

A function is a named block of code that you can run again and again. You give it values, it does its work, and it can send a result back with `return`.

DEF AND RETURN

```
def total(price, qty):  
    return price * qty  
def names the function. return sends a value back to  
the code that called it  
  
total(2.5, 4) → 10.0  
to call the function, write its name, then the values in ()  
  
def greet(name):  
    print("Hello", name)  
this function has no return line  
  
greet("Ada") is None → True  
greet() prints, but the value it sends back is None
```

PARAMETERS AND ARGUMENTS

A parameter is a name in the def line (price, rate). An argument is the value you pass in the call (10, 0.1).

```
def with_tax(price, rate=0.2):  
    return round(price * (1 + rate), 2)  
  
with_tax(10) → 12.0  
rate is not given, so it uses its default, 0.2  
  
with_tax(10, 0.1) → 11.0  
passed by position: in the same order as in the def line  
  
with_tax(rate=0.5, price=10) → 15.0  
passed by keyword: name=value, in any order  
  
with_tax() → raises TypeError  
price has no default, so you must pass it
```

*ARGS AND **KWARGS

The * and ** in the def line collect the extra arguments. The names args and kwargs are only a habit: *nums works the same.

```
def show(*args, **kwargs):  
    return args, kwargs  
  
def add_all(*nums):  
    return sum(nums)  
  
show(1, 2) → ((1, 2), {})  
args holds the extra values passed by position, as a tuple  
  
show(size="L", qty=3)  
→ ((), {'size': 'L', 'qty': 3})  
kwargs holds the extra name=value pairs, as a dict  
  
add_all(4, 5, 6) → 15  
use *args when the number of values can change
```

* AND ** IN A CALL: UNPACK

```
def total(price, qty):  
    return price * qty  
  
order = [2.5, 4]  
total(*order)  
→ 10.0  
* spreads a list into separate arguments  
  
order = {"price": 3, "qty": 2}  
total(**order)  
→ 6  
** spreads a dict into name=value arguments
```

/ AND * IN THE DEF LINE

Names before / are passed by position only. Names after * are passed by keyword only.

```
def move(x, y, /, *, speed=1):  
    return x, y, speed  
  
move(1, 2, speed=5) → (1, 2, 5)  
x and y by position, speed by keyword  
  
move(1, 2, 5) → raises TypeError  
speed comes after *, so you must name it  
  
move(x=1, y=2) → raises TypeError  
x and y come before /, so you must not name them
```

RETURN SEVERAL VALUES

```
def min_max(scores):  
    return min(scores), max(scores)  
  
min_max([70, 92, 81]) → (70, 92)  
the comma makes a tuple: one value that holds two values  
  
low, high = min_max([70, 92, 81])  
high  
→ 92  
unpack the tuple: each name gets one value
```

SCOPE: WHERE A NAME CAN BE USED

```
score = 10  
  
def reset():  
    score = 0  
reset()  
score  
→ 10  
a name set inside a function is local: a new name that exists only inside the function  
  
def add_point():  
    global score  
    score += 1  
global lets the function change the score outside it. Use it rarely: return the new value instead  
  
def counter():  
    n = 0  
    def step():  
        nonlocal n  
        n += 1  
        return n  
    return step  
nonlocal lets step() change n, which belongs to counter()  
  
step = counter()  
step(), step()  
→ (1, 2)  
step() keeps n between calls
```

LAMBDA AND FUNCTIONS AS VALUES

```
people = [("Ada", 36), ("Linus", 28),  
          ("Grace", 45)]  
  
sorted(people, key=lambda p: p[1])  
→ [('Linus', 28), ('Ada', 36), ('Grace', 45)]  
lambda p: p[1] is a small function with no name: it takes p and returns p[1]  
  
list(map(str.upper, ["ada", "hedy"]))  
→ ['ADA', 'HEDY']  
pass the function itself, without ()  
  
size = len  
size("Grace")  
→ 5  
a function is a value: you can give it another name
```

DOCSTRINGS AND TYPE HINTS

```
def area(w: float, h: float) -> float:  
    """Area of a w by h rectangle."""  
    return w * h  
the string on the first line is the docstring: help(area) shows it. The hints (: float, → float) say which types you expect  
  
area("ab", 3) → 'ababab'  
Python does not check the hints when the code runs
```

A TINY DECORATOR

```
def shout(func):  
    def wrapper(name):  
        return func(name).upper()  
    return wrapper  
a decorator takes a function and returns a new function. The new one calls the old one and does extra work  
  
@shout  
def hello(name):  
    return f"hello {name}"  
@shout here is the same as hello = shout(hello)  
  
hello("Hedy") → 'HELLO HEDY'  
the new function makes the result upper case
```

WATCH OUT

```
def add(item, cart=[]):  
    cart.append(item)  
    return cart  
add("tea")  
add("pie")  
→ ['tea', 'pie']  
the default list is made once and shared by every call. Use cart=None, then cart = [] inside  
  
def double(n):  
    n * 2  
double(4)  
→ None  
there is no return, so the result is lost and you get None  
  
def total(a, b):  
    print(a + b)  
print only shows the value; total() returns None, so total(1, 2) + 1 raises TypeError. Use return  
  
sorted(names, key=len())  
TypeError: pass len, not len(). len is the function, len() calls it  
  
list = ["tea"]  
list("ab")  
→ raises TypeError  
the name list now hides the built-in list(): choose another name, such as items
```