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** (executed) by its name

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

Schematically:

Example: summing up a list

Defining function **sumup()**:

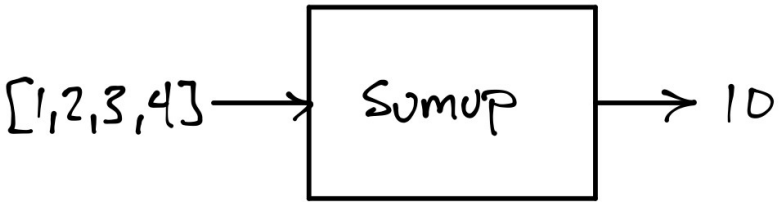
```
def sumup(values):
    total = 0
    for v in values:
        total = total + v
    return total
```

values: argument (input)

total: returned (output)

Invoking (using) the function:

```
nums1 = [1, 2, 3, 4]
tot1 = sumup(nums1)      calls function on nums1
print(tot1)              prints 10
```



Benefits of functions:

- Make repeat calculations easier:
  Don't have to write the same code multiple times

- Make code shorter and logic cleaner and clearer:

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)
```

- Make code more reliable:
  One version to get right rather than several
  Coder mantra: "*Don't repeat yourself*"

Optional arguments with default values:

Defining a function with a default:

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

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

In addition, *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, 0)
tot3 = sumup2(vals, start=0)
tot4 = sumup2(vals, 10)
```

tot1 →	60	default
tot2 →	60	positional
tot3 →	60	keyword
tot4 →	70	positional

```
tot5 = sumup2(vals, start=20)
```

tot5 → 80 keyword

Can use keyword arguments selectively:

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

```
vals = [10,20,30]
tot6 = sumup3(vals)
tot7 = sumup3(vals, power=2)
```

tot6 → 60

tot7 → 1400

Variable scoping:

What values do variables have *inside* and *outside* the function?

Roughly speaking:

- Functions have their own copies of variables
- Changes inside functions don't change variables outside

Example:

Define a function:

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

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 a parameter
	a → 5+2 → 7	Revised in function
	c → 30	Read from outside
	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 clobber outside variables
- Can be written without knowing every possible context

Note: best practice about passing variables to functions:

- Include all outside variables in the argument list
- Avoid using other external variables