PROGRAMMING FUNDAMENTALS

CHAPTER 5

# FUNCTIONS IN PYTHON

Lei Wang (Griffith University)

**THE COURSE STORY**
**Week 1 CALCULATE -> values, types, expressions**
**Week 2 INTERACT -> variables, input, output, conversion**
**Week 3 DECIDE -> Boolean expressions and selection**
**Week 4 REPEAT -> for, while, range, break, continue**
**Week 5 ORGANISE & REUSE -> functions, parameters, return, scope**

You can now make programs calculate, interact, decide and repeat.

The next challenge is organisation.

Functions let you turn a meaningful task into a named, reusable building block.

**THE BIG IDEA**
A function is a design tool: give a meaningful job a name, give it clean inputs, let it do its work, and give a useful result back.

# How to Use This

This is a student-book chapter.

Read the explanation first, trace the examples by hand, then run the code.

When you see a prediction challenge, do not run it immediately. Write down the expected output or returned value first.

When you see a function, use the four-question method: What goes IN? What does it DO? What comes OUT? What stays INSIDE?

**Important:** The aim is not to memorise syntax. The aim is to understand how functions change the way you design, trace, test and reuse programs.

# 1. Why Functions?

As programs grow, repeated blocks of code create maintenance problems. If the same logic appears in three places, a bug may need three fixes. A function lets us write the logic once and reuse it.

```
def classify_temperature(temperature):
    if temperature > 80:
        return "WARNING"
    return "SAFE"

status = classify_temperature(85)
print(status)
```

The caller can use a meaningful question such as classify_temperature(...) without repeatedly thinking about every implementation detail.

**THREE REASONS**

1. Reuse
2. Reduce complexity and defects
3. Control program flow

# 2. Decomposition: Break a Large Problem into Jobs

A large task can often be decomposed into smaller responsibilities.

| Large task | Possible function |
|---|---|
| Get user input | get_input() |
| Validate a value | validate_input(value) |
| Calculate | calculate_result(value) |
| Classify | classify_result(value) |
| Display | display_report(result) |

A strong function usually has one clear, meaningful responsibility. This makes it easier to understand, test and reuse.

**DESIGN QUESTION**

Do not ask: 'Where can I put def?'
Ask: 'What meaningful job deserves a name?'

# 3. Defining and Calling Functions

```
def say_hello():
    print("Hello!")

say_hello()
say_hello()
say_hello()
```

def creates the function definition. A call such as say_hello() executes its body. Defining a function does not execute it. In a normal script, define the function before calling it.

**COMMON MISTAKE**
Correct definition + no call = nothing happens.

# 4. The Anatomy of a Function

```
def calculate_area(width, height):
    return width * height
```

| Part | Meaning |
|---|---|
| def | Keyword starting a function definition. |
| calculate_area | Function name. |
| width, height | Parameters. |
| Indented body | Statements that perform the task. |
| return width * height | Sends a result back. |

## Parameter vs Argument

```
def greet(name): # parameter
    print("Welcome,", name)

greet("Alice") # argument
```

**REMEMBER**
Parameter = name in the function definition.
Argument = actual value supplied in the call.

# 5. return: The Most Important New Idea

```
def square(x):
    return x * x
```

```
result = square(5)
print(result)
```

Arguments go in, the function performs work, and return sends a value back to the caller. The caller can store it, print it, compare it, or use it in another calculation.

**MENTAL MODEL**

ARGUMENTS IN -> FUNCTION BODY -> RETURN VALUE OUT

# 6. print() Is Not return

```
def show_double(x):
    print(x * 2)

def get_double(x):
    return x * 2


a = show_double(5)
b = get_double(5)
print("a =", a)
print("b =", b)
```

| print | return |
|---|---|
| Displays information on screen. | Sends a value back to the caller. |
| Useful for user-facing output. | Useful when other code needs the result. |
| Does not provide the displayed value to the caller. | Immediately exits the function. |

Here a is None while b is 10. This is a high-value quiz/exam distinction.

# 7. None and Early return

```
def say_hello():
    print("Hello")

x = say_hello()
print(x)
```

There is no return statement, so x becomes None.

None is not 0; it means there is no useful returned value.

```
def classify_temperature(t):
    if t < 0:
        return "INVALID"
    if t > 80:
        return "CRITICAL"
    return "NORMAL"
```

return immediately exits the function. Once Python reaches a return, later statements in that function are not executed.

**TRACE RULE**

When tracing a function, STOP at the first return that is reached.

# 8. Multiple Parameters and Argument Styles

```
def final_balance(principal, rate, periods):
    return principal * (1 + rate) ** periods

print(final_balance(1000, 0.05, 3))
```

A function may have multiple parameters. Positional arguments are matched in order.

```
def describe_mission(name, crew):
    print(name, "has", crew, "crew members.")

describe_mission("Apollo", 3)
describe_mission(name="Apollo", crew=3)
describe_mission(crew=3, name="Apollo")
```

Keyword arguments identify parameters by name, which can improve clarity and allow the call order to differ.

## Default Values

```
def greet(name, message="Welcome"):
    print(message, name)

greet("Alice")
greet("Alice", "Good morning")
```

A default is used when the caller omits that argument. Required parameters come before default-valued parameters.

# 9. Returning More Than One Value

```
def min_max(a, b):
    if a < b:
        return a, b
    return b, a

small, large = min_max(8, 3)
print(small)
print(large)
```

Python conveniently lets a function return multiple values separated by commas, which can then be unpacked into variables. Functions can return any Python object, not just a single number.

**USE THIS WHEN**

A task naturally produces several related results and returning them together makes the interface clearer.

# 10. Docstrings and help()

```
def square(x):
    """Return the square of x."""
    return x * x

help(square)
```

A docstring is the first statement inside a function and documents its purpose.

Meaningful names plus useful docstrings are usually more valuable than comments on every line.

# 11. Scope: Keeping Variables in the Right Place

Scope means the region of a program where a name can be accessed.

```
def demo():
    x = 10
    print(x)

demo()
print(x)
```

The x created inside demo is local to that function. The outside print cannot access it.

## Global Variables

```
score = 10

def show_score():
    print(score)

show_score()
```

A function can normally read an existing global variable, but the course design rule is to minimise hidden global dependencies.

```
score = 10

def increase_score(score):
    return score + 1

score = increase_score(score)
```

**PREFERRED COMMUNICATION**

**Use parameters to send information in and return values to send information out.**

**This makes a function easier to test and reuse.**

# 12. Shadowing / Eclipsing

```
x = 1

def f():
    x = 2
    print("inside:", x)

print("before:", x)
f()
print("after:", x)
```

The output is before: 1, inside: 2, after: 1.

The local x hides the global x inside f(); it does not change the global variable.

## Parameters Are Local Names

```
def increase(x):
    x = x + 1
    print("inside:", x)

y = 1
increase(y)
print("outside:", y)
```

For this integer example, changing the local parameter x does not change y outside the function.

# 13. Scope vs Lifetime

| Concept | Question |
|---|---|
| Scope | Where can this name be accessed? |
| Lifetime | How long does this variable exist? |

Local variables normally exist while their function is executing.

Global variables normally remain for the program's lifetime unless explicitly deleted.

These ideas are related but not identical.

# 14. Functions Also Organise Control Flow

```
def step():
    print("inside function")

print("A")
step()
print("B")
```

The order is A, inside function, B.

A call temporarily moves execution into the function, then execution returns to the call site.

```
def double(x):
    return x * 2

def quadruple(x):
    return double(double(x))

print(quadruple(5))
```

Functions can call other functions.

This composition lets larger tasks use smaller building blocks.

## 15. Recursion: Recognise the Idea

```
def countdown(n):
    if n == 0:
        return
    print(n)
    countdown(n - 1)

countdown(3)
```

Recursion is a function calling itself.

A reachable stopping condition is essential.

Here, recursion is a concept to recognise; for and while remain the main repetition techniques.

**CONNECTION TO WEEK 4**

Week 4: use loops for repetition.

Week 5: functions package meaningful work so it can be reused.

## 16. Built-in Functions You Already Know

```
print()
input()
type()
int()
float()
str()
bool()
range()
```

You have actually been using functions since Week 1.

Other useful built-ins include len, round, abs, sorted, ord and chr.

You do not need to memorise every built-in; learn to recognise what Python already provides and look it up when needed.

### Functions vs Methods

```
print(len("Python")) # function text = "python"
print(text.upper()) # method
print(text.isdigit()) # method
```

A built-in function is called by its name. A method is called through an object using dot notation.

```
command = input("Command: ").strip().lower()

if command == "scan":
    print("Starting scan...")
```

strip() removes surrounding whitespace; lower() produces lowercase text. Both are string methods.

This is a practical connection to your earlier input-processing work.

## 17. Imported Functions and Modules

```
import math

print(math.sqrt(25))
print(math.ceil(3.2))
print(math.floor(3.8))
```

Modules group useful functionality. The important skill is not memorising every library function, but knowing that functionality exists, knowing how to import it, and knowing how to read its documentation.

**PROGRAMMER SKILL**
When you do not know a function: check its documentation, inspect help(), try a tiny example, then integrate it.

## 18. The Function Design Recipe

1. Name the meaningful job.
2. Identify the information the job needs.
3. Turn those inputs into parameters.
4. Decide what the caller needs back.
5. Use return for reusable results.
6. Keep internal details in local variables.
7. Write a short docstring.
8. Test the function independently before integrating it.

```
def classify_temperature(temperature):
    """Return SAFE, WARNING, CRITICAL, or INVALID."""
    if temperature < 0:
        return "INVALID"
    if temperature >= 100:
        return "CRITICAL"
    if temperature > 80:
        return "WARNING"
```

```
    return "SAFE"
```

This function has one clear job, takes one input, returns one result, and does not secretly depend on external state.

## 19. Test Small Functions First

```
print(classify_temperature(-1))
print(classify_temperature(0))
print(classify_temperature(80))
print(classify_temperature(80.1))
print(classify_temperature(100))
```

| Test | Purpose |
|---|---|
| -1 | Invalid case |
| 0 | Lower boundary |
| 80 | Threshold boundary |
| 80.1 | Just above threshold |
| 100 | Critical boundary |

Functions make debugging more local: if classification is wrong, you can test classify_temperature() without running the whole application.

## 20. Functions Connect Weeks 1-4

| Week | Core learning | Role inside functions |
|---|---|---|
| 1 | Values, types, operators, expressions | Calculate and transform values. |
| 2 | Variables, input, output, conversion | Receive values and return results. |
| 3 | Boolean expressions, if/elif/else | Make decisions inside a function. |
| 4 | for, while, range, break, continue | Repeat work inside a function. |
| 5 | Functions, parameters, return, scope | Package the work behind a reusable interface. |

A function can contain everything you already know: variables, input, expressions, decisions, loops, break and continue. The new idea is packaging that logic behind a clean interface.

**BIG PICTURE**

Weeks 1-4 taught you how to make a program work.

Week 5 starts teaching you how to make a program well organised.

## 21. A Preview: Mission Control v5

Last week Mission Control v4 could scan sensors, make decisions, count events, search for critical conditions and report. It is now large enough that clear pieces deserve names.

| Job | Possible function |
|---|---|
| Classify one temperature | classify_temperature(temp) |
| Calculate average | calculate_average(total, count) |
| Check critical condition | is_critical(temp, limit=100) |

| | |
|---|---|
| Produce summary | summarise(...) |
| Scan sensors | scan_sensors() |
| Coordinate application | run_mission_control() |

This week's workshop will refactor v4 into these kinds of functions.

The key design question is not 'Where can I put def?' but 'What meaningful job deserves a name?'

```
def is_critical(temperature, limit=100):
    return temperature >= limit

print(is_critical(105))
print(is_critical(95, limit=90))
```

This combines a default parameter, keyword argument, Boolean expression and return value.

The workshop will build on this pattern.

# 22. Predict Before You Run

## Challenge A - print vs return

```
def f(x):
    print(x * 2)

result = f(4)
print("result:", result)
```

Predict the exact output and the value of result.

## Challenge B - early return

```
def check(x):
    if x > 5:
        return "big"
    return "small"

print(check(8))
print(check(3))
```

Predict both outputs.

## Challenge C - shadowing

```
x = 10

def test():
    x = 20
    print(x)

test()
print(x)
```

Does the final print show 10 or 20? Explain.

### Challenge D - defaults and keywords

```
def power(x, exponent=2):
    return x ** exponent

print(power(3))
print(power(3, 3))
print(power(exponent=4, x=2))
```

Predict all three results.

## 23. Common Week 5 Mistakes

| Mistake | Correct idea |
|---|---|
| Define but forget to call | def creates; f() executes. |
| Parameter vs argument | Parameter is in def; argument is in call. |
| Use print when caller needs a result | Return the value. |
| No return means 0 | No return statement means None. |
| Expect code after return to execute | return exits immediately. |
| Use local variable outside | Local names stay in their function. |
| Rely on globals for communication | Prefer parameters and return values. |
| Bad default ordering | Required parameters come before defaults. |
| Confuse methods and functions | text.upper() vs len(text). |
| Use module without import | Import the required module. |
| Memorise everything | Learn to read documentation. |

## 24. Exam and Quiz Strategy

When given a function-tracing question, use this routine:

1. Identify the function name and parameters.
2. Write down the arguments at the call.
3. Track local variables separately from outer variables.
4. Follow conditions in order.
5. Stop immediately when return is reached.
6. If there is no return, record None.
7. Only then determine what is printed or assigned.

```
def process(value, limit=10):
    if value > limit:
        return value * 2
    return value + 1
```

| Question | Answer |
|---|---|
| Parameters? | value and limit |
| Default? | limit = 10 |
| process(12)? | 24 |
| process(7)? | 8 |
| What does it print? | Nothing |

This is exactly the kind of function analysis students should be able to explain in an exam: inputs, local variables, control flow and return value.

**HIGH-VALUE TRAP**

Always separate: 'What appears on screen?' from 'What value comes back?'

# 25. Practical Practice

## Practice 1 - Number Toolkit

Write is_even(n), is_positive(n), and square(n). Ask the user for a number and use the three functions.

## Practice 2 - Temperature Toolkit

Write classify_temperature(t) and calculate_average(total, count). Keep input/output outside the calculation functions.

## Practice 3 - Function Documentation

Write a function with a useful docstring. Then run help(your_function). Improve the docstring until a classmate can understand the function without reading its body.

## Practice 4 - Library Explorer

```
help(round)
help(str.strip)
help(str.lower)
```

For each, record: purpose, inputs, and returned result.

# 26. Chapter Checklist

- [ ] I can define and call a function.
- [ ] I can distinguish a parameter from an argument.
- [ ] I understand return and that it exits the function.
- [ ] I can explain print vs return.
- [ ] I know that no return statement gives None.
- [ ] I can use multiple, positional, keyword and default arguments.
- [ ] I can return and unpack multiple values.
- [ ] I can write a useful docstring and use help().
- [ ] I understand local/global scope and shadowing.
- [ ] I understand basic variable lifetime.
- [ ] I can distinguish functions from methods.
- [ ] I know how to import a module and inspect its functionality.
- [ ] I can design a function with one clear responsibility.
- [ ] I can test a function independently.
- [ ] I can explain how functions organise Weeks 1-4 knowledge.

# 27. Key Takeaways

| Idea | Remember |
|---|---|

| | |
|---|---|
| Function definition | def creates a reusable named task. |
| Function call | f(...) executes the task. |
| Parameter | Local name receiving an argument. |
| Argument | Actual value supplied at the call. |
| return | Sends a value back and exits. |
| None | Default result when no value is returned. |
| Keyword argument | Pass by parameter name. |
| Default parameter | Optional value used when omitted. |
| Multiple return | return a, b can be unpacked. |
| Local scope | Internal variables stay inside the function. |
| Global state | Avoid hidden dependencies where possible. |
| Method | object.method(...) uses dot notation. |
| Module | import module gives access to grouped functionality. |
| Design | Clear job -> clear name -> clean inputs -> clean outputs. |

**THE FOUR QUESTIONS**

**1. What information goes IN?**
**2. What does the function DO?**
**3. What information comes OUT?**
**4. What variables belong only INSIDE?**

**These four questions form a powerful mental model for reading and designing functions.**

# 28. Looking Ahead

Functions are the beginning of a larger programming idea: abstraction. You can now package a meaningful task behind a clean interface.

In our Mission Control v5 workshop, you will refactor the Week 4 application rather than build a completely new system.

The loops and decisions stay; the architecture becomes cleaner.

**WEEK 5 TAKEAWAY**

CLEAR JOB -> CLEAR NAME -> CLEAN INPUTS -> CLEAN OUTPUTS -> TEST -> REUSE

# 29. Reflection

**What is one piece of code you have written that would benefit from becoming a function?**

Your notes: _____

**Why is return usually more reusable than print?**

Your notes: _____

**What is the difference between a parameter and an argument?**

Your notes: _____

**What scope mistake can you now avoid?**

Your notes: _____

**When would a keyword argument make a function call clearer?**

Your notes: _____

**What built-in function or method would you like to learn more about?**

Your notes: _____

**What meaningful job deserves a function in Mission Control?**

Your notes: _____

# Week 5 Deep-Dive: Practical Skills to Master

## A. The clean function interface

A function is easiest to reuse when its interface is obvious.

Prefer this pattern: the caller supplies the data, the function performs the computation, and the function returns the result.

Keep user interaction and presentation separate unless the function's actual purpose is to display something.

```
def calculate_average(total, count):

    return total / count



average = calculate_average(300, 5)
print("Average:", average)
```

**WHY THIS IS BETTER**

calculate_average() can be reused by a report, a test, another calculation, or Mission Control.

A function that only prints the average would make the result harder for other code to use.

## B. Trace a function like a computer

Use a small trace table. For example:

```
def mystery(x):

    x = x + 10

    return x * 2



x = 5
print(mystery(x))
print(x)
```

| Moment | Name | Value | Important point |
|---|---|---|---|
| Before call | global x | 5 | Caller has x = 5 |
| Inside function | local x | 5 then 15 | Parameter is local |
| At return | returned value | 30 | 15 * 2 |
| After call | global x | 5 | Global x did not change |

**EXPECTED OUTPUT**

30

5

## C. Five high-value prediction challenges

These are deliberately similar to the kinds of questions that easily cause mistakes in a quiz.

### Challenge 1

```
def f(x):

    print(x * 2)



def g(x):

    return x * 2



a = f(4)
b = g(4)
print("a =", a)
print("b =", b)
```

**Before running:** What are a and b?

### Challenge 2

```
def alert(message="READY"):

    print("ALERT:", message)



alert()
alert("LOW FUEL")
```

**Before running:** What are the two output lines?

### Challenge 3

```
def calculate(a, b):

    return a + b, a * b



total, product = calculate(3, 4)
print(total)
print(product)
```

**Before running:** What are total and product?

### Challenge 4

```
x = 10

def test():

    x = 20
```

```
    print(x)



test()
print(x)
```

**Before running:** Does the final print show 10 or 20?

### Challenge 5

```
def process(value, limit=10):

    if value > limit:

        return value * 2

    return value + 1
```

**Before running:** What are the parameters? What does process(12) return? What does process(7) return? What does it print?

Answers: Challenge 1 -> a is None and b is 8. Challenge 2 -> ALERT: READY and ALERT: LOW FUEL. Challenge 3 -> total is 7 and product is 12. Challenge 4 -> final print is 10. Challenge 5 -> parameters are value and limit; process(12) returns 24; process(7) returns 8; it prints nothing.

## D. More useful Python around functions

You are also learning an important professional habit: use existing Python functionality rather than reinventing everything.

```
# String methods
text = " Mission Ready "
print(text.strip())
print(text.lower())
print(text.upper())
print("123".isdigit())
print("a,b,c".split(",")) # Formatting
value = 10.0 / 3
print("value = {:.2f}".format(value)) # Standard library
import math
print(math.sqrt(16))
print(math.ceil(5 / 3))
print(math.floor(5 / 3)) # Import one function directly
from math import sqrt
print(sqrt(16))
```

**DO NOT MEMORISE EVERYTHING**

A strong programmer knows how to find out.

Use help(), read documentation, inspect the function's inputs and return value, then test a tiny example before integrating it.

## E. Scope: a visual mental model

```
score = 10

def update(score):

    score = score + 1

    return score



score = update(score)
print(score)
```

Think of the function as having its own workspace. The parameter score is local to update(). The returned value is the clean communication channel back to the caller.

**DESIGN RULE**

Prefer explicit inputs and outputs over hidden global state.

This makes functions easier to test, understand and reuse.

## F. Function quality checklist

[ ] Can I describe the function's job in one sentence?

[ ] Does the function have a meaningful name?

[ ] Are all necessary inputs parameters?

[ ] Is the returned value clear?

[ ] Does the function avoid unnecessary printing?

[ ] Does it avoid unnecessary dependence on globals?

[ ] Are local variables used for internal work?

[ ] Is there a useful docstring?

[ ] Have I tested normal cases?

[ ] Have I tested boundary cases?

[ ] Can another part of the program reuse the result?