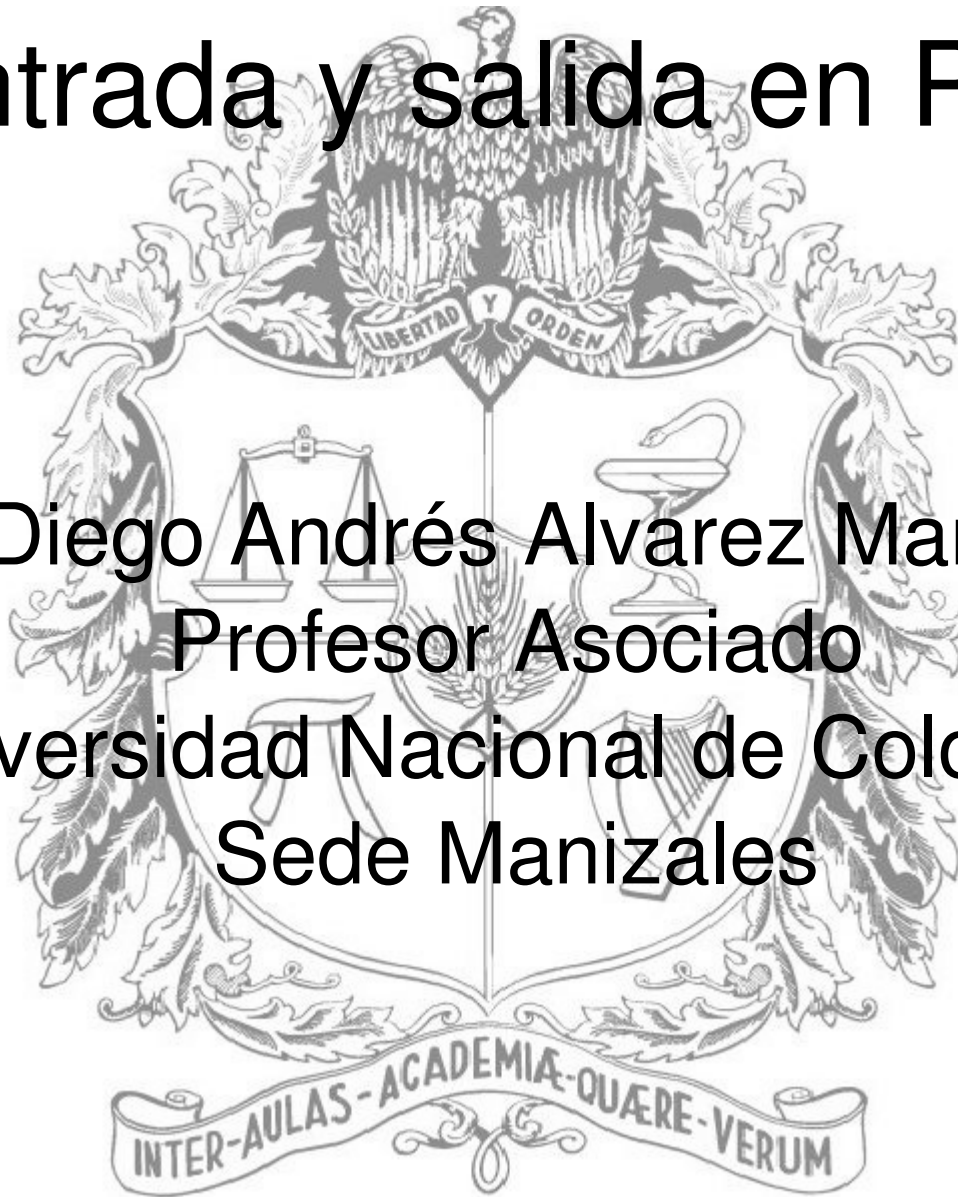


# 09 – Entrada y salida en Python 3

Diego Andrés Álvarez Marín  
Profesor Asociado  
Universidad Nacional de Colombia  
Sede Manizales



# Entrada salida

- Tipos de entrada
  - Mouse
  - Teclado
- Tipos de salida
  - Pantalla (gráficos y texto)
  - Sonido

# Lectura de datos desde el teclado con `input()`

## **`input`**(*[prompt]*)

If the *prompt* argument is present, it is written to standard output without a trailing newline. The function then reads a line from input, converts it to a string (stripping a trailing newline), and returns that. When EOF is read, `EOFError` is raised. Example:

```
>>> s = input(' --> ')
--> Monty Python's Flying Circus
>>> s
"Monty Python's Flying Circus"
```

If the `readline` module was loaded, then `input()` will use it to provide elaborate line editing and history features.

# Lectura de datos desde el teclado

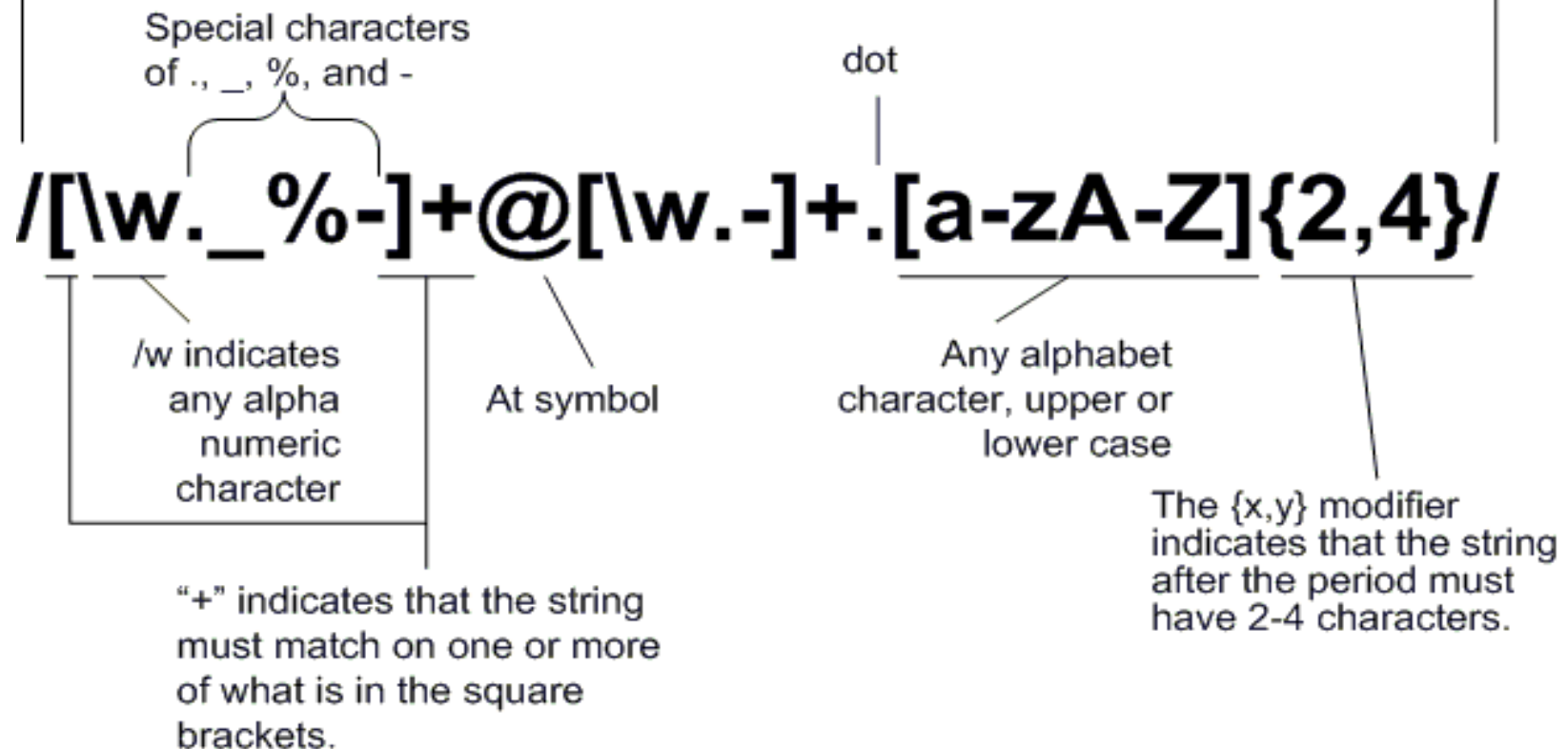
```
>>> x = input('Entre un número = ')
Entre un número = 123.34
>>> x
'123.34'
>>> x = input('Entre una lista = ') # observe que se almacenará en una cadena
Entre una lista = [1, 2, 3]
>>> x
'[1, 2, 3]'
>>> x = input() # sin texto
Hola
>>> x
'Hola'
>>> x = input() # Presionando Ctrl+C
Traceback (most recent call last):
  File "<pyshell#17>", line 1, in <module>
    x = input() # Presionando Ctrl+C
  File "/usr/lib/python3.4/idlelib/PyShell.py", line 1384, in readline
    line = self._line_buffer or self.shell.readline()
KeyboardInterrupt
```

Se lanzan unas excepciones

```
>>> x = input() # Presionando Ctrl+D
Traceback (most recent call last):
  File "<pyshell#18>", line 1, in <module>
    x = input() # Presionando Ctrl+D
EOFError: EOF when reading a line
```

En Linux: Ctrl+D  
En Windows: Ctrl+Z y luego ENTER

Ruby regular expression begin  
and end markers (forward slash)



**username @ domain . qualifier (com/net/tv/...)**

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!



BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!



IT'S HOPELESS!

EVERYBODY STAND BACK.



I KNOW REGULAR EXPRESSIONS.



NOTA: lenguajes como PERL, PYTHON, MATLAB tienen un muy buen soporte para expresiones regulares. Les aconsejo sinceramente aprender a manejar las expresiones regulares cuando tengan problemas con validar entradas de texto, o procesar una gran cantidad de datos en archivos. **Es una herramienta que los puede sacar de apuros en más de una ocasión.**

# Expresiones regulares en Python

- Ver:
  - <https://docs.python.org/3/library/re.html>
  - [http://en.wikipedia.org/wiki/Regular\\_expression](http://en.wikipedia.org/wiki/Regular_expression)

# Salida por pantalla con el comando print()

```
>>> nombre = input('Entre su nombre = ')
Entre su nombre = Pepito
>>> apellido = input('Entre su apellido = ')
Entre su apellido = Pérez
>>> edad = int(input('Entre su edad = '))
Entre su edad = 20
>>> print(nombre, apellido, 'tiene', edad, 'años')
Pepito Pérez tiene 20 años
>>> |
```



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

Print *objects* to the text stream *file*, separated by *sep* and followed by *end*. *sep*, *end* and *file*, if present, must be given as keyword arguments.

All non-keyword arguments are converted to strings like `str()` does and written to the stream, separated by *sep* and followed by *end*. Both *sep* and *end* must be strings; they can also be `None`, which means to use the default values. If no *objects* are given, `print()` will just write *end*.

The *file* argument must be an object with a `write(string)` method; if it is not present or `None`, `sys.stdout` will be used. Since printed arguments are converted to text strings, `print()` cannot be used with binary mode file objects. For these, use `file.write(...)` instead.

Whether output is buffered is usually determined by *file*, but if the *flush* keyword argument is true, the stream is forcibly flushed.

*Changed in version 3.3:* Added the *flush* keyword argument.

# print()

```
>>> for i in range(10):  
...     print(i, end=" ", "  
...     if i == 9:  
...         print("\b\b.")  
...  
0, 1, 2, 3, 4, 5, 6, 7, 8, 9.
```

El **end** se utiliza para evitar el `\n` después de la salida y/o para terminar la salida con una cadena diferente.

Un `print()` solo se utiliza para imprimir un `\n`

# Opciones para la interpolación de cadenas con Python 3

- Opción 1: Formateo con %
  - <https://docs.python.org/3/library/stdtypes.html#printf-style-string-formatting>
- Opción 2: Formateo con .format() (nuevo en Python 3.0 y luego se exportó a Python 2.7)
  - <https://docs.python.org/3/library/string.html#format-string-syntax>
- Opción 3: Formateo f-strings (nuevo en Python 3.6)
  - [https://docs.python.org/3/reference/lexical\\_analysis.html#f-strings](https://docs.python.org/3/reference/lexical_analysis.html#f-strings)
- Opción 4: Plantillas de cadenas
  - <https://docs.python.org/3/library/string.html>

# Opción 1: string interpolation

Funciona de forma similar al comando `sprintf()` de lenguaje C

```
>>> 'Hola %s' % "Pedro"
'Hola Pedro'
>>> 'Hola %d' % 123.4
'Hola 123'
>>> 'Hola %d' % 123
'Hola 123'
>>> 'Hola %f' % 123312.12312
'Hola 123312.123120'
>>> 'Hola %7.2f' % 123312.12312
'Hola 123312.12'
>>> x = 3.987982374198723
>>> x
3.987982374198723
>>> 'x = %7.2f' % x
'x =      3.99'
>>> nombre = 'Pepito'
>>> apellido = 'Pérez'
>>> edad = 20
>>> "%s %s tiene %d años" % nombre, apellido, edad
Traceback (most recent call last):
  File "<pyshell#11>", line 1, in <module>
    "%s %s tiene %d años" % nombre, apellido, edad
TypeError: not enough arguments for format string
>>> "%s %s tiene %d años" % (nombre, apellido, edad)
'Pepito Pérez tiene 20 años'
>>> print("%s %s tiene %d años" % (nombre, apellido, edad))
Pepito Pérez tiene 20 años
```

# Pasando los argumentos con un diccionario

```
In [10]: errno = 50159747054
```

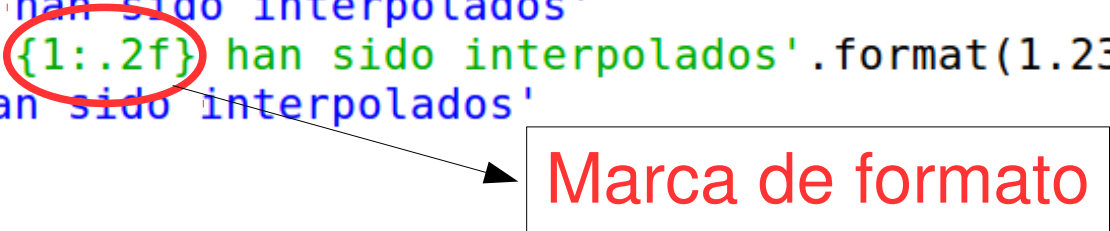
```
In [11]: name = 'Bob'
```

```
In [12]: 'Hey %(name)s, there is a 0x%(errno)x error!' % \
...: { "name": name, "errno": errno }
```

```
Out[12]: 'Hey Bob, there is a 0xbadc0ffee error!'
```

# Opción 2: interpolación de cadenas con el método str.format()

```
>>> 'El número {0} has sido interpolado'.format(1.234)
'El número 1.234 has sido interpolado'
>>> 'Los números {0} y {1} han sido interpolados'.format(1.234, 9.999)
'Los números 1.234 y 9.999 han sido interpolados'
>>> 'Los números {1} y {0} han sido interpolados'.format(1.234, 9.999)
'Los números 9.999 y 1.234 han sido interpolados'
>>> 'Los números {0:.1f} y {1:.2f} han sido interpolados'.format(1.234, 9.999)
'Los números 1.2 y 10.00 han sido interpolados'
>>>
>>> nombre = 'Pepito'
>>> profesión = 'Ingeniero Civil'
>>> '{1} tu eres {0}. ¿Cierto?'.format(profesión, nombre)
'Pepito tu eres Ingeniero Civil. ¿Cierto?'
>>> '{} tu eres {}'.format(nombre, profesión)
'Pepito tu eres Ingeniero Civil. ¿Cierto?'
>>> '{xxx} tu eres {yyy}. ¿Cierto?'.format(yyy=profesión, xxx=nombre)
'Pepito tu eres Ingeniero Civil. ¿Cierto?'
>>> '{1} tu eres {yyy}. ¿Cierto?'.format(yyy=profesión, nombre)
SyntaxError: non-keyword arg after keyword arg
>>> '{0} tu eres {yyy}. ¿Cierto?'.format(nombre, yyy=profesión)
'Pepito tu eres Ingeniero Civil. ¿Cierto?'
>>> '{1} {0} es {yyy}.'.format('Pérez', nombre, yyy=profesión)
'Pepito Pérez es Ingeniero Civil.'
>>> |
```



Marca de formato

Ver : <https://docs.python.org/3/library/stdtypes.html#str.format>

```
>>> 'Una {cadena} entre {0} corchetes'.format(123)
```

```
Traceback (most recent call last):
```

```
File "<pyshell#0>", line 1, in <module>
```

```
    'Una {cadena} entre {0} corchetes'.format(123)
```

```
KeyError: 'cadena'
```

```
>>> 'Una {{cadena}} entre {{{0}}}' corchetes'.format(123)
```

```
'Una {cadena} entre {123} corchetes'
```

# Las marcas de formato

`{[field_name] [!conversion] [:format_spec]}`

- **field\_name** (argumento opcional) especifica el objeto cuyo valor debe ser formateado e insertado en la cadena.
- **conversion** (argumento opcional): se precede por el `!`. No abordaremos este parámetro en el curso.
- **format\_spec** (argumento opcional) se precede por un `:`. Especifica la forma de representar el `field_name` en la cadena.

Esta diapositiva y las siguientes son tomadas de:

- <https://docs.python.org/3/library/string.html#format-string-syntax>
- <https://docs.python.org/3/library/string.html#format-examples>



{[**field\_name**] [!conversion] [:format\_spec]}

- **field\_name** (argumento opcional) especifica el objeto cuyo valor debe ser formateado e insertado en la cadena.

```
>>> 'Artículo: {a:5d}, Precio: {p:8.2f}'.format(a=453, p=59.058)
'Artículo:   453, Precio:   59.06'
>>> 'Artículo: {:_5d}, Precio: {:_8.2f}'.format(453, 59.058)
'Artículo:   453, Precio:   59.06'
>>> 'Artículo: {0:5d}, Precio: {1:8.2f}'.format(453, 59.058)
'Artículo:   453, Precio:   59.06'
```

The positional argument specifiers can be omitted, so '{} {}' is equivalent to '{0} {1}'

# Ejemplos

Accessing arguments by position:

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'
>>> '{} , {} , {}'.format('a', 'b', 'c') # 3.1+ only
'a, b, c'
>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'
>>> '{2}, {1}, {0}'.format(*'abc')        # unpacking argument sequence
'c, b, a'
>>> '{0}{1}{0}'.format('abra', 'cad')    # arguments' indices can be repeated
'abracadabra'
```

Accessing arguments by name:

```
>>> 'Coordinates: {latitude}, {longitude}'.format(latitude='37.24N', longitude='-115.81W')
'Coordinates: 37.24N, -115.81W'
>>> coord = {'latitude': '37.24N', 'longitude': '-115.81W'}
>>> 'Coordinates: {latitude}, {longitude}'.format(**coord)
'Coordinates: 37.24N, -115.81W'
```

# Ejemplos

Accessing arguments' attributes:

```
>>> c = 3-5j
>>> ('The complex number {0} is formed from the real part {0.real} '
...  'and the imaginary part {0.imag}').format(c)
'The complex number (3-5j) is formed from the real part 3.0 and the imaginary part -5.0.'
>>> class Point:
...     def __init__(self, x, y):
...         self.x, self.y = x, y
...     def __str__(self):
...         return 'Point({self.x}, {self.y})'.format(self=self)
...
>>> str(Point(4, 2))
'Point(4, 2)'
```

Accessing arguments' items:

```
>>> coord = (3, 5)
>>> 'X: {0[0]}; Y: {0[1]}'.format(coord)
'X: 3; Y: 5'
```

{[field\_name] [!conversion] [:format\_spec]}  
[[fill]align][sign][#][0][width][,][.precision][type]

fill	::=	<any character>
align	::=	"<"   ">"   "="   "^"
sign	::=	"+"   "-"   " "
width	::=	integer
precision	::=	integer
type	::=	"b"   "c"   "d"   "e"   "E"   "f"   "F" "g"   "G"   "n"   "o"   "s"   "x"   "X"   "%"

{[field\_name] [!conversion] [:format\_spec]  
[[fill]align][sign][#][0][width][,][.precision][type]}

If a valid *align* value is specified, it can be preceded by a *fill* character that can be any character and defaults to a space if omitted. Note that it is not possible to use `{` and `}` as *fill* char while using the `str.format()` method; this limitation however doesn't affect the `format()` function.

The meaning of the various alignment options is as follows:

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.
'^'	Forces the field to be centered within the available space.

Note that unless a minimum field width is defined, the field width will always be the same size as the data to fill it, so that the alignment option has no meaning in this case.

# Ejemplo

Aligning the text and specifying a width:

```
>>> '{:<30}'.format('left aligned')
'left aligned'
>>> '{:>30}'.format('right aligned')
'right aligned'
>>> '{:^30}'.format('centered')
'centered'
>>> '{:*^30}'.format('centered')  # use '*' as a fill char
'*****centered*****'
```

{[field\_name] [!conversion] [:format\_spec]}  
[[fill]align][sign][#][0][width][,][.precision][type]

The *sign* option is only valid for number types, and can be one of the following:

Option	Meaning
'+'	indicates that a sign should be used for both positive as well as negative numbers.
'-'	indicates that a sign should be used only for negative numbers (this is the default behavior).
space	indicates that a leading space should be used on positive numbers, and a minus sign on negative numbers.

The '#' option causes the “alternate form” to be used for the conversion. The alternate form is defined differently for different types. This option is only valid for integer, float, complex and Decimal types. For integers, when binary, octal, or hexadecimal output is used, this option adds the prefix respective '0b', '0o', or '0x' to the output value. For floats, complex and Decimal the alternate form causes the result of the conversion to always contain a decimal-point character, even if no digits follow it. Normally, a decimal-point character appears in the result of these conversions only if a digit follows it. In addition, for 'g' and 'G' conversions, trailing zeros are not removed from the result.

# Ejemplo

Replacing `%+f`, `%-f`, and `% f` and specifying a sign:

```
>>> '{:+f}; {:+f}'.format(3.14, -3.14) # show it always
'+3.140000; -3.140000'
>>> '{: f}; {: f}'.format(3.14, -3.14) # show a space for positive numbers
' 3.140000; -3.140000'
>>> '{:-f}; {:-f}'.format(3.14, -3.14) # show only the minus -- same as '{:f}; {:f}'
'3.140000; -3.140000'
```

Replacing `%x` and `%o` and converting the value to different bases:

```
>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'
```



{[field\_name] [!conversion] [:format\_spec]}  
[[fill]align][sign][#][0][width][,][.precision][type]

The `' , '` option signals the use of a comma for a thousands separator. For a locale aware separator, use the `' n '` integer presentation type instead.

*Changed in version 3.1:* Added the `' , '` option (see also [PEP 378](#)).

*width* is a decimal integer defining the minimum field width. If not specified, then the field width will be determined by the content.

Preceding the *width* field by a zero (`' 0 '`) character enables sign-aware zero-padding for numeric types. This is equivalent to a *fill* character of `' 0 '` with an *alignment* type of `' = '`.

The *precision* is a decimal number indicating how many digits should be displayed after the decimal point for a floating point value formatted with `' f '` and `' F '`, or before and after the decimal point for a floating point value formatted with `' g '` or `' G '`. For non-number types the field indicates the maximum field size - in other words, how many characters will be used from the field content. The *precision* is not allowed for integer values.

# Ejemplo

Using the comma as a thousands separator:

```
>>> '{:,}'.format(1234567890)
'1,234,567,890'
```

Expressing a percentage:

```
>>> points = 19
>>> total = 22
>>> 'Correct answers: {:.2%}'.format(points/total)
'Correct answers: 86.36%'
```

El 0 sustituye los espacios en blanco por un 0:

```
>>> 'El_{0:010}_formateado.'.format(123)
'El_00000000123_formateado.'
```

{[field\_name] [!conversion] [:format\_spec]}  
[[fill]align][sign][#][0][width][,][.precision][type]

Finally, the *type* determines how the data should be presented.

The available string presentation types are:

Type	Meaning
's'	String format. This is the default type for strings and may be omitted.
None	The same as 's'.

{[field\_name] [!conversion] [:format\_spec]}  
[[fill]align][sign][#][0][width][,][.precision][type]

The available integer presentation types are:

Type	Meaning
'b'	Binary format. Outputs the number in base 2.
'c'	Character. Converts the integer to the corresponding unicode character before printing.
'd'	Decimal Integer. Outputs the number in base 10.
'o'	Octal format. Outputs the number in base 8.
'x'	Hex format. Outputs the number in base 16, using lower- case letters for the digits above 9.
'X'	Hex format. Outputs the number in base 16, using upper- case letters for the digits above 9.
'n'	Number. This is the same as 'd', except that it uses the current locale setting to insert the appropriate number separator characters.
None	The same as 'd'.

# Ejemplo

Replacing `%x` and `%o` and converting the value to different bases:

```
>>> # format also supports binary numbers
>>> "int: {0:d}; hex: {0:x}; oct: {0:o}; bin: {0:b}".format(42)
'int: 42; hex: 2a; oct: 52; bin: 101010'
>>> # with 0x, 0o, or 0b as prefix:
>>> "int: {0:d}; hex: {0:#x}; oct: {0:#o}; bin: {0:#b}".format(42)
'int: 42; hex: 0x2a; oct: 0o52; bin: 0b101010'
```

{[field\_name] [!conversion] [:format\_spec]}  
[[fill]align][sign][#][0][width][,][.precision][type]

In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except 'n' and None). When doing so, `float()` is used to convert the integer to a floating point number before formatting.

The available presentation types for floating point and decimal values are:

Type	Meaning
'e'	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent. The default precision is 6.
'E'	Exponent notation. Same as 'e' except it uses an upper case 'E' as the separator character.
'f'	Fixed point. Displays the number as a fixed-point number. The default precision is 6.
'F'	Fixed point. Same as 'f', but converts <code>nan</code> to <code>NAN</code> and <code>inf</code> to <code>INF</code> .
'g'	<p>General format. For a given precision <code>p &gt;= 1</code>, this rounds the number to <code>p</code> significant digits and then formats the result in either fixed-point format or in scientific notation, depending on its magnitude.</p> <p>Positive and negative infinity, positive and negative zero, and nans, are formatted as <code>inf</code>, <code>-inf</code>, <code>0</code>, <code>-0</code> and <code>nan</code> respectively, regardless of the precision.</p> <p>A precision of <code>0</code> is treated as equivalent to a precision of <code>1</code>. The default precision is 6.</p>

{[field\_name] [!conversion] [:format\_spec]}  
[[fill]align][sign][#][0][width][,][.precision][type]

In addition to the above presentation types, integers can be formatted with the floating point presentation types listed below (except 'n' and None). When doing so, `float()` is used to convert the integer to a floating point number before formatting.

The available presentation types for floating point and decimal values are:

Type	Meaning
'G'	General format. Same as 'g' except switches to 'E' if the number gets too large. The representations of infinity and NaN are uppercased, too.
'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.
None	Similar to 'g', except that fixed-point notation, when used, has at least one digit past the decimal point. The default precision is as high as needed to represent the particular value. The overall effect is to match the output of <code>str()</code> as altered by the other format modifiers.



Nesting arguments and more complex examples:

```
>>> for align, text in zip('<^>', ['left', 'center', 'right']):
...     '{0:{fill}{align}16}'.format(text, fill=align, align=align)
...
'left<<<<<<<<<<'
'^^^^^center^^^^^'
'>>>>>>>>>>right'
>>>
>>> octets = [192, 168, 0, 1]
>>> '{:02X}{:02X}{:02X}{:02X}'.format(*octets)
'C0A80001'
>>> int(_, 16)
3232235521
>>>
>>> width = 5
>>> for num in range(5,12):
...     for base in 'dXob':
...         print('{0:{width}{base}}'.format(num, base=base, width=width), end=' ')
...     print()
...
5      5      5      101
6      6      6      110
7      7      7      111
8      8      10     1000
9      9      11     1001
10     A      12     1010
11     B      13     1011
```



Podemos combinar los diferentes elementos y controlar con mucha precisión el formato:

```
>>> 'El_{0:+#016b}_formateado.'.format(123)↵  
'El_+0b0000001111011_formateado.'
```

No nos extenderemos tanto con las posibilidades de formato para números en coma flotante, pero no nos resistimos a poner algunos ejemplos que deberías analizar:

```
>>> 'El_{0:e}_formateado.'.format(123.45)↵  
'El_1.234500e+02_formateado.'  
>>> 'El_{0:g}_formateado.'.format(123.45)↵  
'El_123.45_formateado.'  
>>> 'El_{0:+e}_formateado.'.format(123.45)↵  
'El_+1.234500e+02_formateado.'  
>>> 'El_{0:10e}_formateado.'.format(123.45)↵  
'El_1.234500e+02_formateado.'  
>>> 'El_{0:10.4g}_formateado.'.format(123.45)↵  
'El_123.45_formateado.'  
>>> 'El_{0:10.2g}_formateado.'.format(123.45)↵  
'El_1.2e+02_formateado.'  
>>> 'El_{0:10.1g}_formateado.'.format(123.45)↵  
'El_1e+02_formateado.'  
>>> 'El_{0:10.0g}_formateado.'.format(123.45)↵  
'El_1e+02_formateado.'  
>>> 'El_{0:.1%}_formateado.'.format(123.45)↵  
'El_12345.0%_formateado.'
```

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print('{0:10} ==> {1:10d}'.format(name, phone))
...
Jack          ==>      4098
Dcab          ==>      7678
Sjoerd        ==>      4127
```

<https://docs.python.org/3/library/string.html#formatstrings>

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

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

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

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

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

For a complete overview of string formatting with `str.format()`, see *Format String Syntax*.

# Opción 3: formateo con f-strings

Tienen una sintaxis similar a `str.format()`:

```
>>> name = "Eric"
>>> age = 74
>>> f"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

```
>>> f"{2 * 37}"
'74'
```

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

Python

```
>>> def to_lowercase(input):
...     return input.lower()

>>> name = "Eric Idle"
>>> f"{to_lowercase(name)} is funny."
'eric idle is funny.'
```

Python

```
>>> F"Hello, {name}. You are {age}."
'Hello, Eric. You are 74.'
```

```
>>> def greet(name, question):
...     return f"Hello, {name}! How's it {question}?"
...
>>> greet('Bob', 'going')
'Hello, Bob! How's it going?'
```

# f-strings de varias líneas

```
>>> 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.'
```

```
>>> message = (
...     f"Hi {name}. "
...     "You are a {profession}. "
...     "You were in {affiliation}."
... )
>>> message
'Hi Eric. You are a {profession}. You were in {affiliation}.'
```

# f-strings de varias líneas

```
>>> 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 """:

Python

```
>>> 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'
```

Here's a speed comparison:

Python

```
>>> import timeit
>>> timeit.timeit("""name = "Eric"
... age = 74
... '%s is %s.' % (name, age)""", number = 10000)
0.003324444866599663
```

Python

```
>>> timeit.timeit("""name = "Eric"
... age = 74
... '{} is {}'.format(name, age)""", number = 10000)
0.004242089427570761
```

Python

```
>>> timeit.timeit("""name = "Eric"
... age = 74
... f'{name} is {age}.'""", number = 10000)
0.0024820892040722242
```

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

Here's a speed comparison:

Python

```
>>> import timeit
>>> timeit.timeit("""name = "Eric"
... age = 74
... '%s is %s.' % (name, age)""", number = 10000)
0.003324444866599663
```

Python

```
>>> timeit.timeit("""name = "Eric"
... age = 74
... '{} is {}'.format(name, age)""", number = 10000)
0.004242089427570761
```

Python

```
>>> timeit.timeit("""name = "Eric"
... age = 74
... f'{name} is {age}.'""", number = 10000)
0.0024820892040722242
```

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

# f-strings con diccionarios

This will work:

Python

```
>>> 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:

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
```



```
In [24]: a = 453
```

```
In [25]: p = 59.058
```

```
In [26]: f'Artículo: {a:5d}, Precio: {p:8.2f}'
```

```
Out[26]: 'Artículo: 453, Precio: 59.06'
```

```
In [27]: f'Artículo: {a:5d}, Precio: {p:.2f}'
```

```
Out[27]: 'Artículo: 453, Precio: 59.06'
```

```
In [28]: ancho = 10
```

```
In [29]: precision = 4
```

```
In [30]: valor = 12.34567
```

```
In [31]: f'resultado: {valor:{ancho}.{precision}}'
```

```
Out[31]: 'resultado: 12.35'
```

```
In [32]: f'La tercera parte es el {1/3*100:.2f} %'
```

```
Out[32]: 'La tercera parte es el 33.33 %'
```

# Opción 4: Plantillas de cadenas

```
>>> from string import Template
>>> t = Template('Hey, $name!')
>>> t.substitute(name=name)
'Hey, Bob!'
```

Nota: esta opción se debe usar en caso que se tenga información delicada que podría ser hackeada. Ver por ejemplo:

<https://realpython.com/python-string-formatting/>  
<http://lucumr.pocoo.org/2016/12/29/careful-with-str-format/>

You see here that we need to import the `Template` class from Python's built-in `string` module. Template strings are not a core language feature but they're supplied by the `string` module in the standard library.

Another difference is that template strings don't allow format specifiers. So in order to get the previous error string example to work, you'll need to manually transform the `int` error number into a hex-string:

Python

```
>>> templ_string = 'Hey $name, there is a $error error!'
>>> Template(templ_string).substitute(
...     name=name, error=hex(errno))
'Hey Bob, there is a 0xbadc0ffee error!'
```

# Referencias

- Wikipedia
- <http://www.inventwithpython.com/>
- <http://www.diveintopython3.net/>
- Documentación de Python:
  - <https://docs.python.org/3/tutorial/index.html>
  - <https://docs.python.org/3/>
- Marzal Varó, Andrés; Gracia Luengo, Isabel; García Sevilla, Pedro (2014). Introducción a la programación con Python 3. Disponible en: <http://dx.doi.org/10.6035/Sapientia93>
- <https://realpython.com/python-f-strings/>
- <https://realpython.com/python-string-formatting/>