PROGRAMMING FOUNDATIONS FOR ANALYTICS

Pocket Reference: Topics 5 & 6

Loops, List Comprehensions and User-Defined Functions on one sheet. Keep it open beside your Jupyter notebook — look up the syntax, copy the pattern, change the values.

- Topic 5 · Loops & Comprehensions
- Topic 6 · User-Defined Functions
- Common mistake — read twice

05A Loops

while · for · range · enumerate · nested · break & continue

5A.1 The while loop

Repeats **while a condition stays True**. Use it when you don't know how many rounds you need.

SYNTAX

```
while condition:
    do_something
```

```
balance = 2
while balance < 12:
    balance += 2
    print(f"${balance} left")
print("Enough!")

# $4 left … $12 left, then Enough!
```

The golden rule: something inside the loop must eventually make the condition False. Forget it and the loop runs forever — press the ■ stop button in Jupyter, or Kernel → Interrupt.

5A.2 The for loop

Steps through **every item in a sequence** and stops on its own. No more `list[0]`, `list[1]`, `list[2]`…

SYNTAX

```
for item in sequence:
    do_something
```

```
hdb_flats = [45, 65, 90, 110, 115, 130]

for area in hdb_flats:
    print(area, "sqm")

# 45 sqm / 65 sqm / 90 sqm / … / 130 sqm
```

`area` is just a name you choose. Pick one that describes a **single** item: `for sale in daily_sales`, not `for x in daily_sales`.

Works on lists, strings, and `range()`.

5A.3 Collect results in a new list

The three-step pattern you will use constantly: empty list → loop → append.

SYNTAX

```
new_list = []
for item in sequence:
    result = do_something(item)
    new_list.append(result)
```

```
prices = [4.50, 5.00, 3.80, 6.40]
prices_gst = []

for price in prices:
    price_gst = round(price * 1.09, 2)
    prices_gst.append(price_gst)

print(prices_gst)
# [4.91, 5.45, 4.14, 6.98]
```

Create the empty list **before** the loop, not inside it — otherwise it is wiped clean on every round and you end up with one item.

5A.4 range() — loop a fixed number of times

FORM	GIVES
`range(5)`	`0 1 2 3 4`
`range(2, 6)`	`2 3 4 5`
`range(0, 10, 2)`	`0 2 4 6 8`
`range(5, 0, -1)`	`5 4 3 2 1`

The stop number is always excluded — same rule as slicing. `range(5)` gives five numbers but never reaches 5.

```
for n in range(1, 6):
    print(f"5 x {n} = {5 * n}")
```

Wrap it in `list()` to see what it holds: `list(range(5))`.

5A.5 enumerate() — position and item together

When you need to know **which** item you are on as well as its value.

SYNTAX

```
for index, item in enumerate(sequence):
    do_something
```

```
hdb_flats = [45, 65, 90, 110, 115, 130]

for index, area in enumerate(hdb_flats):
    print(f"Flat {index}: {area} sqm")

# Flat 0: 45 sqm
# Flat 1: 65 sqm  …
```

Counting starts at 0. To start at 1 — handy for menus — use `enumerate(menu, 1)`.

Two names on the left, because each round hands you a **pair**. One name gives you the pair as a tuple instead.

5A.6 Nested loops

For **every** round of the outer loop, the inner loop runs all the way through.

```
for row in range(3):
    for col in range(7):
        print("=", end="")
    print()

# =======
# =======
# =======
```

- [ ] **Outer loop** controls the rows (3 of them).
- [ ] **Inner loop** controls the columns (7 each time).
- [ ] `end=""` stops `print()` from jumping to a new line.
- [ ] **Bare** `print()` after the inner loop ends the row.

Total rounds = outer × inner. Here that is 3 × 7 = 21 prints.

5A.7 break and continue

KEYWORD	DOES	USE WHEN
`break`	Leaves the loop completely	You found what you needed
`continue`	Skips to the next round	This item isn't useful

```
correct_password = "kopi123"

while True:
    password = input("Please enter password: ")
    if password != correct_password:
        print("Wrong password. Try again.")
        continue
    print("Correct password. Login successful!")
    break
```

`while True:` never ends on its own. It **must** contain a `break`, or you have written an infinite loop.

05B List Comprehensions

One line instead of a loop · filter · transform · flatten

5B.1 A loop and an append, on one line

SYNTAX

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

Read it left to right: *"give me **x**2, for each **x** in **range(10)**."*

LOOP WAY	COMPREHENSION

`squares = []`
`for x in range(10):`
 	
`squares.append(x**2)`
	`squares = [x**2 for x in range(10)]`

```
# [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Square brackets are not decoration — they are what makes the result a **list**.

5B.2 Filter — keep only what matches

SYNTAX

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

```
nums = [1, 4, 9, 16, 25, 36]
even_nums = [x for x in nums if x % 2 == 0]
# [4, 16, 36]
```

Items that fail the condition are simply **left out** — the new list is shorter than the old one.

There is **no else** in a filter. The `if` sits at the **end**, with nothing after it.

5B.3 Transform — change every value

No `if` at all. Every item goes in, every item comes out — changed.

```
prices = [4.50, 5.00, 3.80, 6.40]
prices_gst = [round(p * 1.09, 2) for p in prices]
# [4.91, 5.45, 4.14, 6.98]
```

```
words = ["kopi", "teh", "milo"]
[w.upper() for w in words]
# ['KOPI', 'TEH', 'MILO']
```

The new list is always the **same length** as the original.

5B.4 Choose between two values

SYNTAX

```
[ true_value if condition else false_value for
item in iterable ]
```

```
scores = [45, 72, 38, 90]
results = ["Pass" if s >= 50 else "Fail" for s in
scores]
# ['Fail', 'Pass', 'Fail', 'Pass']
```

The position of the `if` is everything. At the **front** with an `else` → labels every item. At the **end** with no `else` → filters items out. Put a filter at the front and Python raises a SyntaxError.

5B.5 Flatten a list of lists

SYNTAX

```
[ expression for sub in outer for item in sub ]
```

```
nested = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[item for sub in nested for item in sub]
# [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Read the two `for` clauses in the **same order** you would write nested loops: outer first, inner second.

Why flatten? A single flat list lets you use `sum()`, `max()`, `min()` and filters across *all* the groups at once instead of group by group.

5B.6 Which shape do I need?

YOU WANT	SHAPE	LENGTH
Change every item	`[f(x) for x in xs]`	Same
Keep some items	`[x for x in xs if c]`	Shorter
Label every item	`[a if c else b for x in xs]`	Same
Flatten groups	`[x for g in gs for x in g]`	Longer

If a comprehension takes more than one line to understand, write the plain loop instead. Short beats clever.

06 User-Defined Functions

def · parameters · return · docstrings · tuples · scope · lambda

6.1 Define it, then call it

SYNTAX

```
def function_name():
    body

function_name()
```

```
def cafe_menu():
    print("1. Latte")
    print("2. Cappuccino")

cafe_menu()   # the call runs the body
```

- [ ] `def` is the keyword that starts a definition.
- [ ] `()` **and** `:` end the header line — always both.
- [ ] **Indent 4 spaces** for the body.
- [ ] **Defining is not running.** Nothing happens until you call it.

6.2 Parameter vs argument

```
import math

def circle_area(radius):      # radius = PARAMETER
    return math.pi * radius**2

circle_area(6)                # 6 = ARGUMENT
```

PARAMETER	ARGUMENT
The placeholder written inside the `def`	The real value you hand in when you call
Waiting to be filled	Here, `radius` becomes 6

6.3 Several parameters

```
def trapezium_area(h, a, b):
    return 0.5 * (a + b) * h
```

For `a=6, b=10, h=4`:

CALL	RESULT	WHY
`trapezium_area(6, 10, 4)`	`42.0`	Wrong — h got 6
`trapezium_area(4, 6, 10)`	`32.0`	Right — order matches
`trapezium_area(a=6, b=10, h=4)`	`32.0`	Right — names win

Positional arguments must be in order. **Keyword** arguments can be in any order. **Default** values cover the usual case: `def calc_gst(amount, rate=0.09)` — call it with one argument and 0.09 is used.

6.4 print vs return — the #1 confusion

print shows a value on screen and hands back **None**. **return** hands the value back so you can store, reuse and combine it.

PRINT — JUST SHOWS IT	RETURN — HANDS IT BACK

`def gst(a):`
`  print(a * 0.09)`
 	
`gst(50) + gst(30)`
	`def gst(a):`
	`  return a * 0.09`
	
	`gst(50) + gst(30)`

✗ **TypeError** — you cannot add two Nones.	✓ **7.2** — real values come back.

Rule of thumb: **print to look, return to use.** A function with no `return` gives back `None`.

6.5 Docstrings

A short description in triple quotes, placed **directly under the def line**. It shows up in `help()` and when you hover in the editor.

```
def calc_gst(amount):
    """Return the 9% GST for an amount."""
    return amount * 0.09

help(calc_gst)   # shows the docstring
```

✗ AVOID	✓ WRITE
`"""function"""`	`"""Return 9% GST for an amount.`
`"""this does stuff"""`	`"""Return (total, average) of s`
`"""calc"""`	`"""Return the discounted price.`

Good docstring = **start with a verb**, one line, say what comes back.

6.6 Return several values (tuples)

A **tuple** is like a list but written with round brackets and **cannot be changed** once created.

LIST	TUPLE
`[10, 20, 30]`	`(10, 20, 30)`
Add, remove, edit	Fixed after creating

```
import math

def circle_info(r):
    return 2*math.pi*r, math.pi*r**2

result = circle_info(5)   # one tuple: (31.42, 78.54)
c, a = circle_info(5)     # unpacked: c = 31.42, a = 78.54
```

The number of names on the left must match the number of values returned. `x, y, z = circle_info(5)` raises a **ValueError** — only 2 to unpack.

6.7 Local vs global scope

Local = created inside a function, exists only there. **Global** = defined at the top of the notebook, readable everywhere.

```
def add_all(numbers):
    total = 0        # local
    for n in numbers:
        total += n
    return total

print(total)   # ✗ NameError — total is gone
```

```
gst_rate = 0.09      # global

def calc(amount):
    return amount * gst_rate   # ✓ can read it

print(calc(100))     # 9.0
```

To get a value **out** of a function, `return` it. Don't reach in from outside.

6.8 lambda — a one-line function

SYNTAX

```
name = lambda params: expression
```

THE LONG WAY	THE LAMBDA
`def double(x):` `  return x * 2`	`double = lambda x: x * 2`

```
print(double(5))   # 10
```

There is no `return` — the expression *is* the result. Good for short throwaway logic, especially as an argument to `sorted()`.

Keep `def` for anything with more than one step, a docstring, or a name worth reading.

07 When to use what

Pick the right tool in one glance

Which loop?

I have a list or a string	`for item in seq`
I need to repeat N times	`for i in range(N)`
I need the position too	`enumerate(seq)`
I don't know how many rounds	`while condition`
Rows and columns	`nested for`

If you can count the rounds in advance, a `for` loop is safer — it can never run forever.

Loop or comprehension?

Building one new list	`comprehension`
Printing as you go	`for loop`
Several steps per item	`for loop`
Running totals or counters	`for loop`
Needs break or continue	`for loop`

A comprehension always produces **a list**. If you don't want a list back, you want a loop.

Writing a function

Caller needs the value	`return`
Only showing it on screen	`print`
Two results to hand back	`return a, b`
A usual value that rarely changes	`rate=0.09`
One short expression	`lambda`

Before you ask for help — run this checklist

- [ ] **Is my while loop still running?** Check that something inside changes the condition. Stop it with the ■ button.
- [ ] **Did I create the empty list before the loop?** Inside the loop it resets every round.
- [ ] **Did I remember range() excludes the stop number?** `range(1, 5)` is 1, 2, 3, 4.
- [ ] **Is my comprehension's `if` in the right place?** Front with `else` = label. End without `else` = filter.
- [ ] **Did I return or just print?** A function that prints gives back `None` .
- [ ] **Did I actually call the function?** A `def` on its own runs nothing.
- [ ] **Am I reading a local variable from outside?** Return it instead.

Check yourself

1 · How many times does this print?

```
for i in range(2, 6):
    print(i)
```

4 times — 2, 3, 4, 5. The stop number is excluded.

2 · What is the output?

```
[x*2 for x in range(3)]
```

[0, 2, 4] — every item transformed, same length.

3 · How long is the result?

```
n = [1, 2, 3, 4, 5]
[x for x in n if x > 3]
```

2 items — `[4, 5]` . A filter makes the list shorter.

4 · What does `r` hold?

```
def f(x):
    print(x + 1)

r = f(5)
```

None — it prints 6 but returns nothing.

5 · What is the result?

```
def area(h, a, b):
    return 0.5 * (a + b) * h

area(b=10, a=6, h=4)
```

32.0 — keyword arguments work in any order.

6 · Which line fails?

```
def g():
    n = 5
    return n

g()
print(n)
```

print(n) — NameError. `n` is local and gone once `g()` ends.
