D = b^2 - 4ac$.

Algorithm

Challenge: Given a, b, c (with $a \neq 0$), decide quickly: Are there **no**, **one**, or **two** real roots?

- There's a well-known *recipe*: compute the discriminant $D = b^2 - 4ac$.
- $D < 0 \Rightarrow$ no real roots; $D = 0 \Rightarrow$ one real root; $D > 0 \Rightarrow$ two real roots.

Algorithm

Challenge: Given a, b, c (with $a \neq 0$), decide quickly: Are there **no**, **one**, or **two** real roots?

- There's a well-known *recipe*: compute the discriminant $D = b^2 - 4ac$.
- $D < 0 \Rightarrow$ no real roots; $D = 0 \Rightarrow$ one real root; $D > 0 \Rightarrow$ two real roots.
- We'll first write it the long way, then turn it into a **function** (a named recipe).

Motivation #1 — Quadratic via discriminant

Algorithm

Steps:

1. Read a, b, c.
2. Compute $D = b^2 - 4ac$.
3. If $D < 0$: `no real roots`. If $D = 0$: One real root.
4. If $D > 0$: two real roots.

Motivation #1 — First pass (no function yet)

```
a = float(input("a? "))
b = float(input("b? "))
c = float(input("c? "))
D = b*b - 4*a*c
if D < 0:
    print("no real roots")
elif D == 0:
    print("one real root")
else:
    print("two real roots")
```

Clear but not reusable. Every new quadratic = copy/paste the same steps.

Motivation #2 — Turn the recipe into a function

```
def discriminant(a, b, c):
    return b*b - 4*a*c

def solve_quadratic(a, b, c):
    D = discriminant(a, b, c)
    if D < 0:
        return "no real roots"
    elif D == 0:
        return "one real root"
    else:
        return "two real roots"

kind = solve_quadratic(1, -3, 2)   # x^2 - 3x + 2
print(kind)
```

*Now we **call** it by name with parentheses. No copy/paste. Easy to test.*

Why functions? The payoff

- **Reuse**: same named recipe, many inputs. Faster labs, fewer mistakes.

Why functions? The payoff

- **Reuse**: same named recipe, many inputs. Faster labs, fewer mistakes.
- **Clarity**: `solve_quadratic(a,b,c)` reads like English.

Why functions? The payoff

- **Reuse**: same named recipe, many inputs. Faster labs, fewer mistakes.
- **Clarity**: `solve_quadratic(a,b,c)` reads like English.
- **Testing**: try known cases quickly: $(1, -3, 2)$, $(1, 2, 1)$, $(1, 0, 1)$.

Algorithm

A **function** is a named block of code. It runs *only* when you **call** it. It can take inputs (parameters) and can **return** a result. Main win: avoid repetition and make code easier to read.

- Define once with `def`; call many times with `name(...)`

Algorithm

A **function** is a named block of code. It runs *only* when you **call** it. It can take inputs (parameters) and can **return** a result. Main win: avoid repetition and make code easier to read.

- Define once with `def`; call many times with `name(...)`
- Returns a value with `return`; no `return` means it returns *None*

Algorithm

A **function** is a named block of code. It runs *only* when you **call** it. It can take inputs (parameters) and can **return** a result. Main win: avoid repetition and make code easier to read.

- Define once with `def`; call many times with `name(...)`
- Returns a value with `return`; no `return` means it returns *None*
- Body must be **indented** — indentation defines the function's block

Defining and calling a function

```
def greet():
    print("Hello from a function")   # body is indented

greet()   # call: name followed by parentheses
greet()
```

Definition uses `def` *and a colon. Calls use the name with parentheses.*

Return values

```
def get_greeting():
    return "Hello from a function"   # send a value back

message = get_greeting()             # store the result
print(message)
print(get_greeting())                # or use it directly
```

- Hitting **return** ends the function immediately

Return values

```
def get_greeting():
    return "Hello from a function"   # send a value back

message = get_greeting()             # store the result
print(message)
print(get_greeting())                # or use it directly
```

- Hitting **return** ends the function immediately
- If you omit return, Python returns *None* by default

Motivation — write once, reuse (DRY)

Without functions (repetition)

```
temp1 = 77
c1 = (temp1 - 32) * 5 / 9
print(c1)

temp2 = 95
c2 = (temp2 - 32) * 5 / 9
print(c2)

temp3 = 50
c3 = (temp3 - 32) * 5 / 9
print(c3)
```

With a function (reuse)

```
def fahrenheit_to_celsius(f):
    return (f - 32) * 5 / 9

print(fahrenheit_to_celsius(77))
print(fahrenheit_to_celsius(95))
print(fahrenheit_to_celsius(50))
```

Same logic, cleaner code, easier to test and change.

Function names — quick rules

- Start with a letter or _ ; only letters, digits, _

Function names — quick rules

- Start with a letter or _ ; only letters, digits, _
- Case-sensitive: `myFunction` $\neq$ `myfunction`

Function names — quick rules

- Start with a letter or `_` ; only letters, digits, `_`
- Case-sensitive: `myFunction` $\neq$ `myfunction`
- Prefer clear names: `calculate_sum()`, `clean_text()`, `to_celsius()`

pass as a placeholder

Algorithm

You can't have an empty function body. Use **pass** as a placeholder while sketching your code.

```
def todo():
    pass   # placeholder so the file runs

def greet(name):
    # TODO: implement later
    pass

print(todo())   # returns None (no explicit return)
```

- Lets you define structure first, fill details later

pass as a placeholder

Algorithm

You can't have an empty function body. Use **pass** as a placeholder while sketching your code.

```
def todo():
    pass   # placeholder so the file runs

def greet(name):
    # TODO: implement later
    pass

print(todo())   # returns None (no explicit return)
```

- Lets you define structure first, fill details later
- Keeps your file runnable while you build step by step

Putting it together — tiny toolkit

```
def greet(who):
    print("Hello,", who)

def fahrenheit_to_celsius(f):
    return (f - 32) * 5 / 9

def placeholder():
    pass  # to be implemented

greet("Alice")
print(fahrenheit_to_celsius(77))
```

You now know: **def**, *calls*, **return**, *indentation, and* **pass**.

Algorithm

Parameter = the name in the `def` line.
Argument = the actual value you pass when calling.
Rule of thumb: **Match the number and the order** of parameters when you call a function.

- Define once: `def greet(name): ...` (*name* is a parameter)

Function inputs: parameters vs arguments

Algorithm

Parameter = the name in the `def` line.
Argument = the actual value you pass when calling.
Rule of thumb: **Match the number and the order** of parameters when you call a function.

- Define once: `def greet(name): ...` (*name* is a parameter)
- Call with values: `greet("Emil")` (*"Emil"* is an argument)

Algorithm

Parameter = the name in the `def` line.
Argument = the actual value you pass when calling.
Rule of thumb: **Match the number and the order** of parameters when you call a function.

- Define once: `def greet(name): ...` (*name* is a parameter)
- Call with values: `greet("Emil")` (*"Emil"* is an argument)
- If you pass too few/many, Python raises an error; if you swap order, you change the meaning

Function inputs: examples

```
def full_name(first, last):
    print(first, last)

full_name("Emil", "Refsnes")   # OK: 2 args, correct order
full_name("Refsnes", "Emil")   # Still OK, but order changed the output

full_name("Emil")              # TypeError: missing 1 required
    positional argument
full_name("Emil", "Refsnes", "Extra")  # TypeError: too many positional
    arguments
```

*For now: respect **number** and **position**.*

Scope — one simple rule

Algorithm

Variables made inside a function stay inside that function. If you need a value outside, **return** it and capture it.

- Don't expect to use a function's temporary names from the outside

Scope — one simple rule

Algorithm

Variables made inside a function stay inside that function. If you need a value outside, **return** it and capture it.

- Don't expect to use a function's temporary names from the outside
- Prefer: compute inside → `return` result → store it outside

Scope — what you can and can't see

```
def area(w, h):
    a = w * h           # 'a' is created inside the function
    return a            # send the value back

print(area(3, 4))       # 12
print(a)                # NameError: 'a' is not defined here
```

Fix: store the returned value outside if you need it later.

Scope — the right pattern

```
def area(w, h):
    a = w * h
    return a

result = area(3, 4)     # capture the returned value
print(result)           # 12
```

*Make values inside; **return** them; use them outside. Clean, predictable, testable.*

Algorithm

You already know `split()`. It turns a string into a **list** of pieces.

From text to lists in one move

```
line = "alice bob  charlie"
names = line.split()          # ["alice", "bob", "charlie"]

csv  = "10,20,30"
nums = csv.split(",")         # ["10", "20", "30"]
```

- A **list** is an ordered, changeable sequence (duplicates allowed).

From text to lists in one move

```
line = "alice bob  charlie"
names = line.split()          # ["alice", "bob", "charlie"]

csv  = "10,20,30"
nums = csv.split(",")         # ["10", "20", "30"]
```

- A **list** is an ordered, changeable sequence (duplicates allowed).
- Mental model: a numbered shelf with slots **0,1,2,...**.

From text to lists in one move

```
line = "alice bob  charlie"
names = line.split()          # ["alice", "bob", "charlie"]

csv  = "10,20,30"
nums = csv.split(",")         # ["10", "20", "30"]
```

- A **list** is an ordered, changeable sequence (duplicates allowed).
- Mental model: a numbered shelf with slots **0,1,2,...**.
- We'll learn to **access**, **loop**, and **modify** these shelves.

Lists — create, inspect, access

```
fruits = ["apple", "banana", "cherry"]  # literal
print(len(fruits))                      # 3
print("banana" in fruits)               # True

print(fruits[0])    # "apple"  (first)
print(fruits[-1])   # "cherry" (last)
```

- Create with `[]` (or `list(...)`).

Lists — create, inspect, access

```
fruits = ["apple", "banana", "cherry"]  # literal
print(len(fruits))                      # 3
print("banana" in fruits)               # True

print(fruits[0])    # "apple"  (first)
print(fruits[-1])   # "cherry" (last)
```

- Create with **`[]`** (or **`list(...)`**).
- Size with **`len(lst)`**; membership with **`x in lst`**.

Lists — create, inspect, access

```
fruits = ["apple", "banana", "cherry"]  # literal
print(len(fruits))                      # 3
print("banana" in fruits)               # True

print(fruits[0])    # "apple"  (first)
print(fruits[-1])   # "cherry" (last)
```

- Create with **`[]`** (or **`list(...)`**).
- Size with **`len(lst)`**; membership with **`x in lst`**.
- Indexing is 0-based; **negative indices** count from the end.

Looping over a list

When you only need values

```
for name in names:
    print(name)
```

When you need positions

```
for i in range(len(names)):
    print(i, names[i])
```

- Prefer the value-only loop; switch to indices when necessary.

```
print(", ".join(names))   # "alice, bob, charlie"
```

Looping over a list

When you only need values

```
for name in names:
    print(name)
```

When you need positions

```
for i in range(len(names)):
    print(i, names[i])
```

- Prefer the value-only loop; switch to indices when necessary.
- **Tip**: for friendly output, join strings:

```
print(", ".join(names))   # "alice, bob, charlie"
```

Changing a list (add, insert, remove)

```
fruits = ["apple", "banana", "cherry"]
fruits.append("orange")          # add at end
fruits.insert(1, "kiwi")         # put at index 1
fruits.remove("banana")          # remove by value (first match)
last = fruits.pop()              # remove & return last item
print(fruits)

more = ["mango", "papaya"]
fruits.extend(more)              # add another list's items
```

Note: mid-list insert/remove shifts later items; end-append is simplest.

Quick slicing

```
a = ["a","b","c","d","e","f"]
print(a[2:5])    # ["c","d","e"]   start incl., end excl.
print(a[:3])     # ["a","b","c"]   from start
print(a[3:])     # ["d","e","f"]   to end
```

- Use slicing to grab a contiguous chunk.

Quick slicing

```
a = ["a","b","c","d","e","f"]
print(a[2:5])    # ["c","d","e"]   start incl., end excl.
print(a[:3])     # ["a","b","c"]   from start
print(a[3:])     # ["d","e","f"]   to end
```

- Use slicing to grab a contiguous chunk.
- Advanced step sizes (like `a[::2]`) later.

Alias vs copy

```
a = [1,2,3]
b = a             # alias: both names point to same list
b.append(4)
print(a)          # [1,2,3,4]  (surprise!)
```

Make a (shallow) copy

```
a = [1,2,3]
b = a[:]          # or a.copy()
b.append(4)
print(a)          # [1,2,3]
```

Three tiny patterns (from split() to results)

1) Read tokens from one line

```
names = input("names? ").split()   # space-separated
print(len(names), "names read")
```

2) Convert numbers then process

```
nums = input("nums? ").split(",")  # csv
vals = []
for s in nums:
    vals.append(int(s))
print("sum:", sum(vals))            # ok to use built-in sum
```

Three tiny patterns (from split() to results)

3) Filter with a loop

```
short = []
for w in names:
    if len(w) <= 4:
        short.append(w)
print("short names:", " ".join(short))
```

Algorithm

A **list comprehension** builds a new list from an iterable in one compact line:

```
[ expression   for item   in iterable   if condition ]
```

- **Clear intent**: "take items, maybe filter them, transform them."

List comprehensions — why bother?

Algorithm

A **list comprehension** builds a new list from an iterable in one compact line:

```
[ expression   for item   in iterable   if condition ]
```

- **Clear intent**: "take items, maybe filter them, transform them."
- **Fewer lines**: less boilerplate than loops + append.

List comprehensions — why bother?

Algorithm

A **list comprehension** builds a new list from an iterable in one compact line:

```
[ expression   for item   in iterable   if condition ]
```

- **Clear intent**: "take items, maybe filter them, transform them."
- **Fewer lines**: less boilerplate than loops + append.
- **Readable when short**; if it gets long, fall back to a loop.

Comprehensions — basic patterns

From words to upper-case:

```
words = "alice bob charlie".split()
upper = [w.upper() for w in words]    # ['ALICE','BOB','CHARLIE']
```

Filter while building:

```
nums = [10, -3, 7, 0, 5]
positives = [x for x in nums if x > 0]   # [10, 7, 5]
```

Map (transform) while building:

```
lengths = [len(w) for w in words]        # [5, 3, 7]
```

*Think: **filter** with `if`, **transform** with the leading expression.*

From split() to processed numbers

Two simple steps (clear and fast):

```
tokens = input("nums? ").split()          # e.g. "10 -2 7 3"
vals   = [int(s) for s in tokens]         # [10, -2, 7, 3]
evens  = [x for x in vals if x % 2 == 0] # [10, -2]
squares_of_pos = [x*x for x in vals if x > 0]  # [100, 49, 9]
```

- **Parse** then **filter/transform**. Keep steps small and readable.

Conditional transformation (inline if/else)

Choose the output per item:

```
vals = [5, 6, 7, 8]
signed = [x if x % 2 == 0 else -x for x in vals]  # [-5, 6, -7, 8]

words = "a bb ccc dddd".split()
label = ["LONG" if len(w) >= 3 else "short" for w in words]
# ['short', 'short', 'LONG', 'LONG']
```

Structure: `[ A if cond else B for x in items ]`.

Sets: fast *seen?* checks and no duplicates

Algorithm

You've got many items (names, IDs, words) and you only care about two things: **Is it there?** and **keep each item once**. That's a **set**.

- **Deduplicate** instantly (no repeats).

Sets: fast *seen?* checks and no duplicates

Algorithm

You've got many items (names, IDs, words) and you only care about two things: **Is it there?** and **keep each item once**. That's a **set**.

- **Deduplicate** instantly (no repeats).
- **Membership test** is *fast* **(hash table)** — much faster than scanning a list.

Sets: fast *seen?* checks and no duplicates

Algorithm

You've got many items (names, IDs, words) and you only care about two things: **Is it there?** and **keep each item once**. That's a **set**.

- **Deduplicate** instantly (no repeats).
- **Membership test** is *fast* **(hash table)** — much faster than scanning a list.
- Use sets when order doesn't matter and you won't index by position.

Set basics — literal, membership, add/remove

```
students = {"ali", "salma", "amira"}      # curly braces literal
print("salma" in students)                # True (fast membership)

students.add("youssef")                   # add one
students.discard("amira")                 # remove if present (no error
    if absent)

more = ["salma", "fatima", "omar"]
students.update(more)                     # add many from any iterable
print(students)                           # duplicates ignored
```

Note: **{} is an empty dict**, not a set. Empty set is **set()**.

From split() to a clean unique roster

```
line = "ali salma salma amira ALI"
raw  = line.split()                  #
    ['ali','salma','salma','amira','ALI']
names = {w.lower() for w in raw}     # normalize + dedupe with a set
print(names)                         # {'ali','salma','amira'}
```

- **Sets drop duplicates** by design.

From split() to a clean unique roster

```
line = "ali salma salma amira ALI"
raw  = line.split()                  #
    ['ali','salma','salma','amira','ALI']
names = {w.lower() for w in raw}     # normalize + dedupe with a set
print(names)                         # {'ali','salma','amira'}
```

- **Sets drop duplicates** by design.
- **Normalize first** (e.g., `lower()`, strip punctuation) then convert to a set.

Mental model — what sets *are* and *aren't*

- **Unordered**: no fixed positions; **no** `s[0]`.

Mental model — what sets *are* and *aren't*

- **Unordered**: no fixed positions; **no** `s[0]`.
- **Unique**: trying to add the same item again does nothing.

Mental model — what sets *are* and *aren't*

- **Unordered**: no fixed positions; **no** `s[0]`.
- **Unique**: trying to add the same item again does nothing.
- **Mutable container**: you can add/remove items; the items themselves must be hashable (e.g., strings, numbers, tuples of immutables).

Mental model — what sets *are* and *aren't*

- **Unordered**: no fixed positions; **no** `s[0]`.
- **Unique**: trying to add the same item again does nothing.
- **Mutable container**: you can add/remove items; the items themselves must be hashable (e.g., strings, numbers, tuples of immutables).
- **Edge case**: `True` and `1` are considered equal; `False` and `0` too.

Why sets feel fast for `in`

Algorithm

List membership → checks each item ($O(n)$). Set membership → hash table lookup ($\approx O(1)$ average).

```
# Precompute a set for repeated queries:
words = input().split()
vocab = set(words)        # build once

q = input("query? ")
print(q in vocab)         # fast check, use many times
```

Pattern: when you'll ask "seen?" many times, **convert to a set first**.

Set operations — think logic: OR, AND, MINUS, XOR

Union (OR) / Intersection (AND)

```
a = {"ali","salma","omar"}
b = {"salma","fatima"}

all_students = a | b          # union
both = a & b                  #
    intersection
```

Difference (MINUS) / Symmetric diff (XOR)

```
only_in_a = a - b             # in a,
    not in b
in_exactly_one = a ^ b        # in a or
    b, not both
```

These mirror math set laws and are extremely handy for filters.

Three quick patterns you'll use a lot

1) Deduplicate while reading

```
unique_tags = set(input().split())
```

2) Fast filter by membership

```
stop = {"a", "an", "the"}
tokens = input().split()
clean = [w for w in tokens if w not in stop]   # set used for fast checks
```

3) Compare two groups

```
group_A = set(input().split())
group_B = set(input().split())
common  = group_A & group_B
only_A  = group_A - group_B
```

Set comprehension — build + dedupe in one go

```
# Unique first letters (lowercased)
words = "Alice bob anna Omar".split()
initials = {w[0].lower() for w in words}      # {'a','b','o'}

# Filter + normalize + dedupe: keep words with 'a'
a_words = {w.lower() for w in words if 'a' in w.lower()}
```

Structure: `{ expression for item in iterable if condition }`.

Common pitfalls (10-second warnings)

- **No indexing, no ordering**: don't rely on printed order.

Common pitfalls (10-second warnings)

- **No indexing, no ordering**: don't rely on printed order.
- **Empty set**: use `set()`, not `{}`.

Common pitfalls (10-second warnings)

- **No indexing, no ordering**: don't rely on printed order.
- **Empty set**: use `set()`, not `{}`.
- **Numbers vs booleans**: `True == 1`, `False == 0` — they collide in a set.

Common pitfalls (10-second warnings)

- **No indexing, no ordering**: don't rely on printed order.
- **Empty set**: use `set()`, not `{}`.
- **Numbers vs booleans**: `True == 1`, `False == 0` — they collide in a set.
- **Unhashable items**: lists/dicts can't live inside a set; use tuples or frozenset if needed.

Practice — three short tasks

```
# 1) Read a line of names; print how many *unique* names there are.
names = input().split()
print(len(set(names)))

# 2) Stop-words: remove "a an the of" from a second input line.
stop = {"a","an","the","of"}
tokens = input().split()
clean = [w for w in tokens if w.lower() not in stop]
print(" ".join(clean))

# 3) Given two lines of course IDs, print common IDs (any order).
A = set(input().split()); B = set(input().split())
print(A & B)
```

If order later matters, convert the set back to a list and sort it.

Dictionaries: name → value, fast lookup

Algorithm

You know lists (positions). Now you need **labels**. A **dictionary** maps a **key** (name) to a **value** — like contacts: `name` → phone.

- **Fast by key** (hash table, $\approx O(1)$ average).

Dictionaries: name → value, fast lookup

Algorithm

You know lists (positions). Now you need **labels**. A **dictionary** maps a **key** (name) to a **value** — like contacts: `name` → phone.

- **Fast by key** (hash table, $\approx O(1)$ average).
- **Clear meaning**: read `phone["John"]` like English.

Dict basics — literal, access, membership

```
car = {"brand": "Ford", "model": "Mustang", "year": 1964}
print(len(car))              # 3
print(car["model"])          # "Mustang"  (KeyError if missing)
print("model" in car)        # True       (checks keys, fast)
```

- Literal with **{key: value}** pairs; keys must be **hashable** (e.g., str, int, tuple).

Dict basics — literal, access, membership

```
car = {"brand": "Ford", "model": "Mustang", "year": 1964}
print(len(car))              # 3
print(car["model"])          # "Mustang"  (KeyError if missing)
print("model" in car)        # True       (checks keys, fast)
```

- Literal with **{key: value}** pairs; keys must be **hashable** (e.g., str, int, tuple).
- **dict(name="John", age=36)** also works.

Dict basics — literal, access, membership

```
car = {"brand": "Ford", "model": "Mustang", "year": 1964}
print(len(car))              # 3
print(car["model"])          # "Mustang"  (KeyError if missing)
print("model" in car)        # True       (checks keys, fast)
```

- Literal with **{key: value}** pairs; keys must be **hashable** (e.g., str, int, tuple).
- **dict(name="John", age=36)** also works.
- Empty **{}** is a dict (contrast: empty set is **set()**).

Access safely: get with a default

```
user = {"name": "Amina", "role": "student"}

# Direct indexing: raises KeyError if missing
# dept = user["dept"]   # <-- crash if no "dept"

dept = user.get("dept", "N/A")   # safe default
print(dept)                      # "N/A"
```

- **get(key, default)** avoids KeyError.

Access safely: get with a default

```
user = {"name": "Amina", "role": "student"}

# Direct indexing: raises KeyError if missing
# dept = user["dept"]   # <-- crash if no "dept"

dept = user.get("dept", "N/A")   # safe default
print(dept)                      # "N/A"
```

- **get(key, default)** avoids KeyError.
- **Use in to check presence: "dept" in user.**

Changing a dict (add, update, remove)

```
car = {"brand": "Ford", "model": "Mustang", "year": 1964}

car["color"] = "red"                    # add or overwrite one key
car["year"]  = 2020                     # mutate existing

car.update({"owner": "Sara"})           # merge many (iterable of pairs ok)
who = car.pop("owner", None)            # remove by key (safe default)
last = car.popitem()                    # remove last inserted (key, value)
# car.clear()                           # empty the dict
```

Duplicate keys on creation/merge: last write wins.

Keys, values, items — and how to loop

```
car = {"brand":"Ford","model":"Mustang","year":2020}

print(list(car.keys()))      # ['brand','model','year']   (dynamic views)
print(list(car.values()))    # ['Ford','Mustang',2020]
print(list(car.items()))     # [('brand','Ford'),...]

for k in car:                # iterate keys (common)
    print(k, car[k])

for k, v in car.items():     # iterate pairs
    print(k, "->", v)
```

Views update as the dict changes. Use `items()` *for clean key/value loops.*

Why dicts feel fast for lookups

Algorithm

List search $\rightarrow$ scan items ($O(n)$). Dict lookup $\rightarrow$ hash key $\Rightarrow$ direct slot ($\approx O(1)$ average).

- If you will ask "what's the value for this name?" many times, **use a dict**.

Why dicts feel fast for lookups

Algorithm

List search $\rightarrow$ scan items ($O(n)$). Dict lookup $\rightarrow$ hash key $\Rightarrow$ direct slot ($\approx O(1)$ average).

- If you will ask "what's the value for this name?" many times, **use a dict**.
- Build once, query often.

Three mini-workflows (very common)

1) Word count (frequency map)

```
counts = {}
for w in input().split():
    w = w.lower()
    counts[w] = counts.get(w, 0) + 1
```

Dict comprehension — build maps in one line

```
words = "alice bob charlie".split()
lengths = {w: len(w) for w in words}
# {'alice':5, 'bob':3, 'charlie':7}

# Normalize + filter
line = "A a an the apple".split()
stop = {"a","an","the"}
keep = {w.lower(): len(w) for w in line if w.lower() not in stop}
```

Structure: `{ key_expr : value_expr for item in iterable if cond }`.

Common pitfalls

- **KeyError**: `d["x"]` when missing. Prefer `get` or check with `"x" in d`.

Common pitfalls

- **KeyError**: `d["x"]` when missing. Prefer `get` or check with `"x" in d`.
- **Overwrites**: duplicate keys keep the last value.

Common pitfalls

- **KeyError**: `d["x"]` when missing. Prefer `get` or check with `"x" in d`.
- **Overwrites**: duplicate keys keep the last value.
- **Keys must be hashable**: lists/dicts can't be keys; use tuples or strings.

Common pitfalls

- **KeyError**: `d["x"]` when missing. Prefer `get` or check with `"x" in d`.
- **Overwrites**: duplicate keys keep the last value.
- **Keys must be hashable**: lists/dicts can't be keys; use tuples or strings.
- **Copy vs alias**: `d2 = d` shares; use `d.copy()` or `dict(d)`.

Practice — two short tasks

```
# 1) Build a tiny phonebook, then answer 3 queries (fast):
book = {"ali":"0611","salma":"0612","omar":"0613"}
for _ in range(3):
    name = input().strip().lower()
    print(book.get(name, "not found"))

# 2) Word count: print top-1 most frequent word (tie: any).
counts = {}
for w in input().split():
    counts[w] = counts.get(w, 0) + 1
```

Reads naturally, runs fast: that's the point of key → value.

Recap — the big picture

- **Functions** = **named recipe**. Define once with `def`, call with `name(...)`, send results with `return`, sketch with `pass`.

Rule of thumb: pick by *how you access*: position (list), presence (set), label (dict).

Recap — the big picture

- **Functions** = **named recipe**. Define once with **`def`**, call with **`name(...)`**, send results with **`return`**, sketch with **`pass`**.
- **Lists** = **ordered, changeable** sequences. Index with **`0`**..**`n-1`**, iterate, **`append`**/**`insert`**/**`pop`**, slicing.

Rule of thumb: pick by *how you access*: position (list), presence (set), label (dict).

Recap — the big picture

- **Functions** = **named recipe**. Define once with **`def`**, call with **`name(...)`**, send results with **`return`**, sketch with **`pass`**.
- **Lists** = **ordered, changeable** sequences. Index with **`0`**..**`n-1`**, iterate, **`append`**/**`insert`**/**`pop`**, **slicing**.
- **Sets** = **unique items + fast membership**. No indexes, great for "seen?" checks and dedupe.

Rule of thumb: pick by *how you access*: position (list), presence (set), label (dict).

Recap — the big picture

- **Functions** = **named recipe**. Define once with **`def`**, call with **`name(...)`**, send results with **`return`**, sketch with **`pass`**.
- **Lists** = **ordered, changeable** sequences. Index with **`0`**..**`n-1`**, iterate, **`append`**/**`insert`**/**`pop`**, **slicing**.
- **Sets** = **unique items + fast membership**. No indexes, great for "seen?" checks and dedupe.
- **Dicts** = **key → value** mapping. Fast lookup by name; **`get`**, **`items()`** loops, **`update`**.

Rule of thumb: pick by *how you access*: position (list), presence (set), label (dict).

Recap — micro-pattern cheat sheet

Function skeleton

```
def fn(a, b):
    # TODO: implement
    pass
```

List: read, convert, process

```
vals = [int(s) for s in input().split()]
even = [x for x in vals if x % 2 == 0]
```

Set: dedupe + fast membership

```
vocab = set(input().split())
print("alice" in vocab)
```

Dict: counting/frequency map

Recap — pitfalls and fixes

- **List aliasing** (b = a) → both names change; **copy** with `a[:]` or `a.copy()`.

Recap — pitfalls and fixes

- **List aliasing** (b = a) → both names change; **copy** with `a[:]` or `a.copy()`.
- **Empty set** is **`set()`**, not `{}` (that's an empty dict).

Recap — pitfalls and fixes

- **List aliasing** (b = a) → both names change; **copy** with `a[:]` or `a.copy()`.
- **Empty set** is **`set()`**, not `{}` **(that's an empty dict)**.
- **Set has no order/index** → don't rely on order.

Recap — pitfalls and fixes

- **List aliasing** (b = a) → both names change; **copy** with `a[:]` or `a.copy()`.
- **Empty set** is **`set()`**, not `{}` (that's an empty dict).
- **Set has no order/index** → don't rely on order.
- **KeyError on dict access** → **use `get`** or check `key in d`.

Recap — pitfalls and fixes

- **List aliasing** (b = a) → both names change; **copy** with `a[:]` or `a.copy()`.
- **Empty set** is **`set()`**, not `{}` (that's an empty dict).
- **Set has no order/index** → don't rely on order.
- **KeyError on dict access** → **use `get`** or check `key in d`.
- **Unhashable keys/items** → keys in dicts/sets must be hashable (use tuples, not lists).

Recap — pitfalls and fixes

- **List aliasing** (b = a) → both names change; **copy** with `a[:]` or `a.copy()`.
- **Empty set** is **`set()`**, not `{}` **(that's an empty dict)**.
- **Set has no order/index** → don't rely on order.
- **KeyError on dict access** → **use `get`** or check `key in d`.
- **Unhashable keys/items** → keys in dicts/sets must be hashable (use tuples, not lists).
- **Too-clever comprehensions** → if it's long/nested, **switch to a plain loop**.

Exit ticket — 90 seconds

1) Classify each task (list / set / dict):

- Keep student names in the order they signed up.
- Check if an ID has already been seen, fast.
- Map country name to its telephone code.

2) Predict the output:

```
d = {"a":1, "b":2}
d.update({"b":5, "c":7})
print(d.get("x", 0), d["b"])
```

Exit ticket — 90 seconds

3) Fix the bug (choose one):

```
S = {}          # should be an empty set
S.add("x")
```