

Python Handout

English edition

Contents

1	Getting started	3
1.1	Your first program	3
1.2	Comments & code style	3
1.3	Input, process, output	4
2	Variables, types & operators	5
2.1	Variables & assignment	5
2.2	Numbers: int & float	5
2.3	Expressions & type conversion	6
2.4	Booleans & comparison	6
3	Strings	8
3.1	Indexing	8
3.2	Slicing	8
3.3	String methods & length	9
3.4	f-strings	10
4	Selection	11
4.1	if / elif / else	11
4.2	Combining conditions	12
5	Iteration	13
5.1	for loops and range	13
5.2	The accumulator pattern	13
5.3	while loops	14
5.4	Nested loops	14
6	Lists & 2-D lists	16
6.1	Lists	16
6.2	Traversing a list	17
6.3	2-D lists (grids)	17
6.4	List comprehensions	17
6.5	Tuples & sets	18
7	Dictionaries	19
7.1	Add and update	19
7.2	Check and loop	19
8	Functions & abstraction	21
8.1	Defining & calling functions	21
8.2	Return values	21
8.3	Parameters, arguments & scope	22

8.4	Procedural abstraction	22
8.5	Modules & imports	23
9	Errors, exceptions & testing	24
9.1	Errors & debugging	24
9.2	try / except / raise	24
9.3	Testing & robustness	25
10	Files	26
10.1	Reading & writing text files	26
11	Algorithm design	28
11.1	Algorithms & decomposition	28
11.2	Pseudocode & flowcharts	28
11.3	Recursion & the call stack	28
12	Data structures	30
12.1	Abstract Data Types (ADTs)	30
12.2	Stacks	30
12.3	Queues	31
12.4	Linked lists	31
12.5	Hash tables	31
12.6	Binary search trees	32
12.7	Graphs	33
13	Searching, sorting & efficiency	35
13.1	Linear & binary search	35
13.2	Sorting (bubble & insertion)	36
13.3	Algorithmic efficiency	36
13.4	Randomness & simulation	37
14	OOP & paradigms	39
14.1	Classes & objects	39
14.2	Inheritance, encapsulation & polymorphism	40
14.3	Programming paradigms	41
15	Data representation	42
15.1	Bits & binary	42
15.2	Compression	43
16	Computing concepts	44
16.1	What computing is & the design cycle	44
16.2	The Internet	44
16.3	Parallel & distributed computing	44

16.4	Impact of computing	45
17	Putting it together	46
17.1	End-to-end mini-projects	46

1 Getting started

1.1 ■ Your first program

Python runs your code one line at a time. Each line is a statement . A program is just a list of statements that run from top to bottom.

The `print()` function shows text on the screen. This is called output . Text inside quotes is a string .

```
print("Hello, world!")
print("I am learning Python")
```

- Each `print()` starts a new line.
- Quotes can be "double" or 'single' – both make a string.
- A program does nothing until you run it.

Your first program: write → run → see



Exam tip: `print` shows a value; quotes mark a string; indentation is part of the syntax

Source code runs through the interpreter to produce output

1.2 ■ Comments & code style

A comment starts with `#`. Python ignores everything after the `#` on that line. Comments explain your code to people; they do not change what the code does.

```
# This line is a note for humans
print("Hi")          # you can also comment at the end of a line
```

Good style makes code easy to read:

- Use clear names that say what a value means.
- Put one statement on each line.
- Do not add spaces at the start of a normal line. In Python, spacing at the start (indentation) has a special meaning, so a wrong space gives an error .

1.3 ■ Input, process, output

Many programs follow a simple plan: **input** → **process** → **output**. You get some data, do something with it, then show a result.

The `input()` function reads text that the user types. It always gives back a string.

```
name = input("What is your name? ")
print("Hello, " + name)
```

- `input()` waits for the user to type and press Enter.
- Store the typed text in a variable so you can use it later.
- Because `input()` returns a string, change it with `int(...)` first if you need a number.

Common mistakes

- Forgetting the quotes: `print(Hello)` looks for a variable named `Hello` and raises a `NameError`. Text needs quotes: `print("Hello")`.
- A stray space at the **start** of a line: Python reads indentation as structure, so it raises an `IndentationError`.
- Assuming `input()` returns a number. It always returns a string, so wrap it in `int(...)` before doing any maths.

2 Variables, types & operators

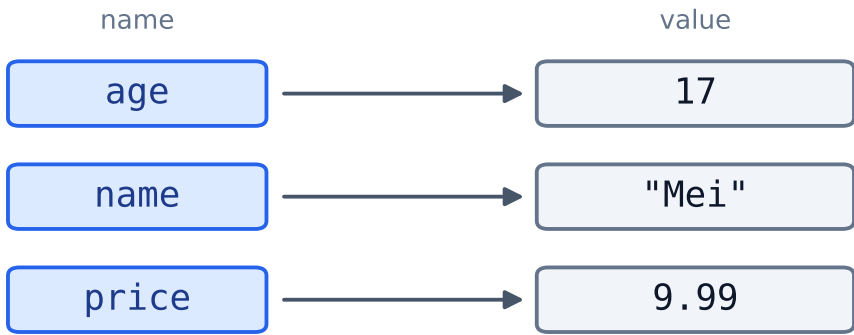
2.1 ■ Variables & assignment

A variable is a name for a value . You make one with =, which is called assignment . The name goes on the left; the value goes on the right.

```
age = 17
name = "Mei"
price = 9.99
print(age, name, price)
```

Now age holds 17. Use the name anywhere you need the value, and change it later:

```
age = 17
age = age + 1 # age is now 18
print(age)
```



variable: name → value in memory

Each variable name points to a value in memory

- The = sign does not mean "equal". It means "store the right side under the left name".
- To test if two values are equal, use == (see below).

2.2 ■ Numbers: int & float

Python has two main number types. An integer (int) is a whole number like 17. A float (float) has a decimal point like 9.99.

These operators work on numbers:

Operator	Meaning	Example	Result
+	add	3 + 2	5
-	subtract	3 - 2	1
*	multiply	3 * 2	6
/	divide (always float)	7 / 2	3.5
//	integer divide	7 // 2	3
%	remainder (modulo)	7 % 2	1
**	power	2 ** 3	8

- / always gives a float, so 4 / 2 is 2.0.
- // and % go together: 17 // 5 is 3, and 17 % 5 is 2.

2.3 ■ Expressions & type conversion

An expression is anything that has a value, like 3 + 4 * 2. Python uses normal maths order (* and / before + and -); add brackets to make the order clear.

input() gives a string, so convert it before doing maths. Changing a value from one type to another is type conversion :

```
age = int("17") # text "17" -> number 17
price = float("9.99") # text -> 9.99
label = str(17) # number -> text "17"
print(age, price, label)
```

- int("abc") fails, so only convert text that looks like a number.
- Mixing types fails too: "age: " + 17 is an error; write "age: " + str(17).

2.4 ■ Booleans & comparison

A Boolean is one of just two values: True or False. A comparison gives back a Boolean.

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

```
print(7 > 2) # True
print(3 == 3.0) # True
age = 20
print(age >= 18) # True
```

Join comparisons with `and`, `or`, `not`:

```
age = 20
print(age >= 13 and age <= 19)    # True only for a teenager
```

Common mistakes

- `/` always gives a float, even `4 / 2` is `2.0`. Use `//` when you want a whole number.
- Using `=` where you mean `==`: `=` assigns a value, `==` tests whether two values are equal.
- Joining a string and a number: `"age: " + 5` raises a `TypeError`. Convert first with `str(5)`.
- Floats are not exact, so `0.1 + 0.2` is not exactly `0.3` – avoid `==` on float results.

3 Strings

3.1 ■ Indexing

A string is text inside quotes. Each character has a position, called its index. The first index is 0, not 1.

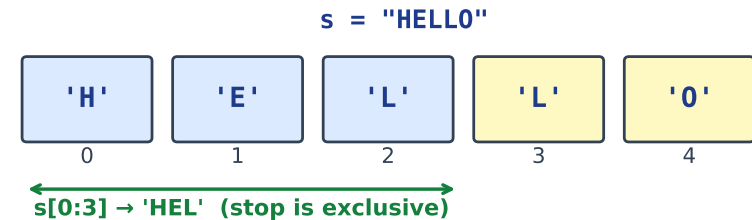
Read one character with square brackets:

```
word = "Python"
print(word[0])    # P (the first character)
print(word[2])    # t
print(len(word))  # 6 (how many characters)
```

- Counting starts at 0, so the last index is `len(word) - 1`.
- A negative index counts back from the end: `word[-1]` is the last character.

```
word = "Python"
print(word[-1])   # n
print(word[-2])   # o
```

- An index that is too large gives an error (an `IndexError`).



Exam tip: slice is half-open [start, stop); omit ends for defaults; step is optional

`s[start:stop]` is a half-open window of characters

3.2 ■ Slicing

A slice takes a part of a string. Write `word[start:end]`. The slice keeps **start** but stops **before end**.

```
word = "Python"
print(word[0:3])    # Pyt (positions 0, 1, 2)
print(word[2:5])    # tho
```

- Leave out **start** to begin at 0; leave out **end** to go to the end.

```
word = "Python"
print(word[:3])    # Pyt
print(word[3:])    # hon
```

- A third number is the step . word[::-1] reverses the string.

```
print("Python"[::-1])    # nohtyP
```

3.3 ■ String methods & length

A method is a function that belongs to a value. You call it with a dot:

```
name = "mei chen"
print(name.upper())    # MEI CHEN
print(name.title())    # Mei Chen
print(len(name))       # 8
```

Strings are immutable : a method returns a **new** string and never changes the original one.

```
name = "mei"
print(name.upper())    # MEI (the returned value)
print(name)            # mei (the original is unchanged)
```

Common methods (each returns a new value):

Method	Meaning	Example	Result
.upper() / .lower()	change case	"Hi".lower()	hi
.strip()	remove edge spaces	" hi ".strip()	hi
.replace(a, b)	swap text	"cat".replace("c", "b")	bat
.split(sep)	break into a list	"a,b".split(",")	['a', 'b']

Join strings with +. This is called concatenation :

```
first = "Mei"
last = "Chen"
print(first + " " + last)    # Mei Chen
```

3.4 ■ f-strings

An f-string builds text from values. Put **f** before the quote, then write {...} around a value.

```
name = "Mei"
age = 17
print(f"{name} is {age} years old")    # Mei is 17 years old
```

- Any expression can go inside the braces.
- {value:.2f} rounds to 2 decimal places .

```
price = 9.5
print(f"Two cost {price * 2}")    # Two cost 19.0
print(f"Pi is about {3.14159:.2f}")    # Pi is about 3.14
```

Common mistakes

- Strings cannot be changed in place: s[0] = "x" raises an error. Build a new string instead.
- Indexing starts at 0; the last character is s[-1], and s[len(s)] is out of range.
- A slice s[a:b] includes a but stops **before** b.
- String methods return a NEW string: s.upper() does nothing unless you store the result.

4 Selection

4.1 ■ if / elif / else

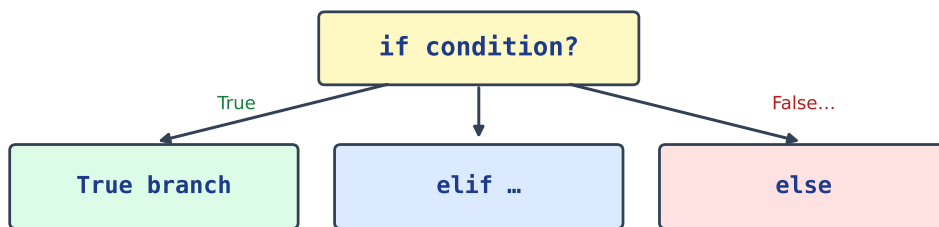
A program chooses what to do with `if`. It runs an indented block only when a condition is true. The `if` line ends with a colon.

```
score = 72
if score >= 60:
    print("pass")
# pass
```

Add `elif` (else-if) for more cases and `else` for "anything else". Python runs the **first** true branch only, then skips the rest.

```
score = 72
if score >= 80:
    print("A")
elif score >= 60:
    print("B")
else:
    print("fail")
# B
```

- Compare values with `==` (equal to), `!=` (not equal to), `<`, `>`, `<=`, `>=`.
- A comparison gives a Boolean – either `True` or `False`.



Exam tip: only one branch runs; conditions are checked top to bottom; indent the body

if / elif / else: only one branch runs

4.2 ■ Combining conditions

Join conditions with `and`, `or`, `not`. `and` needs **both** sides true; `or` needs **either** side true; `not` flips a Boolean.

```
age = 16
has_ticket = True
if age >= 18 and has_ticket:
    print("entry allowed")
else:
    print("entry refused")
# entry refused
```

- Use brackets to make the order clear: `(a or b) and c`.

```
temp = 30
if temp > 25 and not temp > 35:
    print("warm but ok")
# warm but ok
```

Common mistakes

- Write `elif`, not `else if`.
- Every `if / elif / else` line ends with a colon `:`, and its body must be indented.
- `if x = 5:` is an error – use `==` to compare.
- `else` takes no condition; only `if` and `elif` do.

5 Iteration

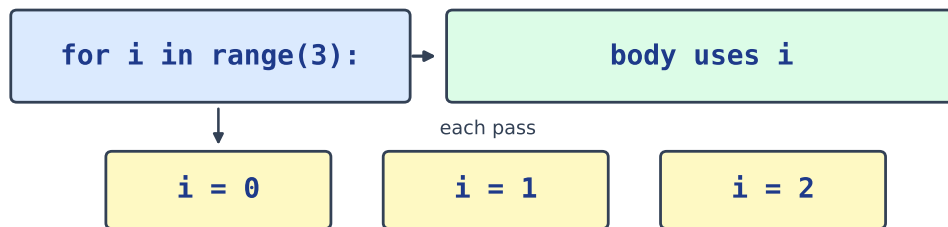
5.1 ■ for loops and range

A loop repeats code. A **for** loop repeats once for each item in a sequence. **range(n)** gives the numbers 0 up to $n - 1$.

```
for i in range(5):
    print(i)
# 0, then 1, 2, 3, 4 (each on its own line)
```

- **range(a, b)** goes from a up to (but not including) b.
- **range(a, b, step)** adds a step each time.

```
for n in range(2, 11, 2):
    print(n)
# 2 4 6 8 10
```



Exam tip: **range(n)** is 0 ... $n-1$; the loop variable takes one value per pass

for i in range(n): body runs with $i = 0 \dots n-1$

5.2 ■ The accumulator pattern

To build a result across a loop, start a variable **before** the loop, then update it each turn. This is the accumulator pattern.

```
total = 0
for n in range(1, 6):
    total = total + n
print(total)
# 15
```

- The same idea counts how many items match a test.

```
count = 0
for letter in "banana":
    if letter == "a":
        count = count + 1
print(count)
# 3
```

5.3 ■ while loops

A **while** loop repeats **as long as** a condition stays true. Change something inside, or it never stops – an infinite loop.

```
n = 1
while n <= 3:
    print(n)
    n = n + 1
# 1 2 3
```

- **break** leaves the loop straight away.

```
total = 0
while True:
    total = total + 10
    if total >= 30:
        break
print(total)
# 30
```

5.4 ■ Nested loops

A loop inside another loop is a nested loop. The inner loop runs fully for **each** turn of the outer loop.

```
for row in range(3):
    line = ""
    for col in range(3):
        line = line + "*"
    print(line)
# ***
# ***
# ***
```

Common mistakes

- **range(n)** goes from 0 to $n - 1$, not 1 to n – the classic off-by-one error.
- Changing a list while looping over it skips items; loop over a copy if you must remove things.
- Forgetting to change the loop variable in a **while** makes it run forever.

- Indentation decides what is inside the loop; a wrongly-indented line runs only once, after the loop.

6 Lists & 2-D lists

6.1 ■ Lists

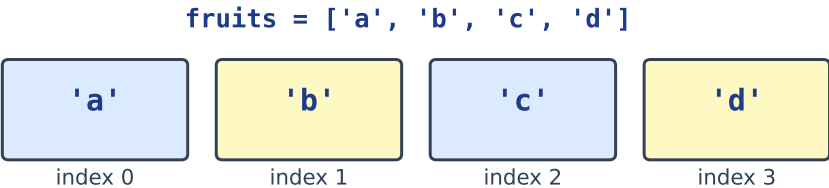
A list holds many values in order, inside []. Each item has an index (from 0).

```
scores = [88, 71, 95]
print(scores[0])      # 88
print(len(scores))    # 3
scores.append(60)      # add to the end
print(scores)          # [88, 71, 95, 60]
```

- Change an item by its index: `scores[1] = 100`.
- A list can grow and shrink; a string cannot.

The everyday list tools:

Tool	Does
<code>a.append(x)</code>	adds <code>x</code> at the end
<code>a.insert(i, x)</code>	inserts <code>x</code> at position <code>i</code>
<code>a.remove(x)</code>	removes the first <code>x</code>
<code>a.pop()</code> / <code>a.pop(i)</code>	removes and returns the last item / item <code>i</code>
<code>a.sort()</code>	sorts the list in place
<code>sorted(a)</code>	returns a NEW sorted list
<code>x in a</code>	is <code>x</code> in the list?
<code>len(a)</code> , <code>sum(a)</code> , <code>max(a)</code> , <code>min(a)</code>	size and quick maths



Exam tip: first item is index 0; last is len(list)-1; negative indices count from the end

List indices start at 0

6.2 ■ Traversing a list

To traverse a list is to visit each item. A `for` loop does this with no index needed.

```
scores = [88, 71, 95]
total = 0
for s in scores:
    total = total + s
print(total)          # 254
```

- Use `enumerate` when you also need the index.

```
for i, name in enumerate(["a", "b"]):
    print(i, name)      # 0 a / 1 b
```

6.3 ■ 2-D lists (grids)

A 2-D list is a list of lists – a grid of rows and columns. Use two indexes: `grid[row][col]`.

```
grid = [[1, 2, 3],
        [4, 5, 6]]
print(grid[0][2])      # 3
print(grid[1][0])      # 4
```

- A nested loop visits every cell.

```
grid = [[1, 2], [3, 4]]
for row in grid:
    for value in row:
        print(value, end=" ")
print()                # 1 2 3 4
```

6.4 ■ List comprehensions

A list comprehension builds a new list in one line: `[expression for item in sequence]`.

```
squares = [x * x for x in range(5)]
print(squares)         # [0, 1, 4, 9, 16]
```

- Add `if` to keep only some items.

```
evens = [n for n in range(10) if n % 2 == 0]
print(evens)           # [0, 2, 4, 6, 8]
```

6.5 ■ Tuples & sets

A tuple is a fixed sequence in round brackets. It cannot be changed after it is made – use one for values that belong together, and unpack it into names.

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

A function that needs to hand back two results returns a tuple:

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

lo, hi = min_max([5, 2, 9])
print(lo, hi)         # 2 9
```

A set stores each value once, with no order. It is perfect for removing duplicates and for fast membership tests.

```
votes = ["red", "blue", "red", "green", "red"]
colours = set(votes)
print(len(colours))   # 3 (duplicates removed)
print("blue" in colours) # True
```

Common mistakes

- `b = a` shares one list. `a.copy()` or `a[:]` creates a new outer list, but nested mutable items remain shared (a shallow copy).
- The last item is `a[-1]`; `a[len(a)]` is out of range.
- `append` adds ONE item; use `extend` or `+` to join another list.
- Building a grid with `[[0]*3]*3` makes three copies of the **same** row. Build the rows in a loop instead.
- A tuple with one item needs a comma: `(5,)`, not `(5)`.
- A set has no order and no duplicates, so you cannot index it with `s[0]`.

7 Dictionaries

A dictionary (dict) stores key → value pairs. You look up a value by its key, not by a number index.

```
student = {"name": "Mei", "score": 88}
print(student["name"])    # Mei
print(student["score"])   # 88
```

7.1 ■ Add and update

Assign to a key to add it, or to change an existing one.

```
student = {"name": "Mei"}
student["score"] = 88      # add a new key
student["score"] = 90      # update the value
print(student)             # {'name': 'Mei', 'score': 90}
```

7.2 ■ Check and loop

Use `in` to test for a key. Loop over the keys, or over `.items()` to get both key and value.

```
student = {"name": "Mei", "score": 90}
print("score" in student)  # True
for key, value in student.items():
    print(key, "=", value)
# name = Mei
# score = 90
```

- `.get(key, default)` returns a default when the key is missing – no error.

```
student = {"name": "Mei"}
print(student.get("age", 0))  # 0
```

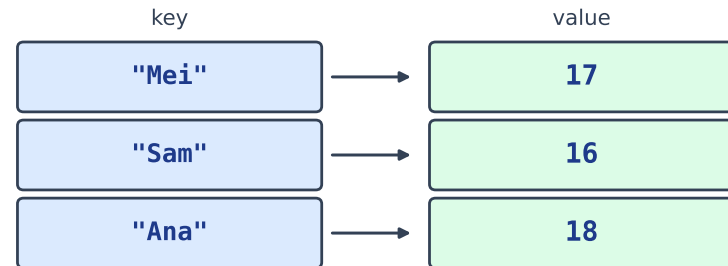
The classic exam pattern – tally how often each value appears:

```
word = "banana"
counts = {}
for letter in word:
    counts[letter] = counts.get(letter, 0) + 1
print(counts)  # {'b': 1, 'a': 3, 'n': 2}
```

- `.get(letter, 0)` supplies 0 the first time a key is seen, so there is no `KeyError`.

Common mistakes

- Reading a missing key with `d[key]` raises a `KeyError`; use `d.get(key)` or test if `key in d` first.
- Assigning `d[key]` again overwrites the old value – keys are unique.
- Keys must be immutable, such as a string or number – a list cannot be a key.



Exam tip: `ages["Mei"]` looks up the value; keys must be unique and immutable

A dictionary maps each key to one value

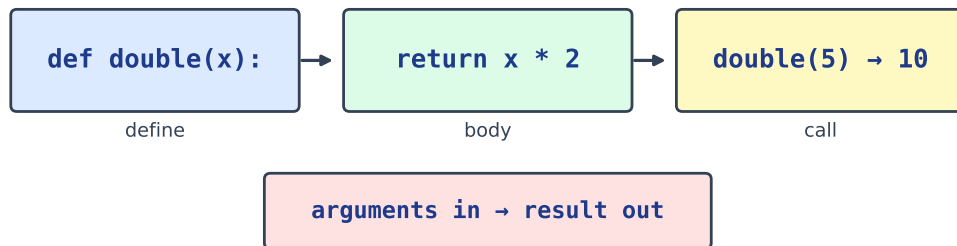
8 Functions & abstraction

8.1 ■ Defining & calling functions

A function is a named block of code you can reuse. Define it with `def`, then call it by name.

```
def greet():  
    print("Hello!")  
  
greet()      # Hello!  
greet()      # Hello!
```

- The code inside runs only when you call the function.



Exam tip: `def` creates the function; `call` runs it; `return` sends a value back to the caller

def defines; call runs; return sends a value back

8.2 ■ Return values

A function can return a value with `return`. The call then stands for that value.

```
def square(n):  
    return n * n  
  
print(square(5))      # 25  
print(square(3) + 1)  # 10
```

- `return` ends the function at once. A function with no `return` gives `None`.

8.3 ■ Parameters, arguments & scope

A parameter is the name in the `def`. An argument is the value you pass in.

```
def power(base, exp):      # base, exp are parameters  
    return base ** exp  
  
print(power(2, 3))        # 8 (2 and 3 are arguments)
```

A variable made inside a function is local – it exists only there. That region is its scope.

```
def f():  
    x = 10      # local to f  
    return x  
  
print(f())      # 10  
# print(x) here would be an error: x is not defined outside f
```

A parameter can have a default value, used when the caller leaves it out:

```
def greet(name, greeting="Hello"):  
    return greeting + ", " + name  
  
print(greet("Mei"))      # Hello, Mei  
print(greet("Sam", "Welcome")) # Welcome, Sam
```

8.4 ■ Procedural abstraction

Procedural abstraction means hiding details behind a name. You use a function by its name and what it does, not by how it works.

```
def area_of_rectangle(w, h):  
    return w * h  
  
print(area_of_rectangle(4, 5)) # 20
```

- A good function does one job, has a clear name, and avoids repeating code.

8.5 ■ Modules & imports

A module is a file of ready-made functions. Bring one in with `import`.

```
import random
random.seed(0)           # makes the result repeatable
print(random.randint(1, 6)) # a dice roll

import math
print(math.sqrt(16))      # 4.0
```

Common mistakes

- A function returns `None` unless you write `return`. Printing is not the same as returning.
- Never use a mutable default like `def f(x=[])` – the same list is shared between all calls.
- A variable made inside a function is local and cannot be seen outside it.
- Run a function with `f()`; writing `f` on its own only names it.

9 Errors, exceptions & testing

9.1 ■ Errors & debugging

Code can fail in three ways. A syntax error breaks Python's rules, so it never runs. A runtime error happens while running, like dividing by zero. A logic error runs but gives the wrong answer.



Exam tip: read the last line of the traceback first — it names the error and the line

Common Python errors: Syntax, Name, Type, Index

```
# A runtime error, caught so this block still finishes:
try:
    print(10 / 0)
except ZeroDivisionError:
    print("cannot divide by zero")
# cannot divide by zero
```

- Python prints a traceback showing where it failed. Read it bottom-up.

9.2 ■ try / except / raise

Put risky code in `try`; if it fails, `except` catches the exception instead of a crash.

```
def to_int(text):
    try:
        return int(text)
    except ValueError:
        return 0
print(to_int("abc")) # 0
```

- Catch a specific type (`ValueError`, `ZeroDivisionError`, ...).
- `raise` makes your own error on purpose.

```
def set_age(age):
    if age < 0:
        raise ValueError("age cannot be negative")
    return age

try:
    set_age(-1)
except ValueError as err:
    print("error:", err)
# error: age cannot be negative
```

9.3 ■ Testing & robustness

A test checks that code gives the right answer. Try normal cases **and** edge cases – empty input, zero, very large values.

```
def average(nums):
    if len(nums) == 0:          # edge case: empty list
        return 0
    return sum(nums) / len(nums)

print(average([2, 4, 6]))      # 4.0
print(average([]))             # 0
```

- Robust code handles strange input gracefully instead of crashing.

Common mistakes

- Do not use a bare `except:` – catch the specific error, e.g. `except ValueError:`.
- A syntax error stops the whole program before it runs, so fix those first.
- Test the edge cases (empty input, zero, the largest value), not just the easy one.

10 Files

10.1 ■ Reading & writing text files

A text file stores text on disk. Open it with `open(name, mode)` where `mode` says read or write. Always use `with`, which closes the file for you.

10.1.1 Writing

Mode "w" writes a new file and erases any old one.

```
with open("notes.txt", "w") as f:
    f.write("first line\n")
    f.write("second line\n")
print("saved")          # saved
```

10.1.2 Reading

Mode "r" (the default) reads. `.read()` returns the whole file as one string.

```
with open("notes.txt", "w") as f:
    f.write("hello\nworld\n")
with open("notes.txt") as f:
    print(f.read().strip())  # hello / world
```

10.1.3 Line by line

Loop over the file to get one line at a time. `.strip()` removes the newline at the end.

```
with open("data.txt", "w") as f:
    f.write("Mei,88\nSam,71\n")
with open("data.txt") as f:
    for line in f:
        name, score = line.strip().split(",")
        print(name, "scored", score)
# Mei scored 88
# Sam scored 71
```

10.1.4 Appending

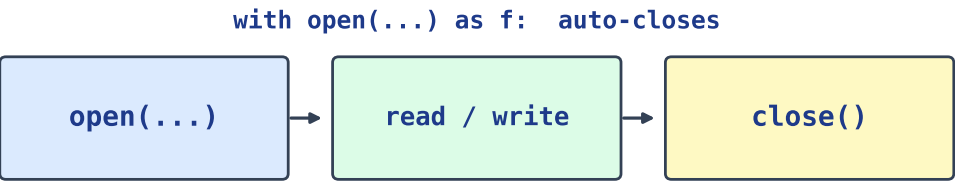
Mode "a" appends – it adds to the end without erasing.

```
with open("log.txt", "w") as f:
    f.write("line 1\n")
with open("log.txt", "a") as f:
    f.write("line 2\n")
with open("log.txt") as f:
    print(f.read().strip())    # line 1 / line 2
```

Mode	Meaning
"r"	read (default)
"w"	write (erases first)
"a"	append (add to the end)

Common mistakes

- Always close a file, or use `with open(...) as f:` which closes it for you.
- `read()` gives the whole file as one string, and each line still ends with `\n`.
- Opening with "w" erases the file first; use "a" to add to the end.



Exam tip: mode 'r' reads, 'w' writes (overwrites); prefer with so the file always closes

open → read/write → close (with auto-closes)

11 Algorithm design

11.1 ■ Algorithms & decomposition

An algorithm is a clear list of steps that solves a problem. Decomposition means breaking a big problem into smaller parts you can solve one at a time.

```
# Algorithm: find the largest number in a list
def largest(nums):
    best = nums[0]
    for n in nums:
        if n > best:
            best = n
    return best

print(largest([3, 9, 2, 7]))    # 9
```

- Abstraction means ignoring detail: you use `largest(...)` without re-reading how it works.

11.2 ■ Pseudocode & flowcharts

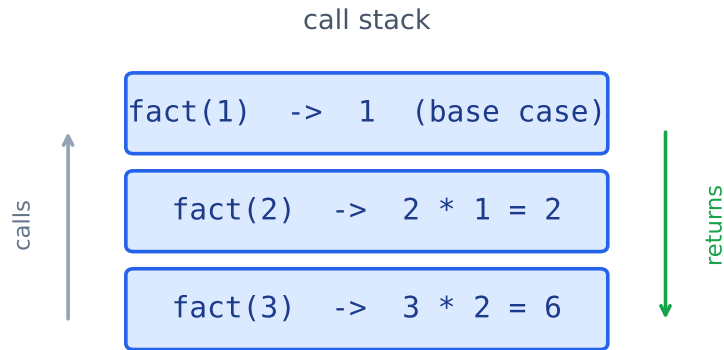
Pseudocode is plain, structured English for an algorithm, written before the real code. It is not run.

```
SET best TO first number
FOR Each number n
    IF n > best
        THEN
            SET best TO n
OUTPUT best
```

A flowchart draws the same plan: a box for each step, a diamond for each decision, and arrows for the order.

11.3 ■ Recursion & the call stack

Recursion is when a function calls itself. It needs a base case (a simple input that returns at once) and a recursive case (it calls itself on a smaller input).



call stack: newest frame on top · returns unwind

The call stack for factorial(3): each call waits, then returns in reverse order

```
def fact(n):
    return 1 if n <= 1 else n * fact(n - 1)

print(fact(5))    # 120
```

- Each paused call sits on the call stack until the call above it returns.

Common mistakes

- Recursion needs a base case, or it calls itself forever and crashes the call stack.
- Pseudocode is for planning – it need not run, but every step must be unambiguous.
- Break a big problem into small named steps before you write any code.

12 Data structures

12.1 ■ Abstract Data Types (ADTs)

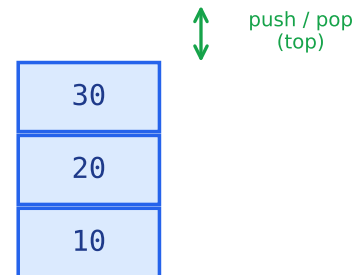
An abstract data type (ADT) describes some data plus the operations on it, separate from how it is built. You use it through its operations, not through its inner storage.

```
# A stack ADT, built on a list
s = []
s.append(1)      # add
s.append(2)
print(s.pop())   # 2 (remove the most recent)
```

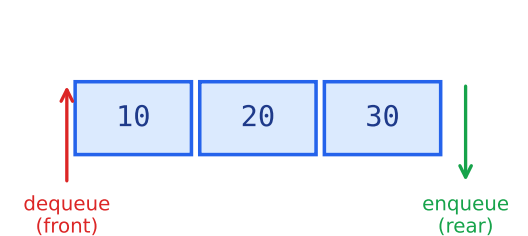
12.2 ■ Stacks

A stack is last-in, first-out (LIFO). You push onto the top and pop from the top.

Stack (LIFO)



Queue (FIFO)



stack: LIFO · queue: FIFO

A stack removes from the top (LIFO); a queue removes from the front (FIFO)

```
stack = []
stack.append("a")
stack.append("b")
print(stack.pop()) # b
print(stack.pop()) # a
```

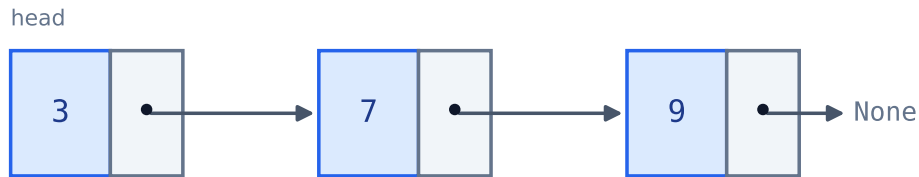
12.3 ■ Queues

A queue is first-in, first-out (FIFO). You enqueue at the back and dequeue from the front.

```
queue = []
queue.append("a")    # enqueue
queue.append("b")
print(queue.pop(0))  # a (dequeue the front)
print(queue.pop(0))  # b
```

12.4 ■ Linked lists

A linked list is a chain of nodes. Each node holds data and a pointer to the next node; the last points to `None`.



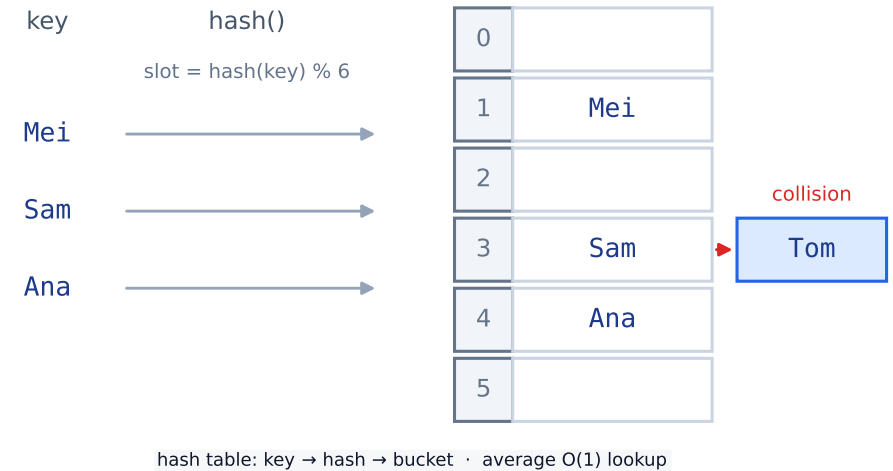
linked list: node → next · O(1) insert at head

A linked list: each node holds data and a pointer to the next node, ending at None

```
n3 = {"data": 3, "next": None}
n2 = {"data": 2, "next": n3}
n1 = {"data": 1, "next": n2}
node = n1
while node is not None:    # traverse to the end
    print(node["data"])
    node = node["next"]
# 1 2 3
```

12.5 ■ Hash tables

A hash table maps a key to a slot with a hash function. Two keys can land in the same slot – a collision. Python's `dict` is a hash table, so lookup is fast.

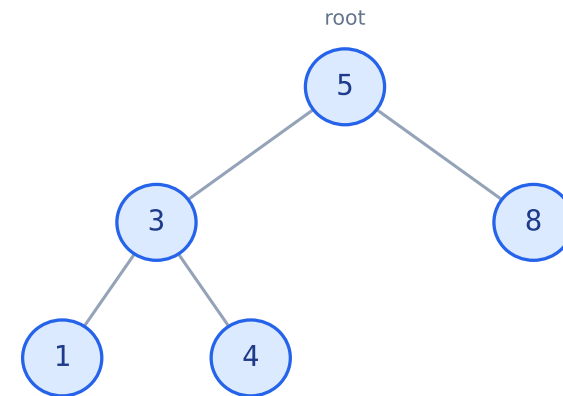


A hash function maps each key to a slot; two keys can collide in one slot

```
table = {}
table["Mei"] = 88
table["Sam"] = 71
print(table["Mei"])    # 88 (fast lookup by key)
```

12.6 ■ Binary search trees

A binary search tree (BST) keeps order: every left child is smaller than its node, every right child is larger. Search stays fast.



in-order traversal: 1 3 4 5 8

BST: left < node < right · search O(log n) if balanced

A binary search tree: smaller values go left, larger values go right

```
def insert(root, val):
    if root is None:
        return {"val": val, "left": None, "right": None}
    if val < root["val"]:
        root["left"] = insert(root["left"], val)
    else:
        root["right"] = insert(root["right"], val)
    return root

def inorder(root):
    if root is None:
        return []
    return inorder(root["left"]) + [root["val"]] + inorder(root["right"])

tree = None
for v in [5, 3, 8, 1, 4]:
    tree = insert(tree, v)
print(inorder(tree))  # [1, 3, 4, 5, 8]
```

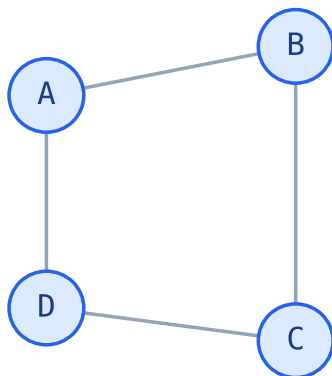
```
graph = {"A": ["B", "D"], "B": ["A", "C"], "C": ["B", "D"], "D": ["A",
↔ "C"]}
for vertex in graph:
    print(vertex, "->", graph[vertex])
# A -> ['B', 'D'] (and so on for B, C, D)
```

Common mistakes

- A stack is last-in-first-out; a queue is first-in-first-out. Do not mix them up.
- Check whether a structure is empty before you pop or dequeue from it.
- In a linked list, losing the head pointer loses the whole list.

12.7 ■ Graphs

A graph is a set of vertices joined by edges. An adjacency list dict of neighbour lists – is a common way to store one. – a



```
graph = {
    "A": ["B", "D"],
    "B": ["A", "C"],
    "C": ["B", "D"],
    "D": ["A", "C"],
}
```

graph: vertices + edges · BFS/DFS traverse

A graph of vertices and edges, with its adjacency-list form

13 Searching, sorting & efficiency

13.1 ■ Linear & binary search

A search finds where a value is. Linear search checks each item in turn, so it works on any list.

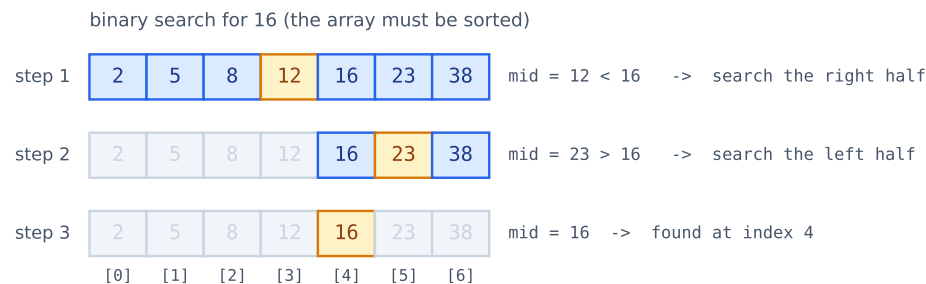
```
def linear_search(items, target):
    for i in range(len(items)):
        if items[i] == target:
            return i
    return -1    # not found

print(linear_search([4, 8, 2, 9], 2))    # 2
```

Binary search is much faster but needs a **sorted** list. It halves the range each step.

```
def binary_search(items, target):
    lo, hi = 0, len(items) - 1
    while lo <= hi:
        mid = (lo + hi) // 2
        if items[mid] == target:
            return mid
        elif items[mid] < target:
            lo = mid + 1
        else:
            hi = mid - 1
    return -1

print(binary_search([1, 3, 5, 7, 9], 7))    # 3
```



binary search: halve the range each step · $O(\log n)$

Binary search halves the range each step – $O(\log n)$ on a sorted list

13.2 ■ Sorting (bubble & insertion)

To sort is to put items in order. Bubble sort repeatedly swaps neighbours that are in the wrong order.

```
def bubble_sort(a):
    a = a[:]    # work on a copy
    for i in range(len(a)):
        for j in range(len(a) - 1 - i):
            if a[j] > a[j + 1]:
                a[j], a[j + 1] = a[j + 1], a[j]
    return a

print(bubble_sort([5, 2, 4, 1]))    # [1, 2, 4, 5]
```

Insertion sort builds a sorted part one item at a time, sliding each new item back into its place:

```
def insertion_sort(a):
    a = a[:]    # work on a copy
    for i in range(1, len(a)):
        key = a[i]
        j = i - 1
        while j >= 0 and a[j] > key:    # shift bigger values right
            a[j + 1] = a[j]
            j -= 1
        a[j + 1] = key    # drop key into the gap
    return a

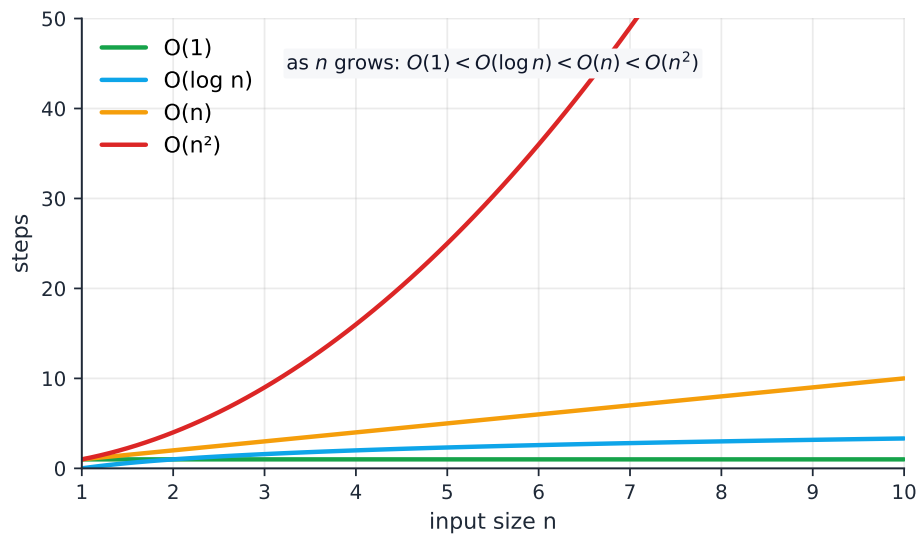
print(insertion_sort([5, 2, 4, 1]))    # [1, 2, 4, 5]
```

- In real code, use Python's built-in `sorted()`:

```
print(sorted([5, 2, 4, 1]))    # [1, 2, 4, 5]
```

13.3 ■ Algorithmic efficiency

Efficiency asks how the work grows as the input grows. We describe it with Big-O.



How the number of steps grows with input size for common complexities

Big-O	Name	Example
$O(1)$	constant	look up a dict key
$O(\log n)$	logarithmic	binary search
$O(n)$	linear	linear search
$O(n^2)$	quadratic	bubble sort

```
def steps(n):          # how many steps a linear scan takes
    count = 0
    for i in range(n):
        count = count + 1
    return count

print(steps(100))      # 100 -> O(n)
```

13.4 ■ Randomness & simulation

The `random` module makes random numbers. Use a seed to make results repeat-able. A simulation runs many random trials to estimate an answer.

```
import random
random.seed(0)
rolls = [random.randint(1, 6) for _ in range(1000)]
print(rolls.count(6))  # about 1/6 of 1000
```

Common mistakes

- Binary search only works on a **sorted** list.
- Big-O tells you how the time GROWS, not the exact time; an $O(n^2)$ method beats $O(n)$ only for tiny inputs.
- Bubble sort is $O(n^2)$ – fine for learning, but slow on large lists.

14 OOP & paradigms

14.1 ■ Classes & objects

A class is a blueprint. An object is one thing built from it (an instance). `__init__` is the constructor that sets up each object; `self` is the object itself.

```
class Dog:
    def __init__(self, name):
        self.name = name          # an attribute
    def speak(self):
        return self.name + " says woof"

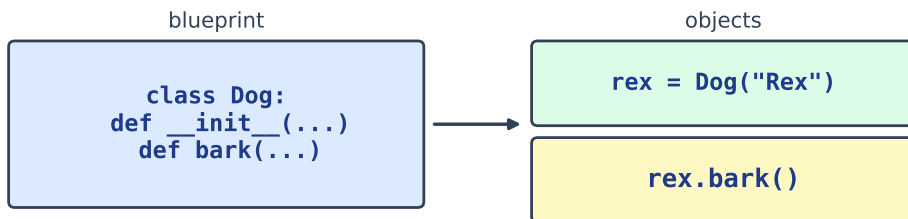
d = Dog("Rex")
print(d.speak())                # Rex says woof
```

- `name` is an attribute (data on the object); `speak` is a method (an action).

Add `__str__` to control what `print(obj)` shows:

```
class Dog:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return f"Dog named {self.name}"

print(Dog("Rex"))               # Dog named Rex
```



Exam tip: class defines the type; calling `ClassName(...)` creates an object (instance)

A class is a blueprint; calling it creates an object

14.2 ■ Inheritance, encapsulation & polymorphism

Inheritance lets a subclass reuse a superclass. Use `super()` to call the parent; override a method to change it.

```
class Animal:
    def speak(self):
        return "some sound"

class Cat(Animal):
    def speak(self):                # override
        return "meow"

print(Cat().speak())              # meow
```

Encapsulation groups data with methods and exposes an intended interface. In Python, a leading underscore marks an intended non-public (often called private) name by convention; it does not prevent outside code from accessing it. Double-leading-underscore name mangling helps avoid accidental subclass name clashes, not enforce security.

```
class Account:
    def __init__(self):
        self._balance = 0          # non-public by convention
    def deposit(self, n):
        self._balance += n
    def balance(self):
        return self._balance

a = Account()
a.deposit(50)
print(a.balance())                # 50
```

Polymorphism means one name, many behaviours – the right `speak` runs for each object.

```
class Cat:
    def speak(self):
        return "meow"

class Cow:
    def speak(self):
        return "moo"

for animal in [Cat(), Cow()]:
    print(animal.speak())          # meow, then moo
```

14.3 ■ Programming paradigms

A paradigm is a style of writing programs. Procedural code is a sequence of steps and functions. Object-oriented code groups data and methods into objects. Declarative code says **what** you want, not how (a list comprehension or SQL).

```
def total(nums):           # procedural
    t = 0
    for n in nums:
        t += n
    return t
print(total([1, 2, 3]))    # 6

print(sum([1, 2, 3]))      # 6 (declarative: same result)
```

Common mistakes

- Every method needs `self` as its first parameter.
- `__init__` sets up a new object and runs automatically when you create one.
- Two objects of the same class have separate attributes; changing one does not change the other.

15 Data representation

15.1 ■ Bits & binary

A bit is a single 0 or 1. Binary is the base-2 number system: each place is worth twice the one to its right (1, 2, 4, 8, ...). Denary (base-10) is our normal numbers.

place	8	4	2	1
	1	1	0	1

$$8 + 4 + 1 = 13$$

binary: powers of 2 · bit k has weight 2^k

Binary place values: 1101 means $8 + 4 + 1 = 13$

```
print(bin(13))           # 0b1101
print(int("1101", 2))    # 13
```

- 8 bits make a byte . A fixed width can overflow (wrap around) when the number is too big.

```
x = 250
x = (x + 10) % 256        # one byte wraps at 256
print(x)                  # 4
```

Hexadecimal (base 16) is a compact way to read binary: one hex digit stands for exactly four bits. Python writes hex with `0x`:

```
print(hex(255))           # 0xff
print(0xFF)               # 255
print(int("ff", 16))      # 255
```

15.2 ■ Compression

Compression makes data smaller. Lossless compression keeps every bit, so you rebuild the original exactly. Lossy compression throws away detail – smaller but not exact – and is used for photos and music.

Run-length encoding is a simple lossless method: store a run (a repeat) as a count plus the value.

```
def rle(text):
    out = ""
    i = 0
    while i < len(text):
        run = 1
        while i + run < len(text) and text[i + run] == text[i]:
            run += 1
        out += str(run) + text[i]
        i += run
    return out

print(rle("AAAABBBCCD")) # 4A3B2C1D
```

Common mistakes

- n bits store 2**n different values, from 0 up to 2**n - 1.
- Lossy compression throws away detail and cannot be undone; lossless can be reversed exactly.

16 Computing concepts

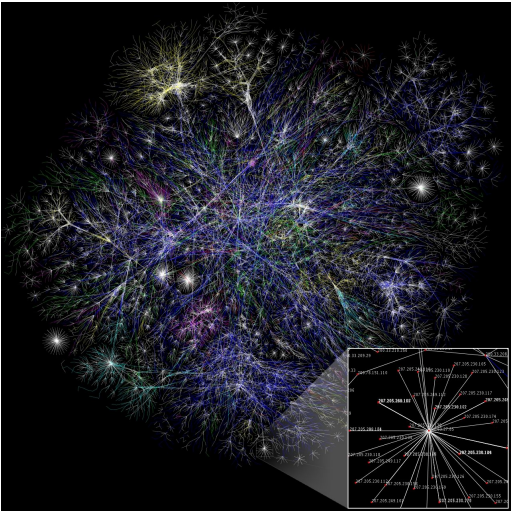
16.1 ■ What computing is & the design cycle

Computing means solving problems with computers: input, process, output. Good software is built in a design cycle – plan, write, test, improve – repeated many times.

- Break a problem down, build a small part, test it, then add more.
- Programmers work in teams and reuse each other’s code.

16.2 ■ The Internet

The Internet is a network of networks. Data is split into packets that travel separately and are put back together at the other end. Shared rules called protocols (such as TCP/IP) make this work. If one path breaks, packets take another route – this is redundancy, which gives fault tolerance.



A map of the Internet: each line is a path between two networks

Layer	Job
HTTP	request and send web pages
TCP	reliable delivery, in order
IP	addressing and routing

16.3 ■ Parallel & distributed computing

Sequential code does one step at a time. Parallel computing does several steps at once on many cores, which can give a speedup. Distributed computing spreads the work across many computers, such as a cloud.

- Not everything can run in parallel: some steps must wait for an earlier result.

16.4 ■ Impact of computing

Computing brings both benefits and harms. The digital divide means not everyone has equal access to it. Software can carry bias from the data it learns from. Respect intellectual property (licences), and protect people's personal data and privacy.

Common mistakes

- The Internet and the World Wide Web are not the same: the Web is one service that runs on top of the Internet.
- More processor cores help only if the work can be split into parts that run at the same time.

17 Putting it together

17.1 ■ End-to-end mini-projects

A mini-project combines earlier ideas: data in a list, a function with selection inside a loop, and printed output. This is also the shape of the AP Create Performance Task.

17.1.1 Project: average mark

```
def average(marks):  
    if len(marks) == 0:  
        return 0  
    return round(sum(marks) / len(marks), 1)  
  
print(average([88, 71, 95, 60])) # 78.5
```

17.1.2 Project: count passes

```
def count_passes(marks, pass_mark=60):  
    passes = 0  
    for m in marks: # iteration  
        if m >= pass_mark: # selection  
            passes += 1  
    return passes  
  
print(count_passes([88, 50, 95, 60])) # 3
```

17.1.3 Project: filter to a new list

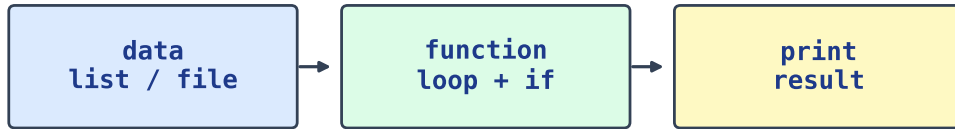
```
def merit(marks):  
    return [m for m in marks if m >= 80]  
  
print(merit([88, 71, 95, 60])) # [88, 95]
```

The AP Create Task wants a list, a parameterised procedure that uses selection and iteration, and some input/output. Each project above is exactly that shape – build small pieces, then join them.

Common mistakes

- Build in small steps and test each part before moving on – do not write it all at once.
- Read the whole task first, then plan the input → process → output before you code.

Mini-project pipeline



Exam tip: combine list + function + selection + loop + clear output—the Create task shape

Mini-project: data → function → printed result