python™ from scratch

A primer for scientists working with Next-Generation-Sequencing data

CHAPTER 4

Writing your own functions

Why functions?

Often the same task needs to be performed at different times in a program.

Copying and pasting lines is error-prone and leads to bloated code.

Functions enable you to reuse bits of code without the need of copying it.

What is a function?

We have seen several examples:

- `print`
- `open`
- ...

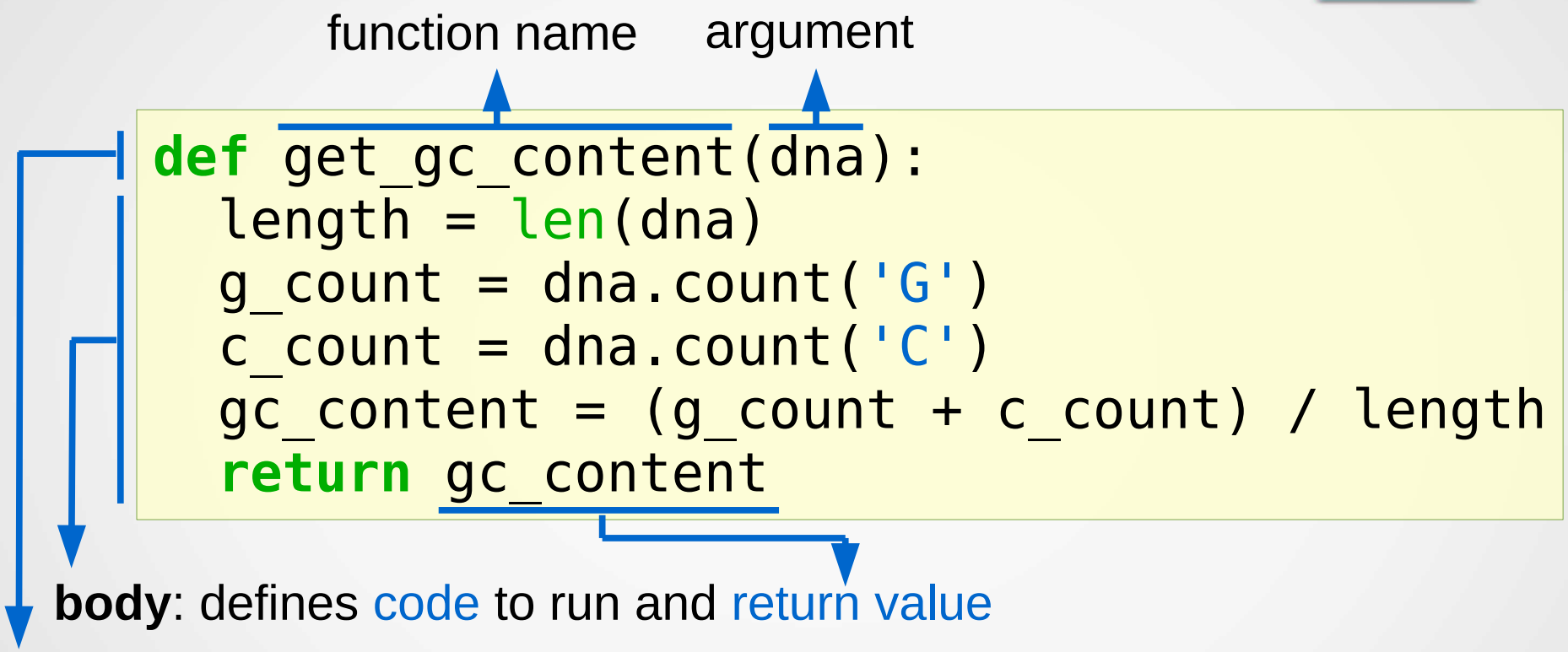
Essentially a function is

- a block of code
- identified and called by a name

Functions can have

- arguments (input values)
- a return value (output)

Defining a function



Note: This function will not work correctly with *Python 2* unless you include the following import at the top.

```
from __future__ import division
```

Calling a function

- functions are called by their defined name:

```
gc_content = get_gc_content("ATAGGCACATT")
```

- only when the function is called, its code is executed
- if a return value is returned by the function it can be assigned to a variable
  - if no **`return`** statement was defined, the function returns the value `None`

Using our function in a program

- a program can comprise several functions and calls to those functions
- programs are usually considered to be more complex and structured than scripts, which often only implement a small subtask

```
def get_gc_content(dna):
  [...]
  return gc_content

my_gc_content = get_gc_content("ATGCGCGATCGATCGAATCG")
print(str(my_gc_content))
print(get_gc_content("ATGCATGCAACTGTAGC"))
print(get_gc_content("aactgtagctagctagcagcgta"))
```

Improving our function

- we can make the output nicer by rounding the return value:

```
def get_gc_content(dna):
  [...]
  return round(gc_content, 2)
```

- Still, the result for the lower case sequence is incorrect. We can make the script **ignore case** by formatting the input parameter:

```
def get_gc_content(dna):
  g_count = dna.upper().count('G')
  c_count = dna.upper().count('C')
  [...]
```

Improving our function

- More flexibility can be achieved by making the **number of significant figures** an input parameter:

```
def get_gc_content(dna, ndigits):
  length = len(dna)
  g_count = dna.upper().count('G')
  c_count = dna.upper().count('C')
  gc_content = (g_count + c_count) / length
  return round(gc_content, ndigits)
```

Keyword arguments

- A slightly different way of **calling** a function (or method):

```
get_gc_content(dna="ATAGGCACATT", ndigits=3)
```

- identical to the call we used before:

```
get_gc_content("ATAGGCACATT", 3)
```

- BUT now we can change the parameter order:

```
get_gc_content(ndigits=3, dna="ATAGGCACATT")
```

- Can make calling of functions **more descriptive**

Parameter defaults

- Some parameters are provided for flexibility but in most cases a reasonable default value can be assumed

```
def get_gc_content(dna, ndigits=2):
  [...]
  return round(gc_content, ndigits)
```

- Now we can **omit the parameter** if we are fine with the default value:

```
gc_content = get_gc_content("ATAGGCACATT")
```

- Especially useful for functions with many input parameters

Encapsulation

- putting bits of code that perform a defined subtask into a function is called encapsulation
- encapsulation enables you to deal with only parts of the code at a time, a method of abstraction
- well-structured code is **much** easier to maintain and to understand
- the whole concept of object-oriented programming is based on encapsulation

Testing functions

- the `assert` function is a means of testing a function's output:

```
assert get_gc_content("ACGT") == 0.5
assert get_gc_content("acgt") == 0.5
assert get_gc_content("ACT", 3) == 0.333
```

- `assert` returns an exception if the comparison returns false, otherwise nothing
- for complex programs with many functions it is **good practice** to write so-called unit tests to make sure functions work as expected at all times
- unit tests are usually stored in separate test scripts

Recap

In this unit you learned:

- **defining** functions
- **calling** functions
- passing **input values** to functions as parameters
  - defining **default values** for parameters
  - passing parameters by name as **keyword arguments**
- returning **output values** from within functions
- **testing** functions
- the concept of **encapsulation**

Exercise 4-1: Amino Acid Content (pt. 1)

- Write a function **`aa_content`** that takes two arguments:
  - a protein sequence
  - an amino acid residue code (single letter)
- The function should return the percentage of the protein that the **amino acid** makes up.
- Use the following assertions to **test** your function:

```
assert aa_content("MSRSLLLRFLLFLLLLPPLP", "M") == 5
assert aa_content("MSRSLLLRFLLFLLLLPPLP", "r") == 10
assert aa_content("MSRSLLLRFLLFLLLLPPLP", "L") == 50
assert aa_content("MSRSLLLRFLLFLLLLPPLP", "Y") == 0
```

Exercise 4-2: Amino Acid Content (pt. 2)

- Modify the function from part one so that it accepts a list of amino acid residues rather than a single one.
- If no list is given, the function should return the percentage of hydrophobic amino acid residues (A, I, L, M, F, W, Y, V).
- Your function should pass the following assertions:

```
assert aa_content("MSRSLLLRFLLFLLLLPPLP", ['M']) == 5
assert aa_content("MSRSLLLRFLLFLLLLPPLP", ['M','L']) == 55
assert aa_content("MSRSLLLRFLLFLLLLPPLP", ['F','S','L']) == 70
assert aa_content("MSRSLLLRFLLFLLLLPPLP") == 65
```