

7. Input and Output

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

7.1. Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the [print\(\)](#) function. (A third way is using the [write\(\)](#) method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are several ways to format output.

- To use [formatted string literals](#), begin a string with `f` or `F` before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between `{` and `}` characters that can refer to variables or literal values.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- The [str.format\(\)](#) method of strings requires more manual effort. You'll still use `{` and `}` to mark where a variable will be substituted and can provide detailed formatting directives, but you'll also need to provide the information to be formatted. In the following code block there are two examples of how to format variables:

```
>>> yes_votes = 42_572_654
>>> total_votes = 85_705_149
>>> percentage = yes_votes / total_votes
```

```
>>> '{:-9} YES votes {:2.2%}'.format(yes_votes, percentage)
' 42572654 YES votes 49.67%'
```

Notice how the `yes_votes` are padded with spaces and a negative sign only for negative numbers. The example also prints percentage multiplied by 100, with 2 decimal places and followed by a percent sign (see [Format Specification Mini-Language](#) for details).

- Finally, you can do all the string handling yourself by using string slicing and concatenation operations to create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width.

When you don't need fancy output but just want a quick display of some variables for debugging purposes, you can convert any value to a string with the [`repr\(\)`](#) or [`str\(\)`](#) functions.

The [`str\(\)`](#) function is meant to return representations of values which are fairly human-readable, while [`repr\(\)`](#) is meant to generate representations which can be read by the interpreter (or will force a [SyntaxError](#) if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, [`str\(\)`](#) will return the same value as [`repr\(\)`](#). Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Some examples:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print(s)
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
>>> hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print(hellos)
'hello, world\n'
```

```
>>> # The argument to repr() may be any Python object:  
>>> repr((x, y, ('spam', 'eggs')))  
"(32.5, 40000, ('spam', 'eggs'))"
```

The [string](#) module contains support for a simple templating approach based upon regular expressions, via [string.Template](#). This offers yet another way to substitute values into strings, using placeholders like `$x` and replacing them with values from a dictionary. This syntax is easy to use, although it offers much less control for formatting.

7.1.1. Formatted String Literals

[Formatted string literals](#) (also called f-strings for short) let you include the value of Python expressions inside a string by prefixing the string with `f` or `F` and writing expressions as `{expression}`.

An optional format specifier can follow the expression. This allows greater control over how the value is formatted. The following example rounds pi to three places after the decimal:

```
>>> import math  
>>> print(f'The value of pi is approximately {math.pi:.3f}.')  
The value of pi is approximately 3.142.
```

Passing an integer after the `:` will cause that field to be a minimum number of characters wide. This is useful for making columns line up.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}  
>>> for name, phone in table.items():  
...     print(f'{name:10} ==> {phone:10d}')  
...  
Sjoerd      ==>      4127  
Jack        ==>      4098  
Dcab        ==>      7678
```

Other modifiers can be used to convert the value before it is formatted. `!a` applies [ascii\(\)](#), `!s` applies [str\(\)](#), and `!r` applies [repr\(\)](#):

```
>>> animals = 'eels'  
>>> print(f'My hovercraft is full of {animals}.')  
My hovercraft is full of eels.
```

```
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

The `=` specifier can be used to expand an expression to the text of the expression, an equal sign, then the representation of the evaluated expression:

```
>>> bugs = 'roaches'
>>> count = 13
>>> area = 'living room'
>>> print(f'Debugging {bugs=} {count=} {area=}')
Debugging bugs='roaches' count=13 area='living room'
```

See [self-documenting expressions](#) for more information on the `=` specifier. For a reference on these format specifications, see the reference guide for the [Format Specification Mini-Language](#).

7.1.2. The String `format()` Method

Basic usage of the [`str.format\(\)`](#) method looks like this:

```
>>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the [`str.format\(\)`](#) method. A number in the brackets can be used to refer to the position of the object passed into the [`str.format\(\)`](#) method.

```
>>> print('{0} and {1}'.format('spam', 'eggs'))
spam and eggs
>>> print('{1} and {0}'.format('spam', 'eggs'))
eggs and spam
```

If keyword arguments are used in the [`str.format\(\)`](#) method, their values are referred to by using the name of the argument.

```
>>> print('This {food} is {adjective}'.format(
...     food='spam', adjective='absolutely horrible'))
This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

```
>>> print('The story of {0}, {1}, and {other}.'.format('Bill', 'Manfred',
...                                              other='Georg'))
The story of Bill, Manfred, and Georg.
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets '[]' to access the keys.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the `table` dictionary as keyword arguments with the `**` notation.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables:

```
>>> table = {k: str(v) for k, v in vars().items()}
>>> message = " ".join([f'{k}: {v}' for k in table.keys()])
>>> print(message.format(**table))
__name__: __main__; __doc__: None; __package__: None; __loader__: ...
```

As an example, the following lines produce a tidily aligned set of columns giving integers and their squares and cubes:

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1    1    1
 2    4    8
 3    9   27
 4   16   64
 5   25  125
 6   36  216
 7   49  343
```

```
8 64 512
9 81 729
10 100 1000
```

For a complete overview of string formatting with [str.format\(\)](#), see [Format String Syntax](#).

7.1.3. Manual String Formatting

Here's the same table of squares and cubes, formatted manually:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' on previous line
...     print(repr(x*x*x).rjust(4))
...
1   1    1
2   4    8
3   9   27
4  16   64
5  25  125
6  36  216
7  49  343
8  64  512
9  81  729
10 100 1000
```

(Note that the one space between each column was added by the way [print\(\)](#) works: it always adds spaces between its arguments.)

The [str.rjust\(\)](#) method of string objects right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods [str.ljust\(\)](#) and [str.center\(\)](#). These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, [str.zfill\(\)](#), which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```
>>> '12'.zfill(5)
'00012'
```

```
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

7.1.4. Old string formatting

The % operator (modulo) can also be used for string formatting. Given `format % values` (where `format` is a string), % conversion specifications in `format` are replaced with zero or more elements of `values`. This operation is commonly known as string interpolation. For example:

```
>>> import math
>>> print('The value of pi is approximately %5.3f.' % math.pi)
The value of pi is approximately 3.142.
```

More information can be found in the [printf-style String Formatting](#) section.

7.2. Reading and Writing Files

`open()` returns a [file object](#), and is most commonly used with two positional arguments and one keyword argument: `open(filename, mode, encoding=None)`

```
>>> f = open('workfile', 'w', encoding="utf-8")
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. `mode` can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The `mode` argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific `encoding`. If `encoding` is not specified, the default is platform dependent (see [open\(\)](#)). Because UTF-8 is the modern de-facto standard, `encoding="utf-8"` is recommended unless you know that you need to use a different encoding. Appending a `'b'` to the mode opens the file in *binary mode*. Binary mode data is read and written as [bytes](#) objects. You can not specify `encoding` when opening file in binary mode.

Table of Contents

- 7. Input and Output
 - 7.1. Fancier Output Formatting
 - 7.1.1. Formatted String Literals
 - 7.1.2. The String `format()` Method
 - 7.1.3. Manual String Formatting
 - 7.1.4. Old string formatting
 - 7.2. Reading and Writing Files
 - 7.2.1. Methods of File Objects
 - 7.2.2. Saving structured data with `json`

Previous topic

- 6. Modules

Next topic

- 8. Errors and Exceptions

This page

- [Report a bug](#)
- [Show source](#)

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

It is good practice to use the `with` keyword when dealing with file objects. The advantage is that the file is properly closed after its suite finishes, even if an exception is raised at some point. Using `with` is also much shorter than writing equivalent `try-finally` blocks:

```
>>> with open('workfile', encoding="utf-8") as f:  
...     read_data = f.read()  
  
>>> # We can check that the file has been automatically closed.  
>>> f.closed  
True
```

If you're not using the `with` keyword, then you should call `f.close()` to close the file and immediately free up any system resources used by it.

Warning: Calling `f.write()` without using the `with` keyword or calling `f.close()` **might** result in the arguments of `f.write()` not being completely written to the disk, even if the program exits successfully.

After a file object is closed, either by a `with` statement or by calling `f.close()`, attempts to use the file object will automatically fail.

```
>>> f.close()  
>>> f.read()  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ValueError: I/O operation on closed file.
```

7.2.1. Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string (in text mode) or bytes object (in binary mode). `size` is an optional numeric argument. When `size` is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most `size` characters (in text mode) or `size` bytes (in binary mode) are read and returned. If the end of the file has been reached, `f.read()` will return an empty string ('').

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by '`\n`', a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
>>> for line in f:
...     print(line, end='')
...
This is the first line of the file.
Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of `string` to the file, returning the number of characters written.

```
>>> f.write('This is a test\n')
15
```

Other types of objects need to be converted – either to a string (in text mode) or a bytes object (in binary mode) – before writing them:

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use `f.seek(offset, whence)`. The position is computed from adding `offset` to a reference point; the reference point is selected by the `whence` argument. A `whence` value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. `whence` can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)      # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid `offset` values are those returned from the `f.tell()`, or zero. Any other `offset` value produces undefined behaviour.

File objects have some additional methods, such as [isatty\(\)](#) and [truncate\(\)](#) which are less frequently used; consult the Library Reference for a complete guide to file objects.

7.2.2. Saving structured data with [json](#)

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the [read\(\)](#) method only returns strings, which will have to be passed to a function like [int\(\)](#), which takes a string like '123' and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called [JSON \(JavaScript Object Notation\)](#). The standard module called [json](#) can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

Note: The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
>>> import json  
>>> x = [1, 'simple', 'list']  
>>> json.dumps(x)  
'[1, "simple", "list"]'
```

Another variant of the [dumps\(\)](#) function, called [dump\(\)](#), simply serializes the object to a [text file](#). So if `f` is a [text file](#) object opened for writing, we can do this:

```
json.dump(x, f)
```

To decode the object again, if `f` is a [binary file](#) or [text file](#) object which has been opened for reading:

```
x = json.load(f)
```

Note: JSON files must be encoded in UTF-8. Use `encoding="utf-8"` when opening JSON file as a [text file](#) for both of reading and writing.

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the [json](#) module contains an explanation of this.

See also: [pickle](#) - the pickle module

Contrary to [JSON](#), [pickle](#) is a protocol which allows the serialization of arbitrarily complex Python objects. As such, it is specific to Python and cannot be used to communicate with applications written in other languages. It is also insecure by default: deserializing pickle data coming from an untrusted source can execute arbitrary code, if the data was crafted by a skilled attacker.