

1.4 Functions, call frames and returned results

Name: _____ Class: _____ Date: _____ **Total: 22 marks**

KEY WORDS function • parameter • argument • contract
• call frame • local variable • scope

Objective

Build small Python functions with clear inputs and returned results. Trace nested calls, distinguish local names from caller names, and test the contract directly.

1 Key knowledge and worked examples

■ One job, one contract

A **function** gives a reusable name to a task. A **parameter** is a name in the definition; an **argument** supplies an object at a call. State permitted inputs and the promised result in a **contract**.

```
def add_fee(amount):
    """Return a non-negative whole-pence amount plus 3 pence."""
    return amount + 3
```

Calling `add_fee(8)` binds `amount` to the integer 8, calculates 11 and returns it. No output is printed. Calling a function does not automatically copy argument objects.

■ Worked example: Printing is not returning

```
def display_total(amount):
    print(amount + 3)

answer = display_total(8)
print(answer)
```

- Known behaviour: `print` writes text; a function without an explicit `return` returns `None`.
- First, `display_total` prints 11. Next, the call ends and binds `answer` to `None`.
- The second print therefore writes `None`. To return the calculated number, replace the function body's `print` with `return amount + 3`.

A **local variable** belongs to one call. Its **scope** controls where the name is resolved. Rebinding a local integer parameter does not reassign a caller's variable. Changing a mutable object is a different operation, developed in unit 1.5.

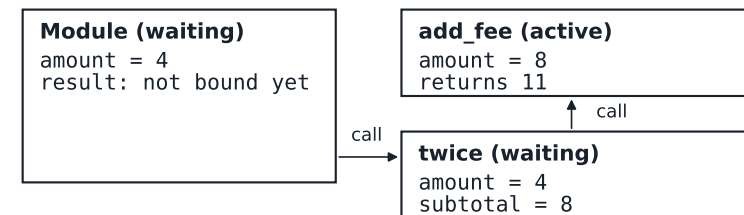
■ Worked example: Trace a nested call

```
def twice(amount):
    subtotal = amount * 2
    return add_fee(subtotal)

amount = 4
result = twice(amount)
print(amount, result)
```

- Known input: the module's `amount` refers to 4. The call gives `twice` a separate parameter name, initially bound to that same integer.
- Inside `twice`, `subtotal` becomes 8. `add_fee(subtotal)` starts a new **call frame** with its own `amount`, bound to 8.
- `add_fee` returns 11; `twice` returns that 11 to the module. Only then is `result` bound. The module's `amount` remains 4.
- The exact printed output is 4 11.

While add_fee is running



After both calls return

Module: `amount = 4, result = 11`

Names are separate. Argument objects are not copied automatically.

The figure shows conceptual frames and control flow, not physical memory addresses. Other functions do not automatically see the caller's local names.

2 Practice

2.1 For `add_fee(8)`, identify the parameter name, the argument value and the returned value. [3]

2.2 Trace the following program. State both printed numbers and explain why they differ. [4]

```
def increase(amount):  
    amount = amount + 1  
    return amount  
  
amount = 4  
new_amount = increase(amount)  
print(amount, new_amount)
```

2.3 A programmer expects `value` below to hold the integer 12. State its actual value, explain the mistake, and give the corrected function body. [3]

```
def triple(number):  
    print(number * 3)  
  
value = triple(4)
```

3 Exam-style questions

3.1 An order uses non-negative integer prices in pence and non-negative integer quantities. Its total is the item subtotal plus a fixed fee of 3 pence, even when quantity is zero.

Write `item_subtotal(unit_pence, quantity)` and `order_total(unit_pence, quantity)`. The second must call the first; both must return results without printing. Give two direct tests, including the zero-quantity boundary, and their expected results. Input validation is outside this task. [6]

3.2 The following program contains two successive calls with the same parameter name. Trace each call and give the three printed numbers. Explain whether object copying is needed to account for the result. [6]

```
def add_fee(amount):  
    return amount + 3  
  
def twice(amount):  
    subtotal = amount * 2  
    return add_fee(subtotal)  
  
amount = 4  
first = twice(amount)  
second = twice(first)  
print(amount, first, second)
```

Solutions

2.1

- Parameter name: `amount`.
- Argument value: the integer 8.
- Returned value: the integer 11.

2.2

- Output: 4 5.
- The parameter initially refers to the integer 4; the addition produces the integer 5.
- Assignment rebinds the function's local `amount`, without rebinding the module's `amount`.
- The returned 5 binds to `new_amount`; the module's `amount` stays 4.

2.3

- `value` holds `None`.
- Printing 12 does not return that integer; reaching the function's end returns `None`.
- Correct body: `return number * 3`.

3.1

```
def item_subtotal(unit_pence, quantity):  
    return unit_pence * quantity  
  
def order_total(unit_pence, quantity):  
    return item_subtotal(unit_pence, quantity) + 3  
  
assert order_total(5, 2) == 13  
assert order_total(5, 0) == 3
```

- `item_subtotal` takes the two inputs and returns their product.
- `order_total` calls that function with its own two inputs.
- It adds exactly 3 to the returned subtotal and returns the result, without printing.
- A non-zero test gives the correct expected result, for example 13 for (5, 2).
- A zero-quantity boundary test expects 3 under the stated contract.
- Both are direct return-value tests, such as assertions; observing printed output alone is insufficient.

3.2

- First call: `subtotal` is 8; `add_fee(8)` returns 11, so `first` is 11.
- Second call: `subtotal` is 22; `add_fee(22)` returns 25, so `second` is 25.
- The module's `amount` remains 4.
- Exact output: 4 11 25.
- Parameters bind to argument objects; no automatic copying is needed.
- Integer calculations and local rebinding do not reassign the caller's name.