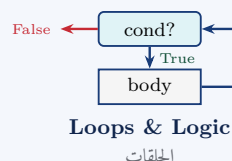
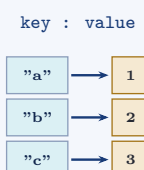


PYTHON PROGRAMMING

مقدمة في البرمجة

Comprehensive Theoretical Summary

تلخيص شامل (نظري)



حسن إسماعيل صالح

تخصص هندسة الذكاء الاصطناعي - الجامعة الإسلامية بغزة

إعداد الطالب حسن إسماعيل صالح تحت إشراف الدكتور محمد الحنجوري

التاريخ: ١٤ يوليو ٢٠٢٦

بسم الله الرحمن الرحيم

الحمد لله رب العالمين، والصلاة والسلام على سيدنا محمد وعلى آله وصحبه أجمعين.

يسرني أن أضع بين أيديكم هذا الملخص لمادة مقدمة في البرمجة بلغة بايثون، والذي قمت بإعداده خلال دراستي للمساق بهدف جمع أهم الأفكار والمفاهيم والقوانين الواردة في المادة بصورة مرتبة ومبسطة تساعد على فهمها ومراجعتها بشكل أسرع. وقد حاولت في هذا العمل تلخيص المحتوى الأساسي للمادة مع التركيز على النقاط المهمة التي يحتاجها الطالب أثناء الدراسة أو قبل الامتحانات، وذلك لتسهيل الطريق على الطلبة الذين سيدرسون هذا المساق في المستقبل بإذن الله.

كما أتقدم بجزيل الشكر والتقدير إلى الأستاذ الدكتور الفاضل محمد الحنجوري، لما بذله من جهد كبير في شرح المادة وتوضيح مفاهيمها، ولأسلوبه المميز في التدريس الذي ألهمنا وكان له أثر كبير في فهمنا للمادة والاستفادة منها.

هذا الملخص ما هو إلا جهد بشري قابل للخطأ أو النقص، وقد كتبته بنية الإفادة ونشر العلم بين الطلبة. أسأل الله أن ينفع به، وأن يجعله عملاً خالصاً لوجهه الكريم. كما أرجو من كل من ينتفع به أن يذكر شهداءنا وجرحانا بدعوة صادقة.

والحمد لله رب العالمين.

Contents المحتويات

02	Elementary Programming البرمجة الأساسية	p. 5
-----------	---	-------------

03	Mathematical Functions, Strings, and Objects الدوال الرياضية والسلاسل النصية والكائنات	p. 13
-----------	--	--------------

04	Selections الاجمل الشرطية	p. 17
-----------	-------------------------------------	--------------

05	Loops الحلقات التكرارية	p. 22
-----------	-----------------------------------	--------------

06	Functions الدوال	p. 27
-----------	----------------------------	--------------

10	Lists القوائم	p. 32
-----------	-------------------------	--------------

11	Multidimensional Lists القوائم متعددة الأبعاد	p. 39
-----------	---	--------------

13	Files and Exception Handling الملفات ومعالجة الاستثناءات	p. 43
-----------	--	--------------

14 Tuples, Sets, and Dictionaries

الصفوف والمجموعات والقواميس

p. 48

Common Ideas and Cautions

Before the chapters, here are the habits that quietly save marks in this course. Most of them I learned by losing points first, so keep them in the back of your head whenever you trace Python code.

- ★ When in doubt, don't guess the output, trace it. Start at the first line of the module (or inside `main()` if the program defines one and calls it), go top to bottom one statement at a time, and write down each value as it changes.
- ★ Watch the types before the values. In Python 3, `/` is *always* real (float) division, while `//` is floor division and `%` is remainder. So `1 / 2` is `0.5` but `1 // 2` is `0`.
- ★ Indentation *is* the syntax. A block is defined purely by its indentation level, not by braces. A stray space or a mix of tabs and spaces is a real error (`IndentationError/TabError`), not a cosmetic one.
- ★ `input()` always returns a **string**. If you need a number you must convert it with `eval()`, `int()`, or `float()` first, or `"2" + "3"` surprises will follow.
- ★ Most "wrong answer" questions are really off-by-one or short-circuit questions in disguise, so count loop iterations carefully and stop reading **and/or** the instant the result is decided.
- ★ List indices are 0-based; `lst[-1]` is the last element. Going past the end raises `IndexError` at run time, it does not silently return garbage.
- ★ Python is case sensitive and there is no statement terminator: one statement per line, and the newline ends it. A backslash `\` or an open bracket lets a statement continue onto the next line.
- ★ Numbers and strings are **immutable** (passed effectively by value); lists, sets, and dictionaries are **mutable** (passed by sharing). If a function changes a list argument and you see it outside, that is why.

CHAPTER 2 Elementary Programming

البرمجة الأساسية

02

★ How a Python program actually runs

الفكرة: بايثون تُنفذ سطرًا بـ سطر بواسطة مُفسّر. حدّد نوع كل قيمة أولاً، ثم احسب من الداخل للخارج.

The thing that made Python click for me is that there is no separate "compile then run" step I can lean on: an **interpreter** reads my `.py` file top to bottom and executes each statement the moment it reaches it. So the recipe for tracing any program is exactly the reading order.

1. **Read a statement, run it, move on.** Names must exist *before* the line that uses them, because the interpreter has not seen anything below the current line yet.
2. **Every value has a type.** 20 is an `int`, 3.14159 is a `float`, "radius" is a `str`. The type decides which version of an operator runs.
3. **An assignment `x = expr`** first evaluates the whole right-hand side to one value, then binds the name `x` to it. Reading a `print(...)` argument list works the same way: evaluate each piece, then display.
4. **Errors have a stage too.** A `SyntaxError` stops the program before anything runs; a `NameError`, `TypeError`, or `ZeroDivisionError` happens *during* execution at the exact line that misbehaved.

Here is the kind of margin trace I scribble for `radius = 20; area = radius * radius * 3.14159`:

Step	Why	Result
<code>radius = 20</code>	bind name to an int	<code>radius</code> → 20
<code>radius * radius</code>	both int	400
<code>400 * 3.14159</code>	one float pulls it to float	1256.636
store in <code>area</code>	bind name to the float	<code>area</code> → 1256.636

- ★ **Introducing programming with an example (ComputeArea):** A first program usually performs a **simple computation**. The classic example computes the area of a circle from a radius. It shows the three pieces every beginner program has: assign a value, compute with it, then display the result.

```

1 # Assign a radius
2 radius = 20 # radius is now 20
3 # Compute area
4 area = radius * radius * 3.14159
5 # Display results
6 print("The area for the circle of radius", radius, "is", area)
```

Output: The area for the circle of radius 20 is 1256.636

- ★ **Tracing a program's execution:** A method of reviewing how a program works by writing down the new value of each variable after each line runs. Line 2 assigns 20 to `radius`; line 4 computes 1256.636 into `area`; line 6 sends the message to the console. Tracing is the single most reliable way to predict output and find bugs.
- ★ **The `print` function and joining pieces:** `print` displays its arguments separated by spaces. If instead you build *one* string with `+`, every piece must already be a string, so numbers must be converted with `str()` first:

```
1 print("The area for the circle of radius " + str(radius) + " is  
  " + str(area))
```

Key distinction: `print(a, b)` inserts spaces and converts automatically; `"..." + str(a)` + `"..."` concatenates strings and requires manual `str()`.

★ **Reading input from the console:** Interactive programs read data typed by the user.

- **input function:** `variable = input("Enter a string: ")` shows the prompt, waits for the user to type and press *Enter*, and returns what was typed **as a string**.
- **eval function:** `value = eval(stringVariable)` evaluates a string as if it were a Python expression. `eval("51 + (54 * (3 + 2))")` returns 321. This is how early programs turn typed text into numbers.

★ **Reading numbers:** Because `input` yields a string, wrap it in `eval` to get a number:

```
1 radius = eval(input("Enter a value for radius: "))  
2 area = radius * radius * 3.14159  
3 print("The area for the circle of radius", radius, "is", area)
```

★ **Reading several numbers at once:** `eval(input(...))` can evaluate a comma-separated list, which produces a tuple you unpack into several variables (this is the **simultaneous assignment** idea used early):

```
1 number1 = eval(input("Enter the first number: "))  
2 number2 = eval(input("Enter the second number: "))  
3 number3 = eval(input("Enter the third number: "))  
4 average = (number1 + number2 + number3) / 3  
5 print("The average of", number1, number2, number3, "is", average  
  )
```

Caution: `eval` on raw user input runs whatever expression the user typed; it is convenient for a course but unsafe for real programs. Prefer `int(...)` or `float(...)` in production code.

★ **Identifiers:** An identifier is the name of a program element such as a variable or function.

- An identifier is a sequence of letters, digits, and underscores (`_`).
- It **must start with a letter or an underscore**; it **cannot start with a digit**.
- It **cannot be a reserved word** (keyword). Reserved words have a special meaning to Python.
- It can be of **any length**, and Python is **case sensitive** (`area`, `Area`, and `AREA` are different).

★ **Python keywords (reserved words):** include `and`, `as`, `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `False`, `finally`, `for`, `from`, `global`, `if`, `import`, `in`, `is`, `lambda`, `None`, `nonlocal`, `not`, `or`, `pass`, `raise`, `return`, `True`, `try`, `while`, `with`, `yield`. These cannot be used as identifiers.

★ **Variables and assignment:** A variable references a value; the assignment operator `=` binds a name to the value of an expression. The syntax is `variable = expression`.

```

1 x = 1                # Assign 1 to variable x
2 radius = 1.0         # Assign 1.0 to variable radius
3 x = 5 * (3 / 2) + 3 * 2 # Assign the value of the expression
                        # to x
4 x = y + 1            # Assign the addition of y and 1 to x
5 area = radius * radius * 3.14159 # Compute area

```

Caution: the target of an assignment must be a variable on the *left*. Writing `1 = x` is wrong. A variable must be assigned before it is used, otherwise `NameError`.

- ★ **Reassigning and chaining:** A variable can be reassigned (`x = x + 1` reads the old value, adds one, stores it back). You can assign the same value to several variables at once with `i = j = k = 1`.
- ★ **Simultaneous assignment:** `var1, var2, ..., varn = exp1, exp2, ..., expn` evaluates *all* right-hand expressions first, then assigns them in order. This is why swapping two variables needs **no** temporary variable in Python:

```

1 # Common approach in other languages: use a temp
2 temp = x
3 x = y
4 y = temp
5 # Pythonic swap with simultaneous assignment:
6 x, y = y, x

```

- ★ **Named constants:** A variable's value may change during execution, but a **named constant** represents permanent data that never changes.

- Python has **no special syntax** for constants; you simply create an ordinary variable.
- **Convention:** to distinguish a constant from a variable, name it in **all uppercase** letters (e.g., `PI = 3.14159`).

Benefits: you avoid retyping the same value, you can change it in one place, and a descriptive name makes the program easier to read.

- ★ **Numeric data types:** Python's two core numeric types are:

- **Integers** (`int`): whole numbers such as 3, 4, -100.
- **Floating-point** (`float`): numbers with a decimal point such as 3.0, 4.0, -0.5.

Numeric operators:

Name	Meaning	Example	Result
+	Addition	<code>34 + 1</code>	35
-	Subtraction	<code>34.0 - 0.1</code>	33.9
*	Multiplication	<code>300 * 30</code>	9000
/	Float division	<code>1 / 2</code>	0.5
//	Integer (floor) division	<code>1 // 2</code>	0
**	Exponentiation	<code>4 ** 0.5</code>	2.0
%	Remainder (modulo)	<code>20 % 3</code>	2

Key: in Python 3, `/` always yields a `float` even for `4 / 2` (which is 2.0); use `//` when you want an integer result.

★ **Floor division (//) and the floor function:** Integer division with // gives the **floor** of the true quotient, i.e. the largest integer *not greater than* the result. This is exactly `math.floor` applied to the division.

- With positive numbers it behaves as expected: `37 // 5` is 7 (since `37 / 5` is 7 remainder 2).
- **The trap:** with a negative operand it rounds *down*, not toward zero: `37 // -5` is -8, *not* -7.

Reference points: `math.floor(1.95)` is 1, `math.floor(1.0)` is 1, `math.floor(-1.95)` is -2, `math.floor(-2.05)` is -3. The floor is the integer “underneath” the number unless the number is already whole.

★ **The modulo operator (%):** gives the remainder. It is very useful in programming.

- An **even** number % 2 is always 0; an **odd** number % 2 is always 1. So % tells you whether a value is even or odd.
- **Day-of-week example:** if today is Saturday (day 1) and you meet in 10 days, the day is `(1 + 10) % 7 == 4`, i.e. the 4th day, Tuesday.
- **Negative values:** `37 % -5` is -3, because Python defines the sign of % to match the *divisor* and keeps the identity below consistent with floor division.

The identity that ties them together: for any `x`, `y`: `x == (x // y) * y + (x % y)`. Since `37 // -5` is -8, we get `-5 * -8 == 40`, and to reach 37 we must add -3, which is exactly `37 % -5`.

★ **Overflow and underflow:**

- **Overflow** happens when a *float* value is too large in size to store; e.g., `245.0 ** 1000` raises `OverflowError: (34, 'Result too large')`. (Python ints, by contrast, grow arbitrarily large and do not overflow.)
- **Underflow** happens when a floating-point number is too small (too close to zero) to store; Python approximates it to 0. You normally need not worry about underflow.

★ **Scientific notation:** Floating-point literals may be written in scientific notation using `e` or `E` for the exponent (either case). `1.23456e+2` (same as `1.23456e2`) equals 123.456, and `1.23456e-2` equals 0.0123456.

★ **Evaluating arithmetic expressions:** A Python expression is evaluated using the same precedence rules as ordinary arithmetic. To translate a fraction, put the whole numerator over the whole denominator using parentheses. For instance

$$\frac{3+4x}{5} - \frac{10(y-5)(a+b+c)}{x} + 9\left(\frac{4}{x} + \frac{9+x}{y}\right)$$

becomes `(3 + 4 * x) / 5 - 10 * (y - 5) * (a + b + c) / x + 9 * (4 / x + (9 + x) / y)`.

★ **Evaluation order example:** `3 + 4 * 4 + 5 * (4 + 3) - 1` evaluates as: parentheses first (7), then the two multiplications (16 and 35), then additions and subtraction left to right, giving 53.

★ **Augmented assignment operators:** shorthand that combine an operator with assignment. There are **no spaces** inside them.

Operator	Name	Example	Equivalent
<code>+=</code>	Addition assignment	<code>i += 8</code>	<code>i = i + 8</code>
<code>-=</code>	Subtraction assignment	<code>i -= 8</code>	<code>i = i - 8</code>
<code>*=</code>	Multiplication assignment	<code>i *= 8</code>	<code>i = i * 8</code>
<code>/=</code>	Float division assignment	<code>i /= 8</code>	<code>i = i / 8</code>
<code>//=</code>	Integer division assignment	<code>i //= 8</code>	<code>i = i // 8</code>
<code>%=</code>	Remainder assignment	<code>i %= 8</code>	<code>i = i % 8</code>
<code>**=</code>	Exponent assignment	<code>i **= 8</code>	<code>i = i ** 8</code>

Caution: `x op= e` means `x = x op (e)` where the whole right side is one unit, so `x *= a + b` is `x = x * (a + b)`, not `x * a + b`.

★ **Type conversion and rounding:** The general form is `datatype(value)`.

- `int(4.5)` is 4 (truncates toward zero, no rounding), `float(4)` is 4.0, `str(4)` is "4".
- `round(4.6)` is 5, and `round(4.5)` is 4 (Python rounds half to the nearest *even* value, "banker's rounding").

int vs. eval on strings: both can turn "45" into the integer 45, but they differ on several inputs:

Situation	<code>int("...")</code>	<code>eval("...")</code>
Integer string "4"	4	4
Non-integer string "4.3"	error	4.3
Expression string "4 + 3"	error	7
Leading zeros "004"	4	error

★ **Problem: keeping two digits after the decimal point (sales tax).** Compute 6% sales tax and display it with exactly two decimals. The trick is to scale by 100, truncate with `int`, then divide back:

```

1 purchaseAmount = eval(input("Enter purchase amount: "))
2 tax = purchaseAmount * 0.06
3 print("Sales tax is", int(tax * 100) / 100.0)
```

For `purchaseAmount = 197.55`: `tax * 100` is 1185.3, `int(tax * 100)` is 1185, and `int(tax * 100) / 100.0` is 11.85.

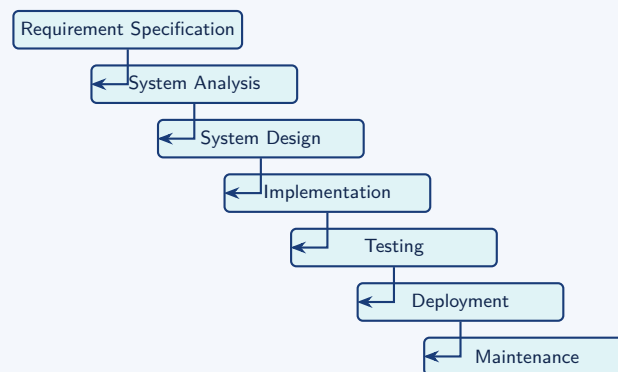
★ **Obtaining the current time with `time.time()`:** `time.time()` returns the current time in **seconds** (with millisecond precision) elapsed since the **Unix epoch**, midnight January 1, 1970 GMT. From this single number you compute the current second, minute, and hour using `//` and `%`:

```

1 import time
2 currentTime = time.time()           # Get current time in
   seconds
3 totalSeconds = int(currentTime)     # Whole seconds since
   the epoch
4 currentSecond = totalSeconds % 60   # Seconds of the
   current minute
5 totalMinutes = totalSeconds // 60
6 currentMinute = totalMinutes % 60   # Minutes of the
   current hour
7 totalHours = totalMinutes // 60
8 currentHour = totalHours % 24       # Hours of the
   current day
9 print("Current time is", currentHour, ":", currentMinute, ":",
   currentSecond, "GMT")

```

The software development process: building software is a sequence of stages, often drawn as a waterfall where each stage feeds the next. The essence of analysis and design is **Input, Process, Output (IPO)**.



- ★ **Requirement Specification:** a formal process to understand and document *what* the software must do; close interaction between users and developers.
- ★ **System Analysis:** analyze the business process in terms of data flow and identify the system's input and output.
- ★ **System Design:** design the components; use levels of abstraction to decompose the problem into manageable parts.
- ★ **Implementation:** translate the design into programs written in a language like Python; involves coding, testing, and debugging.
- ★ **Testing:** ensure the code meets the requirement specification and weed out bugs, usually by an independent team.
- ★ **Deployment:** make the product available for use.
- ★ **Maintenance:** fix newly discovered bugs and incorporate changes so the product keeps working in a changing environment.

- ★ **Problem: computing loan payments.** The user enters the annual interest rate, number of years, and loan amount; the program computes the monthly and total payment. The monthly

payment formula is

$$\text{monthlyPayment} = \frac{\text{loanAmount} \times \text{monthlyInterestRate}}{1 - \frac{1}{(1 + \text{monthlyInterestRate})^{\text{numberOfYears} \times 12}}}$$

and `totalPayment = monthlyPayment * numberOfYears * 12`. The annual rate entered as a percentage is turned into a monthly rate with `annualInterestRate / 1200`.

```
1 annualInterestRate = eval(input("Enter annual interest rate, e.g.
  ., 7.25: "))
2 monthlyInterestRate = annualInterestRate / 1200
3 numberOfYears = eval(input("Enter number of years as an integer,
  e.g., 5: "))
4 loanAmount = eval(input("Enter loan amount, e.g., 120000.95: "))
5 monthlyPayment = loanAmount * monthlyInterestRate / (1 - 1 / (1
  + monthlyInterestRate) ** (numberOfYears * 12))
6 totalPayment = monthlyPayment * numberOfYears * 12
7 print("The monthly payment is", int(monthlyPayment * 100) / 100)
8 print("The total payment is", int(totalPayment * 100) / 100)
```

★ **Case study: computing distances.** Given two points, the distance is $\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$. In Python you compute a square root with the `** 0.5` power operator:

```
1 x1, y1 = eval(input("Enter x1 and y1 for Point 1: "))
2 x2, y2 = eval(input("Enter x2 and y2 for Point 2: "))
3 distance = ((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)) **
  0.5
4 print("The distance between the two points is", distance)
```

Code Output & Identification Tasks

★ Show the output of the following code:

```
1 print(3.5 * 4 / 2 - 2.5)
2 print(1 / 2, 1 // 2, 5 % 2, -7 // 2)
```

Output:

4.5

0.5 0 1 -4 (note `-7 // 2` floors to -4, not -3)

★ What is the difference between these two lines?

```
1 print("radius", radius, "area", area)
2 print("radius " + str(radius) + " area " + str(area))
```

Answer: the first uses `print`'s comma form, which inserts spaces and converts numbers automatically. The second builds a single string with `+`, so every non-string must be wrapped in `str()` or a `TypeError` occurs.

★ Identify and fix the errors:

```
1 radius = input("Enter radius: ")
2 area = radius * radius * 3.14159
3 print("Area is " + area)
```

Fixes:

- **Line 1:** `input` returns a *string*; multiplying a string by itself on line 2 fails. Wrap it: `radius = eval(input("Enter radius: "))`.
- **Line 3:** you cannot concatenate a string with a number; use `print("Area is", area)` or `"Area is " + str(area)`.

Note 1: Integer division by zero (`5 // 0` or `5 % 0`) raises `ZeroDivisionError` and stops the program.

Note 2: In Python 3, `/` between two integers still gives a `float` (`6 / 3` is `3.0`); use `//` for an integer.

Note 3: Operator precedence inside parentheses always dictates execution order, innermost first.

CHAPTER 3 Mathematical Functions, Strings, and Objects

الدوال الرياضية والسلاسل النصية والكائنات

★ How functions, characters, and strings behave

كل قيمة في بايثون كائن؛ والحرف ليس نوعاً مستقلاً بل سلسلة نصية طولها واحد. افهم ذلك تحتفِ المفاجآت.

Two ideas unlock this chapter. First, Python has a big **standard library**, and `math` is one module in it; `import math` makes its functions available as `math.name(...)`. Second, Python has **no separate character type**: a single character is just a **string of length one**. That is why `ord` and `chr` bridge between a one-character string and its numeric code.

- ▷ **Every datum is an object.** A number is an object, a string is an object. `type(x)` tells you its class and `id(x)` its identity. Operations you can call *on* an object with the dot are its **methods**.
- ▷ **Characters live on a code scale.** 'a' is code 97, 'A' is 65, '0' is 48. Because letters are consecutive, `ord(ch) - ord('a')` turns 'a'..'z' into 0..25 — the workhorse trick for counting letters.
- ▷ **Formatting is its own mini-language.** `format(item, spec)` lays a value into a field of a chosen width and precision without changing the value itself.

★ **Built-in functions:** Python provides functions that are always available without importing anything. Common ones from this chapter:

- `max(2, 3, 4)` returns 4; `min(2, 3, 4)` returns 2.
- `round(3.51)` returns 4; `round(3.4)` returns 3 (rounds to the nearest integer).
- `abs(-3)` returns 3 (absolute value).
- `pow(2, 3)` returns 8 (same as `2 ** 3`).
- Others include `int`, `float`, `str`, `eval`, `input`, `print`, `ord`, `chr`, `len`, `sum`.

★ **Standard modules and the `math` module:** Python ships with a large set of **standard modules**; you get them when you download Python. To use one you **import** it. The `math` module provides mathematical functions.

Function	Description	Example
<code>fabs(x)</code>	absolute value as a float	<code>fabs(-2)</code> is 2.0
<code>ceil(x)</code>	rounds up to nearest integer	<code>ceil(2.1)</code> is 3
<code>floor(x)</code>	rounds down to nearest integer	<code>floor(2.1)</code> is 2
<code>exp(x)</code>	e^x	<code>exp(1)</code> is 2.71828
<code>log(x)</code>	natural logarithm of x	<code>log(2.71828)</code> is 1.0
<code>log(x, base)</code>	logarithm of x to given base	<code>log(100, 10)</code> is 2.0
<code>sqrt(x)</code>	square root	<code>sqrt(4.0)</code> is 2
<code>sin(x)</code>	sine (x in radians)	<code>sin(3.14159/2)</code> is 1
<code>cos(x)</code>	cosine (x in radians)	<code>cos(3.14159/2)</code> is 0
<code>degrees(x)</code>	radians $\rightarrow$ degrees	<code>degrees(1.57)</code> is 90
<code>radians(x)</code>	degrees $\rightarrow$ radians	<code>radians(90)</code> is 1.57

Also available: `asin`, `acos`, `tan`, `atan`, and the constants `math.pi` and `math.e`.

★ **Problem: compute angles of a triangle.** Given three points, the angles come from the law of cosines using `math.acos` and `math.sqrt` on the side lengths. It is the canonical example of importing `math` and combining several of its functions:

```

1 import math
2 x1, y1, x2, y2, x3, y3 = eval(input("Enter three points: "))
3 a = math.sqrt((x2 - x3) * (x2 - x3) + (y2 - y3) * (y2 - y3))
4 b = math.sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3))
5 c = math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
6 A = math.degrees(math.acos((a * a - b * b - c * c) / (-2 * b * c
7 )))
8 B = math.degrees(math.acos((b * b - a * a - c * c) / (-2 * a * c
9 )))
10 C = math.degrees(math.acos((c * c - b * b - a * a) / (-2 * a * b
11 )))
12 print("The three angles are", round(A * 100) / 100.0, round(B *
13 100) / 100.0, round(C * 100) / 100.0)

```

- ★ **Strings and characters:** A **string** is a sequence of characters. String literals are enclosed in matching **single quotes** (') or **double quotes** ("). Python has **no separate character data type**; a single-character string *is* a character.

```

1 letter = 'A'           # Same as letter = "A"
2 numChar = '4'          # Same as numChar = "4"
3 message = "Good morning" # Same as message = 'Good morning'

```

Convention (from the book): use double quotes for strings with more than one character and single quotes for single-character or empty strings, to match how other programming languages treat characters.

- ★ **Unicode and ASCII:** Python characters use **Unicode**, an encoding scheme for representing international characters. **ASCII** is a small subset of Unicode, covering codes \u0000 to \u007f. You can put a Unicode character in a string with a \u escape followed by four hex digits (e.g., "\u03b1" is the Greek letter alpha).

- ★ **Functions ord and chr:** bridge a character and its numeric code.

- ord(ch) returns the code of a character: ord('a') is 97.
- chr(code) returns the character for a code: chr(98) is 'b'.

- ★ **Escape sequences for special characters:** a backslash followed by a letter represents a character that is otherwise hard to type.

Escape	Name	Numeric	Unicode
\b	Backspace	8	\u0008
\t	Tab	9	\u0009
\n	Linefeed (newline)	10	\u000A
\f	Formfeed	12	\u000C
\r	Carriage Return	13	\u000D
\\	Backslash	92	\u005C
\'	Single Quote	39	\u0027
\"	Double Quote	34	\u0022

- ★ **Printing without a newline (the end argument):** By default print ends its output with a newline. Passing end='...' replaces that ending string, letting you keep printing on the same line:

```

1 print("AAA", end = ' ')
2 print("BBB", end = ' ')
3 print("CCC", end = '***')
4 print("DDD", end = '***')

```

displays: AAA BBBCCC***DDD***

- ★ **The str function:** converts a number into a string. `str(3.4)` is '3.4' and `str(3)` is '3'. This is needed before joining a number to a string with +.
- ★ **The string concatenation operator (+):** joins two strings. "Welcome " + "to " + "Python" is 'Welcome to Python'. To join a number you first convert it: `chapterNo = 2`, "Chapter " + `str(chapterNo)` is 'Chapter 2'.
- ★ **Reading strings from the console:** use `input` without `eval` to read a plain string:

```

1 s1 = input("Enter a string: ")
2 s2 = input("Enter a string: ")
3 print("s1 is " + s1)
4 print("s2 is " + s2)

```

- ★ **Case study: minimum number of coins.** The user enters an amount in dollars and cents; the program reports the fewest dollars, quarters, dimes, nickels, and pennies that make it (4 quarters, 10 dimes, 20 nickels, 100 pennies per dollar). It is the standard `//` and `%` decomposition: convert to cents, then peel off each denomination from largest to smallest.

```

1 amount = eval(input("Enter an amount, for example, 11.56: "))
2 remainingAmount = int(amount * 100)           # convert to cents
3 numberOfOneDollars = remainingAmount // 100   # dollars
4 remainingAmount = remainingAmount % 100
5 numberOfQuarters = remainingAmount // 25      # quarters
6 remainingAmount = remainingAmount % 25
7 numberOfDimes = remainingAmount // 10         # dimes
8 remainingAmount = remainingAmount % 10
9 numberOfNickels = remainingAmount // 5        # nickels
10 remainingAmount = remainingAmount % 5
11 numberOfPennies = remainingAmount            # pennies

```

For 11.56 the report is: 11 dollars, 2 quarters, 0 dimes, 1 nickel, 1 penny.

- ★ **Introduction to objects and methods:** In Python **everything is an object**. Objects of the same kind have the same **type**.

- `id(object)` returns the object's unique identity (an address-like number).
- `type(object)` returns the object's type: `type(3)` shows `int`, `type(3.0)` shows `float`, `type("Welcome")` shows `str`.

These two are rarely used in real programs but are good pedagogical tools for understanding objects.

- ★ **Methods:** operations you perform on an object are called **methods**, and they are invoked only *from* a specific object using the dot. For example, the `str` type has methods `lower()` and `upper()`:

```

1 s = "Welcome"
2 s1 = s.lower()    # Invoke the lower method -> 'welcome'
3 s2 = s.upper()    # Invoke the upper method -> 'WELCOME'

```

Another useful one is `strip()`, which removes whitespace from both ends: `"\t Welcome\n".strip()` is `'Welcome'`.

- ★ **Formatting numbers and strings:** `format(item, format-specifier)` returns a string that lays `item` into a field of a given width and precision. It is often desirable to display numbers in a fixed format, e.g. an interest amount to two decimals: `format(interest, ".2f")`.
- ★ **Formatting floating-point numbers:** the specifier `"10.2f"` means **field width 10, precision 2**, conversion code `f` (fixed-point). The decimal point counts as one of the ten spaces. Writing for a blank:

```

1 print(format(57.467657, "10.2f"))    # -> "    57.47"
2 print(format(12345678.923, "10.2f")) # -> "12345678.92"
3 print(format(57.4, "10.2f"))         # -> "    57.40"
4 print(format(57, "10.2f"))           # -> "    57.00"

```

- ★ **Formatting in scientific notation:** change the conversion code from `f` to `e`: `format(57.467657, "10.2e")` gives `" 5.75e+01"`.
- ★ **Formatting as a percentage:** conversion code `%` multiplies by 100 and appends a percent sign: `format(0.53457, "10.2%")` gives `" 53.46%"`.
- ★ **Justifying format:** by default numbers are **right justified**. Put a `<` in the specifier to **left justify**: `format(57.467657, "<10.2f")` gives `"57.47"` padded on the right.
- ★ **Formatting integers:** conversion codes `d`, `x`, `o`, `b` format an integer in decimal, hexadecimal, octal, or binary, with an optional width. `format(59832, "10d")` is decimal; `format(59832, "10x")` is hexadecimal `e9b8`.
- ★ **Formatting strings:** conversion code `s` formats a string in a field of a given width. `format("Welcome to Python", ">20s")` right-justifies it in a 20-wide field; `"<20s"` left-justifies. If the string is longer than the width, the full string is printed unchanged.

CHAPTER 4 Selections

الجميل الشرطية

04

★ How the branching logic runs

الشرط تعبير منطقي قيمته *True* أو *False*. اقرأه، ثم ادخل الفرع الذي تحدده قيمته فقط.

Selection is just "compute a Boolean, then take one road." The reliable trace is: evaluate the condition to *True* or *False*, then execute *only* the matching branch and skip the rest.

- ▷ **Indentation marks the branch.** The body of an *if* is exactly the indented block beneath it. Dropping the indentation (or the colon *:*) is a real syntax error, not a style nit.
- ▷ ***elif* chains are mutually exclusive.** In an *if* / *elif* / *else* ladder, Python tests conditions top to bottom and runs the body of the **first** one that is *True*, then exits the whole statement. Order matters: put the tighter condition first.
- ▷ **Short-circuit and / or.** *a* and *b* stops at *a* if *a* is false; *a* or *b* stops at *a* if *a* is true. This is both an efficiency rule and a safety rule (it lets you guard a risky test).

★ **Boolean data type:** often you need to compare two values. The result of a comparison is a **Boolean** value, *True* or *False*. Example: *b = (1 > 2)* assigns *False* to *b*.

★ **Comparison (relational) operators:** there are six.

Operator	Name
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal to
!=	not equal to

Caution: *==* tests equality; a single *=* is assignment. Mixing them up is a classic bug.

★ **Problem: a simple math learning tool (AdditionQuiz).** Generate two random single-digit integers, ask the student to add them, and report whether the answer is right. It uses *random* and a Boolean comparison:

```
1 import random
2 number1 = random.randint(0, 9)           # a random single digit
3 number2 = random.randint(0, 9)
4 answer = eval(input("What is " + str(number1) + " + " + str(
    number2) + "? "))
5 print(number1, "+", number2, "=", answer, "is", number1 +
    number2 == answer)
```

★ **Generating random numbers:** the *random* module provides:

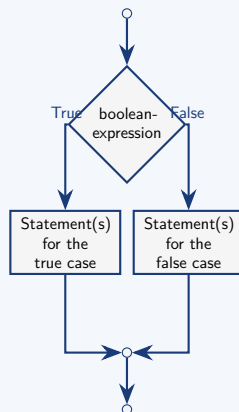
- *random.randint(a, b)* returns a random integer between *a* and *b* **inclusive**.
- *random.random()* returns a random float in *[0.0, 1.0)*.

One-way and two-way if statements. ★ A **one-way if** executes its indented body only if the condition is True; otherwise nothing happens.

```
1 if radius >= 0:
2     area = radius * radius * 3.14159
3     print("The area for the circle of
        radius", radius, "is", area)
```

★ A **two-way if-else** runs the if body when the condition is True and the else body otherwise.

```
1 if radius >= 0:
2     area = radius * radius * 3.14159
3     print("The area for the circle of
        radius", radius, "is", area)
4 else:
5     print("Negative input")
```



Two-way if-else

★ **The colon and indentation define the block.** Unlike braces in other languages, Python uses a trailing colon on the header line and a consistent indentation for the body.

★ **Simple if demo:** if `number % 5 == 0`: then print "HiFive"; if `number % 2 == 0`: then print "HiEven". For input 30 both fire (HiFive, then HiEven); for 4 only HiEven.

★ **Note (common trap):** the body must be indented. if `i > 0`: on its own line *then* `print(...)` indented is correct; putting the print at the same indentation as the if is wrong.

★ **Nested if and multi-way if-elif-else:** An if can be placed inside another if to form a **nested if** (there is no limit to the nesting depth). Nested ifs form a decision tree but grow hard to read. A chain of **elif** is the flattened, preferred form when the conditions are mutually exclusive options.

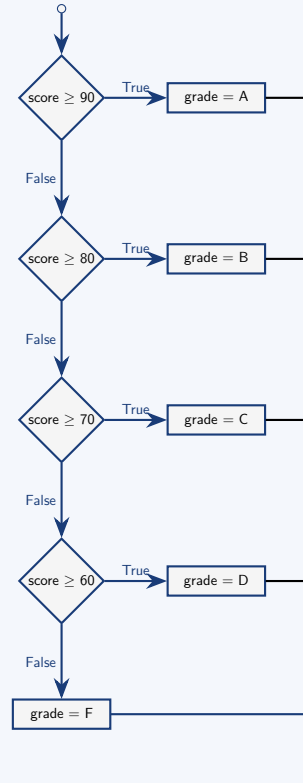
★ **Guessing Birthday / Chinese Zodiac** are classic selection case studies: the birthday trick adds set values based on yes/no answers, and the Zodiac uses `year % 12` to pick one of twelve animals with an **if-elif** ladder.

Preferred form: grade assignment with if-elif-else.

```

1  if score >= 90.0:
2      grade = 'A'
3  elif score >= 80.0:
4      grade = 'B'
5  elif score >= 70.0:
6      grade = 'C'
7  elif score >= 60.0:
8      grade = 'D'
9  else:
10     grade = 'F'

```



Multi-way if-elif-else

Trace (score = 70.0): score >= 90 is False, >= 80 is False, >= 70 is True ⇒ grade = 'C', then exit the whole statement.

★ Common errors and a tip:

- **Wrong indentation.** With `radius = -20`, writing the area line indented under `if radius >= 0:` is correct; leaving it un-indented makes it run *always*, a logic error.
- **Forgotten colon.** Every `if`, `elif`, `else` header ends with `:`.
- **Syntax vs. logic error.** A missing colon is a *syntax* error caught immediately; computing the wrong branch is a *logic* error that runs silently.
- **Tip:** a two-way test that only sets a Boolean, like `if number % 2 == 0: even = True` ...`else: even = False`, is shorter written directly as `even = number % 2 == 0`.

★ Logical operators: combine Boolean expressions.

- `not` — logical negation.
- `and` — logical conjunction (True iff both operands are true).
- `or` — logical disjunction (True iff at least one is true).

p	q	not p	p and q	p or q
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

- ★ **Short-circuit evaluation:** in `p1 and p2`, if `p1` is False, `p2` is not evaluated. In `p1 or p2`, if `p1` is True, `p2` is not evaluated.
- ★ **Examples (TestBooleanOperators):** for a number, "divisible by 2 *and* 3" is `number % 2 == 0 and number % 3 == 0`; "divisible by 2 *or* 3" uses `or`; "by 2 or 3 but not both" uses `(number % 2 == 0 or number % 3 == 0) and not (number % 2 == 0 and number % 3 == 0)`.
- ★ **Range trap:** do *not* write `1 <= day <= 31` carelessly in other languages, but note Python *does* support chained comparisons, so `1 <= day <= 31` is valid and means `(1 <= day) and (day <= 31)`.

- ★ **Problem: determining a leap year.** A year is a leap year if it is divisible by 4 but not by 100, or if it is divisible by 400:

```
1 year = eval(input("Enter a year: "))
2 isLeapYear = (year % 4 == 0 and year % 100 != 0) or (year % 400
  == 0)
3 print(year, "is a leap year?", isLeapYear)
```

- ★ **Problem: lottery.** Generate a random two-digit number, read the user's two-digit guess, and award prizes: exact-order match \$10,000; same digits any order \$3,000; one digit matches \$1,000. It combines `random`, digit extraction with `// 10` and `% 10`, and an `if-elif` ladder with logical operators.
- ★ **Problem: computing taxes.** Tax depends on filing status (0..3) and income bracket. The clean structure is an outer `if-elif` on status, each branch computing tax for its brackets, with a final `else` for a wrong status.

```
1 if status == 0:
2     # Compute tax for single filers
3 elif status == 1:
4     # Compute tax for married filing jointly
5 elif status == 2:
6     # Compute tax for married filing separately
7 elif status == 3:
8     # Compute tax for head of household
9 else:
10    # Display wrong status
```

- ★ **Conditional expression (ternary):** a compact one-line selection.

expression1 if boolean-expression else expression2

- `y = 1 if x > 0 else -1` sets `y` to 1 when `x > 0`, otherwise -1.
- `print("number is even" if number % 2 == 0 else "number is odd")`.

- ★ **Operator precedence (high to low):**

- `()` *parentheses*

- +var, -var	<i>unary plus and minus</i>
- **	<i>exponentiation</i>
- not	
- *, /, //, %	<i>multiplication, division, floor division, remainder</i>
- +, -	<i>binary addition and subtraction</i>
- <, <=, >, >=	<i>comparison</i>
- ==, !=	<i>equality</i>
- and	
- or	
- =, +=, -=, *=, /=, //=, %=	<i>assignment</i>

★ **Operator associativity:** when two operators have the *same* precedence, associativity decides order. **All binary operators except assignment are left-associative**, so $a - b + c - d$ is $((a - b) + c) - d$. **Assignment operators are right-associative**, so $a = b = c = 5$ binds 5 rightward first.

CHAPTER 5 Loops

الحلقات التكرارية

05

★ How a loop actually runs

الحلقة تختصر التكرار. تتبع عدّاد الدوران وشرط الاستمرار في عمودين منفصلين، وستعرف بالضبط متى تخرج.

The motivation is simple: printing a line a hundred times by hand is tedious, so a loop does the repeating for you. The reliable way to trace one is to keep two columns — the control variable and the condition — and step one full pass at a time.

- ▷ **while first tests, then runs.** It checks the continuation condition *before* each pass; if the condition is already `False` the body never runs. Something inside the body must move the condition toward `False`, or the loop never ends.
- ▷ **for walks a sequence.** `for v in range(...)` binds `v` to each value the range produces and runs the body once per value. The count is fixed the moment the range is built.
- ▷ **Off-by-one lives at the boundary.** `range(a, b)` *excludes* `b`. `while count < 100` runs for `count = 0..99`, i.e. 100 times. Always check the first and last iteration explicitly.

The `while` loop.

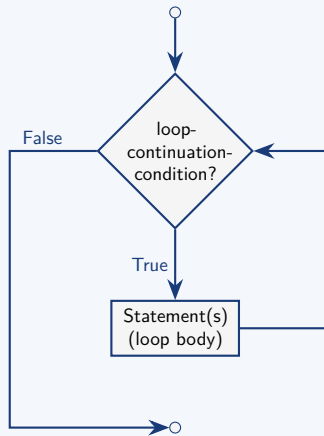
★ **Motivation:** to print a string a hundred times you would need a hundred statements. A loop replaces them with a few lines.

★ **Syntax:** the header `while loop-continuation-condition:` is followed by an indented **loop body**.

```
1 count = 0
2 while count < 100:
3     print("Programming is fun!")
4     count = count + 1
```

★ **Trace (with `count < 2`):** `count = 0` → condition true → print, `count = 1` → still true → print, `count = 2` → condition false → exit and run the next statement after the loop.

★ **Infinite loop:** if the body never changes the condition, the loop runs forever. Make sure the control variable advances.



while loop flow

- ★ **Problem: an advanced math learning tool.** A loop lets you generate several questions in one run and count the correct answers, instead of one question per run.
- ★ **Problem: guessing numbers.** Repeatedly prompt the user until the guess equals a randomly generated number, telling them each time whether the guess is too low or too high so they can narrow in.

```

1 import random
2 number = random.randint(0, 100)
3 guess = -1
4 while guess != number:
5     guess = eval(input("Enter your guess: "))
6     if guess == number:
7         print("Yes, the number is", number)
8     elif guess > number:
9         print("Your guess is too high")
10    else:
11        print("Your guess is too low")

```

- ★ **Ending a loop with a sentinel value:** often the number of iterations is not known in advance. A **sentinel value** is a special input that signals the end of the loop. Below, entering 0 ends the input:

```

1 data = eval(input("Enter an integer (the input ends if it is 0):"))
2 sum = 0
3 while data != 0:
4     sum += data
5     data = eval(input("Enter an integer (the input ends if it is 0): "))
6 print("The sum is", sum)

```

- ★ **Caution — don't use floats for equality in loop control.** Floating-point values are approximations, so a loop that expects a float to hit exactly 0 may never stop. This code is an **infinite loop** in disguise:

```

1 item = 1
2 sum = 0
3 while item != 0:      # item may never be EXACTLY 0
4     sum += item
5     item -= 0.1

```

`item` starts at 1 and drops by 0.1; because 0.1 is not exact in binary, `item` never lands precisely on 0. Use an integer counter or a `</>` test instead of `==/!=` on floats.

★ **The for loop:** implements counter-controlled loops cleanly. The general syntax is

```

1 for var in sequence:
2     # Loop body

```

A for loop over a range is the compact form of the while pattern "initialize / test / advance".

★ **The range function (three forms):**

- `range(b)` yields 0, 1, ..., b-1. E.g. `range(3)` gives 0, 1, 2.
- `range(a, b)` yields a, a+1, ..., b-1 (b excluded). E.g. `range(4, 8)` gives 4, 5, 6, 7.
- `range(a, b, step)` yields a, a+step, ... up to but not including b. `range(3, 9, 2)` gives 3, 5, 7; a negative step counts down, `range(5, 1, -1)` gives 5, 4, 3, 2.

Note: the numbers in `range` must be integers.

★ **Nested loops:** a loop inside another loop. For each pass of the outer loop, the inner loop runs completely. Classic example — a multiplication table:

```

1 print("      Multiplication Table")
2 for i in range(1, 10):
3     for j in range(1, 10):
4         print(format(i * j, "4d"), end='')    # align each
                                                product
5     print()                                    # jump to the
                                                next line

```

The inner for prints one row; the outer for repeats for every row.

★ **Minimizing numerical errors:** numeric errors with floating-point numbers are unavoidable, but you can reduce them. Summing `0.01 + 0.02 + ... + 1.0`, adding the *smaller* numbers first (or using a well-chosen loop) keeps the running error smaller. The point is to be aware that the float sum may be slightly off (e.g. 49.5 vs. an exact fraction).

★ **Problem: greatest common divisor (Euclid).** For all integers a, b: `gcd(a, b) = gcd(a % b, b)`, which shrinks the problem each step. A loop tries divisors up to the smaller number:

```

1 n1 = eval(input("Enter first integer: "))
2 n2 = eval(input("Enter second integer: "))
3 gcd = 1
4 k = 2
5 while k <= n1 and k <= n2:
6     if n1 % k == 0 and n2 % k == 0:
7         gcd = k
8         k += 1
9 print("The greatest common divisor for", n1, "and", n2, "is",
      gcd)

```

★ **Problem: predicting future tuition.** Tuition is \$10,000 and rises 7% each year; how many years until it doubles? A `while` loop multiplies and counts:

```

1 year = 0
2 tuition = 10000
3 while tuition < 20000:
4     year += 1
5     tuition = tuition * 1.07
6 print("Tuition will be doubled in", year, "years")
7 print("Tuition will be $" + format(tuition, ".2f"), "in", year,
      "years")

```

★ **Problem: Monte Carlo simulation (estimating π).** A Monte Carlo method uses random numbers and probability. To estimate π , throw random points into the square $[-1, 1] \times [-1, 1]$ (area 4) and count how many land in the unit circle (area π). Since $\text{circleArea}/\text{squareArea} = \pi/4$, we get $\pi \approx 4 \times \text{numberOfHits}/\text{trials}$:

```

1 import random
2 NUMBER_OF_TRIALS = 1000000
3 numberOfHits = 0
4 for i in range(NUMBER_OF_TRIALS):
5     x = random.random() * 2 - 1
6     y = random.random() * 2 - 1
7     if x * x + y * y <= 1:
8         numberOfHits += 1
9 pi = 4 * numberOfHits / NUMBER_OF_TRIALS
10 print("PI is", pi)

```

★ **Using `break` and `continue`:**

- `break` immediately **exits** the innermost loop.
- `continue` **skips the rest of the current iteration** and jumps to the loop's next pass.

```

1 # break example
2 sum = 0
3 number = 0
4 while number < 20:
5     number += 1
6     sum += number
7     if sum >= 100:
8         break                # leave the loop as soon as sum reaches
                               100
9
10 # continue example
11 sum = 0
12 number = 0
13 while number < 20:
14     number += 1
15     if number == 10 or number == 11:
16         continue            # skip adding 10 and 11
17     sum += number

```

The `break` version stops early (sum 105 at number 14); the `continue` version skips two values (sum 189).

- ★ **Guessing number revisited:** the same guessing game reads much more cleanly with `while True:` and a `break` on the correct guess, rather than a sentinel condition in the header.

- ★ **Problem: displaying prime numbers.** Print the first 50 primes, 10 per line. An integer greater than 1 is **prime** if its only positive divisors are 1 and itself (so 2, 3, 5, 7 are prime; 4, 6, 8, 9 are not). The solution decomposes into: for each candidate number, test primality by trying divisors; count primes; print 10 per line.

```

1 NUMBER_OF_PRIMES = 50
2 NUMBER_OF_PRIMES_PER_LINE = 10
3 count = 0
4 number = 2
5 while count < NUMBER_OF_PRIMES:
6     isPrime = True                # assume prime until shown
                                   otherwise
7     divisor = 2
8     while divisor <= number / 2:
9         if number % divisor == 0:
10             isPrime = False      # found a factor -> not prime
11             break                # no need to keep testing
12     divisor += 1
13     if isPrime:
14         count += 1
15         print(format(number, "5d"), end='')
16         if count % NUMBER_OF_PRIMES_PER_LINE == 0:
17             print()              # 10 per line
18     number += 1

```

CHAPTER 6 Functions

الدوال

06

★ How a function call actually works

الاستدعاء يقفز إلى جسم الدالة بقيم المعاملات، ثم يعود بالقيمة إلى مكان النداء. تتبع القفزة والعودة.

A function packages statements under a name so you can reuse them. When I trace a call I picture a jump: control leaves the calling line, the **arguments** are copied into the **parameters**, the body runs, and **return** hands a value back to exactly the spot that called.

- ▷ **Definition vs. call.** `def name(params):` *defines* the recipe but runs nothing. Nothing happens until a later line *calls* `name(args)`.
- ▷ **Arguments flow into parameters left to right.** `k = max(i, j)` copies `i` into `num1` and `j` into `num2`, runs `max`, and the returned value replaces `max(i, j)` so it can be stored in `k`.
- ▷ **Return ends the function.** The first `return` that executes stops the function and sends its value back. A function with no `return` value hands back the special value `None`.

★ **Opening problem — why functions:** to sum 1..10, 20..37, and 35..49 you would repeat almost identical loops three times. A single function `sum(i1, i2)` removes the duplication:

```

1 def sum(i1, i2):
2     result = 0
3     for i in range(i1, i2 + 1):
4         result += i
5     return result
6
7 def main():
8     print("Sum from 1 to 10 is", sum(1, 10))
9     print("Sum from 20 to 37 is", sum(20, 37))
10    print("Sum from 35 to 49 is", sum(35, 49))
11
12 main()    # Call the main function

```

★ **Defining a function:** a function is a collection of statements grouped to perform an operation. Its parts:

- **Function header:** `def functionName(formal parameters):`.
- **Formal parameters:** the names listed in the header.
- **Function body:** the indented statements, ending (for a value-returning function) with a `return`.

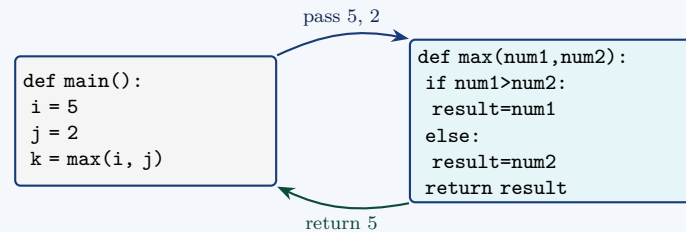
```

1 def max(num1, num2):    # function header; num1, num2 are formal
2     if num1 > num2:      parameters
3         result = num1
4     else:
5         result = num2
6     return result        # return value

```

★ **Calling a function:** `z = max(x, y)` — here `x` and `y` are the **actual parameters (arguments)**. A value-returning function is used wherever a value is expected.

Tracing a function invocation for `k = max(i, j)` with `i = 5, j = 2`:



1. `main` runs; `i` becomes 5, `j` becomes 2.
2. `max(i, j)` copies `i`→`num1` (5) and `j`→`num2` (2), and control jumps into `max`.
3. `num1 > num2` is true, so `result` is 5; `return result` sends 5 back.
4. The returned 5 replaces `max(i, j)` and is stored in `k`; then `main` prints it.

A **call stack** tracks these jumps: each active function gets a frame holding its variables; when a function returns, its frame is popped.

★ **Functions with and without return values:** some functions compute and **return** a value; others just perform an action (e.g. printing) and return nothing.

```

1  def printGrade(score):           # performs an action, returns
    nothing
2      if score >= 90.0:
3          print('A')
4      elif score >= 80.0:
5          print('B')
6      else:
7          print('C')
8
9  def getGrade(score):             # returns a value
10     if score >= 90.0:
11         return 'A'
12     elif score >= 80.0:
13         return 'B'
14     else:
15         return 'C'
  
```

★ **The None value:** a function that does not return a value is a **void** function in other languages; in Python it returns the special value `None`. `None` means "no object." A function whose body never hits a `return` value gives `None`:

```

1  def sum(number1, number2):
2      total = number1 + number2    # computes but never returns
3
4  print(sum(1, 2))                 # prints None
  
```

★ Passing arguments by position and by keyword:

- **Positional arguments:** matched to parameters by order. For `def nPrintln(message, n):`, the call `nPrintln("Welcome to Python", 5)` prints the message 5 times. Order matters: `nPrintln(5, "Welcome")` would be wrong.
- **Keyword arguments:** matched by name, so order is free: `nPrintln(n=4, message="Good")`.
- **Mixing:** positional arguments must come *before* keyword arguments. `nPrintln("Good", n=4)` is fine; `nPrintln(message="Good", 4)` is a syntax error.

★ Pass by value (and by sharing): Python passes the *reference value* of the argument to the parameter — described simply as passing the value.

- **Immutable arguments (numbers, strings):** the original is **not** affected by changes to the parameter inside the function.
- **Mutable arguments (lists):** the parameter refers to the *same* list, so changes made inside the function are visible outside — this is **pass by sharing**.

```

1 def m(number, numbers):
2     number = 1001          # rebinds a local name only
3     numbers[0] = 5555      # mutates the shared list
4
5 x = 1
6 y = [1, 2, 3]
7 m(x, y)
8 print("x is", x)          # x is 1 (unchanged)
9 print("y[0] is", y[0])    # y[0] is 5555 (changed)

```

★ Modularizing code: functions reduce redundant code and enable reuse; they also **modularize** a program, improving its quality. Putting the GCD logic in its own `gcd(n1, n2)` function and importing it (`from GCDFunction import gcd`) gives three benefits:

1. It **isolates** the GCD computation from the rest of the program, so the logic is clear and the program is easier to read.
2. Any errors in computing the GCD are **confined** to the `gcd` function, narrowing debugging.
3. The `gcd` function can be **reused** by other programs.

Note: if you define two functions with the same name in one module, there is no syntax error, but the *later* definition prevails.

★ Subtle issue — default arguments created once: a mutable default value (like `[]`) is created **only once**, when the function is defined, and is *shared* across calls that use the default. This surprises many beginners:

```

1 def add(x, lst = []):
2     if not (x in lst):
3         lst.append(x)
4     return lst
5
6 print(add(1))                # [1]
7 print(add(2))                # [1, 2]  <- same default list
8                               reused!
9 print(add(3, [11, 12, 13]))  # [11, 12, 13, 3]
10 print(add(4))               # [1, 2, 4]

```

The safe idiom uses `None` as the default and creates a fresh list inside: `def add(x, lst=None):`
`if lst is None: lst = []`.

- ★ **Problem: converting decimals to hexadecimals.** A function turns a decimal integer into its hex string by repeatedly taking `% 16` for the next digit and `// 16` to shrink the number, mapping 10..15 to A..F:

```

1 def decimalToHex(decimalValue):
2     hex = ""
3     while decimalValue != 0:
4         hexValue = decimalValue % 16
5         hex = toHexChar(hexValue) + hex
6         decimalValue = decimalValue // 16
7     return hex
8
9 def toHexChar(hexValue):
10    if 0 <= hexValue <= 9:
11        return chr(hexValue + ord('0'))
12    else: # 10 <= hexValue <= 15
13        return chr(hexValue - 10 + ord('A'))

```

For 1234 this returns "4D2".

- ★ **Scope of variables:** the scope is the part of the program where a variable can be referenced.

- A variable created inside a function is a **local variable**; its scope runs from its creation to the end of that function, and it cannot be used outside.
- A variable created outside all functions is a **global variable**, accessible to all functions in its scope.

```

1 globalVar = 1
2 def f1():
3     localVar = 2
4     print(globalVar) # OK: reads the global
5     print(localVar)  # OK: local
6 f1()
7 print(globalVar)     # 1
8 print(localVar)      # ERROR: localVar is out of scope

```

- ★ **Shadowing and the global statement:** if a function assigns to a name that also exists globally, it creates a *new local* by default (the global is unchanged). To bind a local variable in the global scope, or create a variable in a function and use it outside, use a **global** statement:

```

1 x = 1
2 def increase():
3     global x
4     x = x + 1
5 increase()
6 print(x) # 2 (the global x was changed)

```

- ★ **Scope trap in loops and if:** a variable created only inside an `if` branch (e.g. `y` assigned only when `x > 0`) does not exist if that branch never ran, so using it later raises an error. Also, after `for i in range(0, 5):`, the variable `i` still exists (holding its last value, 4) after the loop.

- ★ **Default arguments (normal use):** a parameter can have a default value, used when the caller omits that argument:

```

1 def printArea(width = 1, height = 2):
2     area = width * height
3     print("width:", width, "\theight:", height, "\tarea:", area)
4
5 printArea()                # width=1, height=2
6 printArea(4, 2.5)         # positional
7 printArea(height = 5, width = 3) # keyword
8 printArea(width = 1.2)     # default height=2
9 printArea(height = 6.2)    # default width=1

```

- ★ **Returning multiple values:** a function can return several values by listing them; the caller unpacks them with multiple targets. This sort returns two numbers in non-descending order:

```

1 def sort(number1, number2):
2     if number1 < number2:
3         return number1, number2
4     else:
5         return number2, number1
6
7 n1, n2 = sort(3, 2)      # n1 is 2, n2 is 3

```

- ★ **Function abstraction and its benefits:** think of a function body as a **black box** with an optional input and an optional return value; the caller does not need to know the internal implementation.

- **Write once, reuse anywhere.**
- **Information hiding:** hide the implementation from the user.
- **Reduce complexity.**

- ★ **Stepwise refinement:** apply abstraction to *developing* programs. Use the **divide and conquer** strategy (also called stepwise refinement) to decompose a large problem into sub-problems, then decompose those into smaller, manageable ones. The Print Calendar case study is the classic example: `printCalendar` splits into `readInput` and `printMonth`, which split further into `printMonthTitle`, `printMonthBody`, `getStartDay`, `getTotalNumOfDays`, `getNumOfDaysInMonth`, `getMonthName`, and `isLeapYear`.

- ★ **Top-down vs. bottom-up implementation:**

- **Top-down:** implement functions from the top of the structure chart downward, using a **stub** (a simple, incomplete placeholder) for functions not yet written, so you can test the caller.
- **Bottom-up:** implement from the bottom up, writing a small test program (a **driver**) for each function.
- **Both** implement incrementally, isolate errors, and make debugging easier; they can be combined.

CHAPTER 10 Lists

القوائم

10

★ How lists behave

القائمة كائن قابل للتعديل يحمل تسلسلاً من العناصر بفهرسة تبدأ من الصفر. الاسم يشير إليها، فالنسخ ليس دائماً نسخاً.

A list is one object that holds a sequence of elements, indexed from 0. Two facts govern almost every list bug. First, a list is **mutable**: methods like **append** change it in place. Second, a variable holds a **reference** to the list, so `list2 = list1` makes two names for *one* list, not a copy.

- ▷ **Index range is 0..len-1.** `lst[0]` is the first element and `lst[-1]` is the last. Going past the end raises `IndexError`; there is no silent garbage.
- ▷ **Slices copy a sub-list.** `lst[i:j]` returns a new list of elements `i..j-1`; the end index is excluded, exactly like `range`.
- ▷ **Sharing vs. copying.** To truly duplicate a list, use `[x for x in lst]` or `[] + lst`, never a bare `=`.

★ **Opening problem:** to read a hundred numbers, compute their average, and count how many are above it, you need to keep *all* the numbers so you can compare each to the average. A single list stores them all:

```

1  NUMBER_OF_ELEMENTS = 5          # use 5 instead of 100 for
    simplicity
2  numbers = []                    # Create an empty list
3  sum = 0
4  for i in range(NUMBER_OF_ELEMENTS):
5      value = eval(input("Enter a new number: "))
6      numbers.append(value)
7      sum += value
8  average = sum / NUMBER_OF_ELEMENTS
9  count = 0                        # number of elements above average
10 for i in range(NUMBER_OF_ELEMENTS):
11     if numbers[i] > average:
12         count += 1
13 print("Average is", average)
14 print("Number of elements above the average is", count)

```

★ **Creating lists:** using the list class or the bracket shorthand.

```

1  list1 = list()                  # empty list
2  list2 = list([2, 3, 4])         # elements 2, 3, 4
3  list3 = list(["red", "green", "blue"])
4  list4 = list(range(3, 6))       # 3, 4, 5
5  list5 = list("abcd")           # 'a', 'b', 'c', 'd'
6
7  list1 = []                      # same as list()
8  list2 = [2, 3, 4]              # same as list([2, 3, 4])
9  list4 = [2, "three", 4]        # a list may mix types

```

★ **List methods:** operations invoked on a list `s` with the dot.

Method	Result
<code>s.append(x)</code>	add item <code>x</code> to the end of the list
<code>s.insert(i, x)</code>	insert item <code>x</code> at index <code>i</code>
<code>s.remove(x)</code>	remove the first occurrence of <code>x</code>
<code>s.index(x)</code>	return the index of item <code>x</code>
<code>s.count(x)</code>	number of times <code>x</code> appears
<code>s.sort()</code>	sort the items in place
<code>s.reverse()</code>	reverse the items in place
<code>s.extend(d)</code>	append all items of <code>d</code> (same as <code>s + d</code>)
<code>s.pop(i) / s.pop()</code>	remove and return item at <code>i</code> (or the last)

★ **Functions and operators for lists:**

- `len(s)`, `max(s)`, `min(s)`, `sum(s)` work on lists of numbers.
- `random.shuffle(s)` randomly reorders the list in place.
- Membership: `x in s` / `x not in s`; concatenation `s + d`; repetition `s * n`.
- Iteration: `for item in s:` traverses elements left to right.

```

1 list1 = [2, 3, 4, 1, 32]
2 len(list1)    # 5
3 max(list1)    # 32
4 min(list1)    # 1
5 sum(list1)    # 42
6 import random
7 random.shuffle(list1)    # e.g. [4, 1, 2, 32, 3]
```

★ **The indexer operator `[]`:** accesses an element by position. For `myList = [5.6, 4.5, 3.3, 13.2, 4.0, 34.33, 34.0, 45.45, 99.993, 11123]`, `myList[0]` is 5.6, `myList[5]` is 34.33. Indices run `0..len-1`.

★ **The `+`, `*`, and slicing operators:**

```

1 list1 = [2, 3]; list2 = [1, 9]
2 list3 = list1 + list2    # [2, 3, 1, 9]
3 list3 = 2 * list1        # [2, 3, 2, 3]
4 list4 = list3[2 : 4]     # [2, 3] (indices 2 and 3)
```

★ **Negative indexing and membership:** `list1[-1]` is the last element, `list1[-3]` the third from the end. `2 in list1` is a Boolean test; `2.5 in list1` is `False` if absent.

★ **Index errors and the off-by-one trap:** accessing `list1[6]` or `list1[-7]` on a 6-element list raises `IndexError: list index out of range`. A loop written `while i <= len(lst)` overshoots by one; the correct bound is `while i < len(lst)`.

★ **List comprehension:** a concise way to build a list from a sequence — brackets holding an expression followed by a `for` clause and optional `if` clauses.

```

1 list1 = [x for x in range(0, 5)]           # [0, 1, 2, 3, 4]
2 list2 = [0.5 * x for x in list1]          # [0.0, 0.5, 1.0, 1.5,
    2.0]
3 list3 = [x for x in list2 if x < 1.5]      # [0.0, 0.5, 1.0]

```

★ **Comparing lists:** comparison operators compare lists **element by element** (lexicographically). For `list1 = ["green", "red", "blue"]` and `list2 = ["red", "blue", "green"]`: `list2 == list1` is `False`, `list2 != list1` is `True`, and `list2 > list1` compares the first differing elements.

★ **Splitting a string into a list:** the string method `split` breaks a string on a separator.

```

1 "Welcome to the US".split()              # ['Welcome', 'to', 'the', 'US']
    (whitespace)
2 "34#13#78#45".split("#")                 # ['34', '13', '78', '45']
3 "09/20/2012".split("/")                  # ['09', '20', '2012']

```

Reading and shifting a list: `s = input(...)`, then `lst = [eval(x) for x in s.split()]` converts a whitespace-separated line into a list of numbers.

★ **Problem: deck of cards.** A deck of 52 cards is `deck = [x for x in range(0, 52)]`. With suits and ranks lists, card number `i` has suit `deck[i] // 13` and rank `deck[i] % 13`; `random.shuffle(deck)` shuffles, and the first four cards are dealt.

★ **Copying lists — the sharing trap:** assignment does **not** copy a list; it makes a second name for the same object. After `list2 = list1`, `id(list1) == id(list2)`: they are the same list, so changing one changes the other.

– **Correct copy:** `list2 = [x for x in list1]` or simply `list2 = [] + list1`, both of which build a genuinely new list.

★ **Passing lists to functions:** you can pass a named list or an **anonymous list**. `printList([3, 1, 2, 6, 4, 2])` passes a list with no reference variable.

★ **Pass by value / by sharing revisited:** numbers and strings are immutable, so a function cannot change the caller's value; a *list* argument is shared, so changes made inside the function are visible outside.

★ **Returning a list from a function:** a function can build and return a list. Note a list already has an in-place `reverse()` method, but you can also return a reversed copy:

```

1 def reverse(lst):
2     result = []
3     for element in lst:
4         result.insert(0, element)    # push each to the front
5     return result
6
7 list1 = [1, 2, 3, 4, 5, 6]
8 list2 = reverse(list1)              # [6, 5, 4, 3, 2, 1]

```

★ **Problem: counting occurrences of each letter.** Generate 100 random lowercase letters into a list, then count each letter using a 26-element counts list indexed by `ord(chars[i]) - ord('a')`:

```

1 counts = 26 * [0]                                # 26 zeros
2 for i in range(len(chars)):
3     counts[ord(chars[i]) - ord('a')] += 1

```

Visual Walkthrough ★ Searching & Sorting

These are the two algorithms graders love to trace by hand. Watch how the list *changes* at every step, not just the final answer. **Binary search** throws away half the list each round; **selection sort** locks one more element into its final place each pass. **Linear search** simply scans left to right until it finds the key or runs out.

★ **Searching lists:** searching looks for a specific element (the **key**) in a list. Two standard approaches are **linear search** and **binary search**.

★ **Linear search:** compares the key sequentially with each element until a match is found (returns its index) or the list is exhausted (returns -1).

```

1 def linearSearch(lst, key):
2     for i in range(len(lst)):
3         if key == lst[i]:
4             return i
5     return -1

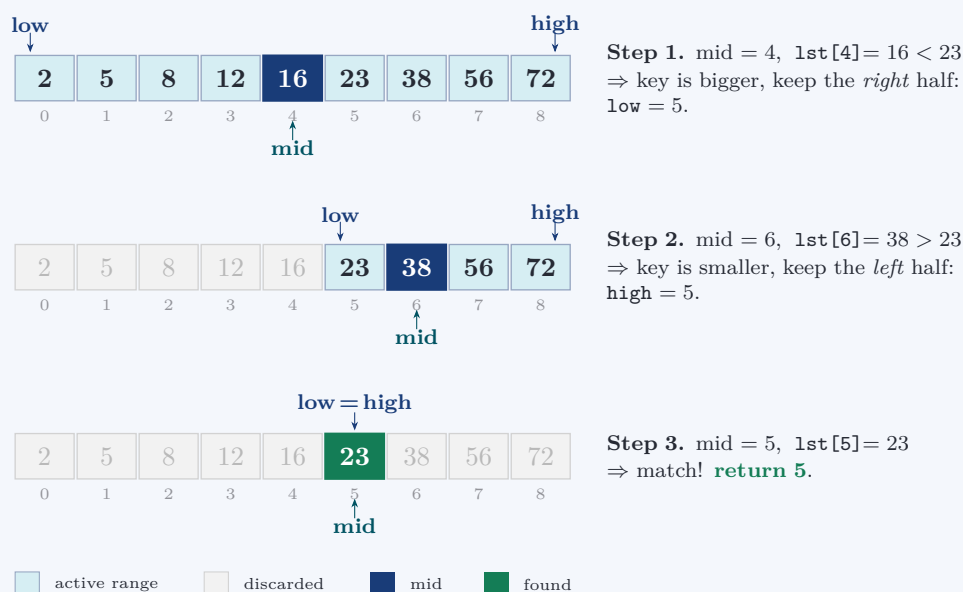
```

Linear search is useful for small or unsorted lists but is inefficient for large lists.

Binary Search — step by step

محاكاة البحث الثنائي

Binary search requires a *sorted* list. Searching for **key = 23**, each step compares the **mid** element and discards the half that cannot contain the key.



★ The search space went $9 \rightarrow 4 \rightarrow 1$ elements: that halving is why binary search is $O(\log n)$ while linear search is $O(n)$. The loop runs **while high $\geq$ low** — the equality matters, since the match at

Step 3 happens exactly when `low == high`. Binary searches are more efficient but require the list to be pre-sorted.

★ **Sorting lists:** arranging elements in order. Two simple algorithms are **selection sort** and **insertion sort**.

★ **Selection sort:** repeatedly find the smallest element in the unsorted part and swap it into the first unsorted position, locking one more element in place each pass.

```

1 def selectionSort(lst):
2     for i in range(len(lst) - 1):
3         currentMin = lst[i]
4         currentMinIndex = i
5         for j in range(i + 1, len(lst)):
6             if currentMin > lst[j]:
7                 currentMin = lst[j]
8                 currentMinIndex = j
9         if currentMinIndex != i:      # swap only if needed
10            lst[currentMinIndex] = lst[i]
11            lst[i] = currentMin

```

Selection Sort — multi-step

محاكاة فرز الاختيار

Each pass scans the unsorted part, picks the smallest value (teal), and swaps it with the first unsorted slot (gold). The green prefix is permanently sorted.



Pass 1. smallest is 8 (idx 4) → swap into idx 0.



Pass 2. smallest of the rest is 13 (idx 3) → swap into idx 1.



Pass 3. smallest of the rest is 14 (idx 3) → swap into idx 2.



Pass 4. smallest of the rest is 29 (idx 4) → swap into idx 3.



Done. the last element is already in place — list fully sorted.

0 1 2 3 4
 sorted
 min found
 swapped out

★ For n elements selection sort always does $n - 1$ passes regardless of input, giving $O(n^2)$ comparisons. The swap only happens when `currentMinIndex != i`, so an already-placed element costs no write.

★ **Insertion sort:** sorts by repeatedly *inserting* the next unsorted element into its correct place within the already-sorted prefix, shifting larger elements right to make room.

```
1 def insertionSort(lst):
2     for i in range(1, len(lst)):
3         currentElement = lst[i]
4         k = i - 1
5         while k >= 0 and lst[k] > currentElement:
6             lst[k + 1] = lst[k]          # shift larger element
              right
7             k -= 1
8         lst[k + 1] = currentElement     # insert into the gap
```

For [2, 9, 5, 4, 8, 1, 6], the sorted prefix grows one element per pass: insert 9, then 5, then 4, and so on until the whole list is ordered.

CHAPTER 11 Multidimensional Lists

القوائم متعددة الأبعاد

11

★ How 2D lists are laid out

القائمة ثنائية البعد قائمة من القوائم؛ صفُّ تلو الآخر. الوصول يكون بفهرس الصف ثم فهرس العمود.

A two-dimensional list is simply a **list of rows**, where each row is itself a list of values. You reach a cell with *two* indices: `matrix[row][column]`. Once that picture is in your head, every 2D operation is just a loop over rows with an inner loop over columns.

- ▷ **Rows first, then columns.** `matrix[i]` is the whole *i*-th row (a list); `matrix[i][j]` is one element. `len(matrix)` counts rows, `len(matrix[0])` counts columns of the first row.
- ▷ **Rows can differ in length.** Because each row is an independent list, a 2D list is not required to be rectangular — though most examples are.
- ▷ **Build empty rows before filling.** `matrix.append([])` adds a new empty row, then you `matrix[row].append(value)` into it.

★ **Processing two-dimensional lists:** view a 2D list as a list of rows accessed by a **row index**, with values inside each row accessed by a **column index**. Example:

```
1 matrix = [
2     [1, 2, 3, 4, 5],
3     [6, 7, 0, 0, 0],
4     [0, 1, 0, 0, 0],
5     [1, 0, 0, 0, 8],
6     [0, 0, 9, 0, 3]
7 ]
8 matrix[0]      # [1, 2, 3, 4, 5] (the whole first row)
9 matrix[0][0]   # 1
10 matrix[4][4]  # 3
```

★ **Initializing with input values:** create an empty list, then append a new empty row for each row and read the columns into it:

```
1 matrix = []
2 numberOfRows = eval(input("Enter the number of rows: "))
3 numberOfColumns = eval(input("Enter the number of columns: "))
4 for row in range(numberOfRows):
5     matrix.append([])                # add an empty new row
6     for column in range(numberOfColumns):
7         value = eval(input("Enter an element and press Enter: "))
8         matrix[row].append(value)
9 print(matrix)
```

★ **Initializing with random values:** the same structure, filling each cell with `random.randint(0, 99)`:

```
1 import random
2 matrix = []
3 for row in range(numberOfRows):
4     matrix.append([])
5     for column in range(numberOfColumns):
6         matrix[row].append(random.randint(0, 99))
```

★ **Printing a 2D list:** loop over rows, then over columns (or iterate directly):

```
1 for row in range(len(matrix)):
2     for column in range(len(matrix[row])):
3         print(matrix[row][column], end=" ")
4     print()    # newline after each row
```

★ **Summing all elements:** keep a running total across every cell.

```
1 total = 0
2 for row in matrix:
3     for value in row:
4         total += value
5 print("Total is", total)
```

★ **Summing elements by column:** for each column index, add that column's element from every row.

```
1 for column in range(len(matrix[0])):
2     total = 0
3     for row in range(len(matrix)):
4         total += matrix[row][column]
5     print("Sum for column", column, "is", total)
```

★ **Which row has the largest sum:** track the maximum row-sum and its index.

```
1 maxRow = sum(matrix[0])    # sum of the first row
2 indexOfMaxRow = 0
3 for row in range(1, len(matrix)):
4     if sum(matrix[row]) > maxRow:
5         maxRow = sum(matrix[row])
6         indexOfMaxRow = row
7 print("Row", indexOfMaxRow, "has the maximum sum of", maxRow)
```

★ **Random shuffling of a 2D list:** `random.shuffle` works on a 1D list; to shuffle every element of a 2D list, swap each `matrix[row][column]` with a randomly chosen `matrix[i][j]`:

```

1 import random
2 for row in range(len(matrix)):
3     for column in range(len(matrix[row])):
4         i = random.randint(0, len(matrix) - 1)
5         j = random.randint(0, len(matrix[row]) - 1)
6         matrix[row][column], matrix[i][j] = matrix[i][j], matrix
            [row][column]

```

- ★ **Sorting a 2D list:** the `sort` method orders the rows by their *first* element; ties are broken by the second element, then the third, and so on.

```

1 points = [[4, 2], [1, 7], [4, 5], [1, 2], [1, 1], [4, 1]]
2 points.sort()
3 # [[1, 1], [1, 2], [1, 7], [4, 1], [4, 2], [4, 5]]

```

- ★ **Passing 2D lists to functions:** a 2D list is passed like any object (by sharing). A function can build one and return it, or accept one and process it, e.g. `accumulate(m)` sums all elements by looping rows then `sum(row)`.

- ★ **Problem: grading a multiple-choice test.** Students' answers are a 2D list (one row per student), and the correct answers are a 1D key list. For each student, compare every answer to the key and count matches:

```

1 answers = [
2     ['A', 'B', 'A', 'C', 'C', 'D', 'E', 'E', 'A', 'D'],
3     ['D', 'B', 'A', 'B', 'C', 'A', 'E', 'E', 'A', 'D'],
4     # ... one row per student
5 ]
6 keys = ['D', 'B', 'D', 'C', 'C', 'D', 'A', 'E', 'A', 'D']
7 for i in range(len(answers)):
8     correctCount = 0
9     for j in range(len(answers[i])):
10         if answers[i][j] == keys[j]:
11             correctCount += 1
12     print("Student", i, "'s correct count is", correctCount)

```

- ★ **Problem: finding the two nearest points.** Given a list of points (each a 2-element list `[x, y]`), compare every pair to find the smallest distance. The distance between `(x1, y1)` and `(x2, y2)` uses `** 0.5`, and two nested loops (`i` and `j > i`) test each pair once:

```

1 def distance(x1, y1, x2, y2):
2     return ((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1)) **
        0.5
3
4 def nearestPoints(points):
5     p1, p2 = 0, 1                                # track the two nearest
6     shortestDistance = distance(points[p1][0], points[p1][1],
7                                     points[p2][0], points[p2][1])
8     for i in range(len(points)):
9         for j in range(i + 1, len(points)):
10            d = distance(points[i][0], points[i][1],
11                            points[j][0], points[j][1])
12            if shortestDistance > d:
13                p1, p2 = i, j
14                shortestDistance = d
15     return p1, p2

```

- ★ **Multidimensional lists:** lists can nest more than two deep. A three-dimensional list is a list of 2D lists, indexed with *three* brackets. For a `scores` table indexed by student, exam, and part:

```

1 scores = [
2     [[11.5, 20.5], [11.0, 22.5], [15, 33.5]],    # student 0
3     [[4.5, 21.5], [11.0, 22.5], [15, 34.5]],    # student 1
4     # ...
5 ]
6 scores[i][j][k]    # which student -> which exam -> which part

```

- ★ **Problem: guessing birthday with a 3D list.** The birthday-guessing trick stores its five card sets as a 3D list `dates`; the program walks `dates[i][j][k]` to print each card and, for every "yes," adds `dates[i][0][0]` to the running day total:

```

1 day = 0
2 for i in range(5):
3     print("Is your birthday in Set" + str(i + 1) + "?")
4     for j in range(4):
5         for k in range(4):
6             print(format(dates[i][j][k], "4d"), end=" ")
7             print()
8         answer = eval(input("Enter 0 for No and 1 for Yes: "))
9         if answer == 1:
10            day += dates[i][0][0]
11 print("Your birthday is " + str(day) + "!")

```

CHAPTER 13 Files and Exception Handling

الملفات ومعالجة الاستثناءات

13

★ How files and exceptions work

الملف يُفْتَح بنِطٍ محدد، يُقْرَأ أو يُكْتَب، ثم يُغْلَق. والاستثناء خطأً زمن التشغيل تلتقطه *try/except* لتُكَبَّل بأمان.

Program data lives only while the program runs; to keep it you write it to a **file** on disk. Every file interaction follows the same three beats: **open** it in a mode, **read** or **write**, then **close**. Separately, when a program hits an impossible operation at run time it raises an **exception**; *try/except* lets you catch it and keep going instead of crashing.

- ▷ **The mode decides what's allowed.** "r" reads, "w" writes (erasing existing content), "a" appends to the end. Reading a file opened for writing, or vice versa, fails.
- ▷ **Text is strings.** *write* takes a string, so numbers must be turned into strings first, and separated (by a space or newline) so you can read them back apart.
- ▷ **Exceptions travel up until caught.** An uncaught exception aborts the program with a traceback; a matching *except* stops that and runs your recovery code.

- ★ **Motivation:** data stored in variables is temporary and lost when the program ends. To keep data permanently you save it in a **file** on disk, where it can be transported and read later. There are two kinds of files: **text** (essentially strings on disk) and **binary**.
- ★ **Opening a file:** you associate a file object with a physical file by **opening** it. The syntax is `fileVariable = open(filename, mode)`.

Mode	Description
'r'	open for reading only
'w'	open for writing only (creates/overwrites)
'a'	open for appending; data written to the end
'rb'	open for reading binary data
'wb'	open for writing binary data

- ★ **Writing to a file:** open in "w", use `write(str)`, then `close()`.

```
1 outfile = open("Presidents.txt", "w")
2 outfile.write("Bill Clinton\n")
3 outfile.write("George Bush\n")
4 outfile.write("Barack Obama")
5 outfile.close()
```

Note: *write* does *not* add a newline; include `\n` yourself where you want line breaks.

- ★ **Testing file existence:** before overwriting, you can check whether a file already exists with `os.path.isfile`:

```
1 import os
2 if os.path.isfile("Presidents.txt"):
3     print("Presidents.txt exists")
```

★ **Reading from a file:** after opening in "r", the file object offers several read methods.

Method	Result
<code>read()</code>	read the entire remaining contents as one string
<code>read(n)</code>	read the next <code>n</code> characters as a string
<code>readline()</code>	read the next line as a string
<code>readlines()</code>	read all remaining lines into a list
<code>write(s)</code>	write the string <code>s</code> to the file
<code>close()</code>	close the file

```

1 infile = open("Presidents.txt", "r")
2 print(infile.read())           # whole file at once
3 infile.close()
4
5 infile = open("Presidents.txt", "r")
6 print(infile.read(4))          # first 4 characters -> 'Bill'
7 infile.close()
8
9 infile = open("Presidents.txt", "r")
10 line1 = infile.readline()     # one line at a time
11 print(infile.readlines())     # remaining lines as a list
12 infile.close()

```

★ **Appending to a file:** open in "a" to add data to the *end* of an existing file without erasing it:

```

1 outfile = open("Info.txt", "a")
2 outfile.write("\nPython is interpreted\n")
3 outfile.close()

```

★ **Writing and reading numeric data:** write needs strings, so convert numbers with `str()` and separate them with a whitespace character (' ' or '\n') so you can split them apart when reading:

```

1 # writing
2 outfile = open("Numbers.txt", "w")
3 for i in range(10):
4     outfile.write(str(random.randint(0, 9)) + " ")
5 outfile.close()
6 # reading them back
7 infile = open("Numbers.txt", "r")
8 s = infile.read()
9 numbers = [eval(x) for x in s.split()] # convert back to
    numbers
10 infile.close()

```

★ **Problem: counting each letter in a file.** Prompt for a file name, open it, and count the occurrences of each letter (case-insensitive) using a 26-element counts list indexed by `ord(ch) - ord('a')`, guarding with `ch.isalpha()`.

★ **Retrieving data from the web:** you can open a URL like a file with `urllib.request.urlopen` and read its contents:

```

1 import urllib.request
2 infile = urllib.request.urlopen("http://www.yahoo.com/index.html")
3 print(infile.read().decode())    # bytes -> string with decode()

```

★ **Exception handling — the motivation:** if the user enters a file or URL that does not exist, the program is aborted and raises an error (for example an `IOError`) with a traceback. Exception handling lets the program continue running or terminate gracefully instead.

★ **The try ... except clause:** put the risky code in `try`; if an exception of a matching type occurs, control jumps to the corresponding `except` handler.

```

1 try:
2     <body>
3 except ExceptionType1:
4     <handler1>
5 except ExceptionTypeN:
6     <handlerN>
7 except:
8     <handler for any other exception>
9 else:
10    <runs only if no exception occurred>
11 finally:
12    <always runs, exception or not>

```

- `else` runs only when the `try` body raised no exception.
- `finally` runs no matter what — ideal for cleanup like closing a file.

```

1 try:
2     number1, number2 = eval(input("Enter two numbers, separated
3         by a comma: "))
4     result = number1 / number2
5     print("Result is", result)
6 except ZeroDivisionError:
7     print("Division by zero!")
8 except SyntaxError:
9     print("A comma may be missing in the input")
10 except:
11     print("Something wrong in the input")
12 else:
13     print("No exceptions")
14 finally:
15     print("The finally clause is executed")

```

★ **Raising exceptions:** an exception is an object created from a class and **raised** by a function that detects an error. The syntax is `raise ExceptionClass("Something is wrong")`.

```

1 def getArea(radius):
2     if radius < 0:
3         raise RuntimeError("Negative radius")
4     return radius * radius * 3.1415

```

- ★ **Processing exceptions using exception objects:** bind the raised object with `as` to inspect its message:

```

1 try:
2     number = eval(input("Enter a number: "))
3     print("The number entered is", number)
4 except NameError as ex:
5     print("Exception:", ex)

```

- ★ **Python's built-in exception classes:** all inherit from `BaseException` → `Exception` → `StandardError`, with families such as `ArithmeticError` (e.g. `ZeroDivisionError`), `RuntimeError`, `LookupError` (e.g. `IndexError`, `KeyError`), `SyntaxError` (e.g. `IndentationError`), `IOError`, and `OSError`.
- ★ **Custom exception classes:** you can define your own exception by subclassing an existing exception class, giving your program meaningful, domain-specific error types.

- ★ **Binary I/O using pickling:** the `pickle` module writes and reads Python objects in binary form. Open the file in `"wb"` to write and `"rb"` to read, then use `pickle.dump(obj, file)` and `pickle.load(file)`:

```

1 import pickle
2 outfile = open("pickle.dat", "wb")
3 pickle.dump(45, outfile)                # an int
4 pickle.dump(56.6, outfile)              # a float
5 pickle.dump("Programming is fun", outfile) # a string
6 pickle.dump([1, 2, 3, 4], outfile)      # a list
7 outfile.close()
8
9 infile = open("pickle.dat", "rb")
10 print(pickle.load(infile))             # 45
11 print(pickle.load(infile))             # 56.6
12 # ... one load per dumped object
13 infile.close()

```

- ★ **Detecting the end of a file:** if you do not know how many objects were pickled, keep calling `load` until it raises `EOFError`, then catch it to end the read loop:

```
1 import pickle
2 infile = open("numbers.dat", "rb")
3 end_of_file = False
4 while not end_of_file:
5     try:
6         print(pickle.load(infile), end=" ")
7     except EOFError:
8         end_of_file = True
9 infile.close()
10 print("\nAll objects are read")
```

CHAPTER 14 Tuples, Sets, and Dictionaries

الصفوف والمجموعات والقواميس

14

★ How tuples, sets, and dictionaries differ

الصف قائمة غير قابلة للتعديل، والمجموعة عناصر فريدة بلا ترتيب، والقاموس أزواج مفتاح/قيمة للوصول السريع.

Three collections, three jobs. A **tuple** is like a list but **immutable** — use it when the contents should never change. A **set** stores **unique, unordered** elements and answers "is this in here?" fast. A **dictionary** stores **key/value pairs** so you can look a value up by its key, like an indexer that is not limited to integers.

- ▷ **Immutability buys safety and speed.** Once built, a tuple cannot be changed, which prevents accidental modification and makes it more efficient than a list.
- ▷ **Sets drop duplicates and order.** {1, 3, 5} has no positions; you cannot index a set. If order does not matter, sets are more efficient than lists for membership tests.
- ▷ **Dictionaries are keyed lookups.** `d[key]` fetches the value for a key. Keys are unique; assigning to an existing key overwrites its value.

★ **Tuples:** like lists except **immutable** — once created their contents cannot change. If a collection in your program never changes, use a tuple to prevent accidental modification; tuples are also more efficient than lists.

★ Creating tuples:

```
1 t1 = ()                                # empty tuple
2 t2 = (1, 3, 5)                        # three elements
3 t3 = tuple([2 * x for x in range(1, 5)]) # from a list -> (2,
    4, 6, 8)
4 t4 = tuple("abac")                    # from a string -> ('a', 'b', 'a', 'c')
```

★ **Using tuples:** you may use a tuple like a list — index it, slice it, iterate it, and apply `len`, `min`, `max`, `sum`, membership, and concatenation — but you *cannot* modify it. Methods that would change contents (`append`, `sort`, `extend`) are not available. You can, however, combine tuples into a new one (`t3 = t1 + t2`) or duplicate (`t3 = 2 * t1`) since those build new tuples.

★ **Sets:** like lists in that they store a collection, but **unlike lists** the elements are **unique** and **unordered**. If your program does not care about order, a set is more efficient than a list. The syntax uses braces `{ }`.

★ Creating sets:

```

1 s1 = set()                                # empty set (NOT {}, which is
    a dict)
2 s2 = {1, 3, 5}                            # three elements
3 s3 = set([1, 3, 5])                      # from a list
4 s4 = set([x * 2 for x in range(1, 10)])  # from a comprehension
5 s5 = set("abac")                         # from a string -> {'a', 'b', 'c'}

```

★ Manipulating and accessing sets:

- `s.add(x)` adds an element; `s.remove(x)` removes one.
- `len(s)`, `max(s)`, `min(s)`, `sum(s)`, and `x in s` all work.
- `s1.issubset(s2)` tests subset ($s1 \subseteq s2$); `s2.issuperset(s1)` tests superset.
- Equality `s1 == s2` is `True` when they contain the same elements, regardless of order.

★ Comparison operators for sets: because elements are unordered, `>`, `>=`, `<=`, `<` do *not* mean numeric ordering; they mean subset/superset relations.

- `s1 > s2` is `True` if `s1` is a *proper superset* of `s2`.
- `s1 >= s2` is `True` if `s1` is a superset of `s2`.
- `s1 < s2` is `True` if `s1` is a *proper subset* of `s2`.
- `s1 <= s2` is `True` if `s1` is a subset of `s2`.

★ Set operations: each has a method form and an operator form.

Operation	Method	Operator
Union	<code>s1.union(s2)</code>	<code>s1 s2</code>
Intersection	<code>s1.intersection(s2)</code>	<code>s1 & s2</code>
Difference	<code>s1.difference(s2)</code>	<code>s1 - s2</code>
Symmetric diff.	<code>s1.symmetric_difference(s2)</code>	<code>s1 ^ s2</code>

For `s1 = {1, 2, 4}` and `s2 = {1, 3, 5}`: union is `{1, 2, 3, 4, 5}`, intersection is `{1}`, difference `s1 - s2` is `{2, 4}`, symmetric difference is `{2, 3, 4, 5}`.

★ Performance: sets vs. lists: testing membership, adding, and removing are much faster for a large set than for a list, because a set is not searched linearly. For thousands of lookups, a set can be dramatically faster (milliseconds vs. seconds).

★ Why a dictionary: suppose you store a million students and often search by ID. A **dictionary** is the efficient structure for this: it stores elements as **key/value pairs**, and the key acts like an indexer (not limited to integers).

★ Creating a dictionary: braces with **key: value** pairs.

```

1 dictionary = {}                                # empty dictionary
2 dictionary = {"Ali": 40, "Omar": 45}          # two pairs
3 students = {"111-34-3434": "Ali", "132-56-6290": "Omar"}

```

★ Adding and modifying entries: `dictionaryName[key] = value` adds a new pair or overwrites an existing key's value:

```

1 dictionary["Sara"] = 50           # add a new pair
2 students["234-56-9010"] = "Sara" # add by key

```

★ **Deleting entries:** `del dictionary[key]` removes a pair by its key.

★ **Looping entries:** iterating a dictionary yields its *keys*:

```

1 for key in dictionary:
2     print(key + ":" + str(dictionary[key]))

```

★ **The len, in, and equality operators for dictionaries:**

- `len(dictionary)` returns the number of pairs.
- `key in dictionary` tests whether a *key* is present.
- `d1 == d2` is `True` when they hold the same key/value pairs (order does not matter); `!=` is the negation.

★ **Dictionary methods:**

Method	Result
<code>keys()</code>	a sequence of the keys
<code>values()</code>	a sequence of the values
<code>items()</code>	a sequence of (key, value) tuples
<code>clear()</code>	delete all entries
<code>get(key)</code>	the value for the key (or <code>None</code> if absent)
<code>pop(key)</code>	remove the pair for the key and return its value
<code>popitem()</code>	remove and return an arbitrary (key, value) pair

```

1 students = {"111-34-3434": "Ali", "132-56-6290": "Omar"}
2 tuple(students.keys())      # ('111-34-3434', '132-56-6290')
3 tuple(students.values())    # ('Ali', 'Omar')
4 students.get("111-34-3434") # 'Ali'
5 print(students.get("999-34-3434")) # None (absent key, no
   error)
6 students.pop("111-34-3434") # 'Ali', and removes the pair
7 students.clear()           # {}

```

Note: `get` is safer than `d[key]` for a possibly-missing key, since indexing a missing key raises `KeyError` while `get` returns `None`.

★ **Case study: occurrences of words.** Count how often each word appears in a text file and display them in order. A **dictionary** maps each word to its count: for each word, if it is not yet a key, add it with value 1; otherwise increase its value by 1. Treat words case-insensitively (Good and good are the same).

```
1 def processLine(line, wordCounts):
2     line = replacePunctuations(line)      # replace punctuation
3     with spaces
4     words = line.split()                  # split into words
5     for word in words:
6         if word in wordCounts:
7             wordCounts[word] += 1          # increase count
8         else:
9             wordCounts[word] = 1           # new word
10
11 # main flow:
12 wordCounts = {}                          # word -> count
13 for line in infile:
14     processLine(line.lower(), wordCounts)
15 pairs = list(wordCounts.items())          # get (word, count)
16 pairs
17 items = [[y, x] for (x, y) in pairs]      # reverse to (count,
18 word)
19 items.sort()                             # sort by count
20 # print the most frequent words in decreasing order
21 for i in range(len(items) - 1, len(items) - 11, -1):
22     print(items[i][1] + "\t" + str(items[i][0]))
```

The trick of reversing each pair to [count, word] lets the built-in sort order the words by their counts.

تم بحمد الله