

Module 1

1. The way of the program (Ch.1.1–1.7)

The Python programming language

What it is:

Python is a high-level, interpreted, general-purpose language that emphasizes readability and rapid development. "High-level" means code is written closer to human reasoning than machine details; "interpreted" means code is executed by an interpreter line-by-line (no explicit compile step required).

Why used for learning:

- Clean, minimal syntax makes concepts clear.
- Interactive REPL (Read-Eval-Print Loop) encourages experimentation.
- Large standard library and readable style let beginners build useful programs quickly.

Key features:

- Dynamic typing: variables don't have fixed types; values do.
- Indentation enforces block structure (no braces).
- Rich built-in types (numbers, strings, lists, dicts, sets, tuples).
- Functions are first-class (can be passed around).

Example (hello + explanation):

interactive style

```
name = input("What's your name? ") # input returns a string print("Hello,", name)
```

- `input()` reads from the user.
- `print()` displays output.

Tip: Use the REPL to test small snippets — it builds intuition quickly.

Common pitfalls: Expecting variable declarations (not needed), confusing `=` (assignment) with `==` (comparison).

Mini exercise: Run `print(2 + 3 * 4)` and explain why the printed value is 14.

What is a program?

Definition: A program is a precise sequence of instructions a computer executes to accomplish a task. It maps inputs $\rightarrow$ processing $\rightarrow$ outputs.

Thinking like a computer scientist:

- Decompose problems into sub-tasks.
- Represent data appropriately.
- Design clear algorithms and verify their correctness.

Simple program example (sum of two numbers):

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
print("Sum =", a + b)
```

- Input: user numbers (strings converted to int).
- Processing: addition.
- Output: printed sum.

Common pitfalls: Not validating inputs (e.g., user enters non-numeric text).

Mini exercise: Write a short program to compute the area of a circle given radius input.

What is debugging?

Definition: The systematic process of detecting, isolating, and fixing errors (bugs) in code.

Strategy:

1. Reproduce the bug (consistent test case).
2. Narrow down the cause (binary search/print tracing).
3. Fix and re-test.
4. Add tests to prevent regressions.

Techniques: print debugging, assertions, using small test inputs, stepping with a debugger.

Example (print tracing):

```
def avg(a, b):
    print("DEBUG: a=", a, "b=", b)
```

```
return (a + b) / 2 print(avg(5, 7))
```

Tip: Rewriting a failing function from scratch is sometimes faster than chasing subtle bugs.

Mini exercise: Introduce an intentional bug (e.g., off-by-one) in a loop and practice tracing it with prints.

Syntax errors

What: Errors in program grammar that prevent Python from parsing your code. Detected before execution (or at least before running the affected code block).

Common examples:

- Missing colon after if/for/def.
- Unclosed parentheses, brackets, or quotes.
- Invalid indentation.

Example:

```
# SyntaxError: invalid syntax if x > 0
    print("positive")
```

How to fix: Read the error message (it often shows the line), check previous lines for missing symbols or wrong indentation.

Mini exercise: Create three small syntax errors and run them to see Python's error messages.

Runtime errors

What: Errors that occur while a program runs (after parsing). The program starts but fails at some point.

Examples:

- Division by zero.
- Index out of range when accessing lists/strings.
- Attribute errors (calling non-existent functions on objects).

Example:

```
a = 5
b = 0
print(a / b) # ZeroDivisionError at runtime
```

How to handle: Use try/except (later chapters), or add checks before risky operations.

Mini exercise: Trigger and catch a `ValueError` when converting a non-numeric string to `int`.

Semantic errors (logic errors)

What: Program runs and produces output but the output is incorrect — the logic is wrong. These are often the hardest to find.

Example: Bad average formula:

```
def average(a, b):  
    return (a + b) / 1          # should be /2
```

How to detect: Unit tests, manual tracing of algorithm on known inputs, assertions.

Tip: Break algorithm into named variables and print intermediate values — it makes reasoning easier.

Mini exercise: Write a function to compute quadratic formula roots and test with an input where discriminant is known.

Experimental debugging

What: Using experiments — changing inputs, adding prints, simplifying code — to learn program behavior and find bugs.

Steps:

- Create small controlled test cases.
- Isolate components and test them independently.
- Use asserts to check invariants.

Example: To isolate bug in list processing, print the list before and after each transformation step.

Mini exercise: For a function that fails on some inputs, design 5 test cases (edge, typical, empty, large, invalid) to help isolate the bug.

2. Variables, Expressions and Statements (Ch.2.1–2.12)

Values and data types

Definition: A *value* is a piece of data (like 3, "hello"). A *datatype* groups values with similar behaviors and defines which operations are allowed.

Common built-in types:

- int— integers (... -2, -1, 0, 1, 2 ...)
- float— floating-point numbers (decimals)
- str— strings (text)
- bool— boolean values (True/ False)
- compound: list, tuple, dict, set

Inspect type:

```
x = 10
print(type(x)) # <class 'int'>
```

Pitfall: Mixing incompatible types (e.g., adding string and int) causes a runtime error.

Mini exercise: Try `type(3.0)`, `type("3")`, `type(True)` and explain differences.

Variables

Definition: A name that refers to a value. Assignment binds a name to a value.

Examples:

```
count = 10
count = count + 1 # updates the value referred by count
```

Best practices: Use descriptive names (`total_sum`, `radius`) for readability.

Scope (brief): Names defined in a function are local to that function; global names exist at module level (learn more in later chapters).

Mini exercise: Predict outputs of:

```
a = 5
b = a
a = 7
print(b)
```

Variable names and keywords

Rules:

- Must start with a letter (a–z, A–Z) or underscore `_`.
- Followed by letters, digits, or underscores.
- Case sensitive (Value != value).
- Cannot be Python *keywords* (if, for, while, def, class, etc.)

Style conventions (PEP8): snake_case for variables and functions; CamelCase for classes.

Mini exercise: Identify which of these are valid: `_temp`, `2nd`, `my-var`, `class`.

Statements

Definition: A line (or block) of code that performs an action — assignment, function call, control statement.

Examples:

- Assignment: `x = 3`
- Function call: `print(x)`
- Control: `if`, `for`, `while`

Note: Many statements can be chained on one line with `;` — but avoid for readability.

Mini exercise: Convert `x = x + 2` into the augmented assignment form.

Evaluating expressions

What: Expressions combine values, variables and operators to produce new values.

Examples:

- Arithmetic: `3 + 4 * 2`
- String concat: `"hi" + "!"`
- Function call: `len("abc")`

Order of evaluation: Controlled by precedence and parentheses. Evaluate inner parentheses first then follow operator precedence.

Mini exercise: Evaluate `2 + 3 * 4 ** 2` and explain priority.

Operators and operands

Arithmetic operators:

- `+` add, `-` subtract, `*` multiply, `/` float division, `//` integer (floor) division, `%` modulus, `**` exponent.

Comparison operators: `==`, `!=`, `<`, `<=`, `>`, `>=` — produce boolean results.

Logical operators: `and`, `or`, `not`.

Examples:

```
5 // 2    # 2
5 % 2     # 1
2 ** 3    # 8
```

Pitfall: Floating point precision issues (e.g., $0.1 + 0.2 \neq 0.3$ exactly because of binary floats).

Mini exercise: Use % to test whether a number is divisible by 3.

Type-converter functions

Purpose: Convert values between types: int(), float(), str(), bool().

Examples:

```
s = "123"
n = int(s)      # 123
f = float("3.14")
```

Pitfall: int("3.14") raises ValueError. Convert to float first then int if needed:
int(float("3.14")).

Mini exercise: Convert user input string to int safely using try/except.

Order of operations

Precedence (high → low):

1. () parentheses
2. ** exponent
3. *, /, //, %
4. +, -
5. Comparisons, then logical ops.

Rule: Use parentheses to avoid ambiguity and make intent clear.

Mini exercise: Place parentheses in $1 + 2 * 3 ** 2$ to get result 9.

Operations on strings

Strings: Immutable sequences of characters. Common operations:

- Concatenation: "a" + "b" → "ab"
- Repetition: "ha" * 3 → "hahaha"
- Indexing: s[0] first character (0-based).
- Slicing: s[1:4] substring from index 1 up to (but not including) 4.
- Length: len(s).

Example:

```
s = "Python"
print(s[1], s[-1], s[:3], s[3:])
```

Pitfall: Trying to mutate string characters (e.g., `s[0] = "p"`) raises error — strings are immutable.

Mini exercise: Extract the file extension from "document.pdf" using slicing and `rfind/split`.

Input

input(prompt) returns user input as a *string*. Convert to required type as needed.

Example:

```
age_str = input("Age: ")
age = int(age_str)
```

Pitfall: Unsanitized input; always handle conversion errors.

Mini exercise: Write a program that prompts for two numbers and prints their product, handling non-numeric input gracefully.

Composition

Meaning: Combine simple expressions and statements to form complex computations. Build small reusable blocks and compose them.

Example (area of trapezoid):

```
a = float(input("base1: "))
b = float(input("base2: "))
h = float(input("height: "))
area = 0.5 * (a + b) * h
print("Area:", area)
```

Tip: Assign intermediate results to variables with descriptive names — easier to debug.

Mini exercise: Compose a function that computes Body Mass Index (BMI) from weight (kg) and height (m).

The modulus operator

Definition: `a % b` returns the remainder when `a` is divided by `b`.

Common uses:

- Determine parity (`n % 2 == 0` → even).

- Extract digits: last digit $n \% 10$.
- Cycle through indices: $i \% n$.

Example:

```
for i in range(1, 11): if i % 2
    == 0:
        print(i, "is even")
```

Mini exercise: Print last two digits of a number using modulus and integer division.

3. Iteration (inc. Ch.3.3 and additional topics you listed)

Assignment & updating variables Assignment: create

or update a binding: $x = 5$. Updating: often in loops:

```
total = 0
for i in range(1, 6):
    total = total + i      # or total += i
```

Tip: Use augmented assignments ($+=$, $-=$, $*=$, $/=$) for clarity and conciseness.

Mini exercise: Compute factorial of n using variable update in a loop.

The forloop

When to use: iterate over sequence of known items (range, list, string, tuple).

Syntax:

```
for item in sequence: #
    body
```

Examples:

```
for i in range(5):      # 0..4
    print(i)
```

```
for ch in "hello": print(ch)
```

Common pattern: iterate with index when you need both index and value: `for i, val in enumerate(list):`

Mini exercise: Sum squares of numbers from 1 to 10 using a for loop.

The while statement

When to use: repeat until a condition becomes false (unknown number of iterations).

Syntax:

```
while condition: #  
    body
```

Example:

```
n = 5  
while n > 0:  
    print(n) n -  
    = 1
```

Pitfalls: risk of infinite loops if condition never becomes false. Always ensure variable(s) in condition are updated.

Mini exercise: Implement while loop to reverse digits of a positive integer.

The Collatz $3n + 1$ sequence

Definition: For given $n > 0$, repeatedly apply: if n is even $\rightarrow n = n/2$; else $\rightarrow n = 3n + 1$; stop when $n == 1$.

Conjecture: for any starting n , sequence eventually reaches 1 (unproven in math, but works for tested numbers).

Implementation:

```
n = int(input("n = "))  
while n != 1:  
    print(n, end=' ')  
    if n % 2 == 0:  
        n = n // 2  
    else:  
        n = 3 * n + 1  
print(1)
```

Mini exercise: Count iterations for the Collatz sequence for various starting n and find the maximum length up to e.g., 1000.

Tables and two-dimensional tables

Tables: represent structured data (rows and columns) using nested lists (list of lists).

Example matrix:

```
matrix = [  
    [1, 2, 3],  
    [4, 5, 6],  
    [7, 8, 9]  
]  
print(matrix[1][2]) # row 1, col 2  $\rightarrow$  6
```

```
for row in matrix:
    for val in row:
        print(val, end=' ')
    print()
```

Pitfall: Shallow copying nested lists causes aliasing; use list comprehensions or copy module for deep copies.

Mini exercise: Write a function to compute transpose of a matrix represented as nested lists.

break statement

Effect: Exit the nearest enclosing loop immediately.

Example:

```
for i in range(10):
    if i == 5:
        break
    print(i)    # prints 0..4
```

Tip: Use to stop early when target found; avoid overusing as it can make flow harder to follow.

Mini exercise: Use break in a search routine that finds first occurrence of a value in a list.

continue statement

Effect: Skip the remainder of the loop body for current iteration and continue with next iteration.

Example:

```
for i in range(6):
    if i % 2 == 0:
        continue
    print(i)    # prints odd numbers only
```

Mini exercise: Iterate 1..20 and print numbers not divisible by 3 using continue.

Paired data

Meaning: Data that come in related pairs (e.g., name and score). Representations:

- Parallel lists: names[i] pairs with scores[i]
- List of tuples: pairs = [("Alice", 90), ("Bob", 85)] (preferable)
- Dict: {"Alice": 90, "Bob": 85} (if mapping by unique key)

Example (list of tuples):

```
students = [("Alice", 90), ("Bob", 85)]
for name, score in students:
    print(name, "->", score)
```

Mini exercise: Convert two parallel lists names and scores into a list of tuples.

Nested loops for nested data

Use: iterate over matrices, grids, or nested lists.

Example (matrix sum):

```
total = 0
for row in matrix:
    for val in row:
        total += val
```

Pitfall: Be careful with variable names so indices don't collide in nested loops.

Mini exercise: Given a nested list of variable row lengths, print the max element in each row.

4. Functions (Ch.4.4, 4.5)

Functions with arguments and return values

What is a function: Named block of code performing a specific task — promotes reuse and abstraction.

Syntax:

```
def function_name(parameters):
    """Optional docstring describing purpose.""" # body
    return result
```

Arguments vs parameters:

- Parameters: variable names in function definition.
- Arguments: concrete values passed when calling function.

Examples:

```
def greet(name):                # name is a parameter
    return f'Hello, {name}'

print(greet("Mahesh"))          # "Mahesh" is an argument
```

Multiple returns: Functions can return multiple values via tuples:

```
def min_max(seq):
    return min(seq), max(seq)
```

```
lo, hi = min_max([3, 7, 1])
```

Pure functions vs side effects:

- Pure: same output for same inputs and no side effects (makes testing easy).
- Impure: modifies global state, does I/O — use when necessary, but confine side effects.

Default / optional parameters:

```
def power(x, n=2):          # n defaults to 2
    return x ** n
```

Docstrings & readability: Use docstrings to document expected parameter types and returned values.

Pitfalls: Mutating default parameter values (like lists) leads to surprising behavior — use Nonesentinel and create inside.

Example pitfall:

```
def append_to_list(x, lst=[]):
    lst.append(x)
    return lst
```

This reuses same default list across calls — usually undesired

Correct pattern:

```
def append_to_list(x, lst=None):
    if lst is None:
        lst = []
    lst.append(x)
    return lst
```

Mini exercise: Write a function `is_prime(n)` that returns `True` if `n` is prime; test with several values.

Module 2

5. Strings (Ch.5.1–5.3 topics expanded)

Working with strings as single things

Strings as atomic values: You can pass, print, concatenate strings; think of them as single data items unless you need to inspect characters.

Example:

```
s = "Hello, world!"
print(s.upper())
print(s.lower())
```

Useful methods: `upper()`, `lower()`, `strip()`, `replace()`, `split()`, `join()`, `format()`.

Mini exercise: Convert " Hello " to "Hello" using `strip()`.

Working with the parts of a string

String indexing and slicing: Extract characters or substrings using `[i]`, `[i:j]`, `[i:j:k]`.

Examples:

```
s = "python" s[0]
                # 'p'
s[-1]          # 'n' last character s[1:4]
# 'yth'
s[:3]          # 'pyt'
s[::2]         # every second char 'pto'
```

Tip: Slicing is safe: out-of-range indices don't raise errors (they are clamped).

Mini exercise: From "2025-11-03", extract the year, month, and day as integers.

Length

`len(s)` returns number of characters. Useful for loops and bounds checking.

Example:

```
s = "abc"
for i in range(len(s)): print(i, s[i])
```

Pitfall: `len()` yields number of code points — Unicode grapheme clusters may be more complex; usually not an issue for ASCII.

Mini exercise: Show that `len("ñ") == 1` in Python but visual width may differ in terminal.

Traversal and the for loop

Iterate over characters:

```
for ch in "banana":
    print(ch)
```

Use `enumerate` to get index & char:

```
for i, ch in enumerate("banana"): print(i,
    ch)
```

Mini exercise: Count how many vowels are in a string.

Slices

Powerful for substring extraction and stepping. Examples:

```
s = "ABCDEFGHJIJ"
s[2:8]      #'CDEFGH'
s[::3]      # every 3rd char
```

Tip: Negative step reverses: `s[::-1]` returns reversed string.

Mini exercise: Reverse a string using slice.

String comparison

Lexicographic order: strings compare character-by-character using Unicode code points.

Examples:

```
"apple" < "banana" # True
"Apple" < "apple"   # True because 'A' < 'a' in Unicode ordering
```

Tip: For case-insensitive comparisons, use `s.lower()`.

Mini exercise: Sort a list of names case-insensitively.

Strings are immutable

Meaning: Once created, characters in a string cannot be changed. Any "modification" creates a new string.

Implication: Building strings in loops by repeated concatenation can be inefficient; prefer `list + join()` for many pieces.

Efficient pattern:

```
parts = []
for item in items:
    parts.append(str(item))
result = "".join(parts)
```

Mini exercise: Implement `naive_replace_first(s, old, new)` without using built-in `replace`.

in and not operators

Membership test: Returns boolean if substring exists inside string.

Examples:

```
"py" in "python"      # True
"z" not in "abc"       # True
```

Mini exercise: Check whether a sentence contains any of the words in a "blacklist".

A find function

Purpose: Find index of first occurrence of substring or character, or return -1 if not present.

Custom simple implementation:

```
def find_char(s, ch):
    for i in range(len(s)):
        if s[i] == ch:
            return i
    return -1
```

Built-in alternative: `s.find(sub)` and `s.index(sub)` (index raises `ValueError` if not found).

Mini exercise: Implement `find_all(s, ch)` returning a list of all indices where `ch` appears.

Looping and counting

Typical pattern: iterate through string and count occurrences.

Example:

```
def count_char(s, ch):
    count = 0
    for c in s:
        if c == ch:
            count += 1
    return count
```

Mini exercise: Count number of words in a sentence by splitting on whitespace.

Optional parameters (for string functions)

Example function with optional parameter:

```
def greet(name="Guest"):
    print("Hello", name)
```

Application in strings: custom `find` might accept starting index: `s.find(sub, start)`.

Mini exercise: Extend `count_char` to be case-insensitive via optional parameter `case_sensitive=True`.

Built-in findmethod

Usage: `s.find(sub[, start[, end]])`— returns index or -1. Efficient C- implemented search.

Example:

```
s = "abracadabra"
s.find("bra") # 1
s.find("xyz") # -1
```

Mini exercise: Use `find` to implement a simple substring replacement for the first occurrence.

splitmethod

Purpose: Split a string into a list using a separator; default is any whitespace.

Examples:

```
"one two three".split()          # ['one','two','three']
"2025-11-03".split("-")         # ['2025','11','03']
```

Tip: `maxsplit` parameter limits splits.

Mini exercise: Parse CSV line "Alice,90,85,78" into fields and compute average.

Cleaning up your strings

Common tasks: `strip()` remove leading/trailing whitespace, `lower()/upper()` normalize case, `replace()` to remove unwanted characters.

Example normalization pipeline:

```
s = input().strip().lower()
```

Mini exercise: Clean a list of email addresses removing extra whitespace and converting to lowercase.

formatmethod

Purpose: Neatly insert values into strings with readable formatting.

Examples:

```
name = "Alice"
score = 93.456
s = "Name: {}, Score: {:.1f}".format(name, score) # "Name:
Alice, Score: 93.5"
```

Python f-strings (modern):

```
s = f"Name: {name}, Score: {score:.1f}"
```

Mini exercise: Format a table of names and scores with columns aligned.

6. Tuples

Tuples are used for grouping data

Definition: Immutable ordered sequences; useful to represent records (e.g., coordinates).

Example:

```
point = (3, 4)
x, y = point # tuple unpacking
```

Why use tuples: immutability (safety), hashability (can be dictionary keys), small fixed groups.

Mini exercise: Represent RGB color as a tuple and write a function to scale brightness by a factor (producing a new tuple).

Tuple assignment

Parallel assignment/unpacking:

```
a, b, c = (1, 2, 3)
a, b = b, a # swap values
```

Common on return values:

```
min_val, max_val = min_max([3,4,1])
```

Mini exercise: Swap two variables without a temporary variable using tuple assignment.

Tuples as return values

Multiple outputs: Return several values packed into a tuple:

```
def divmod_pair(a, b): return a
    // b, a % b
```

```
q, r = divmod_pair(17, 5) # q=3, r=2
```

Mini exercise: Write a function that given a list returns both sorted list and the median as a tuple.

Composability of data structures

Key idea: Combine lists, tuples, dicts to model complex data:

- List of tuples: [("Alice", 90), ("Bob", 85)]
- Dict with tuple keys: {(x,y): value}
- Nested lists for matrices.

Mini exercise: Model library books where each book is a dict with keys title, authors (list), year, isbn.

7. Lists (many topics)

List values & accessing elements

Definition: Mutable ordered sequences of values, possibly heterogeneous.

Example:

```
lst = [10, "abc", 3.14]
print(lst[0])          # 10
```

Negative indices: lst[-1] last element.

Pitfall: IndexError if index out of range.

Mini exercise: Access every second element using slicing: lst[::2].

List length

len(list) returns number of elements.

Use with loops:

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

Prefer: Direct iteration when index not needed:

```
for x in lst:
    print(x)
```

Mini exercise: Compute average of numeric list elements safely (handle empty list).

List membership

intests membership: x in lst.

Example:

```
if "apple" in fruits: print("found")
```

Mini exercise: Remove duplicates from list preserving order using membership check and new list.

List operations

- Concatenation: `a + b`
- Repetition: `a * 3`
- Append/extend: `append()`, `extend()`
- Insert: `insert(index, value)`

Examples:

```
a = [1,2]
a.append(3)      # [1,2,3]
a.extend([4,5]) # [1,2,3,4,5]
```

Mini exercise: Merge two sorted lists into one sorted list.

List slices

Slicing on lists works like strings, returns a new list.

Examples:

```
a = [0,1,2,3,4]
a[1:4]      # [1,2,3]
a[:2]       # [0,1]
a[::2]      # [0,2,4]
```

Assignment to slice (powerful mutability):

```
a[1:3] = [10, 11]      # changes elements 1 and 2
```

Mini exercise: Use slicing to rotate a list by k positions.

Lists are mutable

You can change elements, append, remove.

Example:

```
a[0] = 100
```

Pitfall: When lists are shared (aliasing), a change via one name affects all references.

List deletion

`del a[i]`, `a.pop(i)`(returns removed item), `a.remove(value)`(removes first matching value).

Examples:

```
del a[2]
x = a.pop() # remove last element
a.remove(3) # remove first occurrence of 3
```

Mini exercise: Implement function to remove all occurrences of a value from list in-place.

Objects and references

Key idea: Variables hold references to objects; lists are objects. Assignment copies reference, not contents.

Example (aliasing):

```
a = [1,2,3]
b = a
b[0] = 100
print(a) # [100,2,3]
```

Cloning: `b = a[:]` or `b = list(a)` or `copy()` to make shallow copy.

Mini exercise: Demonstrate aliasing with nested lists and explain consequences.

Aliasing and cloning lists

Aliasing: multiple names point to same list (see above).

Cloning methods (shallow copy):

```
b = a[:]          # slice copy
b = list(a)
b = a.copy()
```

Deep copy: for nested lists where you want independent inner lists, use `copy.deepcopy()`.

Mini exercise: Show shallow copy vs deep copy behavior with nested lists.

Lists and for loops

Iterate directly:

```
for item in lst:
    process(item)
```

Enumerate: get index + value

```
for i, val in enumerate(lst): print(i, val)
```

Mini exercise: Use enumerate to replace elements while iterating by index.

List parameters (functions that accept list)

By reference: Passing list to function gives function a reference to original list. Modifying inside function mutates caller's list unless you copy.

Example (mutates):

```
def append_one(lst): lst.append(1)
```

```
a = []
append_one(a)
print(a) # [1]
```

If you need immutable behavior, copy the list inside function or return a new list.

Mini exercise: Write a function `scale_list(lst, factor)` that returns new list scaled by factor without modifying input.

List methods

Common: `append`, `extend`, `insert`, `remove`, `pop`, `clear`, `index`, `count`, `sort`, `reverse`.

Examples:

```
a.sort() a.reverse()
a.index(value)
a.count(value)
```

Tip: `sorted(a)` returns new sorted list, leaving `a` unchanged.

Mini exercise: Use `sort` with `key` parameter to sort words by length.

Pure functions and modifiers

- Pure functions: return new results, no side effects — easier to reason about and test.
- Modifiers: functions that change their input lists in place (mutators). Use clear naming or document behavior.

Example pure:

```
def add_one(lst):
    return [x + 1 for x in lst]
```

Example modifier:

```
def add_one_inplace(lst):  
    for i in range(len(lst)): lst[i] += 1
```

Mini exercise: Convert a modifier that multiplies elements by 2 into a pure function.

Functions that produce lists

Generators: functions returning lists built from computations (list comprehensions are common).

Examples:

```
def evens(n):  
    return [i for i in range(n) if i % 2 == 0]
```

Lazy alternative: generator expressions ((i for i in range(n) if ...)) to avoid building full list.

Mini exercise: Write function that returns list of prime numbers up to n (simple sieve or trial division).

Strings and lists

Conversion: list(s) gives list of chars; "".join(list_of_str) creates string.

Use cases: tokenizing text, constructing strings efficiently.

Example:

```
chars = list("hello")  
s = "".join(chars)
```

Mini exercise: Given a sentence, split into words, reverse each word, and join back.

list and range

range() produces sequence of integers; often used to create lists:

- Python 3: list(range(5)) → [0, 1, 2, 3, 4]

Looping with indices:

```
for i in range(len(lst)): lst[i] *= 2
```

Mini exercise: Create a list of squares using list comprehension and range.

Nested lists and matrices

Matrices: modelled as lists of lists (rows). Access `m[r][c]`.

Example transpose:

```
def transpose(m):  
    return [ [row[i] for row in m] for i in range(len(m[0])) ]
```

Pitfall: Creating rows by multiplication `[[0]*n]*m` creates repeated same row (aliasing); prefer list comprehension:

```
matrix = [[0 for _ in range(cols)] for _ in range(rows)]
```

Mini exercise: Implement matrix multiplication for 2D lists (ensure sizes compatible).