Python语言程序设计

# Python Programming

Session 07



Tom Lotz (tom.lotz@outlook.com)

# Content


| | |
|---|---|
| | Review |
| 01 | Defining and calling functions |
| 02 | Parameters and arguments |
| 03 | Return values and local variables |
| 04 | Functions in programs |

# Review




# and: Both Conditions

Both conditions must be True. Here, 15 is greater than 10 and less than 20.

Python code

```
x = 15

print(x > 10 and x < 20)
```

Expected output

```
True
```

# or: At Least One Condition




At least one condition must be True. Both may be True. Here, neither condition is True.

Python code

```
x = 15

print(x < 10 or x > 20)
```

Expected output

```
False
```

# not: Reverse the Result




not changes True to False and False to True.

Python code

```
x = 15

print(not x == 15)
print(not True)
print(not False)
```

Expected output

```
False
False
True
```

# The while Loop




while repeats an indented block as long as its condition is True.

Python code

```
x = 1

while x <= 5:
    print(x)
    x = x + 1

print("Finished")
```

Condition: x <= 5

Iteration: one repetition

Counter: x

Python checks the condition before every repetition.

# A Running Total




A counter controls repetition. An accumulator stores a running total.

Python code

```
total = 0
x = 1

while x <= 5:
    total = total + x
    x = x + 1

print(total)
```

x is the counter.

total is the accumulator.

Each update adds x to the previous total.

Expected output: 15

# Stopping with break




Python code

```
while True:
    command = input("Text (q to quit): ")
    if command == "q":
        break
    print(command)

print("Finished")
```

while True keeps repeating unless something stops it.
break exits the nearest enclosing loop immediately.
The code after the loop still runs.

# Defining and calling functions

# A Function: A Named Block of Code

A function groups instructions under a name. Call the function whenever you want those instructions to run.

Python code

```
def show_message():
    print("Ready")
    print("Let us begin")

show_message()
```

Expected output

```
Ready
Let us begin
```

# Defining and Calling

def defines the function. The indented lines form its body. The function runs only when Python reaches a call.

Python code

```
def show_message():
    print("Ready")

print("Before")
show_message()
print("After")
```

Expected output

```
Before
Ready
After
```

# Practice: Function Calls

## CORE EXERCISE

Predict every output, in order.
How many calls run?

### Python code

```
def show_number():
    print(7)

print("Start")
show_number()
show_number()
print("End")
```

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Change the calls so the function runs three times.

Then use a while loop to call it three times without writing three separate calls.

# Function Calls: Solution

Expected output

```
Start
7
7
End
```

The definition stores the instructions.

Each show_number() call runs the body.

There are two calls, so 7 prints twice.

# Debugging: A Function That Will Not Run

## CORE EXERCISE

Find and repair all three problems.

### Code to repair

```
def show_message()
print("Ready")

show_message
```

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Make the corrected function run twice, with "Again" printed between the two calls.

# Function Debugging: Solution

Add the colon, indent the function body, and use parentheses to call the function.

Python code

```
def show_message():
    print("Ready")

show_message()
```

Expected output

```
Ready
```

# Practice: A Menu Function

## CORE EXERCISE

Define show_menu() with no parameters. It should display:

a: Add
s: Subtract
q: Quit

Call it twice. Do not write the three prints twice.

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Use a while loop to display the menu three times.

B. Ask for a choice after each display. Stop when it is q.

# Menu Function: Solution

The same function body runs for both calls.

Python code

```
def show_menu():
    print("a: Add")
    print("s: Subtract")
    print("q: Quit")

show_menu()
show_menu()
```

# Parameters and arguments

# A Parameter Receives a Value

number is a parameter: a name inside the function.
4 and 9 are arguments: the values supplied by the calls.

Python code

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

show_double(4)
show_double(9)
```

Expected output

```
8
18
```

# Parameters and Arguments

| Code | Meaning |
| --- | --- |
| def show_double(number): | number is the parameter |
| show_double(4) | The argument 4 goes into number |
| show_double(9) | The next call uses a new value |

One definition can work with many different values.
The parameter exists inside the function call.

# Two Parameters and Their Order

Arguments match parameters from left to right.
Reversing the arguments can change the result.

Python code

```
def show_difference(a, b):
    print(a - b)

show_difference(9, 4)
show_difference(4, 9)
```

Expected output

```
5
-5
```

# Practice: Arguments and Variables

## CORE EXERCISE

Predict all three outputs.
Does calling the function change x?

### Python code

```
def show_difference(a, b):
    print(a - b)

x = 9
show_difference(x, 4)
show_difference(4, x)
print(x)
```

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Choose two different argument values that produce a negative result.

Explain why reversing them changes the sign.

# Arguments and Variables: Solution

Expected output

```
5
-5
9
```

First call: a = 9, b = 4.
Second call: a = 4, b = 9.

The function reads these values.
It does not assign a new value to x.

# Practice: Rectangle Perimeter

## CORE EXERCISE

Define show_perimeter(length, width).

Inside the function, calculate and print the perimeter:
2 * (length + width)

Call it with (5, 3), (4, 4), and (8, 2).

Expected perimeters: 16, 16, 20.

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Ask for dimensions outside the function. Pass the values into it.

B. Create show_area(length, width).
Use the same dimensions to show area and perimeter.

# Rectangle Perimeter: Solution

length and width receive a different pair of values for each call.

Python code

```
def show_perimeter(length, width):
    perimeter = 2 * (length + width)
    print(perimeter)

show_perimeter(5, 3)
show_perimeter(4, 4)
show_perimeter(8, 2)
```

Expected output

```
16
16
20
```

# Return values and local variables

# return Sends a Value Back

return gives a result back to the caller.
The caller can store that result in a variable.

Python code

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

result = double(7)
print(result)
```

Expected output

```
14
```

# print() and return

print() displays a value. return makes a value available to the caller.

Program A

```
def double(number):
    print(number * 2)

result = double(7)
print(result)
```

Output: 14, then None

Program B

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

result = double(7)
print(result)
```

Output: 14
result stores the number 14.

# A Missing return Gives None

- A function without an explicit return value returns None.
- None means no value here. It is different from 0.
- A printed number does not become the returned result.
- Return the value if the caller needs to calculate with it.

# return Ends the Function Call

Once a return runs, that call ends.
A function can return numbers, strings, or Boolean values.

Python code

```
def magnitude(number):
    if number < 0:
        return -number
    return number

print(magnitude(-6))
print(magnitude(4))
```

Expected output

```
6
4
```

# Practice: Using a Returned Value

## CORE EXERCISE

Predict both outputs before running.

### Python code

```
def triple(number):
    return number * 3

result = triple(4)
print(result + 2)
print(triple(0))
```

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Use triple() twice to multiply a number by 9.

Store or print the final result.

# Returned Values: Solution

Expected output

```
14
0
```

triple(4) returns 12.
The caller prints 12 + 2.

triple(0) returns 0.

Optional example: triple(triple(2)) returns 18.

# Local Variables

Each call has its own parameters and local variables.
The x outside the function remains unchanged.

Python code

```
x = 100
def double(x):
    result = x * 2
    return result

print(double(4))
print(x)
```

Expected output

```
8
100
```

# Debugging: Result and Local Variable

## CORE EXERCISE

Why does this program fail?
Repair it so the caller prints the sum.

### Code to repair

```
def get_total(a, b):
    total = a + b

get_total(4, 6)
print(total)
```

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Keep total local to the function.
Use the repaired function to calculate two sums, then add those returned results.

# Local Variable: Solution

Return total, then store the returned value outside the function. The original program raises NameError at print(total).

Python code

```
def get_total(a, b):
    total = a + b
    return total

result = get_total(4, 6)
print(result)
```

Expected output

```
10
```

# Practice: Return the Larger Value

## CORE EXERCISE

Write larger(a, b).

Return the larger number.
If both are equal, return that shared value.

Keep input() and print() outside the function.

Test positive and negative values.

(8, 3): 8  (-4, -2): -2
(5, 5): 5

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Use larger() to find the largest of three numbers.

B. Test equal values, zero, and two negative numbers.

# Larger Value: Solution

If a is not greater than b, returning b also handles equality.

Python code

```
def larger(a, b):
    if a > b:
        return a
    else:
        return b

print(larger(8, 3))
print(larger(-4, -2))
print(larger(5, 5))
```

Expected output

```
8
-2
5
```

# Testing a Function Independently

| Call | Expected result | Reason for the test |
|---|---|---|
| larger(8, 3) | 8 | First value is larger |
| larger(3, 8) | 8 | Second value is larger |
| larger(5, 5) | 5 | Equal values |
| larger(-4, -2) | -2 | Both values are negative |
| larger(0, -3) | 0 | Zero and a negative value |

Call the function directly with test values.
Compare the actual result with the expected result.

# Practice: A Function with a while Loop

## CORE EXERCISE

Write sum_to(n).

For a nonnegative integer n, return the sum from 1 to n.

Use a while loop, a counter, and a running total.

For n = 0, return 0.
Print the result outside.

sum_to(5): 15
sum_to(0): 0  sum_to(1): 1

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Write sum_even_to(n) to return the sum of the even numbers from 2 up to n, including n when it is even.

Test both odd and even limits.

# Sum Function: Solution

The counter and total are local. Each call starts them again. return runs after the loop finishes.

Python code

```
def sum_to(n):
    total = 0
    x = 1
    while x <= n:
        total = total + x
        x = x + 1
    return total

print(sum_to(5))
```

# Functions in programs

# Default Arguments

A default value makes an argument optional.
Parameters with defaults come after required parameters.

Python code

```
def total_price(price, quantity=1):
    return price * quantity

print(total_price(6))
print(total_price(6, 4))
```

Expected output

```
6
24
```

# Practice: A Default Delivery Fee

## CORE EXERCISE

Write delivery_total(amount, fee=5).

Return the amount plus the fee.

Call it with amount 20 only, then with amount 20 and fee 0.

Keep printing outside the function.

```
delivery_total(20): 25
delivery_total(20, 0): 20
```

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

Make delivery free when the amount is at least 50.

Test 49, 50, and 51, including a call with a different fee.

# Default Delivery Fee: Solution

Without a second argument, fee uses its default value of 5.

Python code

```
def delivery_total(amount, fee=5):
    return amount + fee

print(delivery_total(20))
print(delivery_total(20, 0))
```

Expected output

```
25
20
```

# Input, Calculation, and Output

Calculation functions receive values and return results.
The calling code handles input and output.

Python code

```
def add(a, b):
    return a + b

a = int(input("First number: "))
b = int(input("Second number: "))
result = add(a, b)
print("Result:", result)
```

# Independent Practice: Calculator Functions

## CORE EXERCISE

Extend the repeating calculator.

1. Define add(a, b), subtract(a, b), and multiply(a, b). Each returns a result. No input or printing inside these functions.
2. Keep the input loop and q exit.
3. Use if/elif/else to call the selected function.
4. Print the returned result. Handle unknown operations.

Check: 12, 5, * gives 60.
Test q immediately too.

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Add a division function. Prevent division by zero.

B. Count successful calculations and display the count on exit.

C. Store successful results in a list and display the history.

D. Accept decimal inputs.

# Optional Hint: Calculator Functions

- Keep the calculation functions above the input loop.
- Replace a calculation such as a + b with a function call.
- Store the returned value, then print it.
- Keep the q check before converting input to an integer.
- An unknown operation should not print an old result.

# Calculator Solution: Calculation Functions

Part 1 of 3. These functions only calculate and return values.

Python code

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b
```

# Calculator Solution: The Input Loop

Part 2 of 3. Add this below the function definitions.

Python code

```
while True:
    first = input("First number (q to quit): ")
    if first == "q":
        break
    a = int(first)
    b = int(input("Second number: "))
    operation = input("Operation (+, -, *): ")
```

# Calculator Solution: Calling the Functions

Part 3 of 3. Continue inside the loop. Goodbye is outside it.

Python code

```
    if operation == "+":
        result = add(a, b)
        print("Result:", result)
    elif operation == "-":
        result = subtract(a, b)
        print("Result:", result)
    elif operation == "*":
        result = multiply(a, b)
        print("Result:", result)
    else:
        print("Unknown operation")
print("Goodbye")
```

# Testing the Function Calculator

| Test | Expected behavior |
| --- | --- |
| 12, 5, * then q | Result: 60, then Goodbye |
| -4, 2, + then q | Result: -2, then Goodbye |
| 3, 8, - then q | Result: -5, then Goodbye |
| Unknown operation after a valid one | Message only, no previous result |
| q immediately | Goodbye, with no conversion error |

Test each function directly as well as through the calculator.
Enter integers in the core version.

# A Simple Module: Another Python File

- A module is a Python file that another program can import.
- Put reusable calculations in calculations.py.
- Keep main.py and calculations.py in the same folder.
- In IDLE, save both files and run main.py with F5.
- Keep keyboard input and the program loop in main.py.

# Two Files Working Together

calculations.py

```
def add(a, b):
    return a + b
```

main.py

```
import calculations

print(calculations.add(9,4))
```

import makes the module available. calculations.add() calls its function.
Expected output: 13. The module itself does not ask for input.

# Practice: Two Files in IDLE

## CORE EXERCISE

Create helpers.py containing your larger(a, b) function.

Create main.py in the same folder.

Import helpers, ask for two integers, call helpers.larger(), and print the returned value.

Run main.py in IDLE.

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Add sum_to(n) to helpers.py. Call it from main.py.

B. Move the calculator functions into calculations.py. Keep the interactive loop in main.py.

# Two Files: Solution

helpers.py

```
def larger(a, b):
    if a > b:
        return a
    else:
        return b
```

main.py

```
import helpers

a = int(input("First: "))
b = int(input("Second: "))
print(helpers.larger(a, b))
```

Save both files before running. Run main.py rather than helpers.py.
Test equal values, zero, and negative values too.

# Independent Check

1. Predict both output values.

Python code

```
def add(a, b=2):
    return a + b

print(add(3))
print(add(3, 4))
```

2. How do a parameter and an argument differ?

3. Why can a function print 10 but return None?

4. How can the caller use a value calculated in a local variable?

# Recap

- def defines a function.
A call runs its body.
- Arguments supply values to parameters.
- return sends a value back and ends that function call.
- Local variables belong to a function call.
- Default arguments supply a value when one is omitted.
- Modules let another program import reusable functions.

Test functions with different inputs before combining them into a program.

# Homework: Time in Minutes

## CORE EXERCISE

Write to_minutes(hours, minutes).
Return hours * 60 + minutes.

In the calling program:

- Ask for whole hours and minutes.
- Reject negative hours.
- Accept minutes only from 0 to 59.
- Ask again until the values are valid.
- Call the function and print its result.

Submit your .py file. Explain it.

2 h 15 min: 135  0 h 0 min: 0
Test minutes -1 and 60 too.

## OPTIONAL CHALLENGES

Finish the core first.
Choose any challenge.

A. Allow another calculation until the user chooses to quit.

B. Put to_minutes() in a separate module and import it.

C. Add from_minutes(total) to print full hours and minutes.