Format with f-strings

Contents

F-strings, or formatted string literals, are a fairly simple way to format values using the **print** function. They are called **f-strings** because you need to prefix a string with the letter 'f' in order to get an f-string. The letter 'f' also indicates that these strings are used for formatting.

To use formatted string literals, begin a string with f or F before the opening quotation mark or triple quotation mark. Inside this string, you can write a Python expression between curly braces **{ }** characters that can refer to variables or literal values. **f-strings** support extensive modifiers that control the final appearance of the output string. Expressions in **f-strings** can be modified by a format specification.

Data Types

There are many ways to represent strings and numbers when using f-strings. The following table shows the ones most commonly used.

Type	Meaning

s	String format—this is the default type for strings.
d	Decimal Integer. Outputs the number in base 10.
n	Number. This is the same as **d** except that it uses the current locale setting to insert the appropriate number separator characters.
e	Exponent notation. Prints the number in scientific notation using the letter ‘e’ to indicate the exponent. The default precision is 6.
f	Fixed-point notation. Displays the number as a fixed-point number. The default precision is 6.
n	Number. This is the same as **g**, except that it uses the current locale setting to insert the appropriate number separator characters.
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

Alignment

There are several ways to align variables in **f-strings**.

Option	**Meaning**
<	Forces the field to be left-aligned within the available space (this is the default for most objects).
>	Forces the field to be right aligned within the available space (this is the default for numbers).
=	Forces the padding to be placed after the sign (if any) but before the digits. This is used for printing fields in the form ‘+000000120’. This alignment option is only valid for numeric types. It becomes the default when ‘0’ immediately precedes the field width.
^	Forces the field to be centered within the available space.

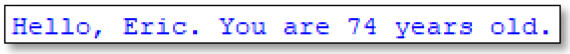
Formatting Numbers

Notice the syntax.

- f precedes what we want to print.
- Everything is enclosed with double quotes "". You can also use single quotes ''.
- Variables are enclosed by curly braces {}.

```
name = "Eric"
age = 74
# Using F-Strings formatting
print(f"Hello, {name}. You are {age} years old.")
```

Example run:



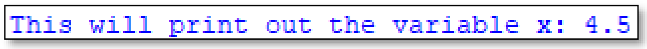
```
Hello, Eric. You are 74 years old.
```

F-strings Examples

The following is a basic example of the use of f-strings:

```
x = 4.5
print(f'This will print out the variable x: {x}')
```

Example run:



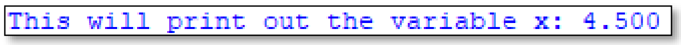
```
This will print out the variable x: 4.5
```

The variable, x, is enclosed in curly braces ({ }) and the f-string understands that x is a float and displays it as assigned.

If you want to change the number of decimals displayed, you would write this:

```
x = 4.5
print(f'This will print out the variable x: {x:.3f}')
```

Example run:

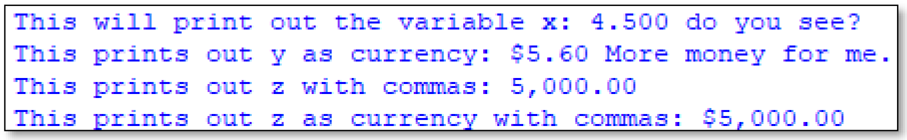


```
This will print out the variable x: 4.500
```

A colon is now added after the variable, which now controls the formatting of the float, specifying that three decimal places are to be displayed.

```
x = 4.5
y = 5.6
z = 5000
print(f'This will print out the variable x: {x:.3f} do you see?')
print(f'This prints out y as currency: ${y:.2f} More money for me.')
print(f'This prints out z with commas: {z:,.2f}')
print(f'This prints out z as currency with commas: ${z:,.2f}')
```

Example run:



```
This will print out the variable x: 4.500 do you see?
This prints out y as currency: $5.60 More money for me.
This prints out z with commas: 5,000.00
This prints out z as currency with commas: $5,000.00
```

This example combines more than one variable with text. The $ formats the variable as currency, the , displays with commas.

Field Width

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

The width specifier sets the width of the value. The value may be filled with spaces or other characters if the value is shorter than the specified width.

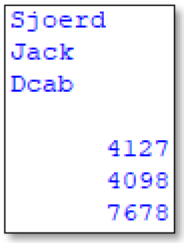
```
# format_width.py
# Illustrates f-string format width
#------------------------------------------------
for x in range(1, 11):
    print(f'{x:02} {x*x:3} {x*x*x:4}')
```

The example prints three columns. Each of the columns has a predefined width. The first column uses 0 to fill shorter values.

```
01   1    1
02   4    8
03   9   27
04  16   64
05  25  125
06  36  216
07  49  343
08  64  512
09  81  729
10 100 1000
```

```
table = ['Sjoerd','Jack','Dcab']
for name in table:
   print(f'{name:10}')
print()
table2 = [4127, 4098, 7678]
for num in table2:
   print(f'{num:10}')
```

Example run:



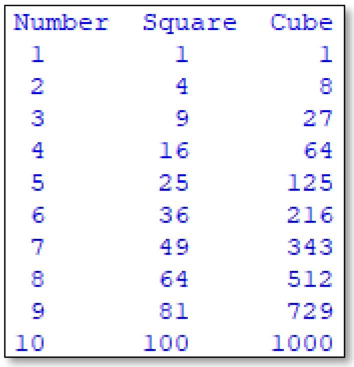
```
Sjoerd
Jack
Dcab

      4127
      4098
      7678
```

For the list, table, the strings are left-aligned in a field width of 10, while the numbers in the list, table2, are right-aligned in a field width of 10.

The following lines produce a tidily aligned set of columns giving integers and their squares and cubes. The tab character (\t) can also be used in an f-string to line up columns, particularly when column headings are used:

```
print(f'Number\tSquare\tCube')
for x in range(1, 11):
   print(f'{x:2d}\t\t{x*x:3d}\t\t{x*x*x:4d}')
```

Example run:

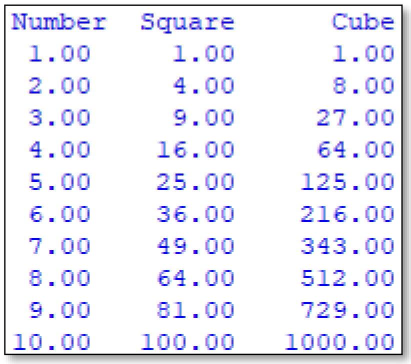


```
Number  Square  Cube
 1        1        1
 2        4        8
 3        9       27
 4       16       64
 5       25      125
 6       36      216
 7       49      343
 8       64      512
 9       81      729
10      100     1000
```

The following takes the previous program and converts x to a float(). This demonstrates formatting of floating-point numbers. This also demonstrates how the use of a value for width will enable the columns to line up. The number after the colon : sets the width.

```
print(f'Number\tSquare\t     Cube')
for x in range(1, 11):
    x = float(x)
    print(f'{x:5.2f}\t{x*x:6.2f}\t{x*x*x:8.2f}')
```

Example run:

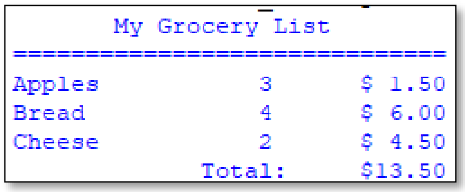


```
Number  Square      Cube
 1.00     1.00      1.00
 2.00     4.00      8.00
 3.00     9.00     27.00
 4.00    16.00     64.00
 5.00    25.00    125.00
 6.00    36.00    216.00
 7.00    49.00    343.00
 8.00    64.00    512.00
 9.00    81.00    729.00
10.00   100.00   1000.00
```

The following grocery list program demonstrates the use of strings, decimals, and floats, as well as tabs for a type of report that is often produced in a typical Python program. Notice the use of the dollar sign ($) just before the variables that are to be displayed as prices.

```
APPLES = .50
BREAD = 1.50
CHEESE = 2.25
numApples = 3
numBread = 4
numCheese = 2
prcApples = 3 * APPLES
prcBread = 4 * BREAD
prcCheese = 2 * CHEESE
strApples = 'Apples'
strBread = 'Bread'
strCheese = 'Cheese'
total = prcBread + prcBread + prcApples
# Center title over 30 spaces
print(f'{"My Grocery List":^30s}')
# Print 30 equals signs =
print(f'{"="*30}')
# Print a tab aligned grocery list
print(f'{strApples}\t{numApples:10d}\t${prcApples:>5.2f}')
print(f'{strBread}\t{numBread:10d}\t${prcBread:>5.2f}')
print(f'{strCheese}\t{numCheese:10d}\t${prcCheese:>5.2f}')
print(f'{"Total:":>19s}\t${total:>4.2f}')
```

Example run:



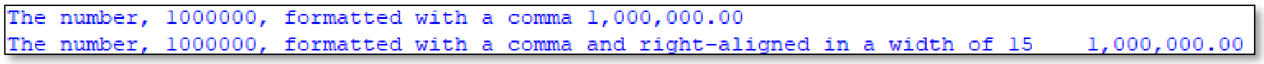
```
       My Grocery List
==============================
Apples           3       $ 1.50
Bread            4       $ 6.00
Cheese           2       $ 4.50
             Total:      $13.50
```

Formatting with Commas

Commas are often needed when formatting large numbers. The following shows the use of commas when numbers are aligned and when numbers do not required alignment.

```
number = 1000000
print (f'The number, 1000000, formatted with a comma {number:,.2f}')
print(f'The number, 1000000, formatted with a comma and right-aligned in a
width of 15 {number:>15,.2f}')
```

Example run:



```
The number, 1000000, formatted with a comma 1,000,000.00
The number, 1000000, formatted with a comma and right-aligned in a width of 15   1,000,000.00
```

Formatting Currency

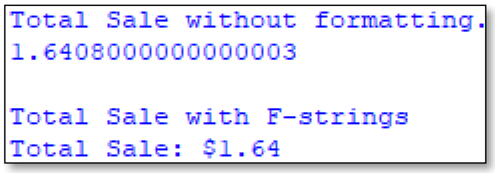
The following example will print out the second line of the program output to two decimal places with comma separators for thousands.

Each of these specifiers can be used separately in any combination.

1. The dollar sign $ specifies currency format. $1000
2. The comma , specifies comma formatting. 1,000
3. The .2f rounds the value to 2 decimal places. 1000.02
4. Put all three together: $1,000.02

```
SALES_TAX = .07
sale = 23.44
totalSale = SALES_TAX * sale
# Without formatting
print("\nTotal Sale without formatting.")
print(totalSale)
# Using F-Strings formatting
print("\nTotal Sale with F-strings")
print(f'Total Sale: ${totalSale:,.2f}')
```

Example run:



```
Total Sale without formatting.
1.6408000000000003

Total Sale with F-strings
Total Sale: $1.64
```

Justify Format

By default, strings are justified to the left. We can use the > character to justify the strings to the right. The > character follows the colon character.

```
# Name: format_justify.py
# Illustrates f-string width format
#----------------------------------------------
s1 = 'a'
s2 = 'ab'
s3 = 'abc'
s4 = 'abcd'

print(f'{s1:>10}')
print(f'{s2:>10}')
print(f'{s3:>10}')
print(f'{s4:>10}')
```

We have four strings of different lengths. We set the width of the output to ten characters. The values are justified to the right.

```
         a
        ab
       abc
      abcd
```

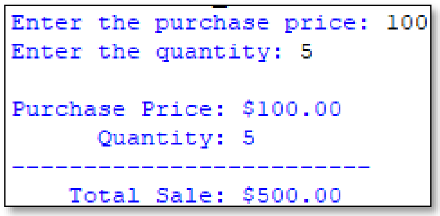
String Literal Formatting

The following line introduces a new concept for f-string formatting, how to format a string literal.

1. Surround the string literal with " → `"Purchase Price:"`
2. Add the f-string formatting curly braces `{}` and colon : → `{"Purchase Price:":}`
3. Add the format specifiers right align > and field width 15.
4. `{"Purchase Price:":>15}`

```
print (f'\n{"Purchase Price:":>15} ${purchase_price:,.2f}')
```

Example run:



```
Enter the purchase price: 100
Enter the quantity: 5

Purchase Price: $100.00
      Quantity: 5
-------------------------
    Total Sale: $500.00
```

Multiline Format

This example presents a multiline f-string. The f-strings are placed between round brackets; each of the strings is preceded with the `f` character.

```
# Name: format_multiline.py
# Purpose: Illustrates f-string multiline format
#-------------------------------------------------
name = 'John Doe'
age = 32
occupation = 'gardener'

msg = (
    f'Name: {name}\n'
    f'Age: {age}\n'
    f'Occupation: {occupation}'
)

print(msg)
```

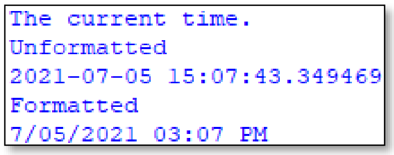
```
Name: John Doe
Age: 32
Occupation: gardener
```

Format Datetime

The following example formats a Python datetime object.

```
# format_datetime.py

import datetime

# Get the current datetime on the local computer
now = datetime.datetime.now()

print("The current time.\nUnformatted")
print(now)

# Format the time
time = f'{now:%m/%d/%Y %I:%M %p}'

# Strip out the leading 0's
time = time.lstrip("0")

print("Formatted")
print(time)
```

The example displays a formatted current datetime. The datetime format specifiers follow the `:` character.



```
The current time.
Unformatted
2021-07-05 15:07:43.349469
Formatted
7/05/2021 03:07 PM
```

Datetime Format Codes

Code	Definition	Example
%a	Weekday name abbreviated	Sun, Mon, Tue, …
%A	Weekday name full	Sunday, Monday, Tuesday, …
%b	Month name abbreviated	Jan, Feb, Mar, …
%B	Month name full	January, February, …
%c	A “random” date and time representation.	Fri Jan 15 16:34:00 1999

%d	Day of the month	[01, 31]
%f	Microsecond	[000000, 999999]
%H	Hour (24h)	[00, 23]
%I	Hour (12h)	[01, 12]
%j	Day of the year	[001, 366]
%m	Month	[01, 12]
%M	Minute	[00, 59]
%p	Locale’s equivalent of either AM or PM.	[AM, PM]
%S	Second	[00, 61]
%U	Week number of the year (Sunday first)	[00, 53]
%w	Weekday number (Sunday=0)	[0, 6]
%W	Week number of the year (Monday first)	[0, 53]
%x	Locale date	01/15/99
%X	Locale time	16:34:00
%y	Year without century	[00, 99]
%Y	Year with century	1999
%z	UTC offset in the form +HHMM or -HHMM or empty string	
%Z	Time zone name or empty string	
%%	A literal ‘%’ character.	