Python Functions

Table of contents

Defining a function 1
The `return` statement 2
Type hints 6
Docstrings 7
General template 7
Worked question — a function that computes but never returns 7

User-defined functions bundle lines of code together so they can be reused, abstracting away complexity. Like mathematical functions, they take input and return output (we've already used built-in functions this way, e.g. `*` and `max`). Multiplication takes two numbers and returns a number; `max` takes two or more numbers and returns a number.

Defining a function

A mathematical function such as $f(x) = x^2 + x + 1$ is written in Python as:

```
>>> def f(x):
...     return x**2 + x + 1
>>> f(3)
13
```

Functions can take multiple parameters, and a call is an expression like any other — so calls nest inside other calls and inside arithmetic:

```
>>> def f(x, y):
...     return x*y
>>> x = f(3, 2)
>>> x
```

```
6
>>> x = f( -f(5, 2) + 12, f(2, 3) )
>>> x
12
```

The inner calls are evaluated before the outer one, so the last line is `f(-10 + 12, 6)`, which is `f(2, 6)`, and so `12`.

Indentation

Four spaces of indentation are significant in Python — they associate a line of code with the control structure above it (here, the function body with its `def`). Get the indentation wrong and you get

```
IndentationError: unexpected indent
```

which is a formatting complaint, not a logic error.

The `return` statement

`return` is a **reserved word** — one that cannot be assigned by us. It designates the value the function hands back **and terminates the function itself**.

> **Definition (`return`).** The `return` statement causes a function to exit and hand back a value to its caller.

Two consequences follow:

1. Statements before the `return` still run (including their side effects).
2. Nothing after the first `return` that is actually reached ever runs — not a further `print`, not a second `return`.

Three variants of the same function show both. In the first, both `print`s are above the `return`, so both run:

```
>>> def example(x):
...     print(1*x)
...     print(2*x)
...     return 3*x

>>> a = example(1)
```

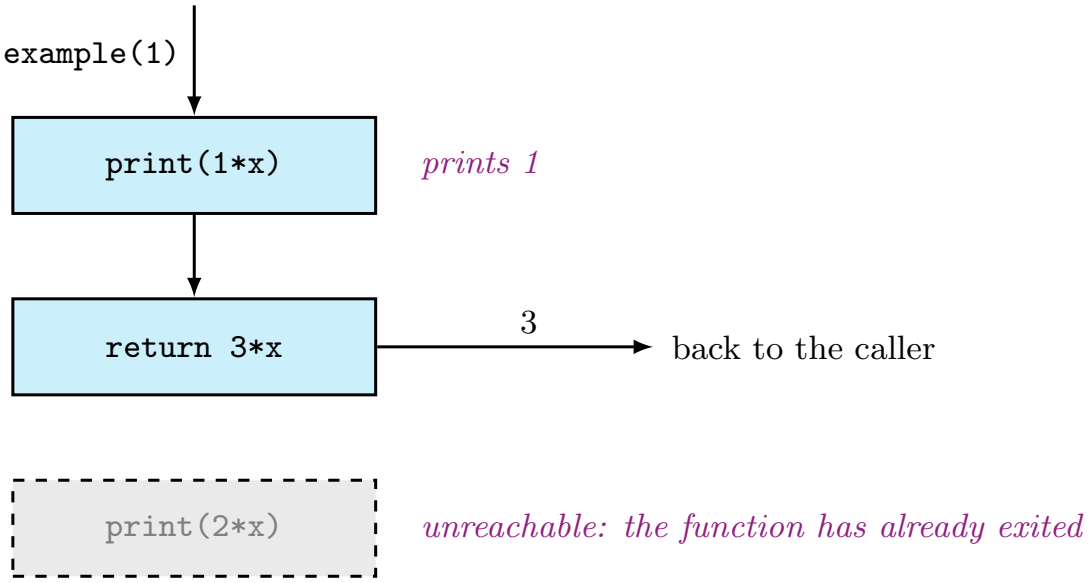

Figure 1: Control through the body of `def example(x): print(1*x); return 3*x; print(2*x)` when called as `example(1)`. The `print` above the `return` runs and displays 1; the `return` hands 3 back and ends the call there, so the `print` below it never executes. The dashed grey box marks that unreachable statement.

```
1
2
>>> a
3
```

Move the `return` up one line and the second `print` is stranded below it:

```
>>> def example(x):
...     print(1*x)
...     return 3*x
...     print(2*x)

>>> a = example(1)
1
>>> a
3
```

Put the `return` first and *nothing* below it runs — not the second `return`, not the `print`. The call produces no output at all:

```
>>> def example(x):
...     return 3*x
...     return 2*x
...     print(1*x)

>>> a = example(1)
>>> a
3
```

If a function has no `return` statement at all, Python presumes `return None` is its last line:

```
>>> def f(x, y):
...     print(x+y)
...     return None      # presumed -- you do not write this
```

so the "no return" case is not special, it just returns `None`:

```
>>> def example(x):
...     print(2*x)
>>> a = example(1)
2
>>> a
>>> type(a)
<class 'NoneType'>
```

`>>> a` displays nothing at all, because the REPL does not echo `None`.

`return` versus `print`

`print` is a function that **returns nothing** and whose **side effect** is that it displays something:

```
>>> x = print("Hello World")
Hello World
>>> type(x)
<class 'NoneType'>
```

The distinction matters because `return` and `print` can look like they are doing the same thing — right up until the result is used in further computation:

```
>>> def f(x, y):
...     print(x+y)
>>> x = f(2, 3)
5
>>> y = f(3, 4)
7
>>> x + y
TypeError: unsupported operand type(s) for +: 'NoneType' and 'NoneType'.
```

Compare with the use of `return`:

```
>>> def f(x, y):
...     return x + y
>>> x = f(2, 3)
>>> y = f(3, 4)
>>> x + y
12
```

`return` is a reserved word, **not a function** — so we write

```
>>> def f(x, y):
...     return x + y
```

and not

```
>>> def f(x, y):
...     return(x + y)
```

Type hints

Type hints annotate the expected type of each parameter and of the return value. In mathematics we would define a mapping triangle_area : $\mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}$; in Python we write that as:

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
>>> triangle_area(2.2, 3.3, 4.4)
3.5147323866832307
```

Type hints are **not enforced** — they are documentation for making the code more readable, nothing more. Annotating the *same* function with `str` everywhere changes nothing about how it runs:

```
>>> def triangle_area(a: str, b: str, c: str) -> str:
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
>>> triangle_area(2.2, 3.3, 4.4)
3.5147323866832307
```

and so this is valid Python too, hints notwithstanding:

```
def foo(x: int, y: int) -> int:
    """
    >>> foo(2, 3)
    6
    """
    return x
ans = foo("oi", 3)
```

Docstrings

A **docstring** (`"""..."""` immediately under the `def` line) documents what a function does and its preconditions, and gives example calls (written like REPL input/output) that double as tests — see python-testing[1]:

```
>>> def triangle_area(a: float, b: float, c: float) -> float:
...     """
...     Return the area of the triangle with sides length <a>,
...     <b>, and <c>.
...     Preconditions: <a>, <b>, <c> are all nonzero positive.
...     >>> triangle_area(3, 4, 5)
...     6.0
...     """
...     s = (a+b+c)/2
...     return (s*(s-a)*(s-b)*(s-c))**0.5
```

Inside a docstring the course refers to a parameter by its name in angle brackets, `<a>`.

General template

```
def function_name(arg0: type, arg1: type, ...) -> type:
    """
    Short description of function for documentation.
    Preconditions (if any).
    >>> function_name(x, y, ...)
    expected output
    """
    ...
    function body
    ...
    return
```

Worked question — a function that computes but never returns

A function that computes a value but never `return`s it still hands back `None`. What is the type of `foo`'s return value?

[1] courses/csse1001/python-testing.pdf

```
def foo(radius: int):
    """
    Precondition:  radius > 0
    """
    area = 3.14159 * radius**2
```

`NoneType`. There is no explicit `return` statement, so the computed `area` is simply discarded when the call ends.

See python-pep8-style-guide[2] for the naming, spacing, and line-length conventions used when writing functions like this.

[2] courses/csse1001/python-pep8-style-guide.pdf