Chapter 9

Handling Strings and Bytes

In this chapter, we present some of the most used methods in strings and bytes objects. Strings are extremely useful to manage most of the output generated from programs, like formatted documents or messages that involve program variables. Bytes allow us, among other uses, to perform input/output operations that make possible the communication between programs by sending/receiving data through different channels in low-level representation.

9.1 Some Built-in Methods for Strings

In Python, all strings are an immutable sequence of *Unicode* characters. *Unicode* is a standard encoding that allows us to have a virtual representation of any character. Here we have some different ways to create a string in Python:

```
a = "programming"
b = 'a lot'
c = '''a string
with multiple
lines'''
d = """Multiple lines with
     double quotation marks """
e = "Three " "Strings" " Together"
f = "a string " + "concatenated"

print(a)
print(b)
print(c)
print(d)
```

```
15 print(e)
16 print(f)
```

```
programming
a lot
a string
with multiple
lines
Multiple lines with
     double quotation marks
Three Strings Together
a string concatenated
```

The type `str` has several methods to manipulate strings. Here we have the list:

```
1 print(dir(str))
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__',
 '__eq__', '__format__', '__ge__', '__getattribute__', '__getitem__',
 '__getnewargs__', '__gt__', '__hash__', '__init__', '__iter__', '__le__',
'__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', 'capitalize', 'casefold',
'center', 'count', 'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal', 'isdigit',
'isidentifier', 'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle',
'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',
'translate', 'upper', 'zfill']
```

Now we show some examples of methods to manipulate strings. We defer the reader to Python documentation for more examples of `string` methods.

The `isalpha()` method checks whether the string contains alpha characters or not. It returns `True` if all the characters are in the alphabet of some language:

```
1 print("abn".isalpha())
```

```
True
```

If there is a number, blank space or punctuation marks, it will return `False`:

```
print("t/".isalpha())
```

```
False
```

The `isdigit()` method returns `True` if all the characters in the string are digits:

```
print("34".isdigit())
```

```
True
```

We can check if a portion of a string includes a specific sub-sequence within it by `startswith()` and `endswith()` methods:

```
s = "I'm programming"
print(s.startswith("I'm"))
print(s.endswith("ing"))
```

```
True
True
```

If we require searching for a sub-sequence anywhere within a string we use the `find(seq)` method, which returns the index of *s* where the argument's sequence `seq` starts:

```
print(s.find("m p"))
```

```
2
```

The index method `index(str, beg=0 end=len(string)-1)` returns the index of where the sequence `str` starts within the string `s`. It always returns the first appearance of the argument `str` in `s` and starts at 0 as other Python indexing cases:

```
print(s.index('g'))
```

```
7
```

If we do not indicate the beginning or ending of the substring, `index()` method will use by default `beg=0` and `end=len(string)-1`. The next example shows how to search a substring that starts at position 4 and ends at position 10:

```
1 print(s.index('o', 4, 10))
```

```
6
```

Python will let us know if we use the right boundaries arguments to search:

```
1 print(s.index('i', 5, 10))
```

```
Traceback (most recent call last):
  File "2.py", line 29, in <module>
    print(s.index('i', 5, 10))
ValueError: substring not found
```

The `split()` method generates a list of words in `s` separated by blank spaces:

```
1 s = "Hi everyone, how are you?"
2 s2 = s.split()
3 print(s2)
```

```
['Hi', 'everyone,', 'how', 'are', 'you?']
```

By default `split()` uses blank spaces. The `join()` method let us to create a string concatenating the words in a list of strings through a specific character. The next example join the words in `s2` using the # character:

```
1 s3 = '#'.join(s2)
2 print(s3)
```

```
Hi#everyone,#how#are#you?
```

We can change portions of a string indicating the sub-sequence that we want to change and the character to replace:

```
1 print(s.replace(' ', '**'))
2 print(s)
```

```
Hi**everyone,**how**are**you?
```

The `partition(seq)` method splits any string at the first occurrence of the sub-sequence `seq`. It returns a tuple with the part before the sub-sequence, the sub-sequence and the remaining portion of the string:

```
1 s5 = s.partition(' ')
2 print(s5)
3 print(s)
```

```
('Hi', ' ', 'everyone, how are you?')
Hi everyone, how are you?
```

As we have seen in previous chapters, we can insert variable values into a string by using `format`:

```
1  # 4.py
2
3  name = 'John Smith'
4  grade = 4.5
5  if grade >= 5.0:
6      result = 'passed'
7  else:
8      result = 'failed'
9
10 template = "Hi {0}, you have {1} the exam. Your grade was {2}"
11 print(template.format(name, result, grade))
```

```
Hi John Smith, you have failed the exam. Your grade was 4.5
```

If we want to include braces within the string, we can escape them by using double braces. In the example below, we print a *Java* class definition:

```
1  # 5.py
2
3  template = """
4  public class {0}
5  {{
6          public static void main(String[] args)
7          {{
8              System.out.println({1});
9          }}
10 }}"""
11
12 print(template.format("MyClass", "'hello world'"));
```

```
public class MyClass
{
        public static void main(String[] args)
```

```
        {
            System.out.println('hello world');
        }
}
```

Sometimes we want to include several variables inside a string. This makes it hard to remember the order in which we have to write them inside of `format`. One solution is to use arguments with `keywords` in the function:

```
1  # 6.py
2
3  print("{} {label} {}".format("x", "y", label="z"))
```

```
x z y
```

```
1  # 7.py
2
3  template = """
4  From: <{from_email}>
5  To: <{to_email}>
6  Subject: {subject}
7  {message}
8  """
9
10 print(template.format(
11     from_email="someone@domain.com",
12     to_email="anyone@example.com",
13     message="\nThis is a test email.\n\nI hope this be helpful!",
14     subject="This email is urgent")
15 )
```

```

From: <someone@domain.com>
To: <anyone@example.com>
Subject: This email is urgent

This is a test email.

I hope this be helpful!
```

We can also use lists, tuples or dictionaries as argument into `format`:

```
# 8.py

emails = ("a@example.com", "b@example.com")
message = {'subject': "You have an email.",
           'message': "This is an email to you."}

template = """
From: <{0[0]}>
To: <{0[1]}>
Subject: {message[subject]} {message[message]}
"""

print(template.format(emails, message=message))
```

```
From: <a@example.com>
To: <b@example.com>
Subject: You have an email. This is an email to you.
```

We can even use a dictionary and index it inside the string:

```
# 9.py

header = {"emails": ["me@example.com", "you@example.com"],
          "subject": "Look at this email."}

message = {"text": "Sorry this is not important."}

template = """
From: <{0[emails][0]}>
To: <{0[emails][1]}>
Subject: {0[subject]}
{1[text]}"""

print(template.format(header, message))
```

```
From: <me@example.com>
To: <you@example.com>
Subject: Look at this email.
Sorry this is not important.
```

We can also pass any object as an argument. For example, we can pass an instance of a class and then access to any of the attributes of the object:

```
1  # 10.py
2
3  class EMail:
4      def __init__(self, from_addr, to_addr, subject, message):
5          self.from_addr = from_addr
6          self.to_addr = to_addr
7          self.subject = subject
8          self.message = message
9
10 email = EMail("a@example.com", "b@example.com", "You have an email.",
11               "\nThe message is useless.\n\nBye!")
12
13 template = """
14 From: <{0.from_addr}>
15 To: <{0.to_addr}>
16 Subject: {0.subject}
17 {0.message}"""
18 print(template.format(email))
```

```
From: <a@example.com>
To: <b@example.com>
Subject: You have an email.

The message is useless.

Bye!
```

We can also improve the format of the strings that we print. For instance, when you have to print a table with data, most of the time you want to show the values of the same variable aligned in columns:

```
# 11.py

items_bought = [('milk', 2, 120), ('bread', 3.5, 800), ('rice', 1.75, 960)]

print("PRODUCT  QUANTITY   PRICE   SUBTOTAL")
for product, price, quantity, in items_bought:
    subtotal = price * quantity
    print("{0:8s}{1: ^9d}    ${2: <8.2f}${3: >7.2f}"
            .format(product, quantity, price, subtotal))
```

```
PRODUCT  QUANTITY   PRICE   SUBTOTAL
milk       120        $2.00    $ 240.00
bread      800        $3.50    $2800.00
rice       960        $1.75    $1680.00
```

Note that within each key there is a dictionary-type item, in other words, before the colon is the index of the argument within the `format` function. The string format is given after the colon. For example, `8s` means that the data is a string of 8 characters. By default, if the string is shorter than 8 characters, the rest is filled with spaces (on the right). If we enter a string longer than 8 characters, it will not be truncated. We can force longer strings to be truncated by adding a dot before the number that indicates the number of characters. As an example, if the format is `{1: ^9d}`:

- 1 corresponds to the index of the argument in the `format` function
- The space after the colon says that the empty spaces must be filled with spaces (by default integer types are filled with zeros)
- The symbol ^ is used to center the number in the available space
- `9d` means it will be an integer up to 9 digits

The order of these parameters, although they are optional, should be from left to right after the colon: character to fill the empty spaces, alignment, size and then type.

In the case of the price, `{2: <8.2f}` means that the used data is the third argument of the `format` function. The free places are filled with spaces; the $<$ symbol means that the alignment is to the left, the number is a float of up to 8

characters, with two decimals. Similarly, in the case of the subtotal `{3:> 7.2f}`, it means that the used data is the fourth argument in the `format` function, the filling character is space, the alignment is to the right, and the number is a 7 digit float, including two decimal places.

9.2 Bytes and I/O

At the beginning of the chapter, we said that Python strings are an immutable collection of Unicode characters. Unicode is not a valid data storage format. We often read information of a string from some file or socket in bytes, not in Unicode. Bytes are the lowest level storage format. They represent a sequence of 8 bits which are described as an integer between 0 and 255, an equivalent hexadecimal between 0 and FF, or a literal (only ASCII characters are allowed to represent bytes).

Bytes can represent anything, such as coded character strings, or pixels of an image. In general, we need to know how they are coded to interpret the correct data type represented by bytes. For example, a binary pattern of 8 bits (one byte) may correspond to a particular character if is decoded as ASCII, or to a different character if is decoded as Unicode.

In Python, bytes are represented with the `bytes` object. To declare that an object is a byte you must add a *b* at the beginning of the object `a`. For example:

```
1  # 12.py
2
3  # What is between quotes is a byte object
4  characters = b'\x63\x6c\x69\x63\x68\xe9'
5  print(characters)
6  print(characters.decode("latin-1"))
7
8  # 61 and 62 are the hexadecimal representation of 'a' and 'b'
9  characters = b"\x61\x62"
10 print(characters.decode("ascii"))
11
12 # 97 and 98 are the corresponding ASCII code of 'a' and 'b'
13 characters = b"ab"
14 print(characters)
15 characters = bytes((97, 98))
16 print(characters)
17 print(characters.decode("ascii"))
```

```
b'clich\xe9'
cliché
ab
b'ab'
b'ab'
ab
```

```
1 # 13.py
2
3 # This generate an error because it is only possible to use ascii literals
4 # to create bytes
5
6 characters = b"áb"
```

```
---------------------------------------------------------------------------
SyntaxError Traceback (most recent call last)
<ipython-input-19-826203d742e2> in <module>()
---> 4 caracteres = b"áb"

SyntaxError: bytes can only contain ASCII literal characters.
```

The space symbol indicates that the two characters after the `x` correspond to a byte in hexadecimal digits. The bytes that coincide with the `ASCII` bytes are immediately recognized. When we print these characters, they appear correctly; the rest are printed as hexadecimal. The `b` reminds us that what is in the right is a `bytes` object, not a `string`. The sentence `characters.decode("latin-1")` decodes the sequence of bytes by using the `"latin-1"` alphabet.

The `decode` method returns a `Unicode` string. If we use another alphabet, we get another string:

```
1 # 14.py
2
3 characters = b'\x63\x6c\x69\x63\x68\xe9'
4 print(characters.decode("latin-1"))
5 print(characters.decode("iso8859-5"))
```

```
cliché
clich
```

To code a string in a different alphabet, we simply have to use the `encode` method from `str`. It is necessary to indicate the encoding alphabet:

```
1  # 15.py
2
3  characters = "estación"
4  print(characters.encode("UTF-8"))  # 8-bit Unicode Transformation Format
5  print(characters.encode("latin-1"))
6  print(characters.encode("CP437"))
7  print(characters.encode("ascii"))  # Can't enconde in ascii the character ó
```

```
b'estaci\xc3\xb3n'
b'estaci\xf3n'
b'estaci\xa2n'

---------------------------------------------------------------------------
UnicodeEncodeError                        Traceback (most recent call last)
<ipython-input-32-844dd5b7b18c> in <module>()
      3 print(characters.encode("latin-1"))
      4 print(characters.encode("CP437"))
----> 5 print(characters.encode("ascii")) # can't enconde in ascii the character ó

UnicodeEncodeError: 'ascii' codec can't encode character '\xf3' in position 6:
ordinal not in range(128)
```

The `encode` method has options to handle the cases where the string we want to code cannot be coded with the requested alphabet. These options are passed in key argument errors. Possible values are: `'strict'` by default to raise a `UnicodeDecodeError` exception, `'replace'` to replace an unknown character with symbol ?, `'ignore'` to skip the character, and `'xmlcharrefreplace'` to create a xml entity that represents a Unicode character.

```
1  # 16.py
2
3  characters = "estación"
4  print(characters.encode("ascii", errors='replace'))
5  print(characters.encode("ascii", errors='ignore'))
```

```
6 print(characters.encode("ascii", errors='xmlcharrefreplace'))
```

```
b'estaci?n'
b'estacin'
b'estaci&#243;n'
```

In general, if we want to code a string and we do not know the alphabet we should use, the best to do is to use *UTF-8* because it is compatible with `ASCII`.

9.3 bytearrays

Just like the name suggests, *bytearrays* are arrays of bytes, that in contrast to bytes, they are mutable. They behave like lists: we can index them with slice notation, and we can add bytes to them with `extend`. To build a *bytearray* we must enter an initial byte:

```
1  # 17.py
2
3  ba_1 = bytearray(b"hello world")
4  print(ba_1)
5  print(ba_1[3:7])
6  ba_1[4:6] = b"\x15\xa3"
7  print(ba_1)
8  ba_1.extend(b"program")
9  print(ba_1)
10 # Here it prints an int, the ascii that corresponds to the letter 'h'
11 print(ba_1[0])
12 print(bin(ba_1[0]))
13 print(bin(ba_1[0])[2:].zfill(8))
```

```
bytearray(b'hello world')
bytearray(b'lo w')
bytearray(b'hell\x15\xa3world')
bytearray(b'hell\x15\xa3worldprogram')
104
0b1101000
01101000
```

Note that the last line is used to print the bits that correspond to the first byte. The `[2:]` is used to start from the third position, because the first two `b'` indicate that the format is binary. When we add `.zfill(8)` we mean that 8 bits will be used to represent a byte. It only makes sense when there are only zeros at the left side.

A one-byte character can be converted to an integer using the `ord` function:

```
1 # 18.py
2
3 print(ord(b"a"))
4 b = bytearray(b'abcdef')
5 b[3] = ord(b'g')  # The ASCII code for g is 103
6 b[4] = 68  # The ASCII code for D is 68, this is the same as b[4] = ord(b'D')
7 print(b)
```

```
97
bytearray(b'abcgDf')
```

9.4 Hands-On Activities

Activity 9.1

The teacher assistants of the *Advanced Programming* course, rescued at the last second all students data from the Computer Science Department's ERP. However, during the download some errors appeared and produced the following changes:

- In some rows, a random natural number followed by a blank space was placed at the beginning of the student's name. For example: *Jonh Smith* changed to *42 John Smith*. Those numbers must be removed.
- For each sequence of letters *n*, an additional *n* was added at the end of the concatenation. For example: *ANNE* became *ANNNE*. These names must be corrected. Assume that there are no names or last names with more than two consecutive *n*.
- Some names changed from uppercase to lowercase. All names must be in uppercase.

The rescued **students** list was saved as a **.csv** file. The main characteristic of this kind of files are:

- Each column is separated by commas

- The first row is the header for all columns.

You must convert all the data to three different formats: **LaTeX**, **HTML**, **Markdown**. The file *FORMATS.md* explains all the different formats and the representation that is required. Consider that the **students.csv** file always have three columns, but an unknown finite number of rows. Assume that there is no missing data. All the required files are available at `https://advancedpythonprogramming.github.io/`

Activity 9.2

One of your friends has to solve a big challenge: to separate a weird audio file that has two audio tracks mixed. Your friend did some research and found out that the file can be modified reading its data as bytes. The problem is that your friend has no idea about bytes, but he knows that you are reading this book, so he asked you to solve this big problem.

The input file, `music.wav` is a mix of two songs. You will have to read this file and generate two other files, `song1.wav` and `song2.wav`, each one with a separate audio. All the required files are available at `https://advancedpythonprogramming.github.io/`

The file `music.wav` has the mixed audio in the following way: From the byte 44 (start counting from 0), one byte corresponds to the first audio, and the other byte corresponds to the second. If `x` and `y` represent the bytes from the first and second audio respectively; the bytes are in the following order:

x y x y x y x y ...

Figure 9.1 shows the structure of a WAV audio file.

Consider the following:

- **The size of each file will change after the separation:** Each output file has half of the original data. Consider this when writing the headers of the output files. **Tip: Use the 4-byte representation for these integers.**
- The rest of the header data can be re-used *without modification* as they appear in the input, but do not forget to add them.

Bonus track

Now we have a new file, `audio.wav`. You suppose to do the same separation previously performed, but now the sampling frequency is different for each output file. In this case, bytes 25 to 28 and 29 to 32 are different depending on

the file. The values are[1]:

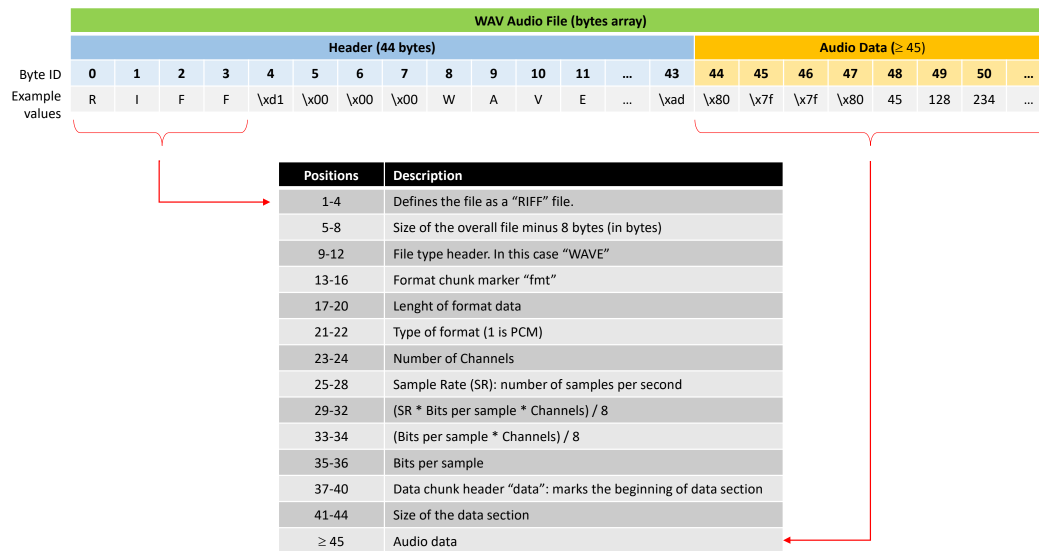
- For the first file, write 11025 in both groups of bytes.
- For the second file, write 22050 in both groups of bytes.

- Run the previous algorithm with this audio and obtain the correct output.

Tips

- `int.from_bytes(bytes, byteorder='little')`: This function transform a byte to the equivalent integer.
- `int.to_bytes(size, byteorder='little')`: This function transforms an integer to bytes of the given size, with the given size of bytes.

[1]The sampling frequency corresponds to the number of samples in one unit of time from a continuous signal during the process of conversion from analogous to digital. The product is a discrete signal that can be reconstructed by a digital audio reproducer when it knows the sampling frequency.



Positions	Description
1-4	Defines the file as a "RIFF" file.
5-8	Size of the overall file minus 8 bytes (in bytes)
9-12	File type header. In this case "WAVE"
13-16	Format chunk marker "fmt"
17-20	Lenght of format data
21-22	Type of format (1 is PCM)
23-24	Number of Channels
25-28	Sample Rate (SR): number of samples per second
29-32	(SR * Bits per sample * Channels) / 8
33-34	(Bits per sample * Channels) / 8
35-36	Bits per sample
37-40	Data chunk header "data": marks the beginning of data section
41-44	Size of the data section
≥ 45	Audio data

Figure 9.1: Figure shows the WAV audio file structure. Note that Byte ID starts from 0 and the position in the structure form 1.