

Day 9 Functions, part 1

Self-Study Guide · About 70 minutes · With answer key

By the end you will be able to

- Define a function with `def`, call it with arguments, and explain the difference between a parameter and an argument.
- Choose between returning a value and printing it, and predict when a function gives back `None`.
- Explain which variables a function can see and change, and return several values at once.
- Break a program into small functions that each do one thing, with a docstring and a few test calls.

What to remember from earlier

- A function is a named instruction you call with parentheses, and the values inside the parentheses are its arguments. ([Day 1: Setup and your first program, 1.4 Printing Text and Numbers](#))
- `None` is the value that means "no value here", and `is None` checks for it. ([Day 2: Variables, numbers and basic types, 2.6 True, False, and None](#))
- A triple-quoted string starts and ends with three quote marks and can run over several lines. ([Day 3: Strings, 3.1 Quotes and Escape Characters](#))
- An accumulator is set before a loop, updated inside it, and used after it. ([Day 5: Loops, 5.6 Counting and Summing](#))
- A list holds values in order; `append()` adds one to the end, and `len()`, `sum()`, `min()`, and `max()` work on the whole list. ([Day 6: Lists](#))
- Changing a list through one name changes it for every name that refers to the same list. ([Day 6: Lists, 6.9 Aliasing Versus Copying](#))
- Writing several values separated by commas makes a tuple, and `a, b = pair` unpacks it into separate names. ([Day 7: Tuples, sets and unpacking](#))

Make a `day09` folder for today's files, open it in your editor, and run each script from its terminal, as on earlier days.

Since Day 1 you have been **calling** functions that someone else wrote: `print`, `len`, `input`, `range`, `sum`. Each one does a job, and you use it by name without thinking about the code inside it. Today you write your own. A function lets you give a name to a few lines of code, run them whenever you like, hand them different values each time, and get an answer back. It is the main tool for turning a long script into small pieces you can understand, reuse, and check one at a time.

9.1 Why Functions

A shop adds 8% sales tax to every price. Here is a script that works out the total for three prices:

tax_repeated.py

```
1 # Three prices, each with 8% sales tax added
2 price = 4.50
3 total = price + price * 0.08
4 print(f"Total: {total:.2f}")
5
6 price = 12.00
7 total = price + price * 0.08
8 print(f"Total: {total:.2f}")
9
10 price = 7.25
11 total = price + price * 0.08
12 print(f"Total: {total:.2f}")
```

```
Total: 4.86
Total: 12.96
Total: 7.83
```

It works, but look at how it is built. The same two lines, the tax formula and the `print`, appear three times with only the price changed. That causes three problems:

- **Changing it is risky.** If the tax rate goes up to 10%, you must find and fix `0.08` in three places. Miss one and one total is wrong, with no error message to warn you.
- **It is hard to read.** Nothing in the code says "add tax". A reader has to work out what `price + price * 0.08` means each time they see it.
- **It is hard to check.** To test the formula you have to run the whole script and inspect every line of output.

A **function** is a named block of code that does one job. You write the code once, give it a name, and then **call** it (run it by name) as many times as you need. On Day 1 you learned that the values you pass in the parentheses are **arguments**; a function can also hand a value back. Here is the same script with a function called `with_tax`:

tax_function.py

```
1 def with_tax(price):
2     return price + price * 0.08
3
4
5 print(f"Total: {with_tax(4.50):.2f}")
6 print(f"Total: {with_tax(12.00):.2f}")
7 print(f"Total: {with_tax(7.25):.2f}")
```

```
Total: 4.86
Total: 12.96
Total: 7.83
```

The output is identical. You will learn every piece of this syntax in the next three subsections; for now, read lines 1 and 2 as "here is how to work out a price with tax", and each `with_tax(...)` on lines 5 to 7 as "use that recipe on this price". The formula now lives in **one** place. This is what functions give you:

Benefit	What it means	In <code>tax_function.py</code>
Reuse	Write the code once and call it as often as you like	One formula, three calls
Naming	The name says what the code does, so the reader does not have to work it out	<code>with_tax(7.25)</code> reads as English
Testing	You can check one small piece on its own by calling it with values you know answers for	Call <code>with_tax(100)</code> and expect 108.0

A good function does **one thing**, and its name says what that thing is. Big programs are built from many small functions, each easy to understand and check on its own.

KEY IDEA

A function gives a name to a block of code so you can run it many times. It lets you reuse code, name an idea, and test a small piece on its own.

WATCH OUT

Copying code instead of writing a function. Copy and paste feels quick, and the script still runs. The cost comes later: every copy is another place to update, and copies drift apart when one is fixed and another is not. A useful rule of thumb: when you are about to paste the same lines a second or third time, make them a function instead.

PAUSE AND TRY

Question 1 Multiple choice

The tax rate changes from 8% to 10%. How many places must you edit in `tax_repeated.py` and in `tax_function.py`?

- A. 1 in `tax_repeated.py`, 1 in `tax_function.py`
- B. 3 in `tax_repeated.py`, 1 in `tax_function.py`
- C. 3 in `tax_repeated.py`, 3 in `tax_function.py`
- D. 6 in `tax_repeated.py`, 3 in `tax_function.py`

9.2 Defining and Calling a Function

To use your own function, you need two steps: **define** it (tell Python its name and what it does) and **call** it (ask Python to do it now).

A **function definition** starts with the keyword `def` (short for "define"):

```
1 def function_name():  
2     statement  
3     statement
```

- `def` is a keyword, followed by the function's name, a pair of parentheses, and a colon. This first line is the **header**.

- The indented lines under it are the **function body**: the code that runs each time the function is called. Indentation groups the body, just as it groups the lines of an `if` or a loop (Days 1, 4, and 5).
- The name follows the same rules as a variable name (Day 2): `snake_case`, letters, digits, and underscores, not starting with a digit. Because a function **does** something, its name is usually a verb or a verb phrase: `say_hello`, `with_tax`, `print_receipt`.

To **call** the function, write its name followed by parentheses, exactly as you call `print()`: `say_hello()`.

WORKED EXAMPLE Defining once, calling twice

Watch the order in which lines run. It is not simply top to bottom.

`say_hello.py`

```
1 def say_hello():
2     print("Hello!")
3     print("Welcome to Day 9.")
4
5
6 print("Start")
7 say_hello()
8 say_hello()
9 print("End")
```

```
Start
Hello!
Welcome to Day 9.
Hello!
Welcome to Day 9.
End
```

Step	Line	What Python does	Prints
1	1–3	Reads the definition and remembers <code>say_hello</code> ; runs nothing	
2	6	Prints	Start
3	7	Calls <code>say_hello</code> : jumps to line 2	
4	2, 3	Runs the body	Hello!, Welcome...
5	7	Body finished: comes back to just after the call	
6	8	Calls <code>say_hello</code> again: runs lines 2 and 3 again	Hello!, Welcome...
7	9	Prints	End

When Python reaches the `def` on line 1, it does **not** run the body. It only stores the instructions under the name `say_hello`, like writing a recipe on a card without cooking it. Each call on lines 7 and 8 is the cooking: Python jumps into the body, runs it from the top, and when the body ends it comes back to the line after the call and carries on. The order in which Python runs a program's lines is called the **flow of execution**; a call is a detour that always returns to where it left.

Now change one thing: keep the definition but take out the calls.

define_only.py

```
1 def say_hello():
2     print("Hello!")
3     print("Welcome to Day 9.")
4
5
6 print("The function is defined.")
```

The function is defined.

Hello! never appears. A definition on its own does nothing you can see; the body runs only when something calls the function.

Notice the two blank lines after each function in these files. Python does not need them, but the official Python style guide (called PEP 8) asks for two blank lines around a top-level function so that each definition stands out from the code around it. All of today's examples follow it.

KEY IDEA

`def name():` defines a function: Python remembers the body but does not run it. `name()` calls the function: Python runs the body, then comes back to the line after the call.

WATCH OUT

Calling a function before its definition. Python reads a file top to bottom (Day 1), and `def` is what creates the name. A call above the `def` uses a name that does not exist yet:

call_before_def.py

```
1 say_hello()
2
3
4 def say_hello():
5     print("Hello!")
```

```
Traceback (most recent call last):
  File "../day09/call_before_def.py", line 1, in <module>
    say_hello()
    ^^^^^^^^^^^
NameError: name 'say_hello' is not defined
```

This is the same `NameError` you met on Day 1 for a misspelled name. At line 1, Python has not yet reached line 4, so it knows nothing called `say_hello`. Put every definition **above** the code that calls it; most programs put all their functions at the top of the file. (Writing `say_hello` without parentheses is a different slip: it names the function but does not call it, so the body does not run.)

PAUSE AND TRY**Question 2** Tracing

What does this program print? Write each line in order.

order_trace.py

```

1 print("A")
2
3
4 def show():
5     print("B")
6
7
8 print("C")
9 show()
10 print("D")
11 show()

```

9.3 Parameters and Arguments

`say_hello` does the same thing every time. `print` and `len` are more useful because you hand them a value to work on. Your functions can take values too.

A **parameter** is a name in the function's header, inside the parentheses, that receives a value when the function is called. An **argument** is the value you write in the call. When you call the function, Python assigns each argument to its parameter, exactly like an assignment statement, and then runs the body. Inside the body, the parameter is an ordinary variable.

An everyday comparison: a recipe card says "add the **fruit**". "fruit" is the parameter, a blank to fill in. When you cook, you decide that today the fruit is apples: "apples" is the argument.

WORKED EXAMPLE Greeting different people

greet_name.py

```

1 def greet(name):
2     print(f"Hi, {name}!")
3
4
5 greet("Ana")
6 greet("Ben")
7 student = "Chen"
8 greet(student)

```

```

Hi, Ana!
Hi, Ben!
Hi, Chen!

```

Call (line)	Argument	name holds while the body runs	Prints
<code>greet("Ana")</code> (5)	"Ana"	"Ana"	Hi, Ana!

<code>greet("Ben")</code> (6)	<code>"Ben"</code>	<code>"Ben"</code>	<code>Hi, Ben!</code>
<code>greet(student)</code> (8)	<code>student</code> , so <code>"Chen"</code>	<code>"Chen"</code>	<code>Hi, Chen!</code>

`name` on line 1 is the parameter. Each call starts by doing `name = <the argument>`, then runs line 2. On line 8, the argument is the variable `student`: Python works out its value, `"Chen"`, and that value is what `name` receives. An argument can be any expression, such as `greet("ana".upper())`; Python works it out first, then passes the result.

A function can have several parameters, separated by commas. The arguments are matched to the parameters **by position**: the first argument goes to the first parameter, the second to the second. Arguments matched this way are called **positional arguments**. Here the same function is called twice with the same two values in a different order:

`describe_pet.py`

```
1 def describe_pet(animal, name):
2     print(f"{name} is a {animal}.")
3
4
5 describe_pet("dog", "Rex")
6 describe_pet("Rex", "dog")
```

```
Rex is a dog.
dog is a Rex.
```

In the first call, `animal = "dog"` and `name = "Rex"`. In the second call, the order is swapped, so `animal = "Rex"` and `name = "dog"`. Python cannot tell that this makes no sense; it only counts positions. When you call a function with several parameters, check the order in its header.

KEY IDEA

Parameters are the names in the `def`; arguments are the values in the call. Each call assigns the arguments to the parameters by position, then runs the body.

WATCH OUT

The wrong number of arguments. Every parameter needs a value, so a call must give one argument per parameter:

`greet_missing.py`

```
1 def greet(name):
2     print(f"Hi, {name}!")
3
4
5 greet()
```

```
Traceback (most recent call last):
  File ".../day09/greet_missing.py", line 5, in <module>
    greet()
    ~~~~~^^
TypeError: greet() missing 1 required positional argument: 'name'
```

The message names the function, says how many arguments are missing, and names the parameter that got nothing. Too many arguments is also a `TypeError`: `greet("Ana", "Ben")` gives `greet()` takes 1 positional argument but 2 were given. Fix the call to match the header. (Older versions of Python may word these messages a little differently.)

PAUSE AND TRY

Question 3 Tracing

What does this program print? Pay attention to which value goes to which parameter in the second call.

show_seat.py

```
1 def show_seat(row, seat):
2     print(f"Row {row}, seat {seat}")
3
4
5 show_seat("C", 7)
6 seat = 2
7 row = "A"
8 show_seat(seat, row)
```

9.4 return: Sending a Value Back

`greet` prints, and that is all it can do. But `len("cat")` does not print anything: it **gives back** 3, and you decide what to do with it, such as store it, compare it, or add 1 to it. To get a result out of your own function, you need the same ability.

The **return statement**, `return expression`, ends the function and sends the value of the expression back to the place where the function was called. That value is the function's **return value**. The call itself then stands for that value: wherever you write `rectangle_area(3, 4)`, Python runs the function and replaces the call with what it returned, the way it replaces `len("cat")` with 3.

WORKED EXAMPLE Areas of two rooms

rectangle_area.py

```

1 def rectangle_area(width, height):
2     return width * height
3
4
5 kitchen = rectangle_area(3, 4)
6 hall = rectangle_area(1.5, 6)
7 print("Kitchen:", kitchen)
8 print("Hall:", hall)
9 print("Total:", kitchen + hall)

```

```

Kitchen: 12
Hall: 9.0
Total: 21.0

```

Line	What happens	Result
5	Call with width = 3, height = 4; line 2 returns 3 * 4; the call becomes 12	kitchen = 12
6	Call with width = 1.5, height = 6; line 2 returns 1.5 * 6; the call becomes 9.0	hall = 9.0
7–9	Ordinary prints; line 9 adds the two stored results	21.0

The function prints nothing itself. It works out a number and hands it back, and the code that called it decides what to do with it: store it on lines 5 and 6, then add the two results on line 9. `hall` is a float because `1.5 * 6` mixes a float with an int (Day 2), and `12 + 9.0` is 21.0 for the same reason.

A function can contain more than one `return`, usually in different branches of an `if`. The first `return` that runs ends the function at once, and any lines after it in the body are skipped. That makes a chain of `if` statements simple to write:

ticket_price.py

```

1 def ticket_price(age):
2     if age < 12:
3         return 5
4     if age >= 65:
5         return 6
6     return 10
7
8
9 print(ticket_price(8))
10 print(ticket_price(70))
11 print(ticket_price(30))
12 print("Family of three:", ticket_price(8) + ticket_price(40) + ticket_price(42))

```

```

5
6
10
Family of three: 25

```

For `ticket_price(8)`, line 2 is `True`, so line 3 returns 5 and the function is over; lines 4 to 6 never run, which is why the second `if` does not need to be an `elif`. For 70, the first test fails, the second passes, and it returns 6. For 30, both tests fail, and line 6 returns the standard price, 10. The last line shows a call used directly inside an expression: `5 + 10 + 10` is 25. Because a function call is an expression, you can use it anywhere a value is allowed: in a calculation, an f-string, a condition, or as the argument of another call.

KEY IDEA

`return value` ends the function and hands `value` back to the caller, where the call is replaced by it. Code in the body after a `return` that runs is skipped.

WATCH OUT

`return` inside a loop by mistake. This function should add up a list (Day 6), but it is one indentation level off:

`return_in_loop.py`

```
1 def total(numbers):
2     result = 0
3     for n in numbers:
4         result += n
5         return result
6
7
8 print(total([4, 5, 6]))
```

4

`return result` is indented inside the `for` loop, so it runs on the **first** pass, right after `result` becomes 4. The function ends there, and 5 and 6 are never added. This is the accumulator pattern from Day 5: the result should be used **after** the loop finishes. Line the `return` up with `for`:

`return_in_loop_fixed.py`

```
1 def total(numbers):
2     result = 0
3     for n in numbers:
4         result += n
5     return result
6
7
8 print(total([4, 5, 6]))
```

15

PAUSE AND TRY**Question 4** Tracing

What does this program print? Work from the inside of each expression outward.

nested_calls.py

```
1 def add(a, b):
2     return a + b
3
4
5 def double(n):
6     return n * 2
7
8
9 result = double(add(2, 3))
10 print(result)
11 print(add(double(1), 4))
```

9.5 Returning Versus Printing

Printing and returning can look alike when you run a small program: in both cases a number shows up somewhere. They do completely different jobs.

- **Printing** shows a value to the **person** at the terminal. Once printed, the program cannot use it again.
- **Returning** hands a value to the **program**, at the place where the function was called. The program can store it, do math with it, or print it later.

What does a function give back if it never reaches a `return`? Every call gives back **something**. A function that ends without running a `return`, or that runs a bare `return` with no value, returns `None`, the Day 2 value that means "no value here".

WORKED EXAMPLE Two ways to double

print_vs_return.py

```
1 def show_double(n):
2     print(n * 2)
3
4
5 def get_double(n):
6     return n * 2
7
8
9 a = show_double(5)
10 b = get_double(5)
11 print("a is", a)
12 print("b is", b)
```

```
10
a is None
b is 10
```

Line	What happens	Prints
9	<code>show_double(5)</code> runs line 2, which prints 10; the body ends with no <code>return</code> , so the call gives back <code>None</code>	10
10	<code>get_double(5)</code> runs line 6, which returns 10; nothing is printed	
11	<code>a</code> holds <code>None</code>	<code>a is None</code>
12	<code>b</code> holds 10	<code>b is 10</code>

The 10 on the first line of output came from **inside** `show_double`, printed while the call was running. The value stored in `a` is `None`. Only `get_double` gave the program a number it can use.

The REPL makes this easy to miss. When you type an expression at the `>>>` prompt, the REPL shows its value, so a returned value **appears** on screen even though nobody called `print`. But the REPL shows nothing for `None`:

```
>>> def get_double(n):
...     return n * 2
...
>>> def show_double(n):
...     print(n * 2)
...
>>> get_double(5)
10
>>> show_double(5)
10
>>> answer = get_double(5)
>>> result = show_double(5)
10
>>> print(answer, result)
10 None
```

`get_double(5)` and `show_double(5)` both show 10, for different reasons: the first is the REPL displaying the return value, and the second is the `print` inside the function. The difference appears when you store the results: storing `get_double(5)` prints nothing, while storing `show_double(5)` still prints 10 and stores `None`. In a script there is no automatic display, so a returned value shows up only if you print it.

	<code>print(value)</code> in the body	<code>return value</code> in the body
Who sees the value	The person at the terminal	The code that called the function
What the call gives back	<code>None</code>	value
Can the caller do math with the result?	No	Yes
Typical use	Showing a final result or a message	Working something out

The rule of thumb: a function that **works something out** should return it, and leave printing to the code that calls it. A function whose whole job is to show something, like `greet`, can print, and returns `None`.

KEY IDEA

Printing shows a value to the person; returning gives it to the program. A function with no return (or a bare `return`) gives back `None`.

WATCH OUT

Using the result of a function that only prints. It is easy to write a function that prints its answer and then try to calculate with it:

`none_math.py`

```
1 def show_double(n):
2     print(n * 2)
3
4
5 total = show_double(5) + 1
6 print(total)
```

```
10
Traceback (most recent call last):
  File "../day09/none_math.py", line 5, in <module>
    total = show_double(5) + 1
            ~~~~~^~
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

The 10 printed, so the function seemed to work. But the call gave back `None`, and `None + 1` has no meaning. `NoneType` is the type of `None`. Whenever you see `NoneType` in an error message, ask which function you expected to return a value. Here the fix is to change line 2 to return `n * 2`.

PAUSE AND TRY**Question 5** Multiple choice

What does this program print?

`shout.py`

```
1 def shout(word):
2     print(word.upper() + "!")
3
4
5 print(shout("hi"))
```

- A. HI!
- B. HI! then None
- C. None

D. HI! then HI!

9.6 Testing a Function

Returning has another payoff: a function that returns a value can be **checked**. On Day 4 you ran the grade program several times, typing a new score each time and reading the result. With a function, you can write the checks once, in code, and run them all in one go.

Testing a function means calling it with inputs whose correct answers you already know, and comparing what it returns with those answers. A simple way to start is to print each result next to the answer you expect, so a mismatch stands out. Choose inputs that matter: a typical value, and the values right at each **boundary** (the edges where the answer changes), because that is where mistakes hide.

WORKED EXAMPLE Checking is_teen

is_teen should return True for ages 13 to 19. The interesting ages are the edges: 12 and 13 at the bottom, 19 and 20 at the top.

is_teen.py

```
1 def is_teen(age):
2     return 13 <= age <= 19
3
4
5 print(is_teen(12), "expected False")
6 print(is_teen(13), "expected True")
7 print(is_teen(19), "expected True")
8 print(is_teen(20), "expected False")
```

```
False expected False
True expected True
True expected True
False expected False
```

Every result matches its expected value. Notice also what the function returns: the chained comparison `13 <= age <= 19` (Day 4) is already True or False, so the function can return it directly. There is no need for `if ...: return True else: return False`.

Now suppose someone writes the comparison with `<` instead of `<=`. The same tests, plus a typical age, catch the mistake immediately:

is_teen_bug.py

```
1 def is_teen(age):
2     return 13 < age < 19
3
4
5 print(is_teen(12), "expected False")
6 print(is_teen(13), "expected True")
7 print(is_teen(16), "expected True")
8 print(is_teen(19), "expected True")
9 print(is_teen(20), "expected False")
```

```
False expected False
False expected True
True expected True
False expected True
False expected False
```

The typical age, 16, passes, so a single test with an everyday value would have missed the bug. The two boundary tests, 13 and 19, fail, and they point straight at the cause: the ends of the range are not included. Change `<` back to `<=` and run the tests again.

Testing like this only works because `is_teen` **returns** its answer. A function that printed "teen" or "not a teen" could be checked only by a person reading the output. Day 20 shows how to turn checks like these into proper automated tests.

KEY IDEA

Test a function by calling it with inputs whose answers you know, especially the values at each boundary, and comparing what it returns with what you expect.

WATCH OUT

Testing only the easy values. A function can pass for every "middle" value and still be wrong at the edges, as `is_teen_bug.py` shows. For every comparison in a function, test the value on each side of the line it draws: for `age >= 18`, test 17 and 18.

PAUSE AND TRY**Question 6** Short answer

A function `shipping_cost(total)` should return 0 for orders of 50 or more and 4.99 for smaller orders. Which four test values would you choose, and what should each return? Explain why you chose them.

9.7 Docstrings

Once you have written a function, other people (and you, a month later) need to know how to use it: what it does, what to pass in, and what comes back, without reading every line of the body. Python has a standard place for that information.

A **docstring** (short for documentation string) is a string written as the **first statement** of a function's body. It describes what the function does. By convention it uses triple quotes (Day 3), even when it fits on one line, and it starts with a short sentence such as "Return the area of a rectangle." Unlike a comment, Python keeps the docstring and attaches it to the function. The built-in function `help()` shows a function's header and docstring:

WORKED EXAMPLE Asking for help

docstring_help.py

```
1 def rectangle_area(width, height):
2     """Return the area of a rectangle."""
3     return width * height
4
5
6 help(rectangle_area)
```

Help on function rectangle_area in module __main__:

```
rectangle_area(width, height)
    Return the area of a rectangle.
```

Line 2 is the docstring. It is a string on a line by itself, so it does nothing when the function runs, but Python stores it with the function. `help(rectangle_area)` passes the function itself (no parentheses after its name, because you are not calling it) and prints its parameters and docstring. (`__main__` is the name Python gives the file you run; Day 13 explains it.)

A longer docstring has a one-line summary, a blank line, then more detail, such as what each parameter means. The closing `"""` goes on its own line:

docstring_long.py

```
1 def with_tax(price, rate):
2     """Return a price with sales tax added.
3
4     price is the price before tax, and rate is the tax rate
5     as a decimal, so 0.08 means 8%.
6     """
7     return price + price * rate
8
9
10 help(with_tax)
11 print(f"{with_tax(20, 0.08):.2f}")
```

Help on function with_tax in module __main__:

```
with_tax(price, rate)
    Return a price with sales tax added.
```

```
    price is the price before tax, and rate is the tax rate
    as a decimal, so 0.08 means 8%.
```

```
21.60
```

A reader who sees only `with_tax(price, rate)` might pass 8 for 8%. The docstring tells them to pass 0.08. In the REPL, `help()` may open a full-screen view of the text; press `q` to go back to the `>>>` prompt.

	Comment (#, Day 1)	Docstring
Where	Any line	First statement of a function's body
Kept by Python?	No, ignored	Yes, shown by <code>help()</code>
Written for	Someone reading the code	Someone using the function
Typical content	Why a tricky line works	What the function does, takes, and returns

KEY IDEA

A docstring is a triple-quoted string on the first line of a function's body. It says what the function does, and `help(function_name)` shows it.

WATCH OUT

Putting the docstring anywhere but first. Only a string that is the very first statement in the body counts:

docstring_late.py

```
1 def rectangle_area(width, height):
2     area = width * height
3     """Return the area of a rectangle."""
4     return area
5
6
7 help(rectangle_area)
```

Help on function rectangle_area in module __main__:

```
rectangle_area(width, height)
```

The string on line 3 comes after another statement, so it is just a string value that Python works out and throws away. `help` shows no description. Move the docstring directly under the `def` line.

PAUSE AND TRY

Question 7 Multiple choice

Which version of this function has a docstring that `help()` will show?

- A. `# Return the larger of a and b.` on the line above `def larger(a, b):`
- B. `"""Return the larger of a and b."""` as the first line of the body
- C. `"""Return the larger of a and b."""` as the last line of the body, after the `return`
- D. `print("Return the larger of a and b.")` as the first line of the body

9.8 Local Variables

Inside `rectangle_area`, the parameters `width` and `height` exist only while the function runs. What happens to a variable you create inside a function once the call is over? And what if the rest of the program already uses the same name?

The **scope** of a name is the part of the program where that name can be used. Parameters, and any variable assigned inside a function's body, are **local variables**: they belong to that one function. Python creates them fresh at the start of each call and throws them away when the call ends. Code outside the function cannot see them, and they do not touch any variable outside with the same name.

Think of each call as getting its own scratch paper. The function does its working on that paper, hands back the answer with `return`, and the paper goes in the bin.

WORKED EXAMPLE Two variables called total

same_name.py

```
1 total = 100
2
3
4 def add_tip(bill):
5     total = bill * 1.2
6     return total
7
8
9 meal = add_tip(50)
10 print("meal:", meal)
11 print("total:", total)
```

```
meal: 60.0
total: 100
```

Moment	Outside the function	Inside <code>add_tip</code> (local)
After line 1	<code>total = 100</code>	(no call yet)
During the call, line 5	<code>total = 100</code>	<code>bill = 50</code> , <code>total = 60.0</code>
After line 9	<code>total = 100</code> , <code>meal = 60.0</code>	(thrown away)

There are two different variables named `total`. Line 5 assigns inside the function, so it creates a **local** `total` that has nothing to do with the one on line 1. The outer `total` is still 100 at the end. The only thing that leaves the function is the return value, 60.0, which line 9 stores in `meal`.

The same rule covers parameters. A parameter is a local name that starts out referring to the argument's value. Assigning a new value to the parameter changes only the local name. But if the argument is a list, the parameter refers to the **same** list, and changing the list (for example with `append`) changes it for the caller too. This is the aliasing you saw on Day 6:

shared_list.py

```
1 def reset(score):
2     score = 0
3
4
5 def add_item(items, item):
6     items.append(item)
7
8
9 points = 42
10 reset(points)
11 print("points:", points)
12
13 cart = ["milk"]
14 add_item(cart, "bread")
15 print("cart:", cart)
```

```
points: 42
cart: ['milk', 'bread']
```

`reset` assigns `score = 0`, which only points the local name `score` at a new value; `points` outside is still 42. `add_item` does not assign to `items`; it calls `append` on the list that `items` and `cart` both refer to, so the caller sees `'bread'`. Assigning to a parameter never affects the caller; changing a list through a parameter does.

KEY IDEA

Parameters and variables assigned inside a function are local: they exist only during the call and are separate from any outside variable with the same name. Only the return value comes out, but a list passed in can be changed in place.

WATCH OUT

Using a local variable outside its function. A beginner often expects a variable made inside a function to still be there afterwards:

local_name_error.py

```
1 def average(a, b):
2     total = a + b
3     return total / 2
4
5
6 print(average(4, 6))
7 print(total)
```

```
5.0
Traceback (most recent call last):
  File ".../day09/local_name_error.py", line 7, in <module>
    print(total)
      ^^^^^
NameError: name 'total' is not defined
```

`total` was local to `average`. When the call on line 6 ended, `total` was thrown away, so at line 7 no variable by that name exists. To get a value out of a function, return it and store the result: `result = average(4, 6)`.

PAUSE AND TRY

Question 8 Tracing

What does this program print? Keep track of the two different `x` variables.

`scope_trace.py`

```
1 x = 5
2
3
4 def change(x):
5     x = x * 10
6     print("inside:", x)
7     return x
8
9
10 y = change(x)
11 print("outside:", x, y)
```

9.9 Global Variables

Variables created outside every function, at the left edge of the file, are **global variables**: their scope is the whole file. Can a function use them?

Reading a global variable inside a function works. When Python meets a name in a function's body that is not local, it looks for a global with that name. That suits a setting the whole program shares, such as a tax rate:

WORKED EXAMPLE Reading a global setting

read_global.py

```

1 tax_rate = 0.08
2
3
4 def with_tax(price):
5     return price + price * tax_rate
6
7
8 print(f'{with_tax(10):.2f}')
9 tax_rate = 0.10
10 print(f'{with_tax(10):.2f}')
```

```

10.80
11.00
```

`tax_rate` is not a parameter and is never assigned inside `with_tax`, so on line 5 Python uses the global. It reads the value **at the moment of the call**: the first call sees 0.08, and after line 9 changes the global to 0.10, the second call sees 0.10.

Assigning to a name inside a function is different. On Day 5 you used `count = count + 1` happily; inside a function, with `count` defined outside, it fails:

counter_bug.py

```

1 count = 0
2
3
4 def add_one():
5     count = count + 1
6
7
8 add_one()
9 print(count)
```

```

Traceback (most recent call last):
  File ".../day09/counter_bug.py", line 8, in <module>
    add_one()
    ~~~~~^
  File ".../day09/counter_bug.py", line 5, in add_one
    count = count + 1
           ^^^^^
```

```

UnboundLocalError: cannot access local variable 'count' where it is not associated with a value
```

This traceback has two parts, because the error happened inside a call. Read it from the bottom: the error is on line 5, **in `add_one`**, and the line above says Python got there from the call on line 8.

Here is why it happens. Before a function runs, Python looks through its whole body. Any name that is **assigned** anywhere in the body is local for the **entire** function. `count` is assigned on line 5, so `count` is local everywhere in `add_one`, including on the right side of that same line. Python works out the right side first, looks for a local `count`, and finds that it has not been given a value

yet. `UnboundLocalError` is the error for "this local variable exists but has no value yet". (Older Python versions word it local variable 'count' referenced before assignment.)

There are two fixes. Python's **global statement** tells it that a name in this function means the global variable:

counter_global.py

```
1 count = 0
2
3
4 def add_one():
5     global count
6     count = count + 1
7
8
9 add_one()
10 add_one()
11 print(count)
```

2

`global count` on line 5 means the assignment on line 6 changes the global `count`. It works, but it makes the function harder to understand and test: calling `add_one()` quietly changes something outside it, and you cannot tell what from the call. The usual fix is the one this whole day has been building: pass the value in as an argument and return the new value.

counter_return.py

```
1 def add_one(count):
2     return count + 1
3
4
5 count = 0
6 count = add_one(count)
7 count = add_one(count)
8 print(count)
```

2

Now everything `add_one` uses comes in through its parameter, and everything it produces goes out through `return`. Lines 6 and 7 make the change visible right where it happens.

KEY IDEA

A function can read a global variable, but assigning to a name makes it local for the whole function. To update a value, pass it in and return the new value; use `global` only when you really must.

WATCH OUT

Expecting a function to change a global by assigning to it. In `same_name.py` (9.8), line 5 assigns `total` inside `add_tip` without reading it first. That raises no error, so it is easy to

believe the global `total` changed; it did not, because the assignment made a local. If a function seems to "forget" its changes once it returns, check whether it assigns to a name you meant as the global, and return the value instead.

PAUSE AND TRY

Question 9 Multiple choice

What does this program print?

scope_choice.py

```
1 level = 1
2
3
4 def level_up(level):
5     level = level + 1
6     return level
7
8
9 def show_level():
10    print("Level", level)
11
12
13 level_up(level)
14 show_level()
15 level = level_up(level)
16 show_level()
```

- A. Level 1 then Level 2
- B. Level 2 then Level 3
- C. Level 1 then Level 1
- D. An UnboundLocalError

9.10 Returning Multiple Values

A `return` statement hands back one value. But some jobs have two answers: dividing minutes into hours **and** minutes, or finding the lowest **and** highest score. Day 7 gave you the tool: write several values separated by commas, and Python packs them into one **tuple**.

To **return multiple values**, write them after `return` separated by commas: `return hours, minutes`. The function still returns one value, a tuple, and the caller can unpack it into separate names, as on Day 7.

WORKED EXAMPLE Minutes to hours and minutes

to_hours.py

```
1 def to_hours(total_minutes):
2     hours = total_minutes // 60
3     minutes = total_minutes % 60
4     return hours, minutes
5
6
7 h, m = to_hours(135)
8 print(f"{h} h {m} min")
9
10 result = to_hours(135)
11 print(result)
```

```
2 h 15 min
(2, 15)
```

For 135 minutes, line 2 gives `135 // 60`, which is 2, and line 3 gives `135 % 60`, which is 15 (Day 2). Line 4 returns the tuple `(2, 15)`. On line 7, `h, m = ...` unpacks it: the first item goes to `h`, the second to `m`. On line 10, the tuple is stored in one name without unpacking, and printing it shows the parentheses: this is what the function really returns.

The values can come straight from other function calls. Here `min` and `max` from Day 6 work out the two answers:

score_range.py

```
1 def score_range(scores):
2     return min(scores), max(scores)
3
4
5 low, high = score_range([72, 95, 64, 88])
6 print("Lowest:", low)
7 print("Highest:", high)
8 print("Spread:", high - low)
```

```
Lowest: 64
Highest: 95
Spread: 31
```

Unpacking matches by **position**, exactly like arguments and parameters: the first name gets the first value returned. Name your variables in the same order as the `return`, and say the order in the docstring when you write one.

KEY IDEA

`return a, b` returns one tuple holding both values. Unpack it with `x, y = function(...)`, giving one name per value, in the same order.

WATCH OUT

Unpacking into the wrong number of names. `to_hours` returns two values, so three names cannot all be filled:

`unpack_wrong.py`

```
1 def to_hours(total_minutes):
2     return total_minutes // 60, total_minutes % 60
3
4
5 h, m, s = to_hours(135)
```

```
Traceback (most recent call last):
  File "../day09/unpack_wrong.py", line 5, in <module>
    h, m, s = to_hours(135)
    ^^^^^^^
ValueError: not enough values to unpack (expected 3, got 2)
```

The message gives both counts: 3 names, 2 values. Check the function's `return` line and give exactly one name per value. (One name with no comma is always fine: it stores the whole tuple.)

PAUSE AND TRY**Question 10** Tracing

17 sweets are shared among 5 people. What does this program print?

`swap_order.py`

```
1 def divide(total, people):
2     each = total // people
3     left = total % people
4     return each, left
5
6
7 left, each = divide(17, 5)
8 print("each:", each)
9 print("left:", left)
```

Putting it together

A teacher keeps each student's quiz scores in a dictionary (Day 8) and wants a report: each student's average and letter grade. Instead of one long loop, the program is broken into small functions that each do one thing:

`class_report.py`

```
1 def average(scores):
2     """Return the average of a list of scores."""
3     return sum(scores) / len(scores)
```

```

4
5
6 def letter_grade(score):
7     """Return the letter grade for a score from 0 to 100."""
8     if score >= 90:
9         return "A"
10    elif score >= 80:
11        return "B"
12    elif score >= 70:
13        return "C"
14    elif score >= 60:
15        return "D"
16    return "F"
17
18
19 def report_line(name, scores):
20     """Return one line of the report for one student."""
21     avg = average(scores)
22     return f"{name}: {avg:.1f} ({letter_grade(avg)})"
23
24
25 class_scores = {
26     "Ana": [92, 88, 95],
27     "Ben": [71, 64, 80],
28     "Chen": [85, 90, 78],
29 }
30 for name, scores in class_scores.items():
31     print(report_line(name, scores))

```

```

Ana: 91.7 (A)
Ben: 71.7 (C)
Chen: 84.3 (B)

```

Function	Its one job	Takes	Returns
average	Work out the mean of a list	a list of scores	a float
letter_grade	Turn a number into a letter	one score	a string
report_line	Build one student's line, using the other two	a name and a list	a string

Follow Ana through the loop at the bottom:

1. `class_scores.items()` gives "Ana" and [92, 88, 95], unpacked into `name` and `scores` (Day 8).
2. `report_line("Ana", [92, 88, 95])` is called. Its parameters are local, so they do not clash with the loop's `name` and `scores`, even though they share the names.
3. Inside it, `average(scores)` returns $275 / 3$, which is 91.666..., stored in the local `avg`.
4. `letter_grade(avg)` is called inside the f-string; 91.666... passes `>= 90`, so it returns "A".
5. The f-string shows `avg` with `:.1f` (Day 3), so `report_line` returns "Ana: 91.7 (A)", and the loop prints it.

Notice what the design buys you:

- Each function is short enough to read in one glance, and its docstring says what it does.
- Only the loop at the bottom prints. The three functions return values, so each can be tested on its own: `letter_grade(90)` should return "A", and `average([80, 90])` should return 85.0.

- `letter_grade` has no reason to know about dictionaries or names, so it could be reused in any program that grades a number.
- Everything a function needs comes in through its parameters; nothing depends on global variables.

Vocabulary

Term	Definition	Example
function	A named block of code that does one job	<code>with_tax</code>
call	Run a function by writing its name and parentheses	<code>say_hello()</code>
function definition	The <code>def</code> statement that creates a function	<code>def greet(name):</code>
header	The first line of a definition: <code>def</code> , name, parameters, colon	<code>def area(w, h):</code>
function body	The indented lines that run when the function is called	<code>print("Hi")</code>
flow of execution	The order in which Python runs a program's lines	a call jumps into the body and back
parameter	A name in the header that receives a value on each call	name in <code>def greet(name):</code>
argument	A value passed in a call	"Ana" in <code>greet("Ana")</code>
positional argument	An argument matched to a parameter by its position	<code>describe_pet("dog", "Rex")</code>
return statement	Ends the function and sends a value back to the caller	<code>return width * height</code>
return value	The value a call gives back	12 from <code>rectangle_area(3, 4)</code>
testing	Calling a function with inputs whose answers you know and comparing	<code>is_teen(13)</code> should be <code>True</code>
boundary	An edge where a function's answer changes; test both sides	12 and 13 for <code>is_teen</code>
docstring	A string that is the first statement of a function's body, describing it	<code>"""Return the area."""</code>
help()	Shows a function's header and docstring	<code>help(with_tax)</code>
scope	The part of a program where a name can be used	the body of a function
local variable	A parameter or variable assigned inside a function; exists only during a call	<code>total</code> in <code>average</code>
global variable	A variable created outside every function; its scope is the whole file	<code>tax_rate</code>
global statement	Makes a name in a function refer to the global variable	<code>global count</code>
UnboundLocalError	Reading a local variable before it has a value	<code>count = count + 1</code> in a function

Summary

- A **function** names a block of code that does one job. `def name(parameters):` defines it without running it; `name(arguments)` calls it, runs the body, and comes back. Functions give you reuse, readable names, and pieces you can test.
- **Parameters** are the names in the header; **arguments** are the values in the call, matched by position. A call with the wrong number of arguments is a `TypeError`.
- `return value` ends the function and hands the value back to the caller. Printing shows a value to the person; returning gives it to the program. A function with no `return` gives back `None`.
- Test a function by calling it with known answers, especially at the **boundaries**. A **docstring** on the first line of the body says what the function does, and `help()` shows it.
- Parameters and variables assigned in a function are **local**. A function can read **global** variables, but assigning to a name makes it local; pass values in and return results instead. `return a, b` returns a tuple, which the caller unpacks.

Practice

Do these in your `day09` folder. Type each program yourself, run it, and only then open the answer.

Question 11 Short answer

Converter and grade calculator as functions. Rewrite two earlier programs so each does its work in a function that **returns** the result:

- Day 2's Fahrenheit-to-Celsius converter: write `fahrenheit_to_celsius(fahrenheit)`, which returns the Celsius temperature. The program should still print
`72 degrees Fahrenheit is 22.2 degrees Celsius`. Add two test calls whose answers you already know.
- Day 4's letter grade calculator: write `letter_grade(score)`, which returns the letter. The program should still read a score with `input()` and print `Grade: B` for 89.5, or
`Scores run from 0 to 100. for a score outside 0 to 100.`

Question 12 Short answer

Primes. A **prime number** is a whole number greater than 1 whose only divisors are 1 and itself: 2, 3, 5, and 7 are prime; 9 is not, because 3 divides it. Write `is_prime(n)`, which returns `True` if `n` is prime and `False` otherwise. Then use it to build a list of all the primes under 100 and print the list and how many there are.

Question 13 Short answer

Count, total, and average. Write a function that takes a list of numbers and returns three values: how many numbers there are, their total, and their average. Unpack the result and print it. Make sure an empty list does not crash your program.

Review questions

Question 14 Tracing

What does this program print?

mystery.py

```
1 def mystery(n):
2     print("got", n)
3     if n > 10:
4         return "big"
5
6
7 print(mystery(15))
8 print(mystery(3))
```

Question 15 Debugging

has_negative should return True if any number in the list is negative. For [3, -1, 4] it prints False. Explain the bug and write the fix.

has_negative.py

```
1 def has_negative(numbers):
2     for n in numbers:
3         if n < 0:
4             return True
5         else:
6             return False
7
8
9 print(has_negative([3, -1, 4]))
```

Question 16 Multiple choice

What does this program print?

rate_scope.py

```
1 rate = 2
2
3
4 def pay(hours, rate):
5     rate = rate + 1
6     return hours * rate
7
8
9 print(pay(4, 10))
10 print(rate)
```

- A. 40 then 2
- B. 44 then 2
- C. 44 then 11
- D. 12 then 3

Question 17 Short answer

A 40-line script reads a list of prices typed by the user, removes any that are negative, works out the total with 8% tax, and prints a receipt. Suggest how to break it into functions: name three or four functions, and for each say what it takes, what it returns, and whether it should print.

Question 18 Multiple choice

You want to test a squaring function with `print(square(4), "expected 16")`. Which version prints 16 expected 16?

- A. `def square(n):` with the body `print(n * n)`
- B. `def square(n):` with the body `return n * n`
- C. `def square(n):` with the body `result = n * n`
- D. `def square(n):` with the body `return` on one line and `n * n` on the next

Go further

The official Python Tutorial covers today's ideas in [More Control Flow Tools: Defining Functions](#), and its section on [Documentation Strings](#) shows the docstring conventions. The parts after that, on default and keyword arguments, are tomorrow's topic, Day 10.

Reflection

Check each statement. If the answer is "not yet," reread the linked subsection.

- ☐ I can explain why a function is better than copying the same code. Reread: [9.1 Why Functions](#)
- ☐ I can define a function with `def` and say when its body runs. Reread: [9.2 Defining and Calling a Function](#)
- ☐ I can tell a parameter from an argument and match arguments by position. Reread: [9.3 Parameters and Arguments](#)
- ☐ I can use `return` to send a value back and use the call in an expression. Reread: [9.4 return: Sending a Value Back](#)
- ☐ I can choose between returning and printing, and spot a function that gives back `None`. Reread: [9.5 Returning Versus Printing](#)
- ☐ I can test a function with known answers, including the boundaries. Reread: [9.6 Testing a Function](#)
- ☐ I can write a docstring and read one with `help()`. Reread: [9.7 Docstrings](#)
- ☐ I can explain which variables are local to a function. Reread: [9.8 Local Variables](#)
- ☐ I can explain why assigning to a global inside a function fails, and fix it with a parameter and `return`. Reread: [9.9 Global Variables](#)
- ☐ I can return several values and unpack them in the right order. Reread: [9.10 Returning Multiple Values](#)

Check Your Reasoning

Answers and explanations for every question, numbered to match.

Question 1

B. In `tax_repeated.py`, `0.08` appears in three formulas (lines 3, 7, and 11), so you must change all three. In `tax_function.py`, the formula appears once, inside the function on line 2. Every call uses that one line, so one edit changes all three totals.

- **A** is wrong: `tax_repeated.py` has three separate copies of the formula, and each is its own code.
- **C** is wrong: the three calls on lines 5 to 7 only pass prices; none of them contains the rate.
- **D** is wrong: each copy in `tax_repeated.py` has `0.08` once, not twice, and the calls in `tax_function.py` hold no rate at all.

Reread 9.1 *Why Functions* if you chose differently.

Question 2

```
A
C
B
D
B
```

Line 1 prints `A`. Lines 4 and 5 are a definition, so nothing prints there, even though `print("B")` is written above `print("C")`. Line 8 prints `C`. Line 9 calls `show`, which prints `B`. Line 10 prints `D`, and line 11 calls `show` again for a second `B`. If you wrote `A B C D B`, you ran the body when Python reached the `def`; it runs only when called.

Reread 9.2 *Defining and Calling a Function* if your output differed.

Question 3

```
Row C, seat 7
Row 2, seat A
```

The first call matches `"C"` to `row` and `7` to `seat`. The second call passes the variable `seat` (2) first and `row` (`"A"`) second. Arguments match parameters **by position**, not by name, so the parameter `row` receives 2 and the parameter `seat` receives `"A"`. The variables outside the function happen to share the parameters' names, but that does not matter. If you answered `Row A, seat 2`, you matched by name.

Reread 9.3 *Parameters and Arguments* if your output differed.

Question 4

```
10
6
```

Line 9: Python must know the argument before it can call `double`, so it calls `add(2, 3)` first, which returns 5. Then `double(5)` returns 10. Line 11: the arguments of `add` are worked out first, left to

right: `double(1)` returns 2, and 4 is just 4. Then `add(2, 4)` returns 6. A call inside another call's parentheses runs first, the same inside-out order as `int(input(...))` on Day 3.

Reread 9.4 *return: Sending a Value Back* if your output differed.

Question 5

B.

```
HI!  
None
```

Python must work out the argument of the outer `print` first, so it calls `shout("hi")`. The body prints `HI!`. The function has no `return`, so the call gives back `None`, and the outer `print` prints that.

- **A** is wrong: it forgets that the outer `print` still prints whatever the call gave back.
- **C** is wrong: the `print` inside the function does run, before the outer one.
- **D** is wrong: it assumes `shout` returns the text it printed. It does not; it returns `None`.

Reread 9.5 *Returning Versus Printing* if you chose differently.

Question 6

Model answer: test 49.99 (expect 4.99) and 50 (expect 0), because 50 is the boundary where the answer changes, and a mistake such as `>` instead of `>=` would show up only there. Add one ordinary value on each side, such as 20 (expect 4.99) and 80 (expect 0), to check that the normal cases work too. Testing 0 (expect 4.99) is also sensible, since an empty order is an edge case.

A correct answer includes 50 itself and a value just below it, gives the expected result for each, and says the boundary is where mistakes usually hide.

Reread 9.6 *Testing a Function* if your reasoning differed.

Question 7

B. A docstring is a string that is the first statement inside the function's body.

- **A** is wrong: a comment is ignored by Python, and it is outside the function anyway.
- **C** is wrong: a string after other statements is not a docstring; this one would never even be reached, because the `return` ends the function first.
- **D** is wrong: `print` is a call, not a string. It would print the sentence every time the function runs, and `help()` would show no description.

Reread 9.7 *Docstrings* if you chose differently.

Question 8

```
inside: 50  
outside: 5 50
```

The call on line 10 passes 5, so the parameter `x`, a local variable, starts at 5. Line 5 makes the local `x` 50, line 6 prints it, and line 7 returns 50, which is stored in `y`. The `x` on line 1 is a different variable and was never assigned again, so it is still 5. If you answered `outside: 50 50`, you treated the parameter as if it were the outer variable.

Reread 9.8 *Local Variables* if your output differed.

Question 9

A.

```

Level 1
Level 2

```

Line 13 calls `level_up(level)`. Inside, the parameter `level` is a local variable, so line 5 changes only the local copy to 2, and the return value is thrown away because nothing stores it. `show_level` reads the global `level`, still 1. Line 15 stores the return value in the global `level`, so it becomes 2, and `show_level` prints `Level 2`.

- **B** is wrong: the first call's result is never stored, so the global does not change on line 13.
- **C** is wrong: line 15 does store the returned 2 in the global `level`.
- **D** is wrong: in `level_up`, `level` is a parameter, so it already has a value when line 5 reads it. `show_level` only reads `level`, so it uses the global.

Reread 9.9 *Global Variables* and 9.8 *Local Variables* if you chose differently.

Question 10

```

each: 2
left: 3

```

`divide(17, 5)` returns `(3, 2)`: 3 each (`17 // 5`) and 2 left over (`17 % 5`). But line 7 unpacks by position into `left`, `each`, so `left` gets 3 and `each` gets 2: the labels are swapped. The function is correct; the caller named the values in the wrong order. Writing `each, left = divide(17, 5)` fixes it.

Reread 9.10 *Returning Multiple Values* if your output differed.

Question 11

One possible solution for the converter:

temperature_function.py

```

1 def fahrenheit_to_celsius(fahrenheit):
2     """Return a Fahrenheit temperature converted to Celsius."""
3     return (fahrenheit - 32) * 5 / 9
4
5
6 fahrenheit = 72
7 celsius = fahrenheit_to_celsius(fahrenheit)
8 print(fahrenheit, "degrees Fahrenheit is", round(celsius, 1), "degrees Celsius")
9
10 # Quick checks with answers we already know
11 print(fahrenheit_to_celsius(212), "expected 100.0")
12 print(fahrenheit_to_celsius(32), "expected 0.0")

```

```

72 degrees Fahrenheit is 22.2 degrees Celsius
100.0 expected 100.0
0.0 expected 0.0

```

The formula is unchanged from Day 2, but now it lives in a function that returns the exact result. Rounding and printing stay outside, because they are about how to **show** the result, not how to

work it out. The boiling and freezing points of water make good tests because you know their answers without a calculator.

One possible solution for the grade calculator, typing 89.5:

grade_function.py

```
1 def letter_grade(score):
2     """Return the letter grade for a score, or None if it is not 0 to 100."""
3     if not 0 <= score <= 100:
4         return None
5     elif score >= 90:
6         return "A"
7     elif score >= 80:
8         return "B"
9     elif score >= 70:
10        return "C"
11    elif score >= 60:
12        return "D"
13    else:
14        return "F"
15
16
17 score = float(input("Score: "))
18 grade = letter_grade(score)
19 if grade is None:
20     print("Scores run from 0 to 100.")
21 else:
22     print("Grade:", grade)
```

```
Score: 89.5
Grade: B
```

The if chain from Day 4 moves into the function, and each `print("Grade: ...")` becomes a return. For a score outside 0 to 100 there is no letter to give back, so the function returns `None` ("no value here"), and the code that calls it checks with `is None` and prints the message. Keeping `input()` and `print()` outside the function means `letter_grade` can be tested with calls such as `letter_grade(90)` (expect "A") and `letter_grade(59.9)` (expect "F") without typing anything.

Reread 9.4 *return: Sending a Value Back* and 9.5 *Returning Versus Printing* if your result differed.

Question 12

One possible solution:

primes.py

```
1 def is_prime(n):
2     """Return True if n is a prime number, otherwise False."""
3     if n < 2:
4         return False
5     for divisor in range(2, n):
6         if n % divisor == 0:
7             return False
8     return True
9
10
11 primes = []
12 for number in range(1, 100):
13     if is_prime(number):
14         primes.append(number)
15 print(primes)
16 print(len(primes), "primes under 100")
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,
89, 97]
25 primes under 100
```

Numbers below 2 are not prime, so the function returns `False` straight away. Otherwise it tries every possible divisor from 2 up to `n - 1`. The first one that divides evenly (remainder 0, Day 2) proves `n` is not prime, and `return False` ends the function at once, like a `break`. Only if the loop finishes without finding a divisor does Python reach `return True`, which is lined up with `for`, **after** the loop. The bottom of the program is the Day 6 pattern of building a list with `append`; it reads almost like English because the hard part has a name.

To test it, check the edges: `is_prime(1)` should be `False`, `is_prime(2)` should be `True` (the loop has no divisors to try), and `is_prime(9)` should be `False`.

Reread 9.4 *return: Sending a Value Back* if your result differed.

Question 13

One possible solution:

number_stats.py

```

1 def count_total_average(numbers):
2     """Return the count, total, and average of a list of numbers.
3
4     The average is None when the list is empty.
5     """
6     count = len(numbers)
7     total = sum(numbers)
8     average = total / count if count > 0 else None
9     return count, total, average
10
11
12 count, total, average = count_total_average([4, 8, 15, 16, 23, 42])
13 print(f"{count} numbers, total {total}, average {average}")
14
15 count, total, average = count_total_average([])
16 print(f"{count} numbers, total {total}, average {average}")

```

```

6 numbers, total 108, average 18.0
0 numbers, total 0, average None

```

`len` and `sum` (Day 6) give the count and total. The average needs care: for an empty list, `total / count` would be `0 / 0`, a `ZeroDivisionError`. The conditional expression from Day 4 gives `None` in that case, which honestly says "there is no average". `return count, total, average` returns a tuple of three, and each call unpacks it into three names in the same order. The docstring tells callers about the `None`, so they know to check for it.

Reread 9.10 *Returning Multiple Values* if your result differed.

Question 14

```

got 15
big
got 3
None

```

Call	Line 2 prints	<code>n > 10</code>	Returns	Outer print shows
<code>mystery(15)</code>	got 15	True	"big"	big
<code>mystery(3)</code>	got 3	False	None	None

Each outer print must call `mystery` first, so `got ...` always appears before the result. For 3, the `if` is skipped and the body ends without reaching a `return`, so the call gives back `None`. A function that returns a value on only some paths returns `None` on the others.

Reread 9.5 *Returning Versus Printing* and 9.4 *return: Sending a Value Back* if your output differed.

Question 15

On the first pass, `n` is 3. `3 < 0` is `False`, so the `else` runs `return False`, and the function ends after looking at only one number. The `-1` is never checked. "No negatives" is only known after **every** number has been checked, so `return False` belongs after the loop, the same "not found" idea as the loop `else` from Day 5:

has_negative_fixed.py

```

1 def has_negative(numbers):
2     for n in numbers:
3         if n < 0:
4             return True
5     return False
6
7
8 print(has_negative([3, -1, 4]))
9 print(has_negative([3, 1, 4]))
10 print(has_negative([]))

```

```

True
False
False

```

Now the loop returns `True` the moment it finds a negative, and only a loop that finishes without finding one reaches `return False`. The empty list has nothing to check, so it correctly returns `False`.

Reread 9.4 *return: Sending a Value Back* if you missed it.

Question 16

B.

```

44
2

```

`rate` is a parameter of `pay`, so inside the function it is a local variable holding the argument, 10, not the global 2. Line 5 makes the local `rate` 11, and `4 * 11` is 44. The global `rate` is never assigned after line 1, so it still prints 2.

- **A** is wrong: line 5 runs before the `return`, so the `rate` used is 11, not 10.
- **C** is wrong: assigning to the parameter changes only the local name; the global `rate` stays 2.
- **D** is wrong: it treats `rate` inside the function as the global 2 and has line 5 change the global to 3. A parameter with the same name as a global hides the global inside that function.

Reread 9.8 *Local Variables* if you chose differently.

Question 17

Model answer (other good designs exist):

- `read_prices()` takes nothing, asks for prices with `input()` until the user is done, and returns a list of floats.
- `remove_negatives(prices)` takes a list and returns a new list with only the prices of 0 or more. It does not print.
- `total_with_tax(prices, rate)` takes a list and a rate and returns the total with tax added. It does not print.
- `print_receipt(prices, total)` takes the list and the total and prints the receipt. It returns `None`.

A good answer gives each function one job, named with a verb; keeps the calculating functions free of `print` so they can be tested with known values; and passes data in through parameters

and out through return values rather than through global variables. Only the functions whose job is talking to the user call `input()` or `print()`.

Reread *9.1 Why Functions*, *9.5 Returning Versus Printing*, and *Putting it together* if your reasoning differed.

Question 18

B. Only a function that returns its answer gives the caller a value to print or compare; `return n * n` returns 16.

- **A** is wrong: it prints 16 inside the function, then returns `None`, so the test line prints 16 and then `None` expected 16.
- **C** is wrong: `result` is a local variable that is thrown away when the call ends; with no `return`, the call gives back `None`.
- **D** is wrong: a bare `return` ends the function at once and gives back `None`; the line `n * n` never runs.

Reread *9.5 Returning Versus Printing* and *9.6 Testing a Function* if you chose differently.