Theory: Flush and file arguments of print

21 minutes 7 / 8 problems solved

Start practicing

In this topic, we will discuss arguments that can be extremely helpful when working with files. As you know, the `print()` function takes keyword arguments `sep` and `end`. Now it's time to learn that it takes two more: `file` and `flush`.

§1. The file

The `file` argument is in charge of where the function will write given objects to. By default, it is `sys.stdout`, the standard output stream that prints objects on the screen.

Instead of the default value, you can specify the file parameter with the name of the required file. The file should be **opened in advance**, and make sure to close it when you are done:

```
my_file = open('testfile.txt', 'w')
print('This string will be in the file...', file=my_file)
my_file.close()
```

Alternatively, you can set `file` to `sys.stderr`, a standard error stream. It is very useful when debugging a small program because you can print out errors using this method:

```
import sys

def division():
    a = int(input("Set the first number: "))
    b = int(input("Set the second number: "))
    if b != 0:
        print(a / b)
    else:
        # The string below will look like an error message.

        print("The second number cannot be a zero!", file=sys.stderr)
```

§2. The flush

The last argument is the bool parameter `flush`. It can only be used together with specifying the file argument. `False` by default, if set to `True`, it forces the stream to be flushed. But what exactly does it mean?

When writing to files, the output is **buffered** in memory and accumulated until the buffer is full, at which point the buffer gets **flushed**, so the content is written from the buffer to the file. The thing is, it's just more efficient to flush the buffer fewer times, and the user is less likely to notice if the output is not flushed each new line. The **default size** of the buffer depends on the operating system but it is also possible to specify the desired size of the buffer when opening the file (using the keyword argument `buffering`).

Getting back to our topic, if `flush=False`, writing to file takes place when the buffer is full (or when the file is being closed even if the buffer is not full yet). And if `flush=True`, the input is written to the file **straight away**. Below are two examples to understand the difference.

Without the parameter:

Current topic:

✓ Flush and file arguments of print

Topic depends on:

✓ Sep and end arguments of print

✓ Writing files Stage 2

Table of contents:

↑ Flush and file arguments of print

§1. The file

§2. The flush

§3. Print vs Write

§4. Conclusion

Feedback & Comments

```
import time

out = open('file1.txt', 'w')

for i in range(3):
    print(i, file=out)
    time.sleep(5)


# at this moment the file is still empty because the buffer has not been flushed

out.close()

# now the numbers have appeared in the file
```

`time.sleep(5)` makes the program stop for five seconds so that we can notice that the file remains empty till the program is over.

With the `flush` parameter:

```
import time

out2 = open('file2.txt', 'w')

for i in range(3):
    print(i, file=out2, flush=True)
    # the numbers are immediately written in the file
    time.sleep(5)


out2.close()
```

Here, if you open the file before the program is over you will see that some digits are already there.

> Note that `file` and `flush`, just as `sep` and `end`, are keyword arguments. You should explicitly specify them when calling the function.

§3. Print vs Write

You might wonder if there's any difference between the methods `file.write()` and `print()` with the `file` argument. Though they do roughly the same (write a string to a file), there are a few things to keep in mind:

- `print()` renders the given objects, that is, converts them to strings, and there's an opportunity to specify the `sep` and `end` arguments or leave the defaults (a space and `\n` correspondingly), as well as either to flush the stream or not.
- `file.write()`, on the contrary, writes the exact string that is given only when the buffer is full. This also implies that it takes only a string as an argument, and if we want to print a number, for example, we should explicitly convert it into a string.

The first one is a bit more flexible and might be helpful when we don't want to think about rendering objects ourselves, while the second one offers a conscious approach due to explicitness.

§4. Conclusion

Arguments of the `print()` function discussed in this topic provide a very useful opportunity not only to write objects in the system's standard output, but also to make them look like error messages or write them to files. With the help of these arguments, we can do so either straight away or (by default) when the buffer is full, and without rendering the objects ourselves. Keep that in mind when coding further!

Start practicing

Comments (3) Hints (0) Useful links (0) Show discussion