1 Crash Introduction to Python

This chapter kicks off with an example for a simple program in Python. Through this example, we will delve right away into the syntax of this popular programming language. There will be quite a few technical details in this example, which are probably new to anyone with no prior Python experience. The rest of the chapter will then walk the reader briefly yet methodologically through the basic Pythonic syntax. By the end of this chapter, you will be able to write simple programs in Python to solve various problems of a computational nature.

We recommend reading this chapter, and in fact the entire book, alongside with a Python working environment. The code you will see also appears on the book's website, and running it in parallel to reading the material will most certainly improve your understanding, and allow you to explore beyond what we present. Solving the exercises is highly recommended before moving on.

1.1 A Crash Example

Please welcome our first program in Python for this book!

```
1 def GC_content(dna_string):
2     ''' calculate %GC in dna_string (uppercase ACTG) '''
3     GC = 0
4     for nuc in dna_string:     # iterate over dna_string
5         if nuc == "C" or nuc == "G":
6             GC += 1            # equal to GC = GC+1
7     return 100*GC/len(dna_string)
```

Code explained

This code defines a function called `GC_content`. This function receives a single parameter, whose name is `dna_string`. The parameter `dna_string` is expected to be a string that consists of the characters A, T, C, and G only. However, this function does not verify this. Furthermore, in Python variable types are not specified, and we will get back to this point soon. Note the colon at the end of the "title" row.

The body of the function appears next. In Python, indentation defines scopes: we know that lines 2–7 are the definition of the function's body because all these lines are indented to the right. Similarly, code inside loops or conditional statements is also indented, as we will discuss very soon.

Line 2 is a documentation string (also termed "docstring"): this is a string that appears between triple quotes and merely describes the function. It has no effect in terms of the function's operation. Line 3 defines a variable called `GC`, and initializes it to 0. In line 4, there is a loop that iterates over the elements of the input variable `dna_string`. This line should be understood as "use a variable named `nuc` and assign it to the characters of

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

`dna_string` one by one." We chose the name `nuc` because each of these characters represents a single nucleotide. The `#` character denotes the beginning of a programmer's comment (comments have no effect on the program's functionality). In line 5, we use an if-statement (termed conditional statement) to check whether `nuc` is equal to `"C"` (Cytosine) or to `"G"` (Guanine). The equality operator in Python is the equals sign twice, `==`, and the two sub-conditions are combined with a logical `or` relation (termed logical operator). If the condition is true, we increase the variable `GC` by `1`, using the += operator: `GC += 1` is merely a shortcut for `GC = GC+1`. By the end of the loop in lines 4–6, the variable `GC` stores the number of characters in `dna_string` that are `"C"` or `"G"`. Finally, in the last line, the function returns the percentage of `"C"` and `"G"` in the input string to the calling environment.

Once we have written this function, we can use it by calling it:

```
>>> GC_content("AAA")
0.0
>>> GC_content("CCG")
100.0
>>> GC_content("TTAGACCAGTAGAAGTAC")
38.888888888888886
```

The programing notions presented in this example are: function definition and function call, variable, string, loop, if statement, operator, and return value. Let's go over these in more detail now.

1.2 Variables, Basic Types, and Operators

Computer programs use **variables** to store data. Data values in Python and the variables storing these data can have different types. Table 1.1 summarizes several common types in Python.

Numbers in Python come in two types: **integers** of type `int`, for whole numbers, and numbers with a **decimal point** of type `float` (such numbers are represented in the computer's memory in the floating point method, which we will not explain here, thus the type name).

Type `str` is used to store **strings**. A string is a sequence of characters, appearing between quotation marks. Note that strings in Python may be written between double or single quotation marks (`"..."` or `'...'`). Triple-double (`"""..."""`) or triple-single (`'''...'''`) quotation marks are also used in Python in two cases: as documentation strings at the beginning of

Table 1.1. ***Variable types in Python***

Type/class	Meaning	Examples for values of this type
int	integer	`0, 1, -1, 2, -2, 3, -3...`
float	rational number	`3.14  -0.001  8.0`
str	string, sequence of characters	`"Amir"  "!!!"  "45"  "ATTCG"`
list	sequence of elements	`[1, 2, 3]  [55, "hello", 55, -7]`
bool	Boolean	`True / False`

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

functions (as we saw in the earlier example), and for strings that span over more than one line (we will see this later). In any case, a string is started and terminated by matching terminators.

Type `list` is used to store, well, **lists**, which are sequences of elements of any type. Elements in a list are separated by commas, and enclosed within square brackets [ ].

In Python, types are also called **classes**. In fact, every object in Python belongs to a class, which defines its functionality. Python has a built-in function named `type` that returns the class of a value or variable:

```
>>> x = 5
>>> type(x)
<class 'int'>
>>> y = 0.001
>>> type(y)
<class 'float'>
>>> type(True)
<class 'bool'>
>>> s1 = "drosophila"
>>> type(s1)
<class 'str'>
>>> type("45")
<class 'str'>
>>> lst = [1, 2, 3, "hello", 0.001]
>>> type(lst)
<class 'list'>
```

In Python, unlike some other programming languages, such as C and Java, there is no explicit declaration of the type of a variable. This is because variables in Python have dynamic types, which may change whenever a new value is assigned to the variable:

```
>>> x = 5
>>> type(x)
<class 'int'>
>>> x = "drosophila"
>>> type(x)
<class 'str'>
```

Each type in Python supports specific **operators** that can be applied to it. For example, numeric types (`int` or `float`) support the arithmetical operators in Table 1.2.

As you can see, Python[1] has two division operators: one is for "true" division and the other rounds the result down and yields an integer (this is why it is also called integer division):

```
>>> 8/5
1.6
>>> 8//5
1
```

[1] In fact Python version 3, but not the older version 2.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Table 1.2. ***Numeric operators***

Operator	Symbol	Example
Addition	`+`	`x+y`
Subtraction	`-`	`x-y`
Negation	`-`	`-x`
Multiplication	`*`	`x*y`
Division	`/`	`x/y`
Floor division	`//`	`x//y` (this is `x/y` rounded down)
Exponentiation	`**`	`x**y` (this is x^y)
Modulo	`%`	`x%y` (the remainder of `x` divided by `y`)

Table 1.3. ***Comparison operators***

Operator	Symbol	Example
Equal	`==`	`x == y`
Unequal	`!=`	`x != y`
Less than	`<`	`x < y`
Less or equal	`<=`	`x <= y`
Greater than	`>`	`x > y`
Greater or equal	`>=`	`x >= y`

Python defines an **order of precedence** for operators. For example, the expression 3*4+5 will result in 17, since multiplication precedes addition, just as you have learned in elementary school (exponentiation precedes multiplication and division, which in turn precede addition and subtraction). Parentheses are used to impose a different order of evaluation:

```
>>> 3*4+5
17
>>> 3*(4+5)
27
```

We do not find much point in memorizing the precedence order defined by Python, thus we do not present it here. Instead, we recommend using parentheses when in doubt. We will see some operators for strings and lists later in this chapter.

1.3 Comparison, Logical, and Assignment Operators

Python supports several **comparison operators**, as summarized in Table 1.3. The result of a comparison is either `True` or `False`, which are termed **Boolean** values (after the English mathematician George Boole). Expressions of Python's type `bool` are termed Boolean expression, or logical expressions.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
>>> x = 3
>>> y = 4
>>> x == y
False
>>> x == y-1
True
>>> x != 3
False
>>> x < y
True
>>> s1 = "drosophila"      <= this is an assignment
>>> s1 == "Drosophila"     <= this is a comparison
False
>>> [1,2,3] == [2-1,2,2+1]
True
```

Exercise 1

Not every two types are orderable, that is, can be compared by the <, <=, >, >= operators. Try for example:

```
>>> 5 < "5"
```

And observe the resulting error message. However, strings in Python are orderable. You may wonder how order is defined for strings. One could imagine that the order is defined merely by length, but this can be ruled out by a simple check:

```
>>> "cat" < "astronaut"
False
```

Try to figure out how the order relation is defined for strings, by inspecting the following examples, and running additional ones that will support your hypothesis.

```
>>> "cat" < "dog"
True
>>> "cat" < "bee"
False
>>> "cat" < "catalogue"
True
```

Boolean expressions can be combined to form more complex expressions, using the **logical operators** "and," "or," and "not". Here is a short reminder (A, B in Table 1.4 are Boolean expressions):

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Table 1.4. ***Boolean operators***

Operator	Usage	When this expression is True
and	A and B	both A and B are True
or	A or B	at least one of A and B is True
not	not A	A is False

```
>>> x = 3
>>> y = 4
>>> x == 3 and x == y
False
>>> x == y or x == y-1
True
>>> not (x == y or x == y-1)
False
```

The negation operator may be convenient when negating a non-trivial condition. In the last example, we could easily replace

```
not (x == y or x == y-1)
```

by

```
x != y and x != y-1,
```

using simple logical rules. However, when the condition is more complex, negation with the `not` operator may be more convenient.

Exercise 2

What would happen if we removed the () in the last example?

```
>>> not x == y or x == y-1
???
```

Python defines the order of precedence for logical operators too!

In many of the previous examples, we used an **assignment operator**, denoted =. There are several additional operators that involve assignment, which actually combine assignment with modification of a variable. In the `GC_content` example, we already saw this: `GC += 1` is a shortening for `GC = GC+1`. Other assignments of this kind are `-=`, `*=`, `/=`, `//=`, `**=` and even `%=`. Here are several examples (note especially the last one on strings):

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
>>> x = 3
>>> y = 4
>>> x += 1
>>> x
4
>>> x *= 2
>>> x
8
>>> x -= 2*y     <= stands for x = x-2*y
>>> x
0
>>> s = "AAA"
>>> s += "GGG"
>>> s
"AAAGGG"
```

1.4 Type Conversions

Sometimes we need to perform computations that mix more than one type. For example:

```
>>> x = 5 + 3.2
```

Exercise 3

(a) What do you think is the result of the above computation? What is its type?
(b) Suppose you had to vote for the resulting type of the following computation:

```
>>> x = 0.8 + 1.2
>>> type(x)
???
```

The result is obviously the number 2, but is it 2 of type `int` or 2.0 of type `float`? What would you vote for? Why? Check if this is the case in Python as well. In some cases, there is no right or wrong answer, but rather a decision of the designers of the programming language, according to what they find most appropriate, taking into account various considerations, such as consistency, ease of use, or error prevention.

When we mix `float` and `int`, Python first converts the `int` value into `float` (e.g., 5 is converted to its `float` parallel 5.0), and then the expression is computed. This makes sense because often we need to make computations that involve integers and real numbers. Such conversions are called **automatic type casting**.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Exercise 4

Do you think Python would allow adding up a number and a string, involving an automatic casting from `int` to `str` or vice versa? Try for example:

```
>>> x = "5" + 3.2
???
```

Some type mixing is illegal and yields an error. This is the case when the language designers think that a certain operation is more likely to be a bug rather than the intension of the programmer. An error will prevent such commands from going unnoticed.

In addition to the automatic casting done by Python when mixing values of appropriate different types, Python also allows **explicit casting**. Such type casting is forced by the programmer, and is an important feature as we will see in many examples in this book. However, it should be used with caution, as some conversions do not make sense. The explicit conversion in the next examples is done through the functions `int()`, `float()`, `str()`, and `list()`.

```
>>> x = 5.2
>>> int(x)
5

>>> y = 8
>>> float(y)
8.0

>>> str(56)
'56'

>>> int("34")
34

>>> int("hello")
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    int("hello")
ValueError: invalid literal for int() with base 10: 'hello'
```

As you can see, converting from string to integer may yield an error – not everything can be converted to a number. When we ask Python to execute an illegal command, it informs us that there is some problem with the command. Sometimes, the error message is very informative, and sometimes it may look a bit cryptic and more experience will be required to understand it. In this example, the error message may look uninformative, but it merely says that Python tried to interpret the string "hello" as a decimal number, and failed.

Exercise 5

Does the direct conversion from float to int round a number to the closest integer or always round the number down? Is the result of the following 5 or 6?

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
>>> x = 5.8
>>> int(x)
???
```

Conversion from a string to a list creates a list that contains each character in the string as a separate element:

```
>>> list("TATAAA")
['T', 'A', 'T', 'A', 'A', 'A']
```

The other direction, however, is a bit trickier. Let's try:

```
>>> str(['T', 'A', 'T', 'A', 'A', 'A'])
"['T', 'A', 'T', 'A', 'A', 'A']"    <= not what we'd want!
```

What we have here is not a string with the characters contained in the list, as we might expect. The commas between the elements, the quotations marks around each character, as well as the [ ] surrounding the list all became characters in the new string! Although this is a bit early in terms of Python features we have learned, we show below how this conversion should be done appropriately. We note that in Section 1.6.3 we explain the syntax shown here in more detail.

```
>>> "".join(['T', 'A', 'T', 'A', 'A', 'A'])
"TATAAA"
```

1.5 Strings and Lists as Sequences

Like the numerical operators we saw earlier, strings and lists each have their own operators. Since both types are ordered sequences of elements, they have several common operators. Please see two useful operators of strings and lists in Table 1.5.

Here is another example:

```
>>> positive_aa = ["Lysine", "Arginine"]
>>> negative_aa = ["Aspartic acid", "Glutamic acid"]
>>> charged = positive_aa + negative_aa
>>> charged
["Lysine", "Arginine", "Aspartic acid", "Glutamic acid"]
```

Table 1.5. ***Sequence operations***

Operator	Symbol	Example for strings	Example for lists
Concatenation	+	`>>> "droso" + "phila"` `"drosophila"`	`>>> [1,2,3] + [4,5]` `[1,2,3,4,5]`
Duplication	*	`>>> "UAG" * 8` `"UAGUAGUAGUAGUAGUAGUAGUAG"`	`>>> ["a",4] * 3` `["a",4,"a",4,"a",4]`

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Note that assignment operators =+ and *= also work for strings and lists:

```
>>> dna = ""              <= an empty string
>>> dna += "UAG"
>>> dna
'UAG'
>>> dna += "CC"
>>> dna
'UAGCC'

>>> charged = []          <= an empty list
>>> charged += ["Lysine", "Arginine"]
>>> charged += ["Aspartic acid", "Glutamic acid"]
>>> charged
["Lysine", "Arginine", "Aspartic acid", "Glutamic acid"]
```

When working with strings and lists, **indexing** is used to access characters or elements in specific positions (indices). We use `[i]` to probe the *i*th element. The first position is accessed with the index 0, the second one with the index 1, etc. So, legal indices are integers between 0 and the length of the string/list minus 1.

```
>>> organism = "drosophila"
>>> organism[0]
'd'
>>> organism[1]
'r'
>>> organism[5]
'p'
>>> organism[9]
'a'
>>> organism[10]
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    organism[10]
IndexError: string index out of range
```

In the last error message, Pythons tells us we tried to access a string with an illegal index, since the last legal index of a string of length 10 is 9.

Exercise 6

What do you think will the result of these commands be?

```
>>> positive_aa = ["Lysine", "Arginine"]
>>> positive_aa[0][0]
???
>>> positive_aa[1][3]
???
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

The last exercise shows an example for indexing of "higher dimensions," that is, we access the *i*th element of a sequence (string or list), and if this element is a sequence itself, we can index into it as well. This will be very useful later in the book, when we work with lists of strings, or lists containing inner lists. The latter will be used to store tables or matrices. For example, here is a 3 × 3 matrix, represented as a list of lists:

```
>>> mat = [ [1,2,3], [4,5,6], [7,8,9] ]
>>> mat[1][2]
6
```

Checking the **length** of a sequence in Python is easy, using the built-in function `len()`:

```
>>> organism = "drosophila"
>>> len(organism)
10
>>> organism[len(organism)-1]
'a'
>>> organism[len(organism)]
Traceback (most recent call last):
  File "<pyshell#10>", line 1, in <module>
    organism[len(organism)]
IndexError: string index out of range
>>> positive_aa = ["Lysine", "Arginine"]
>>> len(positive_aa * 2)
4
```

Exercise 7

What is the output of the following length tests?

```
>>> mat = [ [1,2] , [3,4] , [5,6] ]
>>> len(mat)
???
>>> len(mat[0])
???
```

Python allows **slicing** a sub-sequence from a given sequence, which is a very convenient feature. To slice a string or list, the start and the end indices are provided between `[ ]` separated by a colon. The resulting slice contains all the elements within the range, not including the end position:

```
>>> organism = "drosophila"
>>> organism[0:4]
'dros'
>>> organism[0:1]
'd'
>>> organism[0:len(organism)]
'drosophila'
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

We can also provide a third integer, to denote the "leap" within the range of indices (otherwise the default leap is 1):

```
>>> organism[0:len(organism):2]
'doohl'
>>> organism[9:3:-1] <= leap of one position backwards
'alihpo'
```

Exercise 8

When we omit the start and/or end indices, Python uses default values for them. What are these default values? Check the output, for example, in the following cases:

```
>>> "drosophila"[3::2]
???
>>> "drosophila"[::-1]
???
```

(The last command is worth remembering, that's an easy way to **reverse** sequences in Python.)

The last useful functionality of strings and lists we present in this section is **membership checking**, using Python's operator `in`.

```
>>> organism = "drosophila"
>>> "p" in organism
True
>>> "phila" in organism
True
>>> "c" in organism
False
>>> 1 in [1,2,3]
True
>>> "1" in [1,2,3]
False
```

Finally, we remark that strings and lists have many differences too. One such fundamental difference is that lists are **mutable**, while strings are **immutable**. This simply means that we can change elements in a list, but we cannot change characters in a string.

```
>>> lst = ["T", "A", "A", "T", "A"]
>>> lst[1] = "G"
>>> lst
["T", "G", "A", "T", "A"]

>>> dna = "TAATA"
>>> dna[1] = "G"
Traceback (most recent call last):
  File "<pyshell#19>", line 1, in <module>
    dna[1] = "G"
TypeError: 'str' object does not support item assignment
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

One way to bypass this limitation is by converting a string to a list, changing (mutating) its elements as needed and then converting them back to a string.

```
>>> dna = "TAATA"
>>> dna_list = list(dna)
>>> dna_list[1] = "G"
>>> dna = "".join(dna_list)
>>> dna
'TGATA'
```

This may look a bit cumbersome, but it does the job.

Exercise 9

Will these commands yield an error or not?

```
>>> Glycine_codons = ["GGU", "GGC", "GGA", "GGG"]
>>> Glycine_codons[3] = "ggg"
>>> Glycine_codons[3][2] = "G"
```

In the next section on functions, we will see more differences between strings and lists: each type supports different functions.

1.6 Functions

1.6.1 Python Built-In Functions

In the previous sections, we saw several commands in Python that involved **functions**. For example, we have encountered `type(...)`, `int(...)`, `str(...)`, `len(...)`, etc. These were all examples for **built-in functions** in Python. This means that these functions are part of the language, and are ready to use whenever we write a program in Python.

Python has many more built-in functions, for example `min`, `max`, and `sum`. Given a list of numbers, these functions return the minimum, maximum, and sum of the list:

```
>>> min([5,2,-4,7,8])
-4
>>> max([5,2,-4,7,8])
8
>>> sum([5,2,-4,7,8])
18
```

You may want to check what happens when these functions are given a list that contains not only numbers, or an input that is not a list at all.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Exercise 10

Try calling the above functions on a list containing only strings as elements. Is this a legal input for each of them?

```
>>> min(["cat", "dog", "bee"])
???
>>> sum(["cat", "dog", "bee"])
???
```

Another useful Python built-in function is `sorted`:

```
>>> sorted([5,2,-4,7,8])
[-4,2,5,7,8]
```

Exercise 11

Does Python's function `sorted` change the list it received as input or does it create a sorted copy of the list? Check this:

```
>>> L = [5,2,-4,7,8]
>>> sorted(L)
[-4,2,5,7,8]
>>> L
???
```

The command `help(__builtins__)` will present all the built-in functions (and other objects, such as classes) that exist in Python. However, in this book we will be using a very small portion of those, and present them when needed.

1.6.2 User-Defined Functions

Writing one's own functions in Python is really necessary for being able to design new programs. This chapter kicked-off with an example of a function that computes the GC content of a DNA sequence given as input. Here is another example for a simple function that takes a number representing the temperature in Fahrenheit and converts it to Celsius (the formula is $°C = (°F - 32)/1.8$).

```
1 def fahr2cel(fahr_temp):
2     ''' input is temperature in Fahrenheit, output in Celsius '''
3     cel_temp = (fahr_temp - 32) / 1.8
4     return cel_temp
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
>>> fahr2cel(86)
30.0
>>> fahr2cel(104.6)
40.33333333333333
```

The general structure for a function's definition is this:

```
def func_name(parameters):
<-tab->  function body
          .
          .
          .
         return value
```

The body of the function includes statements, which are indented by a tab (or a fixed number of spaces). We will see many more user-defined functions in the rest of this chapter, and in the rest of this book.

1.6.3 Class Methods

In addition to the built-in and user-defined function, additional sources for functions are **class methods**. These are functions that belong to specific classes in Python. For example, there are methods that belong to the class of strings. Here are some examples:

```
>>> dna = "ATTGCGGGCTTG"
>>> dna.count("C")
2
>>> dna.count("G")
5
>>> dna.lower()
'attgcgggcttg'
>>> dna.replace("G", "g")
'ATTgCgggCTTg'
>>> dna.find("G")    <= the first index where 'G' is found in dna
3
>>> dna.find("g")    <= if not found
-1                   <= then -1 returned
>>> dna
'ATTGCGGGCTTG'       <= remained unchanged
```

Code explained

The variable `dna`, which belongs to the `str` class, can invoke any method from class `str`. This it does using "`.`" (dot) between the object name and the required method:

```
object.method(parameters)
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

The method `count` receives a string parameter and returns the number of times the parameter appears in the activating string. The method `lower` receives no parameters, and returns a copy of the activating string converted to lower case (without changing it). The method `replace` takes two parameters and replaces every occurrence (in the activating string) of the first one with the second one. The method `find` returns the first occurrence of the input character in the activating string. These are only a few of the methods in class `str`.

Exercise 12

The method `count` can get as input more than a single character. For example:

```
>>> "ATTGCGGGCTTG".count("GC")
2
```

What do you think will be the output in the next example? What can you conclude about how the method `count` treats overlaps in the searched pattern occurrences?

```
>>> "TATAT".count("TAT")
???
```

Similarly, the list class also has its own methods. Here are some useful ones:

```
>>> L = [3,1,1,2,1,4]
>>> L.count(1)
3
>>> L.append(5)          <=  add element at the end of the list
>>> L
[3, 1, 1, 2, 1, 4, 5]
>>> L.insert(3,100)      <= insert 100 at position 3
>>> L
[3, 1, 1, 100, 2, 1, 4]
>>> L.pop(3)             <= remove element in position 3
100
>>> L
[3, 1, 1, 2, 1, 4]
>>> L.remove(2)          <= remove the value 2 from the list
>>> L
[3, 1, 1, 1, 4]
>>> L.reverse()          <=  reverse the order of the list
>>> L
[4, 1, 1, 1, 3]
>>> L.sort()             <=  sort the list
>>> L
[1, 1, 1, 3, 4]
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Code explained

Class `list` also has a method called `count` that works similarly to `str`'s `count`. The only difference is that the parameters are not limited to a string type. `append` adds an element at the end of the list. The method `insert` takes two parameters: the position at which to insert a new element, and the new element to insert. The method `pop` is used to delete an element by its index in the list, while `remove` deletes an element with a specified value; `reverse` and `sort` both change the order of the elements in the activating list in the way reflected by their names. Note that Python's built-in function `sorted` is different from `list`'s `sort`, in that the former receives the list *as a parameter* and creates a sorted copy of it.

Exercise 13

The method `remove` in the class `list` removes an element from the list whose value is given as a parameter. But what happens when there is more than one such element?

```
>>> L = [1,0,0,1,2,1,4]
>>> L.remove(1)
>>> L
???
```

Knowing which methods exist in each class will definitely pay off, as it saves a lot of time and effort writing many lines of code from scratch. We refer the curious readers to Python's documentation, to your favorite search engine, or to the commands `help(str)` and `help(list)` in order to explore the various class methods available. However, whenever we use a method for the first time in this book, we will explain it.

Functions can be used inside the body of other functions. For example, look at the following user-defined function `GC_content2`, which has the same functionality as the old `GC_content` from the beginning of this chapter. This function uses the method count of class string.

```
1 def GC_content2(dna_string):
2       ''' calculate %GC in dna_string (uppercase ACTG) '''
3       C = dna_string.count("C")
4       G = dna_string.count("G")
5       return 100 *(C+G)/len(dna_string)
```

Exercise 14

What would happen if we called `GC_content2` with an input parameter that contains the DNA sequence in lowercase? Try this for example:

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
>>> dna = 'ttagaccagtagaagtac'
>>> GC_content2(dna)
???
```

Suggest a way to overcome this so that the function will be case insensitive.

1.6.4 Return Value(s)

A function's execution is terminated when a `return` command is encountered. The value that appears after the `return` word is propagated back to the calling environment, and can be assigned to a variable for further use:

```
>>> result = GC_content("TTAGACCAGTAGAAGTAC")
>>> print(result)
38.888888888888886
```

Sometimes we write functions but do not need them to return anything. In such a case, we can write `return None` (or simply `return`) at the end of the function, or completely omit the return command. In any of these cases, the function will return the special value `None`, which is the Pythonic way to say "nothing" (which is, as you know, different from not saying anything!).

```
1 def modify(list, i, value):
2     list[i] = value
3     return None
```

```
>>> L = [1,2,3]
>>> res = modify(L, 0, 99)
>>> print(L)
[99,2,3]
>>> print(res)
None
```

Exercise 15

Look at the following function `F` and its usage:

```
def F(x):
    x += 1

>>> res = F(7)
>>> res
???
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

What is the result? Can you explain this? Will the answer change if F is changed to the following?

```
def F(x):
    x += 1
    return x
```

1.6.5 Default Parameters

Parameters of functions can have **default values**. For example:

```
1 def F(a=10, b=20):
2     return a*b
```

The default value for the first parameter is 10, and for the second the value is 20. This means that when we call the function, we may provide less than two parameters, in which case the default values will be assigned to the missing ones:

```
>>> F()
200
>>> F(a=7)
140
>>> F(b=13)
130
>>> F(a=4, b=6)
24
>>> F(4,6)
24
```

Note that we can still call the function normally as we used to, as in the last command above.

1.7 Conditional Statements

The function `GC_content` from the beginning of the chapter used a **conditional statement**, starting with '`if`'. Such a statement splits the program's flow into two or more possible paths. Here is another example: suppose we want to know the physical state of some chemical substance. We get as input the melting `(Tm)` and vaporization `(Tv)` temperatures of the substance, and the room temperature, `T`:

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
1 def state(Tm, Tv, T):
2     ''' return the state of material with Tm and Tv at temperature T '''
3     if T < Tm:
4         return "solid"
5     else:
6         if T < Tv:
7            return "liquid"
8         else:
9            return "gas"
```

Code explained

The function checks the value of the condition `T < Tm`. In case it is `True`, the function terminates and returns the string `"solid"`. Otherwise, it moves on and checks whether `T < Tv`, in which case `"liquid"` is returned. If both conditions were `False`, `"gas"` is returned.

Another way to write an equivalent conditional statement would be:

```
    if T < Tm:
        return "solid"
    elif T < Tv:
        return "liquid"
    else:
        return "gas"
```

note that `elif` is a shortcut for "else if".

The general structure for conditional statements is this:

```
if condition:
 <-tab-> statement
          ...
elif condition:       # 0 or more
 <-tab-> statement
          ...
else:                 # optional
 <-tab-> statement
          ...
```

Often the conditions in a program are composed of several "smaller" conditions, combined by logical operators. In the `GC_content` example, we checked whether the current nucleotide is either cytosine or guanine, and used the logical operator `or`.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

1.8 Iteration and Loops

1.8.1 "for" Loops

Recall the **loop** in the first example of this chapter (`GC_content`). Such loops are usually used in Python to iterate over a collection of elements, for example strings or lists. Here is an additional example:

```
1 def basic_aa_perc(prot):
2     ''' return percentage of basic amino acids in protein '''
3     basic = 0
4     basic_aa_list = ["K", "H", "R"]
5     for aa in prot:
6         if aa in basic_aa_list:
7             basic += 1
8     return 100*basic/len(prot)
```

Code explained

The function computes the percentage of the amino acids lysine, histidine, and arginine in the protein input sequence. In line 5, the variable `aa` (short for amino acid) is implicitly defined, and then assigned characters from `prot` one by one in each iteration. For any such `aa`, if it is either `"K"`, `"H"` or `"R"` the counter is increased.

The general structure of "for" loops is the following (The collections we have seen so far were strings and lists. We will see later that "for" loops are also applicable to other collections.):

```
for var in collection:
<-tab-> statement
            ...
statement       # out of loop
```

Exercise 16

The function, `basic_aa_perc` iterates over the input protein `prot`. We could instead go over the list of basic amino acids. Complete the missing line in the following functions, using class `str`'s method count:

```
def basic_aa_perc(prot):
    basic = 0
    basic_aa_list = ["K", "H", "R"]
    for basic_aa in basic_aa_list:
            complete this line
    return 100*basic/len(prot)
```

Do you expect one of these versions to run faster? We will see later on how to answer such questions, and even measure and compare actual running times.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

1.8.2 "while" Loops

There is another type of loops in Python – "while" loops. As a first example, let us re-write the `GC_content` function:

```
1 def GC_content3(dna_string):
2     ''' calculate %GC in dna string (uppercase ACTG) '''
3     GC = 0
4     i = 0
5     while i < len(dna_string):
6         if dna_string[i] == "C" or dna_string[i] == "G":
7             GC += 1
8         i += 1
9     return 100*GC/len(dna_string)
```

Code explained

The variable `i` is used as the running index. It is initialized to 0, used to access specific elements in the input `dna_string` in line 6, and increased by 1 each iteration in line 8. The loop repeats as long as `i` is smaller than the length of `dna_string`, as specified in line 5. Note that if we forgot to increase `i` in line 8, we would get an infinite loop that would run (theoretically) forever!

The general structure of "while" loops is the following:

```
while   condition:
<-tab-> statement
           ...
statement          # out of loop
```

Note that in "while" loops, the programmer has more responsibility for the correct behavior of the loop. Incorrectly initializing the running index, forgetting to increase it every time, or a wrong loop termination condition will result in an incorrect computation.

"For" loops are more limited than "while" loops: with "for" loops, we can iterate over a given collection of elements, but sometimes we need our program to keep on doing something as long as some condition holds, in which case "while" loops are the only option. For example, suppose we seek all positions in a given protein where a specific restriction site of an enzyme occurs. Here is a function to do the job:

```
1 def cut_positions(seq, pattern):
2     ''' find all positions of pattern within seq '''
3     cut_sites = []
4     site = seq.find(pattern) # get index of pattern in seq, -1 if none
5     while site != -1:
6         cut_sites.append(site)
7         site = seq.find(pattern, site+1) # get next match
8     return all_sites
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Code explained

The method `find` of class `str` can receive one or two parameters. In the former case, as in line 4, it returns the first index in which the input pattern string occurs within `seq`. When given two parameters, as in line 7, the second one specifies at which index of `seq` to start the search. So `seq.find(pattern, site+1)` finds the first occurrence of pattern in `seq` ignoring the first `site` positions, where `site` is the position of the previous match. The loop's condition in line 5 is `site != -1`, because the method returns –1 when no match was found.

1.8.3 Looping with `range`

Python provides an easy way to define a **range of integers** with a constant leap (this may remind you of how slicing works). The command `range(a,b)` generates all the integers between `a` and `b`, not including `b`. Adding a third parameter `range(a,b,c)` provides those integers with a "leap" of `c` positions.

```
>>> for i in range(3,7):
        print(i*2)

6
8
10
12

>>> dna = "ATTG"
>>> for i in range(0,len(dna)):
        print(dna[i]*2)

AA
TT
TT
GG

>>> list(range(3,10,2))
[3,5,7,9]
```

Furthermore, if we provide `range` with only a single parameter, the default starting index is 0:

```
>>> list(range(7))
[0,1,2,3,4,5,6]
```

Note that we can always use a simple "while" loop instead of using `range`, but often the latter is more convenient.

Here is another example, using `range`, of a function that finds the first stop codon in a given DNA sequence, taking into consideration the frame shift:

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
1 def find_stop(dna):
2     stops = ["TAG", "TAA", "TGA"]
3     n = len(dna)
4     for i in range(0, n, 3):
5         if dna[i:i+3] in stops:
6             return i
```

Code explained

The loop in line 4 jumps three positions in each iteration, starting from 0, so the values of `i` are 0, 3, 6, 9... (the last value depends on the length of the string). For every such value of `i`, the function uses slicing to extract the three nucleotides at positions `i`, `i` + 1, `i` + 2 and to check if this codon is a stop codon.

Exercise 17

What will `find_stop` return if no stop codon is found in the DNA input?

1.8.4 Iterating over Sequences – Summary

Let us summarize the three ways we have in Python to iterate over a given sequence, such as a string or a list. The first way we saw accesses the elements directly using a `for` loop:

```
for element in seq:
        do something with element
```

This is the most convenient way, but it is appropriate only when we wish to access merely the elements in the sequence, without any reference to their positions (indices) within the sequence. The second way uses a `while` loop, and iterates over the indices, through which the elements are accessed:

```
i = 0
while i < len(seq):
        do something with seq[i]
        i = i+1
```

The third way uses `range` to iterate over the indices as well:

```
for i in range(len(seq)):
        do something with seq[i]
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Exercise 18

Write a function named `reverse_comp(dna)` that receives a DNA sequence, and returns its reverse complement,

(a) By using Python's `range` to iterate over `dna` in a reverse order.
(b) By reversing the string itself using slicing

1.9 Interactive User Input

It is common for a program to depend on some input from the user in an interactive manner, namely input that is provided by the user at runtime. As an example, recall the function `state` from an earlier section. We could ask the user to provide the room temperature at a given moment during the program's execution. The way to do this in Python is to use the built-in function `input`:

```
>>> room_temp = input("Please provide the current room temperature: ")
```

This command will make your program stand-by, with the message string displayed. The program's execution is paused until the user provides some input and presses <Enter>. Once this happens, the function `input` captures this value and returns it.

```
>>> room_temp = input("Please provide the current temperature: ")
Please provide the current temperature: 29
>>> print(room_temp)
'29'
```

29 is typed by the user at runtime

Note that the function input always returns a string. (Clearly, it has no way of knowing what type of data the user intended to provide – `int`, `str`, etc. It treats any characters typed as a string.) But we have already seen how to convert strings to integers. Now we can get the room temperature from the user at runtime, and use the function `state` as before:

```
1  def get_room_temp():
2      room_temp = input("Please provide the current temperature: ")
3      return int(room_temp)
4
5  def state(Tm, Tv, T):
6      '''return the state of material with Tm and Tv at temperature T '''
7      if T < Tm:
8          return "solid"
9      elif T < Tv:
10         return "liquid"
11     else:
12         return "gas"
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
>>> T = get_room_temp()
Please provide the current temperature: 29
>>> water_state = state(0,100,T)
>>> print("Water at", T, "degrees is", water_state)
Water at 29 degrees is liquid
```

1.10 Containers

The term **container** in Python refers to any element that contains "inner" elements, allowing them to be accessed and iterated over. We have seen two types of containers so far: string and list. Python has additional container types, most notably *tuples*, *sets*, and *dictionaries*, which we describe in this section.

We can roughly classify containers according to two characteristics: **mutability** and **order**. We have encountered the notion of mutability earlier: a mutable object allows changing its "inner" elements whilst keeping its memory address unchanged – lists are mutable, while strings are not. Both strings and lists are ordered containers: their inner elements are ordered by positions (starting at 0). In other words, these are **sequences** of elements. The new container types presented in this section are tuples, sets, and dictionaries (Table 1.6).

Sets and **dictionaries** are unordered collections of elements. "Unordered" means that elements have no indices, thus there is no "first," "second," or "last" elements in them. This is in contrast to lists and strings, in which elements are positioned as a contiguous sequence, and can be accessed using indices 0, 1, 2, etc. Sets in Python resemble a lot of sets in mathematics: they represent an unordered collection of unique elements (unique means that an element cannot appear more than once in a set). In particular, they support operations such as adding and removing elements, union and intersection. Elements of sets are enclosed between {}:

```
>>> s1 = {3,4,5}
>>> 3 in s1
True
>>> s2 = set() <= empty set
>>> s2.add(3)
>>> s2.add(2)
>>> s2
{2, 3}
>>> s2.add(2)
>>> s2
{2, 3}          <= no repetitions in sets
>>> s2.discard(2)
>>> s2
{3}
>>> s2.discard(5)
>>> s2
{3}
>>> s2.union({3,4,5})
{3, 4, 5}
>>> s2.intersection({3,7})
{3}
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Table 1.6. ***Container types***

	Ordered (sequences)	Unordered
Mutable	list [1,2,"hello"]	set {1,2,"hello"}
		dict {1:"a",2:"b",3:"c"}
Immutable	str "TGAAC"	
	tuple (1,2,"hello")	

A very convenient way to remove repetitions from a list is to convert it to a set:

```
>>> list(set([7,3,3,8,7,3]))
[8, 3, 7]
```

Note that since sets are unordered, the original order of the elements in the list is generally not preserved in this process.

Dictionaries in Python are "close relatives" of sets. They are also unordered mutable containers. However, elements in a dictionary have two components: a key and an attached value. In other words, Python's dictionaries are used to map a set of keys to values. To get the mapping of a key, we specify the key inside `[]`, as in the following examples:

```
>>> d1 = dict()  <= empty dictionary
>>> d1["Lys"] = "K"  <= insert a new mapping, "Lys" mapped to "K"
>>> d1
{'Lys': 'K'}
>>> d2 = {"Arg":"R", "Tyr": "Y"}<= initialize a dictionary with 2 keys
>>> d2["Arg"] <= get the value that the key "Arg" is mapped to
'R'
>>> "Arg" in d2
True
>>> d2.keys()
dict_keys(['Tyr', 'Arg'])
>>> list(d2.keys()) <= get all the keys
['Tyr', 'Arg']
>>> list(d2.values()) <= get all the values
['Y', 'R']
>>> d2.pop("Arg")
'R'
>>> d2
{'Tyr': 'Y'}
```

The following dictionary maps DNA codon to the amino acids, according to the universal genetic code:

```
1  universal = {
2      'ttt': 'F', 'tct': 'S', 'tat': 'Y', 'tgt': 'C',
3      'ttc': 'F', 'tcc': 'S', 'tac': 'Y', 'tgc': 'C',
4      'tta': 'L', 'tca': 'S', 'taa': '*', 'tga': '*',
5      'ttg': 'L', 'tcg': 'S', 'tag': '*', 'tgg': 'W',
6      'ctt': 'L', 'cct': 'P', 'cat': 'H', 'cgt': 'R',
```

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
    'ctc': 'L', 'ccc': 'P', 'cac': 'H', 'cgc': 'R',
    'cta': 'L', 'cca': 'P', 'caa': 'Q', 'cga': 'R',
    'ctg': 'L', 'ccg': 'P', 'cag': 'Q', 'cgg': 'R',
    'att': 'I', 'act': 'T', 'aat': 'N', 'agt': 'S',
    'atc': 'I', 'acc': 'T', 'aac': 'N', 'agc': 'S',
    'ata': 'I', 'aca': 'T', 'aaa': 'K', 'aga': 'R',
    'atg': 'M', 'acg': 'T', 'aag': 'K', 'agg': 'R',
    'gtt': 'V', 'gct': 'A', 'gat': 'D', 'ggt': 'G',
    'gtc': 'V', 'gcc': 'A', 'gac': 'D', 'ggc': 'G',
    'gta': 'V', 'gca': 'A', 'gaa': 'E', 'gga': 'G',
    'gtg': 'V', 'gcg': 'A', 'gag': 'E', 'ggg': 'G'   }
```

As can be guessed, a star `'*'` stands for a stop codon in this dictionary. We can use this dictionary to translate a gene into a protein:

```
1  def dna_translate(gene, code=universal):
2      ''' translate a DNA sequence into protein '''
3
4      prot = ""
5      gene = gene.lower() # to match lowercase format of universal
6      for i in range(0,len(gene),3):
7          codon = gene[i:i+3]
8          if codon in code:
9              prot += code[codon]  # get the corresponding amino acid
10         else:
11             prot += "?" # unexpected triplets that are not codons
12     return prot
```

```
>>> dna_translate("ATGGACGGCGGC")
'MDGG'
```

Code explained

The function `dna_translate` receives two parameters: one is the gene to be translated, and the second one is a default parameter set to the universal genetic code defined earlier. If one wishes to compute the translation according to a different genetic code, then the second parameter will be given as input to the function and overload the default value.

Line 4 creates an empty string to which the amino acids will be concatenated, one by one. Line 5 makes the translation case insensitive, by transforming the sequence into lowercase, in case it was not so. The loop in lines 6–11 iterates over the gene, in leaps of three nucleotides each time. Line 7 extracts the three nucleotides starting at position `i`. Line 8 checks if this codon appears in the dictionary of the translation (since the universal code dictionary contains all 64 possible triplets, this case will occur only if

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

the sequence contains characters that are not `a`, `t`, `c`, or `g`). In line 9, the dictionary is accessed to extract the value to which the codon is mapped. This value is a one-letter amino acid, and is concatenated to `prot`. If the membership check failed, `"?"` is appended instead.

Tuples, defined by comma separated elements inside `()`, are the immutable parallel of lists: they are ordered, can store any type of elements, but do not support item assignment:

```
>>> tup = (1,2,3)
>>> tup[2] = 5
Traceback (most recent call last):
  File "<pyshell#69>", line 1, in <module>
    tup[2] = 5
TypeError: 'tuple' object does not support item assignment
```

So tuples are something like "immutable lists." What do we use them for? One of the uses of tuples is as elements in sets, and keys in dictionaries: both must be immutable elements, for reasons beyond our scope here (we just note that it has to do with efficiency, of the kind discussed in the chapter on hashing). So a list can neither be an element in a set, nor a key in a dictionary. This is where tuples come to the rescue.

```
>>> s = {1,2,[3,4,5]}
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    s = {1,2,[3,4,5]}
TypeError: unhashable type: 'list'
>>> s = {1,2,(3,4,5)}
>>> s
{(3, 4, 5), 2, 1}
```

Pay attention that the order of the items in the set was changed in the last running and not by mistake. It is because a set is an unordered collection. Python has an inner order for it, but for us, as outside users, the order seems random.

1.11 Files

Files are used to store long-term data. Unlike variables defined during a program's execution, files can store much larger amounts of data, and these data do not get lost when the program terminates, not even when the computer shuts down. When biologists want to store the genome of an organism or the expression data of various genes in healthy and mutant cells, they store the data in files. The data in the files are later being read by programs to perform various analyses.

Working with text files in Python is very easy. Other formats (such as Excel spreadsheets or xml files) can also be handled, but involve more details and we do not deal with them here. The fundamental text file operations are **opening** and **closing** a file, **reading**

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

its content, and **writing** data to it. Here is an example, in which the content of one file is copied to another, except for the first line:

```
>>> my_file = open("./yeast_genome.txt")  <= ./  for current folder
                                          <= ../  for parent folder
>>> out_file = open("../out.txt", 'w')    <= 'w'  for writing, file will
                                          <= be created if does not exist

>>> first_line = my_file.readline()       <= read until  '\n'  encountered
>>> genome =  my_file.read()              <= read rest of file

>>> print(genome, file=out_file)          <= write  to file specified

>>> my_file.close()
>>> out_file.close()
```

Code explained

Python function `open` is used to open files. The name of the file is given as a string parameter. If the file is located at the same directory as the Python script file, the file's name is enough. Otherwise, you can provide the full path to the location of the file. "./" means the current directory, and "../" points to the parent directory. To write data to a file, it first has to be opened with a special flag 'w'. Then, a simple `print` command, directed to the file, will do the job.

The function `read` is used to load the content of the file. Assuming the file is a simple text file, the content is loaded as a single string type value, and can be stored in a variable. When the file contains line escape characters (the special "end of line" character `'\n'`), we may prefer to read its content line by line and handle each separately, using the `readline` function.

Closing a file terminates the communication with the file system. It is a good habit to close a file once done with it. This is especially important when writing data to the file: until the file is not closed, it is not guaranteed that all the data were actually written to it. This is because writing to a file is an expensive operation, and the computer occasionally prefers to delay it, buffer the data, and write them to the file only when enough data are accumulated.

1.12 Libraries

We conclude this chapter by presenting the use of **libraries**. Many useful objects and functions in Python come as part of libraries, also termed **packages**. A library can be either a standard part of Python or written by third party developers. For example, Python has a `math` library, containing various mathematical functions; a library for random sampling and probability called `random`; a library for time measurements called `time`; and many more. To

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

use a library, we first need to "import" it into the current running environment. Then, we can access all that it has to offer, as shown in the following examples:

```
>>> import random
>>> random.choice("ATCG")      <= function choice of package random
'A'                            <= picked an element uniformly at random
>>> random.choice("ATCG")      <= pick again
'G'
>>> random.choice(range(10))   <= randomly pick from the range 0-9
8

>>> seq = ""
>>> for i in range(10):
        seq += random.choice("ATCG") <= random DNA sequence of length 10

>>> seq
'AGCCACCCGA'

>>> random.random()             <= random decimal number from [0,1)
0.7601425767073609
>>> if random.random() <= 0.3: # if the number picked is between 0-0.3
        print("30% chance this will be printed")

>>> lst = [1,2,3,4]
>>> random.shuffle(lst)         <= randomly permute the list
>>>lst
[2, 1, 3, 4]
```

Exercise 19

Complete the following function `deletion_mutation(dna)` that receives a DNA sequence, and randomly deletes one of its bases.

```
import random
def deletion_mutation(dna):
    i = _____________________     # position to delete
    new_dna = ________________________
    return new_dna
```

Additional library functions will be presented later, when used. We refer the reader to Python's documentation to explore more packages and their functionality.

Reflection

Variables, operators, conditionals, loops, containers, files... we hope you are not too overwhelmed with the amount of information about Python presented in this chapter.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

Although the Python basics presented will be of use in the rest of this book, this does not mean you must be a Python expert to read it. Whenever you encounter something unclear in the code we present later in the book, take your time to play a bit with the code, clarifying the unclear details. This is a good point to remind you that learning a new programming language requires a lot of practice.

Please take your time to look at the problems under the "challenge yourself" section that ends this (and every) chapter. In some problems, you will be asked to use functions that you saw in this chapter. Other problems will require modifying them or even writing new functions. It is always good to remember that Python types have various useful functions that you may consider using. You will find the solutions on our website.

Challenge Yourself

Problem 1 Using dictionaries for protein representation

The following is a dictionary in Python that maps amino acids in 3-letter format to 1-letter format:

```
prot321_dict = {"GLY" : "G", "ALA" : "A", "LEU" : "L", "ILE" : "I",
           "ARG" : "R", "LYS" : "K", "MET" : "M", "CYS" : "C",
           "TYR" : "Y", "THR" : "T", "PRO" : "P", "SER" : "S",
           "TRP" : "W", "ASP" : "D", "GLU" : "E", "ASN" : "N",
           "GLN" : "Q", "PHE" : "F", "HIS" : "H", "XXX" : "X"}
```

(a) This dictionary has two problems:
   1. The value for Valine ("VAL") is missing. Add it to the dictionary.
   2. It has an incorrect key – "XXX." Delete it from the dictionary.

(b) Write a function `prot321(prot)` that takes a protein sequence `prot` in 3-letter representation (separated by spaces). For example: "`GLN ALA GLN ILE`." The function returns the protein in 1-letter representation (and no spaces). For example: '`QAQI`'.

```
>>> prot321("GLN ALA GLN ILE")
'QAQI'
```

Hint: You may want to use the function `split` from class str, to separate the input protein sequence into the triplets. For example:

```
>>> "GLN ALA GLN ILE".split()
['GLN', 'ALA', 'GLN', 'ILE']
```

Problem 2 File and string basics

In the website of this book, you will find a file IME1.txt, containing the DNA sequence of the gene IME1 (of S.Cerevisiae, sequence downloaded from www.yeastgenome.org). We want

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

you to read this sequence from the file, translate it into a protein, and compute the percentage of basic amino acids in it.

Stage A – Reading a file

The sequence is given in FASTA format. In this format, before the gene itself, there is a title row containing basic information on the gene. This row usually starts with the ">" character.

(a) Read the DNA sequence, omitting the title row, into a variable `ime1_dna`.
(b) The length of the gene should be 1083 nucleotides. Check if this is the case (hint: no!).
(c) Check if the sequence contains only the nucleotides `A`, `T`, `C`, `G`. If not, what other characters are in that string?
(d) Another way to answer the previous section is by using a structure called **list comprehension.** This is just another convenient way to form lists in Python.
The general syntax is: [*expression* for *variable_name* in *sequence* if *condition*]

Now, run this command:

```
>>> [char for char in ime1_dna if char not in "ATCG"]
```

(e) Clean the sequence from non-ATCG characters. Use the method `replace` of class str.
(f) Check again the length of the gene. It should now be 1083.
(g) Verify that the length of the gene is divided by 3, using the modulo (%) operator.
(h) What are the last three nucleotides in the sequence? Use string slicing.
(i) Verify, using the dictionary 'universal' (that maps codons to amino acids), which we saw earlier, that the last codon is a stop codon.

Stage B – Translation and analysis

(a) Use the function `dna_translate` to translate the IME1 gene into a protein. Verify that the length of the protein is what you expect.
(b) Note that the last character in the protein is '*', which represents a stop codon. Remove this character from the string (you can use slicing or the method `replace`).
(c) What is the percentage of basic amino acids (lysine – K, histidine – H, arginine – R) in the protein?

Problem 3 Estimating silent mutation probability

Let us define a *complete random mutation*: given a DNA sequence, a randomly picked nucleotide in it will change to one of the other three possibilities, also picked at random. For example, given "`ATTCGG`," assume the first position was picked for the mutation, then `A` will be changed to either `C`, `G`, or `T` at random with equal probabilities.

In this problem, we will empirically estimate the probability for a complete random mutation being silent (that is, not affecting the protein sequence it encodes).

Following is a function `rand_mut`, which takes a DNA sequence, and returns this sequence after a complete random mutation has been applied to it.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

```
def rand_mut(dna):
    ''' return a new dna with a single random mutation '''
    dna = dna.lower()
    position = random.randrange(0, len(dna))
    nuc_targets = list("atcg")
    nuc_targets.remove(dna[position])
    target = random.choice(nuc_targets)
    new_dna = dna[:position] + target + dna[position+1:]
    return new_dna
```

For example:

```
>>> rand_mut("ATTCGG")
'GTTCGG'
>>> rand_mut("ATTCGG")
'ATTCAG'
```

(a) Complete the following function `is_silent`, which applies a random mutation in its input DNA sequence, and returns `True` if this mutation was silent, otherwise `False`.

```
def is_silent(dna):
    new = rand_mut(dna)
    return_____________________________
```

(b) Complete the function `prob_silent`, which takes a DNA sequence, and the desired number of simulations required, and returns the fraction of simulations that yielded a silent random mutation. Note that each simulation generates a single random mutation at the original DNA, that is, the mutations are not accumulated.

```
def prob_silent(dna, num_simulations):
```

```
>>> prob_silent("atc", 10000)
0.2157
>>> prob_silent("atc", 10000)
0.2238
```

Note the different results in the last two executions. This is expected as each time different mutations are randomly generated.

(c) The probability for a silent mutation in the codon 'atc' is 2/9 (why?). Verify this experimentally by enlarging the number of simulations, which increases the significance of the output.

(d) Let us define a random DNA as a DNA sequence in which each position is randomly and uniformly assigned one of the four nucleotides. Write a function `rand_dna(n)` that generates a random DNA of length n.

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press

(e) Estimate the silent mutation probability in a random DNA of length 1000.
(f) Estimate the silent mutation probability in the DNA sequence of the S.Cerevisiae IME1 gene from the previous question.
(g) Discussion: Can you explain the results? Are they surprising?

https://doi.org/10.1017/9781108178327.003 Published online by Cambridge University Press