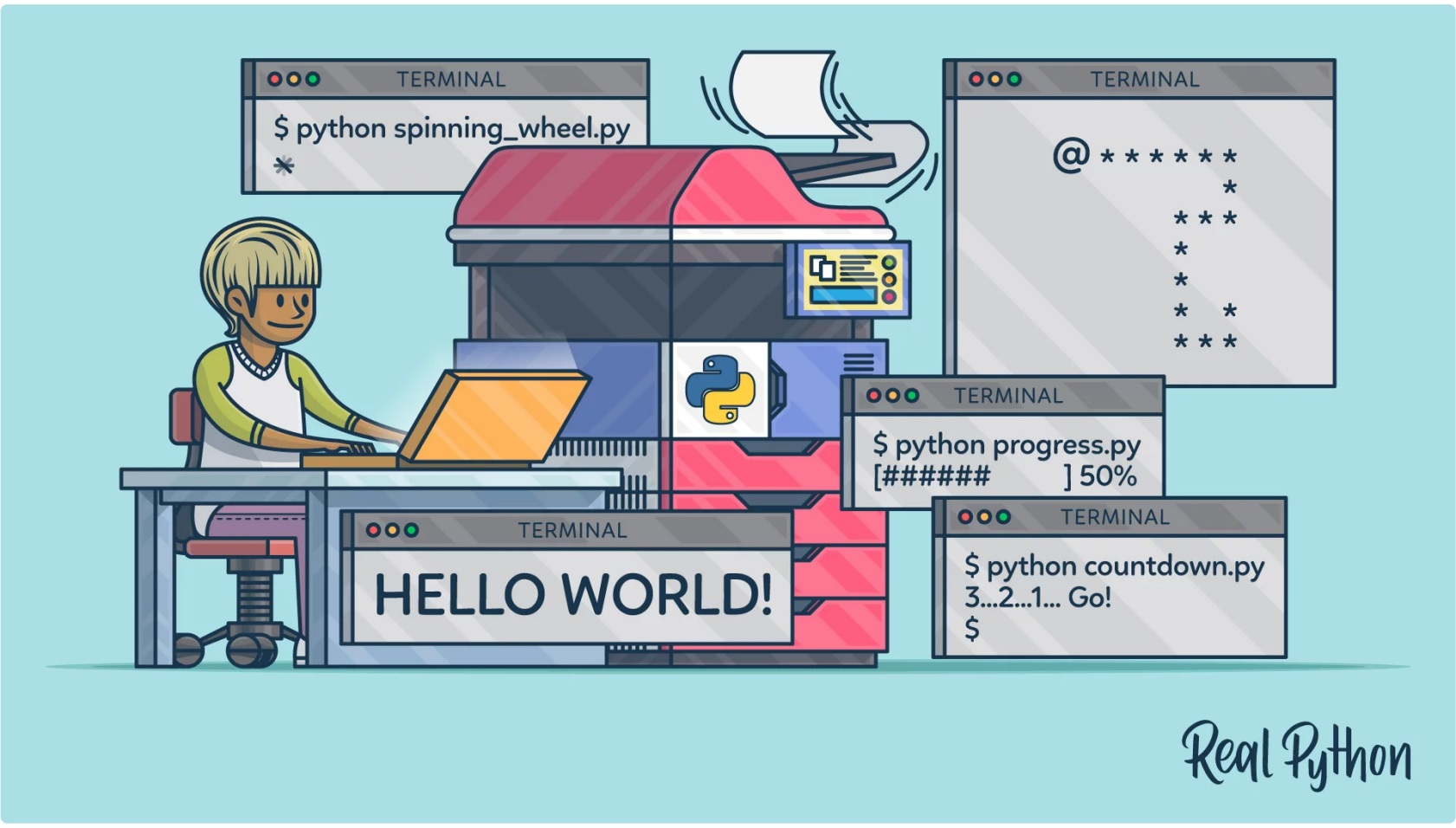


Your Guide to the Python Print Function

by Bartosz Zaczyński Aug 12, 2019 1 Comment basics python

Tweet Share Email

Table of Contents

If you're like most Python users, including me, then you probably started your Python journey by learning about `print()`. It helped you write your very own `hello world` one-liner. You can use it to display formatted messages onto the screen and perhaps find some bugs. But if you think that's all there is to know about Python's `print()` function, then you're missing out on a lot!

Keep reading to take full advantage of this seemingly boring and unappreciated little function. This tutorial will get you up to speed with using Python `print()` effectively. However, prepare for a deep dive as you go through the sections. You may be surprised how much `print()` has to offer!

By the end of this tutorial, you'll know how to:

- Avoid common mistakes with Python's `print()`
- Deal with newlines, character encodings, and buffering
- Write text to files
- Mock `print()` in unit tests
- Build advanced user interfaces in the terminal

If you're a complete beginner, then you'll benefit most from reading the first part of this tutorial, which illustrates the essentials of printing in Python. Otherwise, feel free to skip that part and jump around as you see fit.

> **Note:** `print()` was a major addition to Python 3, in which it replaced the old `print` statement available in Python 2.
>
> There were a number of good reasons for that, as you'll see shortly. Although this tutorial focuses on Python 3, it does show the old way of printing in Python for reference.

Printing in a Nutshell

Let's jump in by looking at a few real-life examples of printing in Python. By the end of this section, you'll know every possible way of calling `print()`. Or, in programmer lingo, you'd say you'll be familiar with the **function signature**.

Calling Print

The simplest example of using Python `print()` requires just a few keystrokes:

Python >>>

```
>>> print()
```

You don't pass any arguments, but you still need to put empty parentheses at the end, which tell Python to actually execute the function rather than just refer to it by name.

This will produce an invisible newline character, which in turn will cause a blank line to appear on your screen. You can call `print()` multiple times like this to add vertical space. It's just as if you were hitting `Enter ↩` on your keyboard in a word processor.

> Newline Character Show/Hide

As you just saw, calling `print()` without arguments results in a **blank line**, which is a line comprised solely of the newline character. Don't confuse this with an **empty line**, which doesn't contain any characters at all, not even the newline!

You can use Python's string literals to visualize these two:

Python

```
'\n'  # Blank line
''    # Empty line
```

The first one is one character long, whereas the second one has no content.

> **Note:** To remove the newline character from a string in Python, use its `.rstrip()` method, like this:
>
> Python >>>
>
> ```
> >>> 'A line of text.\n'.rstrip()
> 'A line of text.'
> ```
>
> This strips any trailing whitespace from the right edge of the string of characters.

In a more common scenario, you'd want to communicate some message to the end user. There are a few ways to achieve this.

First, you may pass a string literal directly to `print()`:

Python >>>

```
>>> print('Please wait while the program is loading...')
```

This will print the message verbatim onto the screen.

> String Literals Show/Hide

Secondly, you could extract that message into its own variable with a meaningful name to enhance readability and promote code reuse:

Python >>>

```
>>> message = 'Please wait while the program is loading...'
>>> print(message)
```

Lastly, you could pass an expression, like string concatenation, to be evaluated before printing the result:

Python >>>

```
>>> import os
>>> print('Hello, ' + os.getlogin() + '! How are you?')
Hello, jdoe! How are you?
```

In fact, there are a dozen ways to format messages in Python. I highly encourage you to take a look at f-strings, introduced in Python 3.6, because they offer the most concise syntax of them all:

Python >>>

```
>>> import os
>>> print(f'Hello, {os.getlogin()}! How are you?')
```

Moreover, f-strings will prevent you from making a common mistake, which is forgetting to type cast concatenated operands. Python is a strongly typed language, which means it won't allow you to do this:

Python >>>

```
>>> 'My age is ' + 42
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    'My age is ' + 42
TypeError: can only concatenate str (not "int") to str
```

That's wrong because adding numbers to strings doesn't make sense. You need to explicitly convert the number to string first, in order to join them together:

Python >>>

```
>>> 'My age is ' + str(42)
'My age is 42'
```

Unless you handle such errors yourself, the Python interpreter will let you know about a problem by showing a traceback.

Note: `str()` is a global built-in function that converts an object into its string representation.

You can call it directly on any object, for example, a number:

Python >>>

```
>>> str(3.14)
'3.14'
```

Built-in data types have a predefined string representation out of the box, but later in this article, you'll find out how to provide one for your custom classes.

As with any function, it doesn't matter whether you pass a literal, a variable, or an expression. Unlike many other functions, however, `print()` will accept anything regardless of its type.

So far, you only looked at the string, but how about other data types? Let's try literals of different built-in types and see what comes out:

Python >>>

```
>>> print(42)                          # <class 'int'>
42
>>> print(3.14)                        # <class 'float'>
3.14
>>> print(1 + 2j)                      # <class 'complex'>
(1+2j)
>>> print(True)                        # <class 'bool'>
True
>>> print([1, 2, 3])                   # <class 'list'>
[1, 2, 3]
>>> print((1, 2, 3))                   # <class 'tuple'>
(1, 2, 3)
>>> print({'red', 'green', 'blue'})    # <class 'set'>
{'red', 'green', 'blue'}
>>> print({'name': 'Alice', 'age': 42})  # <class 'dict'>
{'name': 'Alice', 'age': 42}
>>> print('hello')                      # <class 'str'>
hello
```

Watch out for the `None` constant, though. Despite being used to indicate an absence of a value, it will show up as `'None'` rather than an empty string:

Python >>>

```
>>> print(None)
None
```

How does `print()` know how to work with all these different types? Well, the short answer is that it doesn't. It implicitly calls `str()` behind the scenes to type cast any object into a string. Afterward, it treats strings in a uniform way.

Later in this tutorial, you'll learn how to use this mechanism for printing custom data types such as your classes.

Okay, you're now able to call `print()` with a single argument or without any arguments. You know how to print fixed or formatted messages onto the screen. The next subsection will expand on message formatting a little bit.

Syntax in Python 2 Show/Hide

Separating Multiple Arguments

You saw `print()` called without any arguments to produce a blank line and then called with a single argument to display either a fixed or a formatted message.

However, it turns out that this function can accept any number of **positional arguments**, including zero, one, or more arguments. That's very handy in a common case of message formatting, where you'd want to join a few elements together.

Positional Arguments Show/Hide

Let's have a look at this example:

Python >>>

```
>>> import os
>>> print('My name is', os.getlogin(), 'and I am', 42)
My name is jdoe and I am 42
```

`print()` concatenated all four arguments passed to it, and it inserted a single space between them so that you didn't end up with a squashed message like `'My name isjdoeand I am42'`.

Notice that it also took care of proper type casting by implicitly calling `str()` on each argument before joining them together. If you recall from the previous subsection, a naïve concatenation may easily result in an error due to incompatible types:

Python >>>

```
>>> print('My age is: ' + 42)
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    print('My age is: ' + 42)
TypeError: can only concatenate str (not "int") to str
```

Apart from accepting a variable number of positional arguments, `print()` defines four named or **keyword arguments**, which are optional since they all have default values. You can view their brief documentation by calling `help(print)` from the interactive interpreter.

Let's focus on `sep` just for now. It stands for **separator** and is assigned a single space (`' '`) by default. It determines the value to join elements with.

It has to be either a string or `None`, but the latter has the same effect as the default space:

Python >>>

```
>>> print('hello', 'world', sep=None)
hello world
>>> print('hello', 'world', sep=' ')
hello world
>>> print('hello', 'world')
hello world
```

If you wanted to suppress the separator completely, you'd have to pass an empty string (`''`) instead:

Python >>>

```
>>> print('hello', 'world', sep='')
helloworld
```

You may want `print()` to join its arguments as separate lines. In that case, simply pass the escaped newline character described earlier:

Python >>>

```
>>> print('hello', 'world', sep='\n')
hello
world
```

A more useful example of the `sep` parameter would be printing something like file paths:

Python >>>

```
>>> print('home', 'user', 'documents', sep='/')
home/user/documents
```

Remember that the separator comes between the elements, not around them, so you need to account for that in one way or another:

Python >>>

```
>>> print('/home', 'user', 'documents', sep='/')
/home/user/documents
>>> print('', 'home', 'user', 'documents', sep='/')
/home/user/documents
```

Specifically, you can insert a slash character (`/`) into the first positional argument, or use an empty string as the first argument to enforce the leading slash.

Note: Be careful about joining elements of a list or tuple.

Doing it manually will result in a well-known `TypeError` if at least one of the elements isn't a string:

Python >>>

```
>>> print(' '.join(['jdoe is', 42, 'years old']))
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    print(','.join(['jdoe is', 42, 'years old']))
TypeError: sequence item 1: expected str instance, int found
```

It's safer to just unpack the sequence with the star operator (`*`) and let `print()` handle type casting:

Python >>>

```
>>> print(*['jdoe is', 42, 'years old'])
jdoe is 42 years old
```

Unpacking is effectively the same as calling `print()` with individual elements of the list.

One more interesting example could be exporting data to a comma-separated values (CSV) format:

Python >>>

```
>>> print(1, 'Python Tricks', 'Dan Bader', sep=',')
1,Python Tricks,Dan Bader
```

This wouldn't handle edge cases such as escaping commas correctly, but for simple use cases, it should do. The line above would show up in your terminal window. In order to save it to a file, you'd have to redirect the output. Later in this section, you'll see how to use `print()` to write text to files straight from Python.

Finally, the `sep` parameter isn't constrained to a single character only. You can join elements with strings of any length:

Python >>>

```
>>> print('node', 'child', 'child', sep=' -> ')
node -> child -> child
```

In the upcoming subsections, you'll explore the remaining keyword arguments of the `print()` function.

Syntax in Python 2 Show/Hide

Preventing Line Breaks

Sometimes you don't want to end your message with a trailing newline so that subsequent calls to `print()` will continue on the same line. Classic examples include updating the progress of a long-running operation or prompting the user for input. In the latter case, you want the user to type in the answer on the same line:

Text

```
Are you sure you want to do this? [y/n] y
```

Many programming languages expose functions similar to `print()` through their standard libraries, but they let you decide whether to add a newline or not. For example, in Java and C#, you have two distinct functions, while other languages require you to explicitly append `\n` at the end of a string literal.

Here are a few examples of syntax in such languages:

Language	Example
Perl	`print "hello world\n"`
C	`printf("hello world\n");`
C++	`std::cout << "hello world" << std::endl;`

In contrast, Python's `print()` function always adds `\n` without asking, because that's what you want in most cases. To disable it, you can take advantage of yet another keyword argument, `end`, which dictates what to end the line with.

In terms of semantics, the `end` parameter is almost identical to the `sep` one that you saw earlier:

- It must be a string or `None`.
- It can be arbitrarily long.
- It has a default value of `'\n'`.
- If equal to `None`, it'll have the same effect as the default value.
- If equal to an empty string (`''`), it'll suppress the newline.

Now you understand what's happening under the hood when you're calling `print()` without arguments. Since you don't provide any positional arguments to the function, there's nothing to be joined, and so the default separator isn't used at all. However, the default value of `end` still applies, and a blank line shows up.

Note: You may be wondering why the `end` parameter has a fixed default value rather than whatever makes sense on your operating system.

Well, you don't have to worry about newline representation across different operating systems when printing, because `print()` will handle the conversion automatically. Just remember to always use the `\n` escape sequence in string literals.

This is currently the most portable way of printing a newline character in Python:

Python >>>

```
>>> print('line1\nline2\nline3')
line1
line2
line3
```

If you were to try to forcefully print a Windows-specific newline character on a Linux machine, for example, you'd end up with broken output:

Python >>>

```
>>> print('line1\r\nline2\r\nline3')


line3
```

On the flip side, when you open a file for reading with `open()`, you don't need to care about newline representation either. The function will translate any system-specific newline it encounters into a universal `'\n'`. At the same time, you have control over how the newlines should be treated both on input and output if you really need that.

To disable the newline, you must specify an empty string through the `end` keyword argument:

Python

```
print('Checking file integrity...', end='')
# (...)
print('ok')
```

Even though these are two separate `print()` calls, which can execute a long time apart, you'll eventually see only one line. First, it'll look like this:

Text

```
Checking file integrity...
```

However, after the second call to `print()`, the same line will appear on the screen as:

Text

```
Checking file integrity...ok
```

As with `sep`, you can use `end` to join individual pieces into a big blob of text with a custom separator. Instead of joining multiple arguments, however, it'll append text from each function call to the same line:

Python

```
print('The first sentence', end='. ')
print('The second sentence', end='. ')
print('The last sentence.')
```

These three instructions will output a single line of text:

Text

```
The first sentence. The second sentence. The last sentence.
```

You can mix the two keyword arguments:

Python

```
print('Mercury', 'Venus', 'Earth', sep=', ', end=', ')
print('Mars', 'Jupiter', 'Saturn', sep=', ', end=', ')
print('Uranus', 'Neptune', 'Pluto', sep=', ')
```

Not only do you get a single line of text, but all items are separated with a comma:

Text

```
Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune, Pluto
```

There's nothing to stop you from using the newline character with some extra padding around it:

Python

```
print('Printing in a Nutshell', end='\n * ')
print('Calling Print', end='\n * ')
print('Separating Multiple Arguments', end='\n * ')
print('Preventing Line Breaks')
```

It would print out the following piece of text:

Text

```
Printing in a Nutshell
 * Calling Print
 * Separating Multiple Arguments
 * Preventing Line Breaks
```

As you can see, the `end` keyword argument will accept arbitrary strings.

Note: Looping over lines in a text file preserves their own newline characters, which combined with the `print()` function's default behavior will result in a redundant newline character:

Python >>>

```
>>> with open('file.txt') as file_object:
...     for line in file_object:
...         print(line)
...
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod

tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,

quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
```

There are two newlines after each line of text. You want to strip one of the them, as shown earlier in this article, before printing the line:

Python

```
print(line.rstrip())
```

Alternatively, you can keep the newline in the content but suppress the one appended by `print()` automatically. You'd use the `end` keyword argument to do that:

Python >>>

```
>>> with open('file.txt') as file_object:
...     for line in file_object:
...         print(line, end='')
...
Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod
tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam,
quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
```

By ending a line with an empty string, you effectively disable one of the newlines.

You're getting more acquainted with printing in Python, but there's still a lot of useful information ahead. In the upcoming subsection, you'll learn how to intercept and redirect the `print()` function's output.

Syntax in Python 2 Show/Hide

Printing to a File

Believe it or not, `print()` doesn't know how to turn messages into text on your screen, and frankly it doesn't need to. That's a job for lower-level layers of code, which understand bytes and know how to push them around.

`print()` is an abstraction over these layers, providing a convenient interface that merely delegates the actual printing to a stream or **file-like object**. A stream can be any file on your disk, a network socket, or perhaps an in-memory buffer.

In addition to this, there are three standard streams provided by the operating system:

1. **`stdin`:** standard input
2. **`stdout`:** standard output
3. **`stderr`:** standard error

Standard Streams Show/Hide

In Python, you can access all standard streams through the built-in `sys` module:

Python >>>

```
>>> import sys
>>> sys.stdin
<_io.TextIOWrapper name='<stdin>' mode='r' encoding='UTF-8'>
>>> sys.stdin.fileno()
0
>>> sys.stdout
<_io.TextIOWrapper name='<stdout>' mode='w' encoding='UTF-8'>
>>> sys.stdout.fileno()
1
>>> sys.stderr
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'>
>>> sys.stderr.fileno()
2
```

As you can see, these predefined values resemble file-like objects with `mode` and `encoding` attributes as well as `.read()` and `.write()` methods among many others.

By default, `print()` is bound to `sys.stdout` through its `file` argument, but you can change that. Use that keyword argument to indicate a file that was open in write or append mode, so that messages go straight to it:

Python

```
with open('file.txt', mode='w') as file_object:
    print('hello world', file=file_object)
```

This will make your code immune to stream redirection at the operating system level, which might or might not be desired.

For more information on working with files in Python, you can check out Reading and Writing Files in Python (Guide).

Note: Don't try using `print()` for writing binary data as it's only well suited for text.

Just call the binary file's `.write()` directly:

Python

```
with open('file.dat', 'wb') as file_object:
    file_object.write(bytes(4))
    file_object.write(b'\xff')
```

If you wanted to write raw bytes on the standard output, then this will fail too because `sys.stdout` is a character stream:

Python >>>

```
>>> import sys
>>> sys.stdout.write(bytes(4))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: write() argument must be str, not bytes
```

You must dig deeper to get a handle of the underlying byte stream instead:

Python >>>

```
>>> import sys
>>> num_bytes_written = sys.stdout.buffer.write(b'\x41\x0a')
A
```

This prints an uppercase letter `A` and a newline character, which correspond to decimal values of 65 and 10 in ASCII. However, they're encoded using hexadecimal notation in the bytes literal.

Note that `print()` has no control over character encoding. It's the stream's responsibility to encode received Unicode strings into bytes correctly. In most cases, you won't set the encoding yourself, because the default UTF-8 is what you want. If you really need to, perhaps for legacy systems, you can use the `encoding` argument of `open()`:

Python

```
with open('file.txt', mode='w', encoding='iso-8859-1') as file_object:
    print('über naïve café', file=file_object)
```

Instead of a real file existing somewhere in your file system, you can provide a fake one, which would reside in your computer's memory. You'll use this technique later for mocking `print()` in unit tests:

Python >>>

```
>>> import io
>>> fake_file = io.StringIO()
>>> print('hello world', file=fake_file)
>>> fake_file.getvalue()
'hello world\n'
```

If you got to this point, then you're left with only one keyword argument in `print()`, which you'll see in the next subsection. It's probably the least used of them all. Nevertheless, there are times when it's absolutely necessary.

Syntax in Python 2 Show/Hide

Buffering Print Calls

In the previous subsection, you learned that `print()` delegates printing to a file-like object such as `sys.stdout`. Some streams, however, buffer certain I/O operations to enhance performance, which can get in the way. Let's take a look at an example.

Imagine you were writing a countdown timer, which should append the remaining time to the same line every second:

Text

```
3...2...1...Go!
```

Your first attempt may look something like this:

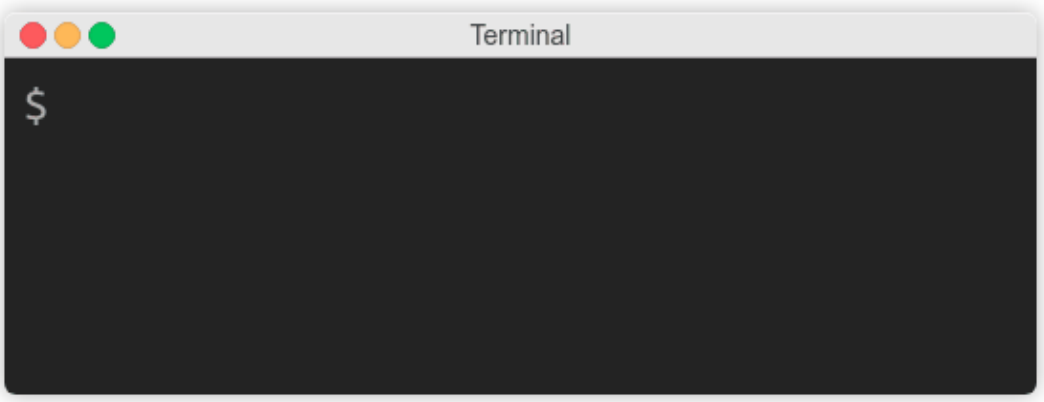
Python

```
import time

num_seconds = 3
for countdown in reversed(range(num_seconds + 1)):
    if countdown > 0:
        print(countdown, end='...')
        time.sleep(1)
    else:
        print('Go!')
```

As long as the `countdown` variable is greater than zero, the code keeps appending text without a trailing newline and then goes to sleep for one second. Finally, when the countdown is finished, it prints `Go!` and terminates the line.

Unexpectedly, instead of counting down every second, the program idles wastefully for three seconds, and then suddenly prints the entire line at once:



That's because the operating system buffers subsequent writes to the standard output in this case. You need to know that there are three kinds of streams with respect to buffering:

1. Unbuffered
2. Line-buffered
3. Block-buffered

Unbuffered is self-explanatory, that is, no buffering is taking place, and all writes have immediate effect. A **line-buffered** stream waits before firing any I/O calls until a line break appears somewhere in the buffer, whereas a **block-buffered** one simply allows the buffer to fill up to a certain size regardless of its content. Standard output is both **line-buffered** and **block-buffered**, depending on which event comes first.

Buffering helps to reduce the number of expensive I/O calls. Think about sending messages over a high-latency network, for example. When you connect to a remote server to execute commands over the SSH protocol, each of your keystrokes may actually produce an individual data packet, which is orders of magnitude bigger than its payload. What an overhead! It would make sense to wait until at least a few characters are typed and then send them together. That's where buffering steps in.

On the other hand, buffering can sometimes have undesired effects as you just saw with the countdown example. To fix it, you can simply tell `print()` to forcefully flush the stream without waiting for a newline character in the buffer using its `flush` flag:

Python

```
print(countdown, end='...', flush=True)
```

That's all. Your countdown should work as expected now, but don't take my word for it. Go ahead and test it to see the difference.

Congratulations! At this point, you've seen examples of calling `print()` that cover all of its parameters. You know their purpose and when to use them. Understanding the signature is only the beginning, however. In the upcoming sections, you'll see why.

Syntax in Python 2 Show/Hide

Printing Custom Data Types

Up until now, you only dealt with built-in data types such as strings and numbers, but you'll often want to print your own abstract data types. Let's have a look at different ways of defining them.

For simple objects without any logic, whose purpose is to carry data, you'll typically take advantage of `namedtuple`, which is available in the standard library. Named tuples have a neat textual representation out of the box:

Python >>>

```
>>> from collections import namedtuple
>>> Person = namedtuple('Person', 'name age')
>>> jdoe = Person('John Doe', 42)
>>> print(jdoe)
Person(name='John Doe', age=42)
```

That's great as long as holding data is enough, but in order to add behaviors to the `Person` type, you'll eventually need to define a class. Take a look at this example:

Python

```
class Person:
    def __init__(self, name, age):
        self.name, self.age = name, age
```

If you now create an instance of the `Person` class and try to print it, you'll get this bizarre output, which is quite different from the equivalent `namedtuple`:

Python >>>

```
>>> jdoe = Person('John Doe', 42)
>>> print(jdoe)
<__main__.Person object at 0x7fcac3fed1d0>
```

It's the default representation of objects, which comprises their address in memory, the corresponding class name and a module in which they were defined. You'll fix that in a bit, but just for the record, as a quick workaround you could combine `namedtuple` and a custom class through inheritance:

Python

```
from collections import namedtuple

class Person(namedtuple('Person', 'name age')):
    pass
```

Your `Person` class has just become a specialized kind of `namedtuple` with two attributes, which you can customize.

Note: In Python 3, the `pass` statement can be replaced with the ellipsis (`...`) literal to indicate a placeholder:

Python

```
def delta(a, b, c):
    ...
```

This prevents the interpreter from raising `IndentationError` due to missing indented block of code.

That's better than a plain `namedtuple`, because not only do you get printing right for free, but you can also add custom methods and properties to the class. However, it solves one problem while introducing another. Remember that tuples, including named tuples, are immutable in Python, so they can't change their values once created.

It's true that designing immutable data types is desirable, but in many cases, you'll want them to allow for change, so you're back with regular classes again.

Note: Following other languages and frameworks, Python 3.7 introduced data classes, which you can think of as mutable tuples. This way, you get the best of both worlds:

Python >>>

```
>>> from dataclasses import dataclass
>>> @dataclass
... class Person:
...     name: str
...     age: int
...
...     def celebrate_birthday(self):
...         self.age += 1
...
>>> jdoe = Person('John Doe', 42)
>>> jdoe.celebrate_birthday()
>>> print(jdoe)
Person(name='John Doe', age=43)
```

The syntax for variable annotations, which is required to specify class fields with their corresponding types, was defined in Python 3.6.

From earlier subsections, you already know that `print()` implicitly calls the built-in `str()` function to convert its positional arguments into strings. Indeed, calling `str()` manually against an instance of the regular `Person` class yields the same result as printing it:

Python >>>

```
>>> jdoe = Person('John Doe', 42)
>>> str(jdoe)
'<__main__.Person object at 0x7fcac3fed1d0>'
```

`str()`, in turn, looks for one of two **magic methods** within the class body, which you typically implement. If it doesn't find one, then it falls back to the ugly default representation. Those magic methods are, in order of search:

1. `def __str__(self)`
2. `def __repr__(self)`

The first one is recommended to return a short, human-readable text, which includes information from the most relevant attributes. After all, you don't want to expose sensitive data, such as user passwords, when printing objects.

However, the other one should provide complete information about an object, to allow for restoring its state from a string. Ideally, it should return valid Python code, so that you can pass it directly to `eval()`:

Python >>>

```
>>> repr(jdoe)
"Person(name='John Doe', age=42)"
>>> type(eval(repr(jdoe)))
<class '__main__.Person'>
```

Notice the use of another built-in function, `repr()`, which always tries to call `.__repr__()` in an object, but falls back to the default representation if it doesn't find that method.

Note: Even though `print()` itself uses `str()` for type casting, some compound data types delegate that call to `repr()` on their members. This happens to lists and tuples, for example.

Consider this class with both magic methods, which return alternative string representations of the same object:

Python

```
class User:
    def __init__(self, login, password):
        self.login = login
        self.password = password

    def __str__(self):
        return self.login

    def __repr__(self):
        return f"User('{self.login}', '{self.password}')"
```

If you print a single object of the `User` class, then you won't see the password, because `print(user)` will call `str(user)`, which eventually will invoke `user.__str__()`:

Python >>>

```
>>> user = User('jdoe', 's3cret')
>>> print(user)
jdoe
```

However, if you put the same `user` variable inside a list by wrapping it in square brackets, then the password will become clearly visible:

Python >>>

```
>>> print([user])
[User('jdoe', 's3cret')]
```

That's because sequences, such as lists and tuples, implement their `.__str__()` method so that all of their elements are first converted with `repr()`.

Python gives you a lot of freedom when it comes to defining your own data types if none of the built-in ones meet your needs. Some of them, such as named tuples and data classes, offer string representations that look good without requiring any work on your part. Still, for the most flexibility, you'll have to define a class and override its magic methods described above.

Syntax in Python 2 Show/Hide

Understanding Python Print

You know **how** to use `print()` quite well at this point, but knowing **what** it is will allow you to use it even more effectively and consciously. After reading this section, you'll understand how printing in Python has improved over the years.

Print Is a Function in Python 3

You've seen that `print()` is a function in Python 3. More specifically, it's a built-in function, which means that you don't need to import it from anywhere:

Python >>>

```
>>> print
<built-in function print>
```

It's always available in the global namespace so that you can call it directly, but you can also access it through a module from the standard library:

Python >>>

```
>>> import builtins
>>> builtins.print
<built-in function print>
```

This way, you can avoid name collisions with custom functions. Let's say you wanted to **redefine** `print()` so that it doesn't append a trailing newline. At the same time, you wanted to rename the original function to something like `println()`:

Python >>>

```
>>> import builtins
>>> println = builtins.print
>>> def print(*args, **kwargs):
...     builtins.print(*args, **kwargs, end='')
...
>>> println('hello')
hello
>>> print('hello\n')
hello
```

Now you have two separate printing functions just like in the Java programming language. You'll define custom `print()` functions in the mocking section later as well. Also, note that you wouldn't be able to overwrite `print()` in the first place if it wasn't a function.

On the other hand, `print()` isn't a function in the mathematical sense, because it doesn't return any meaningful value other than the implicit `None`:

Python >>>

```
>>> value = print('hello world')
hello world
>>> print(value)
None
```

Such functions are, in fact, procedures or subroutines that you call to achieve some kind of side-effect, which ultimately is a change of a global state. In the case of `print()`, that side-effect is showing a message on the standard output or writing to a file.

Because `print()` is a function, it has a well-defined signature with known attributes. You can quickly find its **documentation** using the editor of your choice, without having to remember some weird syntax for performing a certain task.

Besides, functions are easier to **extend**. Adding a new feature to a function is as easy as adding another keyword argument, whereas changing the language to support that new feature is much more cumbersome. Think of stream redirection or buffer flushing, for example.

Another benefit of `print()` being a function is **composability**. Functions are so-called first-class objects or first-class citizens in Python, which is a fancy way of saying they're values just like strings or numbers. This way, you can assign a function to a variable, pass it to another function, or even return one from another. `print()` isn't different in this regard. For instance, you can take advantage of it for dependency injection:

Python

```
def download(url, log=print):
    log(f'Downloading {url}')
    # ...

def custom_print(*args):
    pass  # Do not print anything

download('/js/app.js', log=custom_print)
```

Here, the `log` parameter lets you inject a callback function, which defaults to `print()` but can be any callable. In this example, printing is completely disabled by substituting `print()` with a dummy function that does nothing.

> **Note:** A **dependency** is any piece of code required by another bit of code.
>
> **Dependency injection** is a technique used in code design to make it more testable, reusable, and open for extension. You can achieve it by referring to dependencies indirectly through abstract interfaces and by providing them in a **push** rather than **pull** fashion.
>
> There's a funny explanation of dependency injection circulating on the Internet:
>
> > Dependency injection for five-year-olds
> >
> > When you go and get things out of the refrigerator for yourself, you can cause problems. You might leave the door open, you might get something Mommy or Daddy doesn't want you to have. You might even be looking for something we don't even have or which has expired.
> >
> > What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat.
> >
> > — *John Munsch, 28 October 2009.* (Source)

Composition allows you to combine a few functions into a new one of the same kind. Let's see this in action by specifying a custom `error()` function that prints to the standard error stream and prefixes all messages with a given log level:

Python >>>

```
>>> from functools import partial
>>> import sys
>>> redirect = lambda function, stream: partial(function, file=stream)
>>> prefix = lambda function, prefix: partial(function, prefix)
>>> error = prefix(redirect(print, sys.stderr), '[ERROR]')
>>> error('Something went wrong')
[ERROR] Something went wrong
```

This custom function uses **partial functions** to achieve the desired effect. It's an advanced concept borrowed from the functional programming paradigm, so you don't need to go too deep into that topic for now. However, if you're interested in this topic, I recommend taking a look at the `functools` module.

Unlike statements, functions are values. That means you can mix them with **expressions**, in particular, **lambda expressions**. Instead of defining a full-blown function to replace `print()` with, you can make an anonymous lambda expression that calls it:

Python >>>

```
>>> download('/js/app.js', lambda msg: print('[INFO]', msg))
[INFO] Downloading /js/app.js
```

However, because a lambda expression is defined in place, there's no way of referring to it elsewhere in the code.

> **Note:** In Python, you can't put statements, such as assignments, conditional statements, loops, and so on, in an **anonymous lambda function**. It has to be a single expression!

Another kind of expression is a ternary conditional expression:

Python >>>

```
>>> user = 'jdoe'
>>> print('Hi!') if user is None else print(f'Hi, {user}.')
Hi, jdoe.
```

Python has both conditional statements and conditional expressions. The latter is evaluated to a single value that can be assigned to a variable or passed to a function. In the example above, you're interested in the side-effect rather than the value, which evaluates to `None`, so you simply ignore it.

As you can see, functions allow for an elegant and extensible solution, which is consistent with the rest of the language. In the next subsection, you'll discover how not having `print()` as a function caused a lot of headaches.

Print Was a Statement in Python 2

A **statement** is an instruction that may evoke a side-effect when executed but never evaluates to a value. In other words, you wouldn't be able to print a statement or assign it to a variable like this:

Python

```
result = print 'hello world'
```

That's a syntax error in Python 2.

Here are a few more examples of statements in Python:

- **assignment:** `=`
- **conditional:** `if`
- **loop:** `while`
- **assertion**: `assert`

Note: Python 3.8 brings a controversial **walrus operator** (`:=`), which is an assignment expression. With it, you can evaluate an expression and assign the result to a variable at the same time, even within another expression!

Take a look at this example, which calls an expensive function once and then reuses the result for further computation:

Python

```
# Python 3.8+
values = [y := f(x), y**2, y**3]
```

This is useful for simplifying the code without losing its efficiency. Typically, performant code tends to be more verbose:

Python

```
y = f(x)
values = [y, y**2, y**3]
```

The controversy behind this new piece of syntax caused a lot of argument. An abundance of negative comments and heated debates eventually led Guido van Rossum to step down from the **Benevolent Dictator For Life** or BDFL position.

Statements are usually comprised of reserved keywords such as `if`, `for`, or `print` that have fixed meaning in the language. You can't use them to name your variables or other symbols. That's why redefining or mocking the `print` statement isn't possible in Python 2. You're stuck with what you get.

Furthermore, you can't print from anonymous functions, because statements aren't accepted in lambda expressions:

Python >>>

```
>>> lambda: print 'hello world'
  File "<stdin>", line 1
    lambda: print 'hello world'
                  ^
SyntaxError: invalid syntax
```

The syntax of the `print` statement is ambiguous. Sometimes you can add parentheses around the message, and they're completely optional:

Python >>>

```
>>> print 'Please wait...'
Please wait...
>>> print('Please wait...')
Please wait...
```

At other times they change how the message is printed:

Python >>>

```
>>> print 'My name is', 'John'
My name is John
>>> print('My name is', 'John')
('My name is', 'John')
```

String concatenation can raise a `TypeError` due to incompatible types, which you have to handle manually, for example:

Python >>>

```
>>> values = ['jdoe', 'is', 42, 'years old']
>>> print ' '.join(map(str, values))
jdoe is 42 years old
```

Compare this with similar code in Python 3, which leverages sequence unpacking:

Python >>>

```
>>> values = ['jdoe', 'is', 42, 'years old']
>>> print(*values)  # Python 3
jdoe is 42 years old
```

There aren't any keyword arguments for common tasks such as flushing the buffer or stream redirection. You need to remember the quirky syntax instead. Even the built-in `help()` function isn't that helpful with regards to the `print` statement:

Python >>>

```
>>> help(print)
  File "<stdin>", line 1
    help(print)
               ^
SyntaxError: invalid syntax
```

Trailing newline removal doesn't work quite right, because it adds an unwanted space. You can't compose multiple `print` statements together, and, on top of that, you have to be extra diligent about character encoding.

The list of problems goes on and on. If you're curious, you can jump back to the previous section and look for more detailed explanations of the syntax in Python 2.

However, you can mitigate some of those problems with a much simpler approach. It turns out the `print()` function was backported to ease the migration to Python 3. You can import it from a special `__future__` module, which exposes a selection of language features released in later Python versions.

> **Note:** You may import future functions as well as baked-in language constructs such as the `with` statement.

To find out exactly what features are available to you, inspect the module:

Python >>>

```
>>> import __future__
>>> __future__.all_feature_names
['nested_scopes',
 'generators',
 'division',
 'absolute_import',
 'with_statement',
 'print_function',
 'unicode_literals']
```

You could also call `dir(__future__)`, but that would show a lot of uninteresting internal details of the module.

To enable the `print()` function in Python 2, you need to add this import statement at the beginning of your source code:

Python

```
from __future__ import print_function
```

From now on the `print` statement is no longer available, but you have the `print()` function at your disposal. Note that it isn't the same function like the one in Python 3, because it's missing the `flush` keyword argument, but the rest of the arguments are the same.

Other than that, it doesn't spare you from managing character encodings properly.

Here's an example of calling the `print()` function in Python 2:

Python >>>

```
>>> from __future__ import print_function
>>> import sys
>>> print('I am a function in Python', sys.version_info.major)
I am a function in Python 2
```

You now have an idea of how printing in Python evolved and, most importantly, understand why these backward-incompatible changes were necessary. Knowing this will surely help you become a better Python programmer.

Printing With Style

If you thought that printing was only about lighting pixels up on the screen, then technically you'd be right. However, there are ways to make it look cool. In this section, you'll find out how to format complex data structures, add colors and other decorations, build interfaces, use animation, and even play sounds with text!

Pretty-Printing Nested Data Structures

Computer languages allow you to represent data as well as executable code in a structured way. Unlike Python, however, most languages give you a lot of freedom in using whitespace and formatting. This can be useful, for example in compression, but it sometimes leads to less readable code.

Pretty-printing is about making a piece of data or code look more appealing to the human eye so that it can be understood more easily. This is done by indenting certain lines, inserting newlines, reordering elements, and so forth.

Python comes with the `pprint` module in its standard library, which will help you in pretty-printing large data structures that don't fit on a single line. Because it prints in a more human-friendly way, many popular REPL tools, including JupyterLab and IPython, use it by default in place of the regular `print()` function.

Note: To toggle pretty printing in IPython, issue the following command:

Python >>>

```
In [1]: %pprint
Pretty printing has been turned OFF
In [2]: %pprint
Pretty printing has been turned ON
```

This is an example of **Magic** in IPython. There are a lot of built-in commands that start with a percent sign (`%`), but you can find more on PyPI, or even create your own.

If you don't care about not having access to the original `print()` function, then you can replace it with `pprint()` in your code using import renaming:

Python >>>

```
>>> from pprint import pprint as print
>>> print
<function pprint at 0x7f7a775a3510>
```

Personally, I like to have both functions at my fingertips, so I'd rather use something like `pp` as a short alias:

Python

```
from pprint import pprint as pp
```

At first glance, there's hardly any difference between the two functions, and in some cases there's virtually none:

Python >>>

```
>>> print(42)
42
>>> pp(42)
42
>>> print('hello')
hello
>>> pp('hello')
'hello'  # Did you spot the difference?
```

That's because `pprint()` calls `repr()` instead of the usual `str()` for type casting, so that you may evaluate its output as Python code if you want to. The differences become apparent as you start feeding it more complex data structures:

Python >>>

```
>>> data = {'powers': [x**10 for x in range(10)]}
>>> pp(data)
{'powers': [0,
            1,
            1024,
            59049,
            1048576,
            9765625,
            60466176,
            282475249,
            1073741824,
            3486784401]}
```

The function applies reasonable formatting to improve readability, but you can customize it even further with a couple of parameters. For example, you may limit a deeply nested hierarchy by showing an ellipsis below a given level:

Python >>>

```
>>> cities = {'USA': {'Texas': {'Dallas': ['Irving']}}}
>>> pp(cities, depth=3)
{'USA': {'Texas': {'Dallas': [...]}}}
```

The ordinary `print()` also uses ellipses but for displaying recursive data structures, which form a cycle, to avoid stack overflow error:

Python >>>

```
>>> items = [1, 2, 3]
>>> items.append(items)
>>> print(items)
[1, 2, 3, [...]]
```

However, `pprint()` is more explicit about it by including the unique identity of a self-referencing object:

Python >>>

```
>>> pp(items)
[1, 2, 3, <Recursion on list with id=140635757287688>]
>>> id(items)
140635757287688
```

The last element in the list is the same object as the entire list.

Note: Recursive or very large data sets can be dealt with using the `reprlib` module as well:

Python >>>

```
>>> import reprlib
>>> reprlib.repr([x**10 for x in range(10)])
'[0, 1, 1024, 59049, 1048576, 9765625, ...]'
```

This module supports most of the built-in types and is used by the Python debugger.

`pprint()` automatically sorts dictionary keys for you before printing, which allows for consistent comparison. When you're comparing strings, you often don't care about a particular order of serialized attributes. Anyways, it's always best to compare actual dictionaries before serialization.

Dictionaries often represent JSON data, which is widely used on the Internet. To correctly serialize a dictionary into a valid JSON-formatted string, you can take advantage of the `json` module. It too has pretty-printing capabilities:

Python >>>

```
>>> import json
>>> data = {'username': 'jdoe', 'password': 's3cret'}
>>> ugly = json.dumps(data)
>>> pretty = json.dumps(data, indent=4, sort_keys=True)
>>> print(ugly)
{"username": "jdoe", "password": "s3cret"}
>>> print(pretty)
{
    "password": "s3cret",
    "username": "jdoe"
}
```

Notice, however, that you need to handle printing yourself, because it's not something you'd typically want to do. Similarly, the `pprint` module has an additional `pformat()` function that returns a string, in case you had to do something other than printing it.

Surprisingly, the signature of `pprint()` is nothing like the `print()` function's one. You can't even pass more than one positional argument, which shows how much it focuses on printing data structures.

Adding Colors With ANSI Escape Sequences

As personal computers got more sophisticated, they had better graphics and could display more colors. However, different vendors had their own idea about the API design for controlling it. That changed a few decades ago when people at the American National Standards Institute decided to unify it by defining ANSI escape codes.

Most of today's terminal emulators support this standard to some degree. Until recently, the Windows operating system was a notable exception. Therefore, if you want the best portability, use the `colorama` library in Python. It translates ANSI codes to their appropriate counterparts in Windows while keeping them intact in other operating systems.

To check if your terminal understands a subset of the ANSI escape sequences, for example, related to colors, you can try using the following command:

Shell

```
$ tput colors
```

My default terminal on Linux says it can display 256 distinct colors, while xterm gives me only 8. The command would return a negative number if colors were unsupported.

ANSI escape sequences are like a markup language for the terminal. In HTML you work with tags, such as `<b>` or `<i>`, to change how elements look in the document. These tags are mixed with your content, but they're not visible themselves. Similarly, escape codes won't show up in the terminal as long as it recognizes them. Otherwise, they'll appear in the literal form as if you were viewing the source of a website.

As its name implies, a sequence must begin with the non-printable `Esc` character, whose ASCII value is 27, sometimes denoted as `0x1b` in hexadecimal or `033` in octal. You may use Python number literals to quickly verify it's indeed the same number:

Python >>>

```
>>> 27 == 0x1b == 0o33
True
```

Additionally, you can obtain it with the `\e` escape sequence in the shell:

Shell

```
$ echo -e "\e"
```

The most common ANSI escape sequences take the following form:

Element	Description	Example
`Esc`	non-printable escape character	`\033`
`[`	opening square bracket	`[`
numeric code	one or more numbers separated with `;`	`0`
character code	uppercase or lowercase letter	`m`

The **numeric code** can be one or more numbers separated with a semicolon, while the **character code** is just one letter. Their specific meaning is defined by the ANSI standard. For example, to reset all formatting, you would type one of the following commands, which use the code zero and the letter `m`:

Shell

```
$ echo -e "\e[0m"
$ echo -e "\x1b[0m"
$ echo -e "\033[0m"
```

At the other end of the spectrum, you have compound code values. To set foreground and background with RGB channels, given that your terminal supports 24-bit depth, you could provide multiple numbers:

Shell

```
$ echo -e "\e[38;2;0;0;0m\e[48;2;255;255;255mBlack on white\e[0m"
```

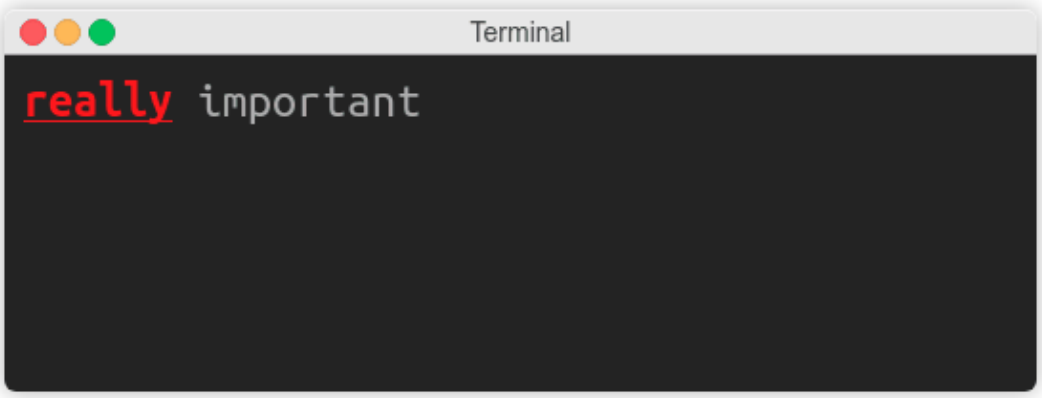
It's not just text color that you can set with the ANSI escape codes. You can, for example, clear and scroll the terminal window, change its background, move the cursor around, make the text blink or decorate it with an underline.

In Python, you'd probably write a helper function to allow for wrapping arbitrary codes into a sequence:

Python >>>

```
>>> def esc(code):
...     return f'\033[{code}m'
...
>>> print(esc('31;1;4') + 'really' + esc(0) + ' important')
```

This would make the word `really` appear in red, bold, and underlined font:



However, there are higher-level abstractions over ANSI escape codes, such as the mentioned `colorama` library, as well as tools for building user interfaces in the console.

Building Console User Interfaces

While playing with ANSI escape codes is undeniably a ton of fun, in the real world you'd rather have more abstract building blocks to put together a user interface. There are a few libraries that provide such a high level of control over the terminal, but `curses` seems to be the most popular choice.

Note: To use the `curses` library in Windows, you need to install a third-party package:

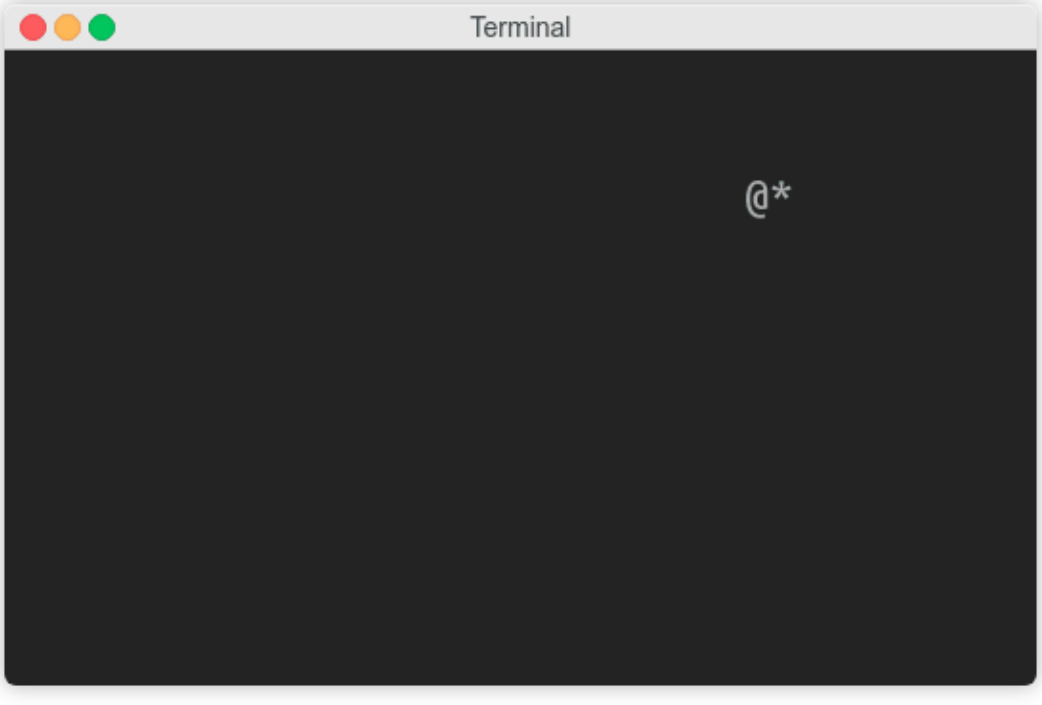
Shell

```
C:\> pip install windows-curses
```

That's because `curses` isn't available in the standard library of the Python distribution for Windows.

Primarily, it allows you to think in terms of independent graphical widgets instead of a blob of text. Besides, you get a lot of freedom in expressing your inner artist, because it's really like painting a blank canvas. The library hides the complexities of having to deal with different terminals. Other than that, it has great support for keyboard events, which might be useful for writing video games.

How about making a retro snake game? Let's create a Python snake simulator:



First, you need to import the `curses` module. Since it modifies the state of a running terminal, it's important to handle errors and gracefully restore the previous state. You can do this manually, but the library comes with a convenient wrapper for your main function:

Python

```
import curses

def main(screen):
    pass

if __name__ == '__main__':
    curses.wrapper(main)
```

Note, the function must accept a reference to the screen object, also known as `stdscr`, that you'll use later for additional setup.

If you run this program now, you won't see any effects, because it terminates immediately. However, you can add a small delay to have a sneak peek:

Python

```
import time, curses

def main(screen):
    time.sleep(1)

if __name__ == '__main__':
    curses.wrapper(main)
```

This time the screen went completely blank for a second, but the cursor was still blinking. To hide it, just call one of the configuration functions defined in the module:

Python

```
import time, curses

def main(screen):
    curses.curs_set(0)  # Hide the cursor
    time.sleep(1)

if __name__ == '__main__':
    curses.wrapper(main)
```

Let's define the snake as a list of points in screen coordinates:

Python

```
snake = [(0, i) for i in reversed(range(20))]
```

The head of the snake is always the first element in the list, whereas the tail is the last one. The initial shape of the snake is horizontal, starting from the top-left corner of the screen and facing to the right. While its y-coordinate stays at zero, its x-coordinate decreases from head to tail.

To draw the snake, you'll start with the head and then follow with the remaining segments. Each segment carries `(y, x)` coordinates, so you can unpack them:

Python

```
# Draw the snake
screen.addstr(*snake[0], '@')
for segment in snake[1:]:
    screen.addstr(*segment, '*')
```

Again, if you run this code now, it won't display anything, because you must explicitly refresh the screen afterward:

Python

```
import time, curses

def main(screen):
    curses.curs_set(0)  # Hide the cursor

    snake = [(0, i) for i in reversed(range(20))]

    # Draw the snake
    screen.addstr(*snake[0], '@')
    for segment in snake[1:]:
        screen.addstr(*segment, '*')

    screen.refresh()
    time.sleep(1)

if __name__ == '__main__':
    curses.wrapper(main)
```

You want to move the snake in one of four directions, which can be defined as vectors. Eventually, the direction will change in response to an arrow keystroke, so you may hook it up to the library's key codes:

Python

```
directions = {
    curses.KEY_UP: (-1, 0),
    curses.KEY_DOWN: (1, 0),
    curses.KEY_LEFT: (0, -1),
    curses.KEY_RIGHT: (0, 1),
}

direction = directions[curses.KEY_RIGHT]
```

How does a snake move? It turns out that only its head really moves to a new location, while all other segments shift towards it. In each step, almost all segments remain the same, except for the head and the tail. Assuming the snake isn't growing, you can remove the tail and insert a new head at the beginning of the list:

Python

```
# Move the snake
snake.pop()
snake.insert(0, tuple(map(sum, zip(snake[0], direction))))
```

To get the new coordinates of the head, you need to add the direction vector to it. However, adding tuples in Python results in a bigger tuple instead of the algebraic sum of the corresponding vector components. One way to fix this is by using the built-in `zip()`, `sum()`, and `map()` functions.

The direction will change on a keystroke, so you need to call `.getch()` to obtain the pressed key code. However, if the pressed key doesn't correspond to the arrow keys defined earlier as dictionary keys, the direction won't change:

Python

```
# Change direction on arrow keystroke
direction = directions.get(screen.getch(), direction)
```

By default, however, `.getch()` is a blocking call that would prevent the snake from moving unless there was a keystroke. Therefore, you need to make the call non-blocking by adding yet another configuration:

Python

```
def main(screen):
    curses.curs_set(0)    # Hide the cursor
    screen.nodelay(True)  # Don't block I/O calls
```

You're almost done, but there's just one last thing left. If you now loop this code, the snake will appear to be growing instead of moving. That's because you have to erase the screen explicitly before each iteration.

Finally, this is all you need to play the snake game in Python:

Python

```
import time, curses

def main(screen):
    curses.curs_set(0)    # Hide the cursor
    screen.nodelay(True)  # Don't block I/O calls

    directions = {
        curses.KEY_UP: (-1, 0),
        curses.KEY_DOWN: (1, 0),
        curses.KEY_LEFT: (0, -1),
        curses.KEY_RIGHT: (0, 1),
    }

    direction = directions[curses.KEY_RIGHT]
    snake = [(0, i) for i in reversed(range(20))]

    while True:
        screen.erase()

        # Draw the snake
        screen.addstr(*snake[0], '@')
        for segment in snake[1:]:
            screen.addstr(*segment, '*')

        # Move the snake
        snake.pop()
        snake.insert(0, tuple(map(sum, zip(snake[0], direction))))

        # Change direction on arrow keystroke
        direction = directions.get(screen.getch(), direction)

        screen.refresh()
        time.sleep(0.1)

if __name__ == '__main__':
    curses.wrapper(main)
```

This is merely scratching the surface of the possibilities that the `curses` module opens up. You may use it for game development like this or more business-oriented applications.

Living It Up With Cool Animations

Not only can animations make the user interface more appealing to the eye, but they also improve the overall user experience. When you provide early feedback to the user, for example, they'll know if your program's still working or if it's time to kill it.

To animate text in the terminal, you have to be able to freely move the cursor around. You can do this with one of the tools mentioned previously, that is ANSI escape codes or the `curses` library. However, I'd like to show you an even simpler way.

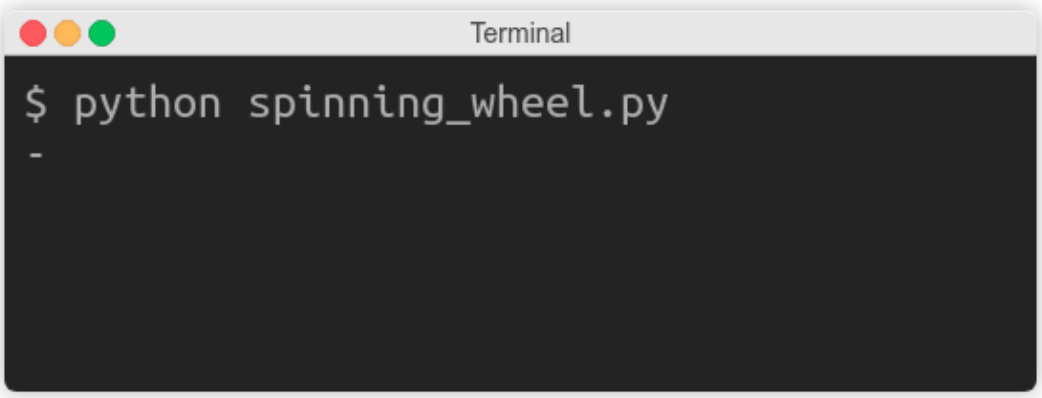
If the animation can be constrained to a single line of text, then you might be interested in two special escape character sequences:

- **Carriage return:** `\r`
- **Backspace:** `\b`

The first one moves the cursor to the beginning of the line, whereas the second one moves it only one character to the left. They both work in a non-destructive way without overwriting text that's already been written.

Let's take a look at a few examples.

You'll often want to display some kind of a **spinning wheel** to indicate a work in progress without knowing exactly how much time's left to finish:



Many command line tools use this trick while downloading data over the network. You can make a really simple stop motion animation from a sequence of characters that will cycle in a round-robin fashion:

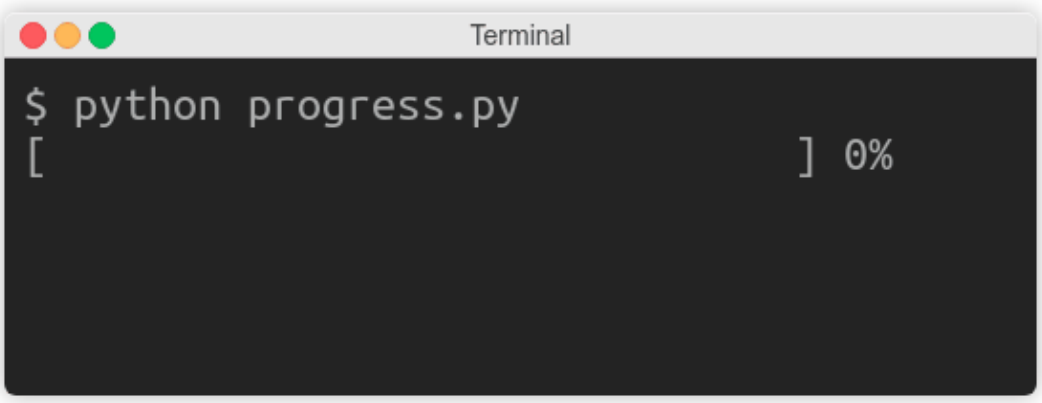
Python

```
from itertools import cycle
from time import sleep

for frame in cycle(r'-\|/-\|/'):
    print('\r', frame, sep='', end='', flush=True)
    sleep(0.2)
```

The loop gets the next character to print, then moves the cursor to the beginning of the line, and overwrites whatever there was before without adding a newline. You don't want extra space between positional arguments, so separator argument must be blank. Also, notice the use of Python's raw strings due to backslash characters present in the literal.

When you know the remaining time or task completion percentage, then you're able to show an animated progress bar:



First, you need to calculate how many hashtags to display and how many blank spaces to insert. Next, you erase the line and build the bar from scratch:

Python

```
from time import sleep

def progress(percent=0, width=30):
    left = width * percent // 100
    right = width - left
    print('\r[', '#' * left, ' ' * right, ']',
          f' {percent:.0f}%',
          sep='', end='', flush=True)

for i in range(101):
    progress(i)
    sleep(0.1)
```

As before, each request for update repaints the entire line.

> **Note:** There's a feature-rich `progressbar2` library, along with a few other similar tools, that can show progress in a much more comprehensive way.

Making Sounds With Print

If you're old enough to remember computers with a PC speaker, then you must also remember their distinctive *beep* sound, often used to indicate hardware problems. They could barely make any more noises than that, yet video games seemed so much better with it.

Today you can still take advantage of this small loudspeaker, but chances are your laptop didn't come with one. In such a case, you can enable **terminal bell** emulation in your shell, so that a system warning sound is played instead.

Go ahead and type this command to see if your terminal can play a sound:

Shell

```
$ echo -e "\a"
```

This would normally print text, but the `-e` flag enables the interpretation of backslash escapes. As you can see, there's a dedicated escape sequence `\a`, which stands for "alert", that outputs a special bell character. Some terminals make a sound whenever they see it.

Similarly, you can print this character in Python. Perhaps in a loop to form some kind of melody. While it's only a single note, you can still vary the length of pauses between consecutive instances. That seems like a perfect toy for Morse code playback!

The rules are the following:

- Letters are encoded with a sequence of **dot** (·) and **dash** (–) symbols.
- A **dot** is one unit of time.
- A **dash** is three units of time.
- Individual **symbols** in a letter are spaced one unit of time apart.
- Symbols of two adjacent **letters** are spaced three units of time apart.
- Symbols of two adjacent **words** are spaced seven units of time apart.

According to those rules, you could be "printing" an SOS signal indefinitely in the following way:

Python

```
while True:
    dot()
    symbol_space()
    dot()
    symbol_space()
    dot()
    letter_space()
    dash()
    symbol_space()
    dash()
    symbol_space()
    dash()
    letter_space()
    dot()
    symbol_space()
    dot()
    symbol_space()
    dot()
    word_space()
```

In Python, you can implement it in merely ten lines of code:

Python

```
from time import sleep

speed = 0.1

def signal(duration, symbol):
    sleep(duration)
    print(symbol, end='', flush=True)

dot = lambda: signal(speed, '·\a')
dash = lambda: signal(3*speed, '–\a')
symbol_space = lambda: signal(speed, '')
letter_space = lambda: signal(3*speed, '')
word_space = lambda: signal(7*speed, ' ')
```

Maybe you could even take it one step further and make a command line tool for translating text into Morse code? Either way, I hope you're having fun with this!

Mocking Python Print in Unit Tests

Nowadays, it's expected that you ship code that meets high quality standards. If you aspire to become a professional, you must learn how to test your code.

Software testing is especially important in dynamically typed languages, such as Python, which don't have a compiler to warn you about obvious mistakes. Defects can make their way to the production environment and remain dormant for a long time, until that one day when a branch of code finally gets executed.

Sure, you have linters, type checkers, and other tools for static code analysis to assist you. But they won't tell you whether your program does what it's supposed to do on the business level.

So, should you be testing `print()`? No. After all, it's a built-in function that must have already gone through a comprehensive suite of tests. What you want to test, though, is whether your code is calling `print()` at the right time with the expected parameters. That's known as a **behavior**.

You can test behaviors by mocking real objects or functions. In this case, you want to mock `print()` to record and verify its invocations.

> **Note:** You might have heard the terms: **dummy**, **fake**, **stub**, **spy**, or **mock** used interchangeably. Some people make a distinction between them, while others don't.
>
> Martin Fowler explains their differences in a short glossary and collectively calls them **test doubles**.

Mocking in Python can be done twofold. First, you can take the traditional path of statically-typed languages by employing dependency injection. This may sometimes require you to change the code under test, which isn't always possible if the code is defined in an external library:

Python

```
def download(url, log=print):
    log(f'Downloading {url}')
    # ...
```

This is the same example I used in an earlier section to talk about function composition. It basically allows for substituting `print()` with a custom function of the same interface. To check if it prints the right message, you have to intercept it by injecting a mocked function:

Python >>>

```
>>> def mock_print(message):
...     mock_print.last_message = message
...
>>> download('resource', mock_print)
>>> assert 'Downloading resource' == mock_print.last_message
```

Calling this mock makes it save the last message in an attribute, which you can inspect later, for example in an `assert` statement.

In a slightly alternative solution, instead of replacing the entire `print()` function with a custom wrapper, you could redirect the standard output to an in-memory file-like stream of characters:

Python >>>

```
>>> def download(url, stream=None):
...     print(f'Downloading {url}', file=stream)
...     # ...
...
>>> import io
>>> memory_buffer = io.StringIO()
>>> download('app.js', memory_buffer)
>>> download('style.css', memory_buffer)
>>> memory_buffer.getvalue()
'Downloading app.js\nDownloading style.css\n'
```

This time the function explicitly calls `print()`, but it exposes its `file` parameter to the outside world.

However, a more Pythonic way of mocking objects takes advantage of the built-in `mock` module, which uses a technique called monkey patching. This derogatory name stems from it being a "dirty hack" that you can easily shoot yourself in the foot with. It's less elegant than dependency injection but definitely quick and convenient.

> **Note:** The `mock` module got absorbed by the standard library in Python 3, but before that, it was a third-party package. You had to install it separately:
>
> Shell
>
> ```
> $ pip2 install mock
> ```
>
> Other than that, you referred to it as `mock`, whereas in Python 3 it's part of the unit testing module, so you must import from `unittest.mock`.

What monkey patching does is alter implementation dynamically at runtime. Such a change is visible globally, so it may have unwanted consequences. In practice, however, patching only affects the code for the duration of test execution.

To mock `print()` in a test case, you'll typically use the `@patch` decorator and specify a target for patching by referring to it with a fully qualified name, that is including the module name:

Python

```
from unittest.mock import patch

@patch('builtins.print')
def test_print(mock_print):
    print('not a real print')
    mock_print.assert_called_with('not a real print')
```

This will automatically create the mock for you and inject it to the test function. However, you need to declare that your test function accepts a mock now. The underlying mock object has lots of useful methods and attributes for verifying behavior.

Did you notice anything peculiar about that code snippet?

Despite injecting a mock to the function, you're not calling it directly, although you could. That injected mock is only used to make assertions afterward and maybe to prepare the context before running the test.

In real life, mocking helps to isolate the code under test by removing dependencies such as a database connection. You rarely call mocks in a test, because that doesn't make much sense. Rather, it's other pieces of code that call your mock indirectly without knowing it.

Here's what that means:

Python

```
from unittest.mock import patch

def greet(name):
    print(f'Hello, {name}!')

@patch('builtins.print')
def test_greet(mock_print):
    greet('John')
    mock_print.assert_called_with('Hello, John!')
```

The code under test is a function that prints a greeting. Even though it's a fairly simple function, you can't test it easily because it doesn't return a value. It has a side-effect.

To eliminate that side-effect, you need to mock the dependency out. Patching lets you avoid making changes to the original function, which can remain agnostic about `print()`. It thinks it's calling `print()`, but in reality, it's calling a mock you're in total control of.

There are many reasons for testing software. One of them is looking for bugs. When you write tests, you often want to get rid of the `print()` function, for example, by mocking it away. Paradoxically, however, that same function can help you find bugs during a related process of debugging you'll read about in the next section.

Syntax in Python 2 Show/Hide

Print Debugging

In this section, you'll take a look at the available tools for debugging in Python, starting from a humble `print()` function, through the `logging` module, to a fully fledged debugger. After reading it, you'll be able to make an educated decision about which of them is the most suitable in a given situation.

> **Note:** Debugging is the process of looking for the root causes of **bugs** or defects in software after they've been discovered, as well as taking steps to fix them.
>
> The term **bug** has an amusing story about the origin of its name.

Tracing

Also known as **print debugging** or **caveman debugging**, it's the most basic form of debugging. While a little bit old-fashioned, it's still powerful and has its uses.

The idea is to follow the path of program execution until it stops abruptly, or gives incorrect results, to identify the exact instruction with a problem. You do that by inserting print statements with words that stand out in carefully chosen places.

Take a look at this example, which manifests a rounding error:

Python >>>

```
>>> def average(numbers):
...     print('debug1:', numbers)
...     if len(numbers) > 0:
...         print('debug2:', sum(numbers))
...         return sum(numbers) / len(numbers)
...
>>> 0.1 == average(3*[0.1])
debug1: [0.1, 0.1, 0.1]
debug2: 0.30000000000000004
False
```

As you can see, the function doesn't return the expected value of `0.1`, but now you know it's because the sum is a little off. Tracing the state of variables at different steps of the algorithm can give you a hint where the issue is.

Rounding Error Show/Hide

This method is simple and intuitive and will work in pretty much every programming language out there. Not to mention, it's a great exercise in the learning process.

On the other hand, once you master more advanced techniques, it's hard to go back, because they allow you to find bugs much quicker. Tracing is a laborious manual process, which can let even more errors slip through. The build and deploy cycle takes time. Afterward, you need to remember to meticulously remove all the `print()` calls you made without accidentally touching the genuine ones.

Besides, it requires you to make changes in the code, which isn't always possible. Maybe you're debugging an application running in a remote web server or want to diagnose a problem in a **post-mortem** fashion. Sometimes you simply don't have access to the standard output.

That's precisely where logging shines.

Logging

Let's pretend for a minute that you're running an e-commerce website. One day, an angry customer makes a phone call complaining about a failed transaction and saying he lost his money. He claims to have tried purchasing a few items, but in the end, there was some cryptic error that prevented him from finishing that order. Yet, when he checked his bank account, the money was gone.

You apologize sincerely and make a refund, but also don't want this to happen again in the future. How do you debug that? If only you had some trace of what happened, ideally in the form of a chronological list of events with their context.

Whenever you find yourself doing print debugging, consider turning it into permanent log messages. This may help in situations like this, when you need to analyze a problem after it happened, in an environment that you don't have access to.

There are sophisticated tools for log aggregation and searching, but at the most basic level, you can think of logs as text files. Each line conveys detailed information about an event in your system. Usually, it won't contain personally identifying information, though, in some cases, it may be mandated by law.

Here's a breakdown of a typical log record:

Text

```
[2019-06-14 15:18:34,517][DEBUG][root][MainThread] Customer(id=123) logged out
```

As you can see, it has a structured form. Apart from a descriptive message, there are a few customizable fields, which provide the context of an event. Here, you have the exact date and time, the log level, the logger name, and the thread name.

Log levels allow you to filter messages quickly to reduce noise. If you're looking for an error, you don't want to see all the warnings or debug messages, for example. It's trivial to disable or enable messages at certain log levels through the configuration, without even touching the code.

With logging, you can keep your debug messages separate from the standard output. All the log messages go to the standard error stream by default, which can conveniently show up in different colors. However, you can redirect log messages to separate files, even for individual modules!

Quite commonly, misconfigured logging can lead to running out of space on the server's disk. To prevent that, you may set up **log rotation**, which will keep the log files for a specified duration, such as one week, or once they hit a certain size. Nevertheless, it's always a good practice to archive older logs. Some regulations enforce that customer data be kept for as long as five years!

Compared to other programming languages, logging in Python is simpler, because the `logging` module is bundled with the standard library. You just import and configure it in as little as two lines of code:

Python

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

You can call functions defined at the module level, which are hooked to the **root logger**, but more the common practice is to obtain a dedicated logger for each of your source files:

Python

```
logging.debug('hello')  # Module-level function

logger = logging.getLogger(__name__)
logger.debug('hello')   # Logger's method
```

The advantage of using custom loggers is more fine-grain control. They're usually named after the module they were defined in through the `__name__` variable.

> **Note:** There's a somewhat related `warnings` module in Python, which can also log messages to the standard error stream. However, it has a narrower spectrum of applications, mostly in library code, whereas client applications should use the `logging` module.
>
> That said, you can make them work together by calling `logging.captureWarnings(True)`.

One last reason to switch from the `print()` function to logging is thread safety. In the upcoming section, you'll see that the former doesn't play well with multiple threads of execution.

Debugging

The truth is that neither tracing nor logging can be considered real debugging. To do actual debugging, you need a debugger tool, which allows you to do the following:

- Step through the code interactively.
- Set breakpoints, including conditional breakpoints.
- Introspect variables in memory.
- Evaluate custom expressions at runtime.

A crude debugger that runs in the terminal, unsurprisingly named **pdb** for "The Python Debugger," is distributed as part of the standard library. This makes it always available, so it may be your only choice for performing remote debugging. Perhaps that's a good reason to get familiar with it.

However, it doesn't come with a graphical interface, so using `pdb` may be a bit tricky. If you can't edit the code, you have to run it as a module and pass your script's location:

Shell

```
$ python -m pdb my_script.py
```

Otherwise, you can set up a breakpoint directly in the code, which will pause the execution of your script and drop you into the debugger. The old way of doing this required two steps:

Python >>>

```
>>> import pdb
>>> pdb.set_trace()
--Return--
> <stdin>(1)<module>()->None
(Pdb)
```

This shows up an interactive prompt, which might look intimidating at first. However, you can still type native Python at this point to examine or modify the state of local variables. Apart from that, there's really only a handful of debugger-specific commands that you want to use for stepping through the code.

> **Note:** It's customary to put the two instructions for spinning up a debugger on a single line. This requires the use of a semicolon, which is rarely found in Python programs:
>
> Python
>
> ```
> import pdb; pdb.set_trace()
> ```
>
> While certainly not Pythonic, it stands out as a reminder to remove it after you're done with debugging.

Since Python 3.7, you can also call the built-in `breakpoint()` function, which does the same thing, but in a more compact way and with some additional bells and whistles:

Python

```
def average(numbers):
    if len(numbers) > 0:
        breakpoint()  # Python 3.7+
        return sum(numbers) / len(numbers)
```

You're probably going to use a visual debugger integrated with a code editor for the most part. PyCharm has an excellent debugger, which boasts high performance, but you'll find plenty of alternative IDEs with debuggers, both paid and free of charge.

Debugging isn't the proverbial silver bullet. Sometimes logging or tracing will be a better solution. For example, defects that are hard to reproduce, such as race conditions, often result from temporal coupling. When you stop at a breakpoint, that little pause in program execution may mask the problem. It's kind of like the Heisenberg principle: you can't measure and observe a bug at the same time.

These methods aren't mutually exclusive. They complement each other.

Thread-Safe Printing

I briefly touched upon the thread safety issue before, recommending `logging` over the `print()` function. If you're still reading this, then you must be comfortable with the concept of threads.

Thread safety means that a piece of code can be safely shared between multiple threads of execution. The simplest strategy for ensuring thread-safety is by sharing **immutable** objects only. If threads can't modify an object's state, then there's no risk of breaking its consistency.

Another method takes advantage of **local memory**, which makes each thread receive its own copy of the same object. That way, other threads can't see the changes made to it in the current thread.

But that doesn't solve the problem, does it? You often want your threads to cooperate by being able to mutate a shared resource. The most common way of synchronizing concurrent access to such a resource is by **locking** it. This gives exclusive write access to one or sometimes a few threads at a time.

However, locking is expensive and reduces concurrent throughput, so other means for controlling access have been invented, such as **atomic variables** or the **compare-and-swap** algorithm.

Printing isn't thread-safe in Python. The `print()` function holds a reference to the standard output, which is a shared global variable. In theory, because there's no locking, a context switch could happen during a call to `sys.stdout.write()`, intertwining bits of text from multiple `print()` calls.

> **Note:** A context switch means that one thread halts its execution, either voluntarily or not, so that another one can take over. This might happen at any moment, even in the middle of a function call.

In practice, however, that doesn't happen. No matter how hard you try, writing to the standard output seems to be atomic. The only problem that you may sometimes observe is with messed up line breaks:

Text

```
[Thread-3 A][Thread-2 A][Thread-1 A]

[Thread-3 B][Thread-1 B]


[Thread-1 C][Thread-3 C]

[Thread-2 B]
[Thread-2 C]
```

To simulate this, you can increase the likelihood of a context switch by making the underlying `.write()` method go to sleep for a random amount of time. How? By mocking it, which you already know about from an earlier section:

Python

```
import sys

from time import sleep
from random import random
from threading import current_thread, Thread
from unittest.mock import patch

write = sys.stdout.write

def slow_write(text):
    sleep(random())
    write(text)

def task():
    thread_name = current_thread().name
    for letter in 'ABC':
        print(f'[{thread_name} {letter}]')

with patch('sys.stdout') as mock_stdout:
    mock_stdout.write = slow_write
    for _ in range(3):
        Thread(target=task).start()
```

First, you need to store the original `.write()` method in a variable, which you'll delegate to later. Then you provide your fake implementation, which will take up to one second to execute. Each thread will make a few `print()` calls with its name and a letter: A, B, and C.

If you read the mocking section before, then you may already have an idea of why printing misbehaves like that. Nonetheless, to make it crystal clear, you can capture values fed into your `slow_write()` function. You'll notice that you get a slightly different sequence each time:

Python

```
[
    '[Thread-3 A]',
    '[Thread-2 A]',
    '[Thread-1 A]',
    '\n',
    '\n',
    '[Thread-3 B]',
    (...)
]
```

Even though `sys.stdout.write()` itself is an atomic operation, a single call to the `print()` function can yield more than one write. For example, line breaks are written separately from the rest of the text, and context switching takes place between those writes.

> **Note:** The atomic nature of the standard output in Python is a byproduct of the Global Interpreter Lock, which applies locking around bytecode instructions. Be aware, however, that many interpreter flavors don't have the GIL, where multi-threaded printing requires explicit locking.

You can make the newline character become an integral part of the message by handling it manually:

Python

```
print(f'[{thread_name} {letter}]\n', end='')
```

This will fix the output:

Text

```
[Thread-2 A]
[Thread-1 A]
[Thread-3 A]
[Thread-1 B]
[Thread-3 B]
[Thread-2 B]
[Thread-1 C]
[Thread-2 C]
[Thread-3 C]
```

Notice, however, that the `print()` function still keeps making a separate call for the empty suffix, which translates to useless `sys.stdout.write('')` instruction:

Python

```
[
    '[Thread-2 A]\n',
    '[Thread-1 A]\n',
    '[Thread-3 A]\n',
    '',
    '',
    '',
    '[Thread-1 B]\n',
    (...)
]
```

A truly thread-safe version of the `print()` function could look like this:

Python

```
import threading

lock = threading.Lock()

def thread_safe_print(*args, **kwargs):
    with lock:
        print(*args, **kwargs)
```

You can put that function in a module and import it elsewhere:

Python

```
from thread_safe_print import thread_safe_print

def task():
    thread_name = current_thread().name
    for letter in 'ABC':
        thread_safe_print(f'[{thread_name} {letter}]')
```

Now, despite making two writes per each `print()` request, only one thread is allowed to interact with the stream, while the rest must wait:

Python

```
[
    # Lock acquired by Thread-3
    '[Thread-3 A]',
    '\n',
    # Lock released by Thread-3
    # Lock acquired by Thread-1
    '[Thread-1 B]',
    '\n',
    # Lock released by Thread-1
    (...)
]
```

I added comments to indicate how the lock is limiting access to the shared resource.

> **Note:** Even in single-threaded code, you might get caught up in a similar situation. Specifically, when you're printing to the standard output and the standard error streams at the same time. Unless you redirect one or both of them to separate files, they'll both share a single terminal window.

Conversely, the `logging` module is thread-safe by design, which is reflected by its ability to display thread names in the formatted message:

Python >>>

```
>>> import logging
>>> logging.basicConfig(format='%(threadName)s %(message)s')
>>> logging.error('hello')
MainThread hello
```

It's another reason why you might not want to use the `print()` function all the time.

Python Print Counterparts

By now, you know a lot of what there is to know about `print()`! The subject, however, wouldn't be complete without talking about its counterparts a little bit. While `print()` is about the output, there are functions and libraries for the input.

Built-In

Python comes with a built-in function for accepting input from the user, predictably called `input()`. It accepts data from the standard input stream, which is usually the keyboard:

Python >>>

```
>>> name = input('Enter your name: ')
Enter your name: jdoe
>>> print(name)
jdoe
```

The function always returns a string, so you might need to parse it accordingly:

Python

```
try:
    age = int(input('How old are you? '))
except ValueError:
    pass
```

The prompt parameter is completely optional, so nothing will show if you skip it, but the function will still work:

Python >>>

```
>>> x = input()
hello world
>>> print(x)
hello world
```

Nevertheless, throwing in a descriptive call to action makes the user experience so much better.

> **Note:** To read from the standard input in Python 2, you have to call `raw_input()` instead, which is yet another built-in. Unfortunately, there's also a misleadingly named `input()` function, which does a slightly different thing.

In fact, it also takes the input from the standard stream, but then it tries to evaluate it as if it was Python code. Because that's a potential **security vulnerability**, this function was completely removed from Python 3, while `raw_input()` got renamed to `input()`.

Here's a quick comparison of the available functions and what they do:

Python 2	Python 3
`raw_input()`	`input()`
`input()`	`eval(input())`

As you can tell, it's still possible to simulate the old behavior in Python 3.

Asking the user for a password with `input()` is a bad idea because it'll show up in plaintext as they're typing it. In this case, you should be using the `getpass()` function instead, which masks typed characters. This function is defined in a module under the same name, which is also available in the standard library:

Python >>>

```
>>> from getpass import getpass
>>> password = getpass()
Password:
>>> print(password)
s3cret
```

The `getpass` module has another function for getting the user's name from an environment variable:

Python >>>

```
>>> from getpass import getuser
>>> getuser()
'jdoe'
```

Python's built-in functions for handling the standard input are quite limited. At the same time, there are plenty of third-party packages, which offer much more sophisticated tools.

Third-Party

There are external Python packages out there that allow for building complex graphical interfaces specifically to collect data from the user. Some of their features include:

- Advanced formatting and styling
- Automated parsing, validation, and sanitization of user data
- A declarative style of defining layouts
- Interactive autocompletion
- Mouse support
- Predefined widgets such as checklists or menus
- Searchable history of typed commands
- Syntax highlighting

Demonstrating such tools is outside of the scope of this article, but you may want to try them out. I personally got to know about some of those through the Python Bytes Podcast. Here they are:

- `bullet`
- `cooked-input`
- `prompt_toolkit`
- `questionnaire`

Nonetheless, it's worth mentioning a command line tool called `rlwrap` that adds powerful line editing capabilities to your Python scripts for free. You don't have to do anything for it to work!

Let's assume you wrote a command-line interface that understands three instructions, including one for adding numbers:

Python

```
print('Type "help", "exit", "add a [b [c ...]]"')
while True:
    command, *arguments = input('~ ').split(' ')
    if len(command) > 0:
        if command.lower() == 'exit':
            break
        elif command.lower() == 'help':
            print('This is help.')
        elif command.lower() == 'add':
            print(sum(map(int, arguments)))
        else:
            print('Unknown command')
```

At first glance, it seems like a typical prompt when you run it:

Shell

```
$ python calculator.py
Type "help", "exit", "add a [b [c ...]]"
~ add 1 2 3 4
10
~ aad 2 3
Unknown command
~ exit
$
```

But as soon as you make a mistake and want to fix it, you'll see that none of the function keys work as expected. Hitting the `Left` arrow, for example, results in this instead of moving the cursor back:

Shell

```
$ python calculator.py
Type "help", "exit", "add a [b [c ...]]"
~ aad^[[D
```

Now, you can wrap the same script with the `rlwrap` command. Not only will you get the arrow keys working, but you'll also be able to search through the persistent history of your custom commands, use autocompletion, and edit the line with shortcuts:

Shell

```
$ rlwrap python calculator.py
Type "help", "exit", "add a [b [c ...]]"
(reverse-i-search)`a': add 1 2 3 4
```

Isn't that great?

Conclusion

You're now armed with a body of knowledge about the `print()` function in Python, as well as many surrounding topics. You have a deep understanding of what it is and how it works, involving all of its key elements. Numerous examples gave you insight into its evolution from Python 2.

Apart from that, you learned how to:

- Avoid common mistakes with `print()` in Python
- Deal with newlines, character encodings and buffering

- Write text to files
- Mock the `print()` function in unit tests
- Build advanced user interfaces in the terminal

Now that you know all this, you can make interactive programs that communicate with users or produce data in popular file formats. You're able to quickly diagnose problems in your code and protect yourself from them. Last but not least, you know how to implement the classic snake game.

If you're still thirsty for more information, have questions, or simply would like to share your thoughts, then feel free to reach out in the comments section below.

About **Bartosz Zaczyński**

Bartosz is a bootcamp instructor, author, and polyglot programmer in love with Python. He helps his students get into software engineering by sharing over a decade of commercial experience in the IT industry.

» More about Bartosz

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Joanna

Mike

Pip, PyPI, Virtualenv: How to Set It All Up

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

Keep Learning

Related Tutorial Categories: basics python

All Tutorial Topics

advanced api basics best-practices community databases data-science devops django docker flask front-end intermediate machine-learning python testing tools web-dev web-scraping

Table of Contents