C: Functions

Rough, practical definition:

A **function** is a **named block of code** that:

1. Starts with a **def** statement
2. Includes subsequent **indented lines**

3. Is **called** or invoked (executed) by its name

4. Accepts [usually] a list of **arguments** or **parameters** (inputs)
5. Returns [usually] a **result** (output)

Schematically:

Some examples to date:

Function call	**Arguments**	**Returned**
a_len = **len(a_list)**	a_list	Length of a_list
print("The length is:", a)	"The length is:", a	Nothing: prints message
sorted_oath = **sorted(oath)**	oath	Sorted list
somelist.append(item)	item	Nothing: somelist updated

Creating a function: summing up a list

Defining function **sumup()**:

```
def sumup(values):
    total = 0
    for v in values:
        total = total + v
    return total
```

values: argument (input)

total: returned (output)

Recent versions of Python allow **type hinting**:

```
def sumup(values: list) -> float:
    total = 0
    for v in values:
        total = total + v
    return total
```

Hints that **values** should be a **list** and the function returns a **float**

Some common types: **str, int, float, bool, list, dict, tuple**

Aside: to find out the type of a variable use the type function:

```
x = "otto"
print( type(x) )
```

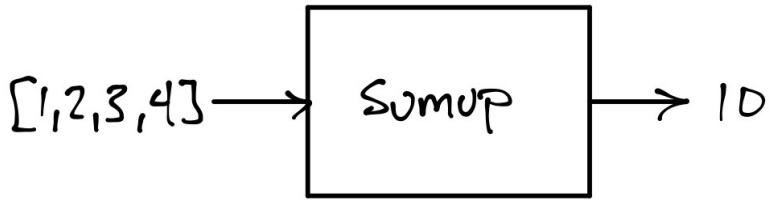
Prints str

Calling the function (using it):

```
nums1 = [1, 2, 3, 4]

tot1 = sumup(nums1)        Calls sumup on nums1

print(tot1)                prints 10
```



Arguments are variables passed from **outside** the function to **inside** it

- Linked by definition and call:

  Definition: def sumup(**values**):

  ↕

  Call: tot1 = sumup(**nums1**)

- In effect, call works like this:

  Implicit assignment at start of function:

  values ← **nums1**

  Makes an *inside* copy of the *outside* variable

- Same process for multiple arguments:

  Outside and inside variables [generally] matched by **position**

Definition: def other(**alpha**, **beta**, **gamma**):

↕ ↕ ↕

Call: res = other(12, "tons", 3.14)

Implicit assignments within other():

alpha ← 12
beta ← "tons"
gamma ← 3.14

Benefits of functions:

1. Make repeat calculations **easier**:

   Don't have to write the same code multiple times

2. Make code shorter and logic cleaner and clearer (more **readable**):

With sumup()	Without sumup()

nums1 = [1,2,3,4]
nums2 = [9,8,7]
 	
tot1 = sumup(nums1)
tot2 = sumup(nums2)
 	
print(tot1,tot2)
	nums1 = [1,2,3,4]
	nums2 = [9,8,7]
	
	tot1 = 0
	for v in nums1:
	  tot1 = tot1 + v
	tot2 = 0
	for v in nums1:
	  tot2 = tot2 + v
	
	print(tot1,tot2)

3. Make code more **reliable**:

   One version to get right rather than several

   Coder mantra: "*Don't repeat yourself*"

Exception to positional matching: **optional** arguments with **default values**:

Defining a function with a default:

```
def sumup2(values: list, start: float = 0) -> float:
    total = start
    for v in values:
        total = total + v
    return total
```

start is optional: if not given, it will default to 0

Type hints for optional arguments go between the name and the equals

With a default, *start*:

1. Can be *omitted* (default argument)
2. Can be given *second with no name* ("positional" argument)
3. Can be given *by name* ("keyword" argument)

Using the function:

```
vals = [10,20,30]

tot1 = sumup2(vals)
tot2 = sumup2(vals, 5)
tot3 = sumup2(vals, start=10)
```

tot1 👉 60 default

tot2 👉 65 **positional**

tot3 👉 70 **keyword**

Very powerful feature: can use keyword arguments *selectively* and *in any order*:

```
def sumup3(values, start=0, power=1):
    total = start
    for v in values:
        total = total + v**power
    return total
```

```
vals = [10,20,30]

tot4 = sumup3(vals, start=5)
tot5 = sumup3(vals, power=2)
tot6 = sumup3(vals, power=2, start=40)
```

tot6 👉 65
tot7 👉 1400
tot8 👉 1440

Very important subtlety with functions: variable **scoping**:

Scope of a variable: portion of code where it is **defined** and **can be used**

Key issue: what values do variables have **inside** and **outside** the function?

Roughly speaking:

- Functions have **their own copies** of variables
- Changes *inside* functions generally **do not** change variables *outside*

Example:

Define a function:

```
def fun(a):
   a = a + 2
   b = a**2  + c
   return b
```

a: argument and also changed within the function

b: set within function

c: from outside the function

Using it:

```
a = 10
b = 20
c = 30
d = fun(5)
print( [a, b, c, d] )
```

What gets printed?

Outside	Inside fun()	How set?
a 👉 10		Set outside
b 👉 20		Set outside
c 👉 30		Set outside
d = fun(5)	a 👉 5	Given as an argument
	a 👉 5+2 👉 7	Revised in function
	c 👉 30	Read from outside function
	b 👉 7**2 + 30 👉 79	Calculated in function
d 👉 79		Returned by function

a 👉 10 Unchanged
b 👉 20 Unchanged
c 👉 30 Unchanged

Printed result: [10,20,30,79]

Why important?

- Functions **can't** accidentally **damage** outside variables
- Can be written without knowing every possible context

Note: best practice about passing variables to functions:

- If possible, include all outside variables in the argument list
- Avoid using other external variables