PROGRAMMING FOUNDATIONS FOR ANALYTICS

Pocket Reference: Topics 3 & 4

Built-In Functions and Logic & Control Flow on one sheet. Keep it open beside your Jupyter notebook — look up the syntax, copy the pattern, change the values.

- Topic 3 · Built-In Functions
- Topic 4 · Logic & Control Flow
- Common mistake — read twice

03 Built-In Functions

Functions & methods · lists · strings · f-strings · libraries

3.1 Function or method? Look for the dot

A **function** is a general helper you call on its own. A **method** belongs to an object and is reached through a dot.

SYNTAX

```
result = function_name(object)   # function
object.method_name()             # method
```

FUNCTION	METHOD
`sorted(my_list)`	`my_list.sort()`
`len("hello")`	`"hello".upper()`
`max(daily_sales)`	`daily_sales.append(190)`

You never need to know **how** a function is written — only what goes in and what comes out.

3.2 round(), max() and help()

```
daily_sales = [120, 135, 142, 128, 180, 165, 190]

best = max(daily_sales)          # 190
average = sum(daily_sales) / 7   # 151.42857142857142
round(average, 2)                # 151.43
```

SYNTAX

```
round(number, ndigits)
```

`ndigits` is optional. Leave it out and you get a whole number: `round(1.58)` → `2`.

Forgot how a function works? Run `help(round)` in a cell. Ask Copilot too, then check its answer against `help()`.

3.3 sorted() vs .sort()

This is the one that catches everyone. **sorted()** gives you a new list. **.sort()** rearranges the original and returns **None**.

```
full = [50, 20, 80]
a = sorted(full)
print(a)      # [20, 50, 80]
print(full)   # [50, 20, 80]  ← untouched
```

```
full = [50, 20, 80]
full.sort()
print(full)   # [20, 50, 80]  ← original order gone
print(full.sort())  # None  ← don't print the call!
```

Both accept `reverse=True` for descending order. `reversed(x)` flips the order without sorting — wrap it in `list()` to see it.

3.4 List methods — the full toolbox

METHOD	DOES
`.append(x)`	Adds x to the end
`.insert(i, x)`	Inserts x at index i
`.pop(i)`	Removes item at i *and hands it back*
`.remove(x)`	Removes the first item equal to x
`.index(x)`	Position of the first item equal to x
`.count(x)`	How many times x appears

```
condo = [53, 72, 111, 152, 221]
condo.index(111)   # 2
condo.count(221)   # 1
list(reversed(condo))  # [221, 152, 111, 72, 53]
```

3.5 try–except: handle a missing item

`.index()` raises a **ValueError** when the item isn't in the list. Catch it instead of letting the program crash.

```
try:
    pos = name_list.index(name)
    print(f"Found in position {pos + 1}")
except ValueError:
    print(f"Name {name} is not found")
```

> Watch the **colon** at the end of each header line and the **4-space indent** underneath. Miss either and you get an IndentationError or SyntaxError.

Python runs the `try` block first. Only if a ValueError happens does it jump to `except`.

3.6 String methods

```
a = " HellO Python "
```

METHOD	DOES	RESULT
`a.upper()`	All capitals	`" HELLO PYTHON "`
`a.lower()`	All small letters	`" hello python "`
`a.strip()`	Trims spaces at both ends	`"HellO Python"`
`a.count('o')`	Counts a smaller string	`1`
`a.find('on')`	Index of first match	`11`
`a.find('ON')`	Not found	`-1`

> **Strings are immutable.** These methods return a **new** string — `a.upper()` alone changes nothing. Store it: `drink = drink.upper()`

3.7 split() and join()

The bridge between text and lists: break text apart, glue a list together.

```
address = "Orchard Road, Istana, Singapore 238823"

parts = address.split(", ")
# ['Orchard Road', 'Istana', 'Singapore 238823']

" | ".join(parts)
# 'Orchard Road | Istana | Singapore 238823'
```

> **Read join() backwards:** the separator goes first, the list goes inside the brackets.

This matters later: a CSV file is just text split on commas — exactly what Pandas does for you in Topic 7B.

3.8 f-string format specifiers

Everything after the `:` inside the braces controls how the value looks.

CODE	EXAMPLE	OUTPUT
`:.2f`	`f"{4.5:.2f}"`	`4.50`
`:,`	`f"{1234567:,}"`	`1,234,567`
`:<10`	`f"[{'teh':<10}]"`	`[teh       ]`
`:>10`	`f"[{'teh':>10}]"`	`[       teh]`
`:^10`	`f"[{'teh':^10}]"`	`[   teh    ]`
`:*^10`	`f"[{'teh':*^10}]"`	`[***teh****]`

They stack, in this order: **fill, align, width, comma, decimals**.

```
f"{9876.5:>10,.2f}"   # '  9,876.50'
```

3.9 Libraries: math and random

A library is ready-made code someone else wrote. `math` and `random` ship with Python — no install needed — but you must **import** them in your notebook before use.

IMPORT STYLE	YOU THEN WRITE	USE WHEN
`import math`	`math.pi, math.sqrt(9)`	You want the whole package (safest, clearest)
`from math import pi`	`pi`	You need one or two specific items
`from math import radians as rad`	`rad(90)`	The real name is long or clashes

```
import math
circumference = 2 * math.pi * radius
area = math.pi * radius ** 2
```

RANDOM	GIVES YOU
`random()`	A float from 0 up to (not including) 1
`randint(a, b)`	A whole number from a to b — **b included**
`choice(a_list)`	One randomly picked item
`shuffle(a_list)`	Shuffles the list in place (returns None)
`seed(n)`	Same seed → same "random" numbers. Use it while testing.

```
from random import choice
winner = choice(["Siti", "Wei Ming", "Kumar", "Mary"])
```

04 Logic & Control Flow

Comparisons · and/or/not · if–elif–else · nested if · ternary

4.1 Comparison operators

Every comparison answers with a **bool** — `True` or `False`. Nothing else.

OPERATOR	ASKS	EXAMPLE
`==`	Equal to?	`3 == 3 → True`
`!=`	Not equal to?	`3 != 5 → True`
`> <`	Greater / less than?	`5 > 8 → False`
`>= <=`	At least / at most?	`7 >= 7 → True`

`=` puts a value into a variable. `==` asks a question. Using `=` inside an `if` is a SyntaxError.

4.2 and · or · not

OPERATOR	TRUE WHEN	EXAMPLE
`and`	**Both** sides are True	`price < 500000 and area >= 70`
`or`	**At least one** side is True	`tiny or huge`
`not`	Flips the answer	`not sold_out`

Order Python works in: comparisons first, then `not`, then `and`, then `or`.

```
age >= 60 or member and weekend
# Python reads this as:
age >= 60 or (member and weekend)
```

Add brackets whenever you mix `and` with `or`. Your reader shouldn't have to remember the rule.

4.3 if / elif / else

SYNTAX

```
if condition_1:
    # runs when condition_1 is True
elif condition_2:
    # runs only if the one above was False
else:
    # runs when nothing above matched
```

- [ ] **Colon** at the end of every `if`, `elif` and `else` line.
- [ ] **Indent 4 spaces** for the block underneath.
- [ ] **elif and else are optional** — an `if` can stand alone.

```
flat = "two_room"

if flat == "three_room":
    print("A 3 room flat")
else:
    print("Looking elsewhere")
```

4.4 First match wins — then it stops

Python checks each condition **in order**, runs the first one that is True, and skips the rest.

```
if area > 70:
    print("big place!")
elif area > 50:
    print("medium, nice!")
else:
    print("pretty small!")
```

AREA	CHECKS	PRINTS
80	80 > 70 ✓	big place!
56	56 > 70 ✗ → 56 > 50 ✓	medium, nice!
45	both ✗	pretty small!

Order matters. Put the **narrowest** condition first — if you check `area > 50` before `area > 70`, the second one can never run.

4.5 Nested if — a decision inside a decision

Use it when the second question only makes sense once the first is answered.

```
if flat == "four_room":
    if area > 70:
        print("A big 4 room")
    else:
        print("A cosy 4 room")
else:
    print("Not a 4 room")
```

The inner `if` is indented **one more level** (8 spaces). It only runs when the outer condition is True.

IndentationError almost always means a block is indented by a different amount from its neighbours. Keep every level at 4 spaces and never mix tabs with spaces.

4.6 Ternary — a decision on one line

SYNTAX

```
value = value_if_true if condition else
value_if_false
```

Read it in the middle first: *"give me a, if a < b, otherwise b."*

LONG WAY	TERNARY

`if a < b:`
`    smaller = a`
`else:`
`    smaller = b`
	`smaller = a if a < b else b`

> Only the **value** goes on each side — never a second `=`. `smaller = a if a < b else smaller = b` is a SyntaxError.

Use a ternary to **pick one value**. If you need to run several statements, use a normal `if`.

05 When to use what

Pick the right tool in one glance

Sorting a list

Keep the original order too	`sorted(x)`
Don't need the old order	`x.sort()`
Highest first	`sorted(x, reverse=True)`
Just flip the order	`list(reversed(x))`

If you ever write `x = x.sort()`, your list becomes `None`. Use `sorted()` when you want a result back.

Finding something

Where is it in a list?	`my_list.index(x)`
Where is it in text?	`text.find(sub)`
How many are there?	`.count(x)`
Is it there at all?	`x in my_list`

`.find()` returns **-1** when nothing matches. `.index()` raises a **ValueError** — wrap it in try–except.

Writing a decision

One thing to check	`if`
Two outcomes	`if / else`
Three or more bands	`if / elif / else`
Second question depends on the first	`nested if`
Picking one value	`ternary`

Before you ask for help — run this checklist

- [ ] **Did I import the library?** `math.sqrt()` fails with a NameError until you run `import math`.
- [ ] **Did I store the result?** String and `sorted()` results are new values — `name.upper()` on its own changes nothing.
- [ ] **Did I print a method that returns None?** `print(x.sort())` and `print(x.append(1))` both show `None`.
- [ ] **Is every header line ended with a colon?** `if`, `elif`, `else`, `try`, `except`.
- [ ] **Is every block indented by 4 spaces?** Same amount for every line in the block.
- [ ] **Did I use == and not = inside the if?**

Check yourself

1 · What is printed?

```
x = [3, 1, 2]
print(x.sort())
```

None — `.sort()` changes `x` but hands nothing back.

2 · What is the output?

```
print(f"{1234.5:,.2f}")
```

1,234.50 — comma for thousands, then 2 decimals.

3 · What does `s` become?

```
s = " Kopi "
s.strip()
```

" Kopi " — unchanged. You must write `s = s.strip()`.

4 · What prints when `area = 80`?

```
if area > 50:
    print("medium")
elif area > 70:
    print("big")
```

medium — the first True wins, so "big" can never run. Reorder the conditions.

5 · What is `label`?

```
score = 45
label = "Pass" if score >= 50
else "Fail"
```

"Fail" — the condition is False, so the value after `else` is used.

6 · Which line raises an error?

```
n = [5, 6]
n.index(6)
n.index(9)
```

n.index(9) — ValueError. Wrap it in try–except.
