Tasks – Diving into the return Statement

Go deeper into **return**: how it differs from **print**, explicit **return None**, **multiple return** statements, using return values in **conditions** and **expressions**, returning from inside a **loop**, returning **multiple values**, and **guard clauses**. Create each file, run it, and check the output.

Run scripts with: `python3 script_name.py`

Part 1 – return vs print

Task 1.1 – return gives a value; print displays it (`return_vs_print.py`)

- Create `return_vs_print.py`.
- Define a function `get_message()` that **returns** the string `"Hello"` (do not print inside the function).
- In the script, call `get_message()` and **assign** the result to a variable. Print that variable. Then print the result of `get_message()` directly: `print(get_message())`. If you only **return** and never **print** the result, nothing would appear; the caller decides what to do with the value (e.g. print it, store it, pass it on).

Expected output:

```
Hello
Hello
```

Task 1.2 – Function that prints vs function that returns (`return_or_print.py`)

- Create `return_or_print.py`.
- Define two functions: (1) `print_sum(a, b)` — **prints** `a + b` and has **no return** (or returns nothing). (2) `return_sum(a, b)` — **returns** `a + b` and does **not** print.

- Call `print_sum(3, 5)` — you see output. Call `result = return_sum(3, 5)` and then `print(result)` — you see the same number. The first does the output itself; the second gives you a value to use.

Expected output:

```
8
8
```

Part 2 – Explicit None and multiple returns

Task 2.1 – Explicit return None (`return_none.py`)

- Create `return_none.py`.
- Define a function `maybe_greet(do_it)` that takes a boolean. If `do_it` is `True`, return the string `"Hi!"`. If `do_it` is `False`, explicitly **return None** (write `return None`). After the function, call `print(maybe_greet(True))` and `print(maybe_greet(False))`. Returning `None` makes the "no value" explicit.

Expected output:

```
Hi!
None
```

Task 2.2 – Multiple return statements (`return_multi.py`)

- Create `return_multi.py`.
- Define a function `sign(n)` that has **several return statements**: if `n < 0` return `-1`, elif `n > 0` return `1`, else return `0`. Only one of these runs per call; **return** exits the function immediately.
- Call and print: `print(sign(-10))`, `print(sign(5))`, `print(sign(0))`.

Expected output:

```
-1
1
0
```

Task 2.3 – return in if/elif/else chain (`return_grade.py`)

- Create `return_grade.py`.
- Define a function `letter_grade(score)` that returns a string: score >= 90 → `"A"`, elif >= 80 → `"B"`, elif >= 70 → `"C"`, else → `"F"`. Use an if/elif/else chain; each branch **returns** a value. No need for a variable if you return directly from each branch.
- Call and print: `print(letter_grade(85))`, `print(letter_grade(72))`, `print(letter_grade(91))`.

Expected output:

```
B
C
A
```

Part 3 – Using the return value

Task 3.1 – Return value in a condition (`return_in_condition.py`)

- Create `return_in_condition.py`.
- Define a function `is_even(n)` that **returns** `True` if `n` is even (e.g. `n % 2 == 0`) and `False` otherwise. Do not print inside the function.
- In the script, use the return value in an **if** condition: e.g. `if is_even(4):` print `"even"`, else print `"odd"`. Do the same for an odd number (e.g. 7). The return value is used as the condition.

Expected output:

```
even
odd
```

Task 3.2 – Return value in an expression (`return_in_expression.py`)

- Create `return_in_expression.py`.
- Define `double(x)` that returns `x * 2` and `add_one(x)` that returns `x + 1`.
- Use return values in expressions: e.g. `print(add_one(double(5)))` (5→10→11) and `print(double(add_one(5)))` (5→6→12). Show that return values can be passed straight into another function or used in arithmetic.

Expected output:

```
11
12
```

Part 4 – return inside loops and multiple values

Task 4.1 – return from inside a loop (`return_in_loop.py`)

- Create `return_in_loop.py`.
- Define a function `first_positive(numbers)` that takes a list of numbers. Loop over the list; when you find the **first** number that is greater than 0, **return** that number immediately (this exits both the loop and the function). If the loop finishes without finding one, **return None** after the loop.
- Call and print: `print(first_positive([-1, 0, 3, 5]))` (should return 3) and `print(first_positive([-1, -2]))` (should return None).

Expected output:

```
3
None
```

Task 4.2 – Return multiple values (tuple) (`return_tuple.py`)

- Create `return_tuple.py`.

- Define a function `min_max(a, b)` that returns **two** values: the smaller and the larger of `a` and `b`. You can write `return a, b` — Python returns this as a tuple. In the script, call the function and **unpack**: e.g. `low, high = min_max(10, 3)`, then print `low` and `high` (you should get 3 and 10).
- Also try `result = min_max(7, 7)` and unpack; both values should be 7.

Expected output (example):

```
3
10
7
7
```

Part 5 – Guard clauses

Task 5.1 – Early return for invalid input (`return_guard.py`)

- Create `return_guard.py`.
- Define a function `divide_safe(a, b)` that returns `a / b` only when `b` is not zero. At the **start** of the function, if `b == 0`, **return None** immediately (and optionally return a string like `"invalid"` instead — if so, print it). Otherwise compute and return `a / b`. This “guard” at the top handles the bad case first.
- Call and print: `print(divide_safe(10, 2))`, `print(divide_safe(10, 0))`.

Expected output (if returning None for b==0):

```
5.0
None
```

Done

You've used: **return** to pass a value back (vs **print** for side effects), **explicit return None**, **multiple return** statements and return in if/elif/else, using return values in **conditions** and

expressions, **return from inside a loop**, **returning multiple values** (tuple and unpacking), and **guard clauses** (early return for invalid input).