**Python Output Using print() function**

We use the print() or eval and expression function to output data to the standard output device (screen). We can also [output data to a file](https://www.programiz.com/python-programming/file-operation)

syntax of the print() function is:

### **Unformatted Console Output**

To display objects to the console, pass them as a comma-separated list of argument to print ().

print(<obj>, ..., <obj>)

By default, print() separates each object by a single space and appends a newline to the end of the output:

>>>

>>> fname = 'Winston'

>>> lname = 'Smith'

>>> print('Name:', fname, lname)

Name: Winston Smith

Any type of object can be specified as an argument to print(). If an object isn’t a string, then print() converts it to an appropriate string representation displaying it:

>>>

>>> a = [1, 2, 3]

>>> type(a)

<class 'list'>

>>> b = -12

>>> type(b)

<class 'int'>

>>> d = {'foo': 1, 'bar': 2}

>>> type(d)

<class 'dict'>

>>> type(len)

<class 'builtin\_function\_or\_method'>

>>> print(a, b, d, len)

[1, 2, 3] -12 {'foo': 1, 'bar': 2} <built-in function len>

**Key arguments in print statement**

print(\*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Here, objects is the value(s) to be printed.

The sep separator is used between the values. It defaults into a space character.

After all values are printed, end is printed. It defaults into a new line.

The file is the object where the values are printed and its default value is sys.stdout (screen).

print(1, 2, 3, 4)

print(1, 2, 3, 4, sep='\*')

print(1, 2, 3, 4, sep='#', end='&')

**Output**

1 2 3 4

1\*2\*3\*4

1#2#3#4&

## Output formatting

Python provides several ways to format output string data the **string modulo operator**, the **string .format() method**, and **f-strings**.

the syntax of the string modulo operator looks like:

<format\_string> % <values>

On the left side of the % operator, <format\_string> is a string containing one or more conversion specifiers. The <values> on the right side get inserted into <format\_string> in place of the conversion specifiers. The resulting formatted string is the value of the expression.

Here’s a print() statement that displays a formatted string using the string modulo operator:

>>> print('%d %s cost $%.2f' % (6, 'bananas', 1.74))

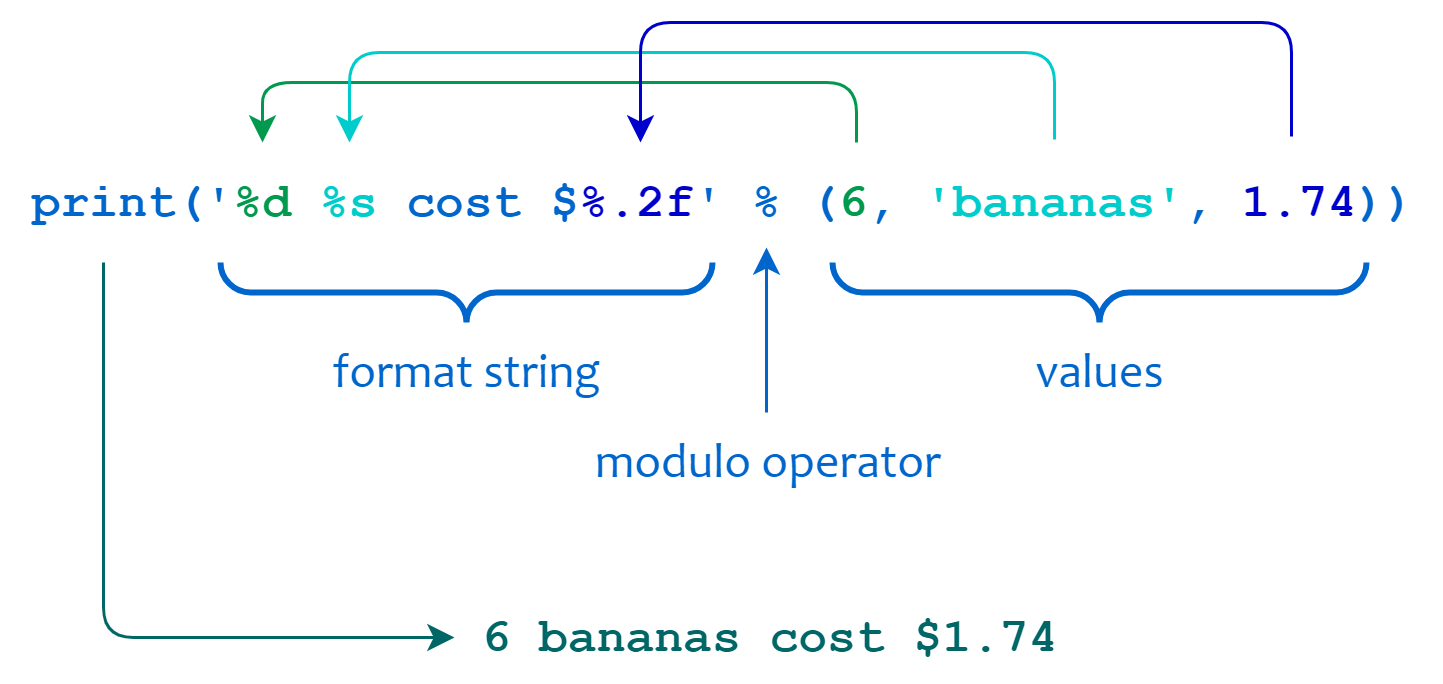
6 bananas cost $1.74

n addition to representing the string modulo operation itself, the '%' character also denotes the conversion specifiers in the format string—in this case, '%d', '%s', and '%.2f'.

In the output, each item from the tuple of values is converted to a string value and inserted into the format string in place of the corresponding conversion specifier:

* The first item in the tuple is 6, a numeric value that replaces '%d' in the format string.
* The next item is the string value 'bananas', which replaces '%s'.
* The last item is the float value 1.74, which replaces '%.2f'.

The resulting string is 6 bananas cost $1.74, as demonstrated in the following diagram:

[](https://files.realpython.com/media/t.176c482e3252.png)

If there are multiple values to insert, then they must be enclosed in a tuple as illustrated above. If there is only one value, then it can appear by itself:

### **Conversion Specifiers**

Conversion specifiers appear in the <format\_string> and determine how values are formatted when they’re inserted.

A conversion specifier begins with a % character and consists of these components:

%[<flags>][<width>][.<precision>]<type>

% and <type> are required. The remaining components shown in square brackets are optional.

The following table summarizes what each component of a conversion specifier does:

| **Component** | **Meaning** |
| --- | --- |
| % | Introduces the conversion specifier |
| <flags> | Indicates one or more flags that exert finer control over formatting |
| <width> | Specifies the minimum width of the formatted result |
| .<precision> | Determines the length and precision of floating point or string output |
| <type> | Indicates the type of conversion to be performed |

### **Conversion Type**

The conversion type, <type>, is the last component of the conversion specifier:

%[<flags>][<width>][.<precision>]**<type>**

It determines the type of conversion the corresponding value undergoes before insertion into the format string. Here’s a table that lists the possible conversion types:

| **<type>** | **Conversion Type** |
| --- | --- |
| d, i, u | Decimal integer |
| x, X | Hexadecimal integer |
| o | Octal integer |
| f, F | Floating point |
| e, E | Exponential |
| g, G | Floating point or Exponential |
| c | Single character |
| s, r, a | String |
| % | Single '%' character |

You’ll see how to use these conversion types in the following sections.

#### **Integer Conversion Types**

The d, i, u, x, X, and o conversion types correspond to integer values.

d, i, and u are functionally equivalent. They all convert the corresponding argument to a string representation of a decimal integer:

>>>

>>> '%d, %i, %u' % (42, 42, 42)

'42, 42, 42'

>>> '%d, %i, %u' % (-42, -42, -42)

'-42, -42, -42'

The value can either be positive or negative. If it is negative, then the resulting value will start with a '-' character.

x and X convert to a string representation of a hexadecimal integer value, and o converts to a string representation of an octal integer value:

>>>

>>> '%x, %X' % (252, 252)

'fc, FC'

>>> '%o' % 16

'20'

x produces lowercase output, and X produces uppercase. (Uppercase 'O' is not a valid conversion type.)

#### **Floating Point Conversion Types**

Conversion types f and F convert to a string representation of a floating point number, while e and E produce a string representing exponential (scientific) notation:

>>>

>>> '%f, %F' % (3.14159, 3.14)

'3.141590, 3.140000'

>>> '%e, %E' % (1000.0, 1000.0)

'1.000000e+03, 1.000000E+03'

e produces lowercase output, and E produces uppercase.

#### **Deep Dive: inf and NaN**

Under some circumstances, a floating-point operation can result in a value that is essentially infinite. The string representation of such a number in Python is 'inf'.

It also may happen that a floating-point operation produces a value that is not representable as a number. Python represents this with the string 'NaN'.

When these values are converted with the string modulo operator, the conversion type character controls the case of the resulting output. f and e produce lowercase output, while F and E produce uppercase:

>>>

>>> x = float('NaN')

>>> '%f, %e, %F, %E' % (x, x, x, x)

'nan, nan, NAN, NAN'

>>> y = float('Inf')

>>> '%f, %e, %F, %E' % (y, y, y, y)

'inf, inf, INF, INF'

This is the only difference between the f and F conversion types.

The g and G conversion types choose between floating point or exponential output, depending on the magnitude of the exponent and the value specified for .<precision>. (See below.) Output is the same as e/E if the exponent is less than -4 or not less than .<precision>. Otherwise, it’s the same as f/F:

>>>

>>> '%g' % 3.14

'3.14'

>>> '%g' % 0.00000003

'3e-08'

>>> '%G' % 0.00000003

'3E-08'

Basically, you can think of these conversion types as making a “reasonable” choice. They’ll produce floating point output if the value in question is reasonably suitable for it, and exponential format otherwise.

Similar to the other floating point conversion types, g produces lowercase output, and G produces uppercase.

#### **Character Conversion Types**

c inserts a single character. The corresponding value may be either an integer or a single-character string:

>>>

>>> '%c' % 97

'a'

>>> '[%c]' % 'y'

'[y]'

The c conversion type supports conversion to [Unicode](https://realpython.com/python-encodings-guide) characters as well:

>>>

>>> '%c' % 8721

'∑'

s, r, and a produce string output using the built-in functions [str()](https://docs.python.org/3/library/stdtypes.html#str), [repr()](https://docs.python.org/3/library/functions.html" \l "repr), and [ascii()](https://docs.python.org/3/library/functions.html#ascii), respectively:

>>>

>>> '%s' % 'foo'

'foo'

>>> '%r' % 'foo'

"'foo'"

>>> '%a' % 'foo'

"'foo'"

The justification and padding of string output can be controlled with the <width> and .<precision> specifiers, as you will see shortly.

#### **Inserting a '%' Character**

To insert a literal '%' character into the output, specify two consecutive % characters in the format string. The first introduces a conversion specifier (as usual), and the second specifies that the conversion type is %, which results in a single '%' character in the output.

In this example, %d%% means a decimal integer conversion type followed by a literal '%' character:

>>>

>>> 'Get %d%% off on %s today only!' % (30, 'bananas')

'Get 30% off on bananas today only!'

Note that the % conversion type doesn’t consume any of the <values> to the right of the string modulo operator.

### **Width and Precision Specifiers**

<width> and .<precision> sit in the middle of the conversion specifier:

%[<flags>]**[<width>][.<precision>]**<type>

They determine how much horizontal space a formatted value occupies.

#### **The <width> Specifier**

<width> specifies the minimum width of the output field. If the output is shorter than <width>, then by default it is right-justified in a field that is <width> characters wide, and padded with ASCII space characters on the left:

>>>

>>> '%5s' % 'foo'

' foo'

>>> '%3d' % 4

' 4'

(The justification and padding character can be modified. See **conversion flags** below.)

If the output length is greater than <width>, then <width> has no effect:

>>>

>>> '%2d' % 1234, '%d' % 1234

('1234', '1234')

>>> '%2s' % 'foobar', '%s' % 'foobar'

('foobar', 'foobar')

Each of these examples specifies a field width of 2. But since the values to be formatted are more than two characters, the result is the same as when no <width> is specified.

#### **The .<precision> Specifier**

.<precision> affects the floating point, exponential, and string conversion types.

For the f, F, e, and E types, .<precision> determines the number of digits after the decimal point:

>>>

>>> '%.2f' % 123.456789

'123.46'

>>> '%.2e' % 123.456789

'1.23e+02'

For the g and G types, .<precision> determines the total number of significant digits before and after the decimal point:

>>>

>>> '%.2g' % 123.456789

'1.2e+02'

String values formatted with the s, r, and a types are truncated to the length specified by .<precision>:

>>>

>>> '%.4s' % 'foobar'

'foob'

It is very common to see <width> and .<precision> used together:

>>>

>>> '%8.2f' % 123.45678

' 123.46'

>>> '%8.3s' % 'foobar'

' foo'

Either of <width> or .<precision> can be specified as an asterisk character (\*), in which case the value to be used is taken from the next item in the <values> tuple:

>>>

>>> '%\*d' % (5, 123)

' 123'

There isn’t much need for this when the <width> value is given as a constant. There isn’t any functional difference between the example given above and this:

>>>

>>> '%5d' % 123

' 123'

But you can also specify <width> and .<precision> by variable:

>>>

>>> for i in range(3):

... w = int(input('Enter width: '))

... print('[%\*s]' % (w, 'foo'))

...

Enter width: 2

[foo]

Enter width: 4

[ foo]

Enter width: 8

[ foo]

This allows the width or precision to be determined at run-time, and potentially change from one execution to the next.

### **Conversion Flags**

Optional conversion <flags> are specified just after the initial % character:

%**[<flags>]**[<width>][.<precision>]<type>

These allow finer control over the display of certain conversion types. <flags> can include any of the characters shown in the following table:

| **Character** | **Controls** |
| --- | --- |
| # | Display of base or decimal point for integer and floating point values |
| 0 | Padding of values that are shorter than the specified field width |
| - | Justification of values that are shorter than the specified field width |
| + ' ' (space) | Display of leading sign for numeric values |

The following sections explain how conversion flags operate in greater detail.

#### **The # Flag**

For the octal and hexadecimal conversion types, the # flag causes base information to be included in the formatted output. For the o conversion type, this flag adds a leading '0o'. For the x and X conversion types, it adds a leading '0x' or '0X':

>>>

>>> '%#o' % 16

'0o20'

>>> '%#x' % 16, '%#X' % 16

('0x10', '0X10')

The # flag is ignored for the decimal conversion types d, i, and u.

For floating point values, the # flag forces the output to always contain a decimal point. Ordinarily, floating point values will not contain a decimal point if there aren’t any digits after it. This flag forces a decimal point to be included:

>>>

>>> '%.0f' % 123

'123'

>>> '%#.0f' % 123

'123.'

>>> '%.0e' % 123

'1e+02'

>>> '%#.0e' % 123

'1.e+02'

This also works for values displayed in exponential notation, as shown.

#### **The 0 Flag**

When a formatted numeric value is shorter than the specified field width, the default behavior is to pad the field with ASCII space characters. The 0 flag causes padding with '0' characters instead:

>>>

>>> '%05d' % 123

'00123'

>>> '%08.2f' % 1.2

'00001.20'

The 0 flag can be used with all the numeric conversion types: d, i, u, x, X, o, f, F, e, E, g, and G.

#### **The - Flag**

When a formatted value is shorter than the specified field width, it is usually right-justified in the field. The - flag causes the value to be left-justified in the specified field instead:

>>>

>>> '%-5d' % 123

'123 '

>>> '%-8.2f' % 123.3

'123.30 '

>>> '%-\*s' % (10, 'foo')

'foo '

You can use the - flag with the string conversion types s, a, and r, as well as all the numeric conversion types. For numeric types, if both 0 and - are present, then 0 is ignored.

#### **The + and ' ' Flags**

By default, positive numeric values do not have a leading sign character. The + flag adds a '+' character to the left of numeric output:

>>>

>>> '%+d' % 3

'+3'

>>> '%+5d' % 3

' +3'

The ' ' (space character) flag causes positive numeric values to be preceded by a space character:

>>>

>>> '% d' % 3

' 3'

These flags have no effect on negative numeric values, which always have a leading '-' character.

### **Specifying Values by Dictionary Mapping**

The <values> inserted into the format string may be specified as a dictionary instead of a tuple. In that case, each conversion specifier must contain one of the dictionary keys in parentheses immediately following the '%' character.

Here’s an example:

>>>

>>> '%d %s cost $%.2f' % (6, 'bananas', 1.74)

'6 bananas cost $1.74'

>>> d = {'quantity': 6, 'item': 'bananas', 'price': 1.74}

>>> '%(quantity)d %(item)s cost $%(price).2f' % d

'6 bananas cost $1.74'

Using this technique, you can specify the inserted values in any order:

>>>

### **Specifying Values by Dictionary Mapping**

The <values> inserted into the format string may be specified as a dictionary instead of a tuple. In that case, each conversion specifier must contain one of the dictionary keys in parentheses immediately following the '%' character.

Here’s an example:

>>>

>>> '%d %s cost $%.2f' % (6, 'bananas', 1.74)

'6 bananas cost $1.74'

>>> d = {'quantity': 6, 'item': 'bananas', 'price': 1.74}

>>> '%(quantity)d %(item)s cost $%(price).2f' % d

'6 bananas cost $1.74'

Using this technique, you can specify the inserted values in any order:

>>>

>>> d = {'quantity': 6, 'item': 'bananas', 'price': 1.74}

>>> '%(quantity)d %(item)s cost $%(price).2f' % d

'6 bananas cost $1.74'

>>> 'It will cost you $%(price).2f for %(item)s, if you buy %(quantity)d' % d

'It will cost you $1.74 for bananas, if you buy 6'

All the conversion specifier items shown above—<flags>, <width>, .<precision>, and <type>—still have the same meaning:

>>>

>>> 'Quantity: %(quantity)03d' % d

'Quantity: 006'

>>> 'Item: %(item).5s' % d

'Item: banan'

**Note:** If you specify <values> by dictionary mapping, then you can’t use \* to specify <width> or .<precision>.

To create an f-string, prefix the string with the letter “ f ”. The string itself can be formatted in much the same way that you would with [str.format()](https://www.geeksforgeeks.org/python-format-function/). F-strings provide a concise and convenient way to embed python expressions inside string literals for formatting.

# Python3 program introducing f-string

val = 'Geeks'

print(f"{val}for{val} is a portal for {val}.")

name = 'Tushar'

age = 23

print(f"Hello, My name is {name} and I'm {age} years old.")

**Output :** 

GeeksforGeeks is a portal for Geeks.

Hello, My name is Tushar and I'm 23 years old.

# Prints today's date with help

# of datetime library

import datetime

today = datetime.datetime.today()

print(f"{today:%B %d, %Y}")

>>> d = {'quantity': 6, 'item': 'bananas', 'price': 1.74}

>>> '%(quantity)d %(item)s cost $%(price).2f' % d

'6 bananas cost $1.74'

>>> 'It will cost you $%(price).2f for %(item)s, if you buy %(quantity)d' % d

'It will cost you $1.74 for bananas, if you buy 6'

All the conversion specifier items shown above—<flags>, <width>, .<precision>, and <type>—still have the same meaning:

>>>

>>> 'Quantity: %(quantity)03d' % d

'Quantity: 006'

>>> 'Item: %(item).5s' % d

'Item: banan'

**Note:** If you specify <values> by dictionary mapping, then you can’t use \* to specify <width> or .<precision>.

>>> x = 5; y = 10

>>> print('The value of x is {} and y is {}'.format(x,y))

The value of x is 5 and y is 10

Here, the curly braces {} are used as placeholders. We can specify the order in which they are printed by using numbers (tuple index).

print('I love {0} and {1}'.format('bread','butter'))

print('I love {1} and {0}'.format('bread','butter'))

**Output**

I love bread and butter

I love butter and bread

We can even use keyword arguments to format the string.

>>> print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))

Hello John, Goodmorning

# Intializing variables

a = 20

b = 10

# addition

sum = a + b

# subtraction

sub = a- b

# Output

print('The value of a is {} and b is {}'.format(a,b))

print('{2} is the sum of {0} and {1}'.format(a,b,sum))

print('{sub\_value} is the subtraction of {value\_a} and {value\_b}'.format(value\_a = a , value\_b = b, sub\_value = sub))

### **Option #2: str.format()**

This newer way of getting the job done was introduced in Python 2.6. You can check out [A Guide to the Newer Python String Format Techniques](https://realpython.com/python-formatted-output/) for more info.

#### **How To Use str.format()**

str.format()  is an improvement on %-formatting. It uses normal function call syntax and is [extensible through the \_\_format\_\_() method](https://www.python.org/dev/peps/pep-3101/#controlling-formatting-on-a-per-type-basis) on the object being converted to a string.

With str.format(), the replacement fields are marked by curly braces:

>>>

>>> "Hello, {}. You are {}.".format(name, age)

'Hello, Eric. You are 74.'

You can reference variables in any order by referencing their index:

>>>

>>> "Hello, {1}. You are {0}.".format(age, name)

'Hello, Eric. You are 74.'

But if you insert the variable names, you get the added perk of being able to pass objects and then reference parameters and methods in between the braces:

>>>

>>> person = {'name': 'Eric', 'age': 74}

>>> "Hello, {name}. You are {age}.".format(name=person['name'], age=person['age'])

'Hello, Eric. You are 74.'

You can also use \*\* to do this neat trick with dictionaries:

>>>

>>> person = {'name': 'Eric', 'age': 74}

>>> "Hello, {name}. You are {age}.".format(\*\*person)

'Hello, Eric. You are 74.'

str.format() is definitely an upgrade when compared with %-formatting, but it’s not all roses and sunshine.

#### **Why str.format() Isn’t Great**

Code using str.format() is much more easily readable than code using %-formatting, but str.format() can still be quite verbose when you are dealing with multiple parameters and longer strings. Take a look at this:

>>>

>>> first\_name = "Eric"

>>> last\_name = "Idle"

>>> age = 74

>>> profession = "comedian"

>>> affiliation = "Monty Python"

>>> print(("Hello, {first\_name} {last\_name}. You are {age}. " +

>>> "You are a {profession}. You were a member of {affiliation}.") \

>>> .format(first\_name=first\_name, last\_name=last\_name, age=age, \

>>> profession=profession, affiliation=affiliation))

'Hello, Eric Idle. You are 74. You are a comedian. You were a member of Monty Python.'

If you had the variables you wanted to pass to .format() in a dictionary, then you could just unpack it with .format(\*\*some\_dict) and reference the values by key in the string, but there has got to be a better way to do this.

## f-Strings: A New and Improved Way to Format Strings in Python

The good news is that f-strings are here to save the day. They slice! They dice! They make julienne fries! Okay, they do none of those things, but they do make formatting easier. They joined the party in Python 3.6. You can read all about it in [PEP 498](https://www.python.org/dev/peps/pep-0498/), which was written by Eric V. Smith in August of 2015.

Also called “formatted string literals,” f-strings are string literals that have an f at the beginning and curly braces containing expressions that will be replaced with their values. The expressions are evaluated at runtime and then formatted using the \_\_format\_\_ protocol. As always, the [Python docs](https://docs.python.org/3/reference/lexical_analysis.html#f-strings) are your friend when you want to learn more.

Here are some of the ways f-strings can make your life easier.

### **Simple Syntax**

The syntax is similar to the one you used with str.format() but less verbose. Look at how easily readable this is:

>>>

>>> name = "Eric"

>>> age = 74

>>> f"Hello, {name}. You are {age}."

'Hello, Eric. You are 74.'

It would also be valid to use a capital letter F:

>>>

>>> F"Hello, {name}. You are {age}."

'Hello, Eric. You are 74.'

Do you love f-strings yet? I hope that, by the end of this article, you’ll answer [>>> F"Yes!"](https://twitter.com/dbader_org/status/992847368440561664).

### **Arbitrary Expressions**

Because f-strings are evaluated at runtime, you can put any and all valid Python expressions in them. This allows you to do some nifty things.

You could do something pretty straightforward, like this:

>>>

>>> f"{2 \* 37}"

'74'

But you could also call functions. Here’s an example:

>>>

>>> def to\_lowercase(input):

... return input.lower()

>>> name = "Eric Idle"

>>> f"{to\_lowercase(name)} is funny."

'eric idle is funny.'

You also have the option of calling a method directly:

>>>

>>> f"{name.lower()} is funny."

'eric idle is funny.'

You could even use objects created from classes with f-strings. Imagine you had the following class:

class Comedian:

def \_\_init\_\_(self, first\_name, last\_name, age):

self.first\_name = first\_name

self.last\_name = last\_name

self.age = age

def \_\_str\_\_(self):

return f"{self.first\_name} {self.last\_name} is {self.age}."

def \_\_repr\_\_(self):

return f"{self.first\_name} {self.last\_name} is {self.age}. Surprise!"

You’d be able to do this:

>>>

>>> new\_comedian = Comedian("Eric", "Idle", "74")

>>> f"{new\_comedian}"

'Eric Idle is 74.'

[The \_\_str\_\_() and \_\_repr\_\_() methods](https://realpython.com/operator-function-overloading/) deal with how objects are presented as strings, so you’ll need to make sure you include at least one of those methods in your class definition. If you have to pick one, go with \_\_repr\_\_() because it can be used in place of \_\_str\_\_().

The string returned by \_\_str\_\_() is the informal string representation of an object and should be readable. The string returned by \_\_repr\_\_() is the official representation and should be unambiguous. Calling str() and repr() is preferable to using \_\_str\_\_() and \_\_repr\_\_() directly.

By default, f-strings will use \_\_str\_\_(), but you can make sure they use \_\_repr\_\_() if you include the conversion flag !r:

>>>

>>> f"{new\_comedian}"

'Eric Idle is 74.'

>>> f"{new\_comedian!r}"

'Eric Idle is 74. Surprise!'

If you’d like to read some of the conversation that resulted in f-strings supporting full Python expressions, you can do so [here](https://mail.python.org/pipermail/python-ideas/2015-July/034726.html).

### **Multiline f-Strings**

You can have multiline strings:

>>>

>>> name = "Eric"

>>> profession = "comedian"

>>> affiliation = "Monty Python"

>>> message = (

... f"Hi {name}. "

... f"You are a {profession}. "

... f"You were in {affiliation}."

... )

>>> message

'Hi Eric. You are a comedian. You were in Monty Python.'

But remember that you need to place an f in front of each line of a multiline string. The following code won’t work:

>>>

>>> message = (

... f"Hi {name}. "

... "You are a {profession}. "

... "You were in {affiliation}."

... )

>>> message

'Hi Eric. You are a {profession}. You were in {affiliation}.'

If you don’t put an f in front of each individual line, then you’ll just have regular, old, garden-variety strings and not shiny, new, fancy f-strings.

If you want to spread strings over multiple lines, you also have the option of escaping a return with a \:

>>>

>>> message = f"Hi {name}. " \

... f"You are a {profession}. " \

... f"You were in {affiliation}."

...

>>> message

'Hi Eric. You are a comedian. You were in Monty Python.'

But this is what will happen if you use """:

>>>

>>> message = f"""

... Hi {name}.

... You are a {profession}.

... You were in {affiliation}.

... """

...

>>> message

'\n Hi Eric.\n You are a comedian.\n You were in Monty Python.\n'

Read up on indentation guidelines in [PEP 8](https://pep8.org/).

### **Speed**

The f in f-strings may as well stand for “fast.”

f-strings are faster than both %-formatting and str.format(). As you already saw, f-strings are expressions evaluated at runtime rather than constant values. Here’s an excerpt from the docs:

“F-strings provide a way to embed expressions inside string literals, using a minimal syntax. It should be noted that an f-string is really an expression evaluated at run time, not a constant value. In Python source code, an f-string is a literal string, prefixed with f, which contains expressions inside braces. The expressions are replaced with their values.” ([Source](https://www.python.org/dev/peps/pep-0498/#abstract))

At runtime, the expression inside the curly braces is evaluated in its own scope and then put together with the string literal part of the f-string. The resulting string is then returned. That’s all it takes.

Here’s a speed comparison:

>>>

>>> import timeit

>>> timeit.timeit("""name = "Eric"

... age = 74

... '%s is %s.' % (name, age)""", number = 10000)

0.003324444866599663

>>>

>>> timeit.timeit("""name = "Eric"

... age = 74

... '{} is {}.'.format(name, age)""", number = 10000)

0.004242089427570761

>>>

>>> timeit.timeit("""name = "Eric"

... age = 74

... f'{name} is {age}.'""", number = 10000)

0.0024820892040722242

As you can see, f-strings come out on top.

However, that wasn’t always the case. When they were first implemented, they had some [speed issues](https://stackoverflow.com/questions/37365311/why-are-literal-formatted-strings-so-slow-in-python-3-6-alpha-now-fixed-in-3-6) and needed to be made faster than str.format(). A special [BUILD\_STRING opcode](https://bugs.python.org/issue27078) was introduced.

## Python f-Strings: The Pesky Details

Now that you’ve learned all about why f-strings are great, I’m sure you want to get out there and start using them. Here are a few details to keep in mind as you venture off into this brave new world.

### **Quotation Marks**

You can use various types of quotation marks inside the expressions. Just make sure you are not using the same type of quotation mark on the outside of the f-string as you are using in the expression.

This code will work:

>>>

>>> f"{'Eric Idle'}"

'Eric Idle'

This code will also work:

>>>

>>> f'{"Eric Idle"}'

'Eric Idle'

You can also use triple quotes:

>>>

>>> f"""Eric Idle"""

'Eric Idle'

>>>

>>> f'''Eric Idle'''

'Eric Idle'

If you find you need to use the same type of quotation mark on both the inside and the outside of the string, then you can escape with \:

>>>

>>> f"The \"comedian\" is {name}, aged {age}."

'The "comedian" is Eric Idle, aged 74.'

### **Dictionaries**

Speaking of quotation marks, watch out when you are working with dictionaries. If you are going to use single quotation marks for the keys of the dictionary, then remember to make sure you’re using double quotation marks for the f-strings containing the keys.

This will work:

>>>

>>> comedian = {'name': 'Eric Idle', 'age': 74}

>>> f"The comedian is {comedian['name']}, aged {comedian['age']}."

The comedian is Eric Idle, aged 74.

But this will be a hot mess with a [syntax error](https://realpython.com/invalid-syntax-python/):

>>>

>>> comedian = {'name': 'Eric Idle', 'age': 74}

>>> f'The comedian is {comedian['name']}, aged {comedian['age']}.'

File "<stdin>", line 1

f'The comedian is {comedian['name']}, aged {comedian['age']}.'

^

SyntaxError: invalid syntax

If you use the same type of quotation mark around the dictionary keys as you do on the outside of the f-string, then the quotation mark at the beginning of the first dictionary key will be interpreted as the end of the string.

### **Braces**

In order to make a brace appear in your string, you must use double braces:

>>>

>>> f"{{70 + 4}}"

'{70 + 4}'

Note that using triple braces will result in there being only single braces in your string:

>>>

>>> f"{{{70 + 4}}}"

'{74}'

However, you can get more braces to show if you use more than triple braces:

>>>

>>> f"{{{{70 + 4}}}}"

'{{70 + 4}}'

### **Backslashes**

As you saw earlier, it is possible for you to use backslash escapes in the string portion of an f-string. However, you can’t use backslashes to escape in the expression part of an f-string:

>>>

>>> f"{\"Eric Idle\"}"

File "<stdin>", line 1

f"{\"Eric Idle\"}"

^

SyntaxError: f-string expression part cannot include a backslash

You can work around this by evaluating the expression beforehand and using the result in the f-string:

>>>

>>> name = "Eric Idle"

>>> f"{name}"

'Eric Idle'

### **Inline Comments**

Expressions should not include comments using the # symbol. You’ll get a syntax error:

>>>

>>> f"Eric is {2 \* 37 #Oh my!}."

File "<stdin>", line 1

f"Eric is {2 \* 37 #Oh my!}."

^

SyntaxError: f-string expression part cannot include '#'