Chapter 9. Strings and Things

Python's `str` type implements Unicode text strings with operators, built-in functions, methods, and dedicated modules. The somewhat similar `bytes` type represents arbitrary binary data as a sequence of bytes, also known as a *bytestring* or *byte string*. Many textual operations are possible on objects of either type: since these types are immutable, methods mostly create and return a new string unless returning the subject string unchanged. A mutable sequence of bytes can be represented as a `bytearray`, briefly introduced in **"bytearray objects"**.

This chapter first covers the methods available on these three types, then discusses the string module and string formatting (including formatted string literals), followed by the `textwrap`, `pprint`, and `reprlib` modules. Issues related specifically to Unicode are covered at the end of the chapter.

Methods of String Objects

`str`, `bytes`, and `bytearray` objects are sequences, as covered in **"Strings"**; of these, only `bytearray` objects are mutable. All immutable-sequence operations (repetition, concatenation, indexing, and slicing) apply to instances of all three types, returning a new object of the same type. Unless otherwise specified in **Table 9-1**, methods are present on objects of all three types. Most methods of `str`, `bytes`, and `bytearray` objects return values of the same type, or are specifically intended to convert among representations.

Terms such as "letters," "whitespace," and so on refer to the corresponding attributes of the `string` module, covered in the following section. Although `bytearray` objects are mutable, their methods returning a bytearray result do not mutate the object but instead return a newly created bytearray, even when the result is the same as the subject string.

For brevity, the term bytes in the following table refers to both bytes and bytearray objects. Take care when mixing these two types, however: while they are generally interoperable, the type of the result usually depends on the order of the operands.

In **Table 9-1**, since integer values in Python can be arbitrarily large, for conciseness we use sys.maxsize for integer default values to mean, in practice, "integer of unlimited magnitude."

Table 9-1. Significant str and bytes methods

capitalize	*s*.capitalize() Returns a copy of *s* where the first character, if a letter uppercase, and all other letters, if any, are lowercase.
casefold	*s*.casefold() **str only**. Returns a string processed by the algorithm described in **section 3.13 of the Unicode standard**. Th similar to *s*.lower (described later in this table) but als into account equivalences such as that between the Ge 'ß' and 'ss', and is thus better for case-insensitive ma when working with text that can include more than jus basic ASCII characters.
center	*s*.center(*n*, *fillchar*=' ', /) Returns a string of length max(len(*s*), *n*), with a copy the central part, surrounded by equal numbers of copi character *fillchar* on both sides. The default fillcha space. For example, 'ciao'.center(2) is 'ciao' and 'x'.center(4, '_') is '_x__'.
count	*s*.count(*sub*, *start*=0, *end*=sys.maxsize, /) Returns the number of nonoverlapping occurrences of substring *sub* in *s*[*start*:*end*].

`decode`	`s.decode(encoding='utf-8', errors='strict')` **`bytes` only**. Returns a `str` object decoded from the `byt` according to the given encoding. `errors` specifies how handle decoding errors: `'strict'` causes errors to rais `UnicodeError` exceptions; `'ignore'` ignores the malfor values, while `'replace'` replaces them with question n (see **"Unicode"** for details). Other values can be registe `codecs.register_error`, covered in Table 9-10.
`encode`	`s.encode(encoding='utf-8', errors='strict')` **`str` only**. Returns a `bytes` object obtained from `str` *`s`* w given encoding and error handling. See **"Unicode"** for details.
`endswith`	`s.endswith(suffix, start=0, end=sys.maxsize, /)` Returns **`True`** when `s[start:end]` ends with the string otherwise, returns **`False`**. *`suffix`* can be a tuple of strir which case `endswith` returns **`True`** when `s[start:end]` with any one of them.
`expandtabs`	`s.expandtabs(tabsize=8)` Returns a copy of *`s`* where each tab character is change one or more spaces, with tab stops every *`tabsize`* chara
`find`	`s.find(sub, start=0, end=sys.maxsize, /)` Returns the lowest index in *`s`* where substring *`sub`* is fo such that *`sub`* is entirely contained in `s[start:end]`. Fo example, `'banana'.find('na')` returns `2`, as does `'banana'.find('na', 1)`, while `'banana'.find('na',` returns `4`, as does `'banana'.find('na', -2)`. `find` retu when *`sub`* is not found.
`format`	`s.format(*args, **kwargs)` **`str` only**. Formats the positional and named argument

	according to formatting instructions contained in the s See **“String Formatting”** for further details.
`format_map`	*s*.`format_map(mapping)` **`str` only**. Formats the mapping argument according to formatting instructions contained in the string *s*. Equiv *s*`.format(**mapping)` but uses the mapping directly. Se **“String Formatting”** for formatting details.
`index`	*s*`.index(`*sub*`,` *start*`=0,` *end*`=sys.maxsize, /)` Like `find`, but raises `ValueError` when *sub* is not found
`isalnum`	*s*`.isalnum()` Returns **`True`** when `len(`*s*`)` is greater than `0` and all cha in *s* are Unicode letters or digits. When *s* is empty, or w least one character of *s* is neither a letter nor a digit, `is` returns **`False`**.
`isalpha`	*s*`.isalpha()` Returns **`True`** when `len(`*s*`)` is greater than `0` and all cha in *s* are letters. When *s* is empty, or when at least one character of *s* is not a letter, `isalpha` returns **`False`**.
`isascii`	*s*`.isascii()` Returns **`True`** when the string is empty or all characters string are ASCII, or **`False`** otherwise. ASCII characters h codepoints in the range U+0000–U+007F.
`isdecimal`	*s*`.isdecimal()` **`str` only**. Returns **`True`** when `len(`*s*`)` is greater than `0` a characters in *s* can be used to form decimal-radix num This includes Unicode characters defined as Arabic dig
`isdigit`	*s*`.isdigit()` Returns **`True`** when `len(`*s*`)` is greater than `0` and all cha

	in *s* are Unicode digits. When *s* is empty, or when at lea character of *s* is not a Unicode digit, `isdigit` returns **`Fa`**
`isidentifier`	`s.isidentifier()` **`str` only**. Returns **`True`** when *s* is a valid identifier acco to the Python language's definition; keywords also satis definition, so, for example, `'class'.isidentifier()` r **`True`**.
`islower`	`s.islower()` Returns **`True`** when all letters in *s* are lowercase. When contains no letters, or when at least one letter of *s* is uppercase, `islower` returns **`False`**.
`isnumeric`	`s.isnumeric()` **`str` only**. Similar to `s.isdigit()`, but uses a broader de of numeric characters that includes all characters defi numeric in the Unicode standard (such as fractions).
`isprintable`	`s.isprintable()` **`str` only**. Returns **`True`** when all characters in *s* are spa (`'\x20'`) or are defined in the Unicode standard as prin Because the null string contains no unprintable charac `''.isprintable()` returns **`True`**.
`isspace`	`s.isspace()` Returns **`True`** when `len(s)` is greater than `0` and all cha in *s* are whitespace. When *s* is empty, or when at least character of *s* is not whitespace, `isspace` returns **`False`**
`istitle`	`s.istitle()` Returns **`True`** when the string *s* is *titlecased*: i.e., with a letter at the start of every contiguous sequence of lette all other letters lowercase (e.g., `'King Lear'.istitle(` returns **`True`**). When *s* contains no letters, or when at l letter of *s* violates the title case condition, `istitle` retu

	`False` (e.g., `'1900'.istitle()` and `'Troilus and Cressida'.istitle()` return `False`).
`isupper`	`s.isupper()` Returns `True` when all letters in *s* are uppercase. Wher contains no letters, or when at least one letter of *s* is lowercase, `isupper` returns `False`.
`join`	`s.join(seq, /)` Returns the string obtained by concatenating the items separated by copies of *s* (e.g., `''.join(str(x) for x in range(7))` returns `'0123456'` and `'x'.join('aeiou')` `'axexixoxu'`).
`ljust`	`s.ljust(n, fillchar=' ', /)` Returns a string of length `max(len(s),n)`, with a copy the start, followed by zero or more trailing copies of ch *fillchar*.
`lower`	`s.lower()` Returns a copy of *s* with all letters, if any, converted to lowercase.
`lstrip`	`s.lstrip(x=string.whitespace, /)` Returns a copy of *s* after removing any leading charact found in string *x*. For example, `'banana'.lstrip('ab'` returns `'nana'`.
`removeprefix`	`s.removeprefix(prefix, /)` **3.9+** When *s* begins with *prefix*, returns the remaind otherwise, returns *s*.
`removesuffix`	`s.removesuffix(suffix, /)` **3.9+** When *s* ends with *suffix*, returns the rest of *s*; otherwise, returns *s*.

`replace`	*s*`.replace(`*old*`, `*new*`, `*count*`=sys.maxsize, /)` Returns a copy of *s* with the first *count* (or fewer, if the fewer) nonoverlapping occurrences of substring *old* re by string *new* (e.g., `'banana'.replace('a', 'e', 2)` re `'benena'`).
`rfind`	*s*`.rfind(`*sub*`, `*start*`=0, `*end*`=sys.maxsize, /)` Returns the highest index in *s* where substring *sub* is fo such that *sub* is entirely contained in *s*`[`*start*`:`*end*`]`. `rf` returns `-1` if *sub* is not found.
`rindex`	*s*`.rindex(`*sub*`, `*start*`=0, `*end*`=sys.maxsize, /)` Like `rfind`, but raises `ValueError` if *sub* is not found.
`rjust`	*s*`.rjust(`*n*`, `*fillchar*`=' ', /)` Returns a string of length `max(len(`*s*`),`*n*`)`, with a copy o the end, preceded by zero or more leading copies of ch *fillchar*.
`rstrip`	*s*`.rstrip(`*x*`=string.whitespace, /)` Returns a copy of *s*, removing trailing characters that a found in string *x*. For example, `'banana'.rstrip('ab'` returns `'banan'`.
`split`	*s*`.split(sep=None, maxsplit=sys.maxsize)` Returns a list *L* of up to `maxsplit+1` strings. Each item o “word” from *s*, where string `sep` separates words. Whe more than `maxsplit` words, the last item of *L* is the sub of *s* that follows the first `maxsplit` words. When `sep` is any string of whitespace separates words (e.g., `'four s` `and seven years'.split(`**None**`, 3)` returns `['four',` `'score', 'and', 'seven years']`). Note the difference between splitting on **None** (any run whitespace characters is a separator) and splitting on `'` (where each single space character, *not* other whitespa

	as tabs and newlines, and *not* strings of spaces, is a sep
	For example:
	`>>> x = 'a  bB'  # two spaces between a and b`
	`>>> x.split()    # or x.split(None)`
	`['a', 'bB']`
	`>>> x.split(' ')`
	`['a', '', 'bB']`
	In the first case, the two-spaces string in the middle is a
	separator; in the second case, each single space is a sep
	so that there is an empty string between the two spaces

`splitlines`	`s.splitlines(keepends=False)` Like `s.split('\n')`. When `keepends` is **`True`**, however, trailing `'\n'` is included in each item of the resulting li (except the last one, if *s* does not end with `'\n'`).
`startswith`	`s.startswith(prefix, start=0, end=sys.maxsize, /)` Returns **`True`** when `s[start:end]` starts with string *pre* otherwise, returns **`False`**. *prefix* can be a tuple of strin which case `startswith` returns **`True`** when `s[start:en` starts with any one of them.
`strip`	`s.strip(x=string.whitespace, /)` Returns a copy of *s*, removing both leading and trailing characters that are found in string *x*. For example, `'banana'.strip('ab')` returns `'nan'`.

`swapcase`	*`s`*`.swapcase()` Returns a copy of *`s`* with all uppercase letters converte lowercase and vice versa.
`title`	*`s`*`.title()` Returns a copy of *`s`* transformed to title case: a capital l the start of each contiguous sequence of letters, with al letters (if any) lowercase.

`translate`	*`s`*`.translate(`*`table`*`, /, delete=b'')`
	Returns a copy of *`s`*, where characters found in *`table`* a
	translated or deleted. When *`s`* is a `str`, you cannot pass
	argument `delete`; *`table`* is a `dict` whose keys are Unic
	ordinals and whose values are Unicode ordinals, Unico
	strings, or **`None`** (to delete the corresponding character)
	example:
	
	`tbl = {ord('a'):None, ord('n'):'ze'}`
	`print('banana'.translate(tbl))  # prints: 'bz`
	
	When *`s`* is a `bytes`, *`table`* is a `bytes` object of length 25
	result of *`s`*`.translate(`*`t`*`, `*`b`*`)` is a `bytes` object with each
	of *`s`* omitted if *`b`* is one of the items of `delete`, and othe
	changed to *`t`*`[ord(`*`b`*`)]`.
	`bytes` and `str` each have a class method named `maketr`
	which you can use to build tables suitable for the respe
	`translate` methods.

`upper`	*`s`*`.upper()` Returns a copy of *`s`* with all letters, if any, converted to uppercase.

[a] This does *not* include punctuation marks used as a radix, such as a dot (`.`) or co

The string Module

The `string` module supplies several useful string attributes, listed in Table 9-2.

Table 9-2. Predefined constants in the `string` module

<table>
<tr><td>ascii_letters</td><td>The string ascii_lowercase+ascii_uppercase (the following two constants, concatenated)</td></tr>
<tr><td>ascii_lowercase</td><td>The string 'abcdefghijklmnopqrstuvwxyz'</td></tr>
<tr><td>ascii_uppercase</td><td>The string 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'</td></tr>
<tr><td>digits</td><td>The string '0123456789'</td></tr>
<tr><td>hexdigits</td><td>The string '0123456789abcdefABCDEF'</td></tr>
<tr><td>octdigits</td><td>The string '01234567'</td></tr>
<tr><td>punctuation</td><td>The string '!"#$%&\'()*+,-./:;<=>?@[\]^_'{|}~' (i.e., all ASCII characters that are deemed punctuation characters in the C locale; does not depend on which locale is active)</td></tr>
<tr><td>printable</td><td>The string of those ASCII characters that are deemed printable (i.e., digits, letters, punctuation, and whitespace)</td></tr>
<tr><td>whitespace</td><td>A string containing all ASCII characters that are deemed whitespace: at least space, tab, linefeed, and carriage return, but more characters (e.g., certain control characters) may be present, depending on the active locale</td></tr>
</table>

You should not rebind these attributes; the effects of doing so are undefined, since other parts of the Python library may rely on them.

The module `string` also supplies the class `Formatter`, covered in the following section.

String Formatting

Python provides a flexible mechanism for formatting strings (but *not* bytestrings: for those, see **"Legacy String Formatting with %"**). A *format string* is simply a string containing *replacement fields* enclosed in braces (`{}`), made up of a *value part*, an optional *conversion part*, and an optional *format specifier*:

```
{value-part[!conversion-part][:format-specifier]}
```

The value part differs depending on the string type:

- For formatted string literals, or *f-strings*, the value part is evaluated as a Python expression (see the following section for details); expressions cannot end in an exclamation mark.
- For other strings, the value part selects an argument, or an element of an argument, to the `format` method.

The optional conversion part is an exclamation mark (`!`) followed by one of the letters `s`, `r`, or `a` (described in **"Value Conversion"**).

The optional format specifier begins with a colon (`:`) and determines how the converted value is rendered for interpolation in the format string in place of the original replacement field.

Formatted String Literals (F-Strings)

This feature allows you to insert values to be interpolated inline surrounded by braces. To create a formatted string literal, put an `f` before the opening quote mark (this is why they're called *f-strings*) of your string, e.g., `f'{value}'`:

```
>>> name = 'Dawn'
>>> print(f'{name!r} is {len(name)} characters long')
```

```
'Dawn' is 4 characters long
```

You can use nested braces to specify components of formatting expressions:

```
>>> for width in 8, 11:
...     for precision in 2, 3, 4, 5:
...         print(f'{2.7182818284:{width}.{precision}}')
...
```

```
     2.7
    2.72
   2.718
  2.7183
        2.7
       2.72
      2.718
     2.7183
```

We have tried to update most of the examples in the book to use f-strings, since they are the most compact way to format strings in Python. Do remember, though, that these string literals are *not* constants—they evaluate each time a statement containing them is executed, which involves runtime overhead.

The values to be formatted inside formatted string literals are already inside quotes: therefore, take care to avoid syntax errors when using value-part expressions that themselves contain string quotes. With four different string quotes, plus the ability to use escape sequences, most things are possible, though admittedly readability can suffer.

F-STRINGS DON'T HELP INTERNATIONALIZATION

Given a format whose contents will have to accommodate multiple languages, it's much better to use the `format` method, since the values to be interpolated can then be computed independently before submitting them for formatting.

Debug printing with f-strings

3.8+ As a convenience for debugging, the last nonblank character of the value expression in a formatted string literal can be followed by an equals sign (=), optionally surrounded by spaces. In this case the text of the expression itself and the equals sign, including any leading and trailing spaces, are output before the value. In the presence of the equals sign, when no format is specified, Python uses the `repr()` of the value as output; otherwise, Python uses the `str()` of the value unless an `!r` value conversion is specified:

```
>>> a = '*-'
>>> s = 12
>>> f'{a*s=}'
```

```
"a*s='*-*-*-*-*-*-*-*-*-*-*-*-'"
```

```
>>> f'{a*s = :30}'
```

```
'a*s = *-*-*-*-*-*-*-*-*-*-*-*-      '
```

Note that this form is *only* available in formatted string literals.

Here's a simple f-string example. Notice that all text, including any white-space, surrounding the replacement fields is copied literally into the result:

```
>>> n = 10
>>> s = ('zero', 'one', 'two', 'three')
>>> i = 2
>>> f'start {"-"*n} : {s[i]} end'
```

```
'start ---------- : two end'
```

Formatting Using format Calls

The same formatting operations available in formatted string literals can also be performed by a call to the string's `format` method. In these cases, rather than the value appearing inline, the replacement field begins with a value part that selects an argument of that call. You can specify both positional and named arguments. Here's an example of a simple `format` method call:

```
>>> name = 'Dawn'
>>> print('{name} is {n} characters long'
... .format(name=name, n=len(name)))
```

```
'Dawn' is 4 characters long
```

```
>>> "This is a {1}, {0}, type of {type}".format("green", "large",
...                                             type="vase")
```

```
'This is a large, green, type of vase'
```

For simplicity, none of the replacement fields in this example contain a conversion part or a format specifier.

As mentioned previously, the argument selection mechanism when using the `format` method can handle both positional and named arguments. The simplest replacement field is the empty pair of braces (`{}`), representing an *automatic* positional argument specifier. Each such replacement field automatically refers to the value of the next positional argument to `format`:

```
>>> 'First: {} second: {}'.format(1, 'two')
```

```
'First: 1 second: two'
```

To repeatedly select an argument, or use it out of order, use numbered replacement fields to specify the argument's position in the list of arguments (counting from zero):

```
>>> 'Second: {1}, first: {0}'.format(42, 'two')
```

```
'Second: two, first: 42'
```

You cannot mix automatic and numbered replacement fields: it's an either-or choice.

For named arguments, use argument names. If desired, you can mix them with (automatic or numbered) positional arguments:

```
>>> 'a: {a}, 1st: {}, 2nd: {}, a again: {a}'.format(1, 'two', a=3)
```

```
'a: 3, 1st: 1, 2nd: two, a again: 3'
```

```
>>> 'a: {a} first:{0} second: {1} first: {0}'.format(1, 'two', a=3)
```

```
'a: 3 first:1 second: two first: 1'
```

If an argument is a sequence, you can use numeric indices to select a specific element of the argument as the value to be formatted. This applies to both positional (automatic or numbered) and named arguments:

```
>>> 'p0[1]: {[1]} p1[0]: {[0]}'.format(('zero', 'one'),
...                                     ('two', 'three'))
```

```
'p0[1]: one p1[0]: two'
```

```
>>> 'p1[0]: {1[0]} p0[1]: {0[1]}'.format(('zero', 'one'),
...                                       ('two', 'three'))
```

```
'p1[0]: two p0[1]: one'
```

```
>>> '{} {} {a[2]}'.format(1, 2, a=(5, 4, 3))
```

```
'1 2 3'
```

If an argument is a composite object, you can select its individual attributes as values to be formatted by applying attribute-access dot notation to the argument selector. Here is an example using complex numbers, which have `real` and `imag` attributes that hold the real and imaginary parts, respectively:

```
>>> 'First r: {.real} Second i: {a.imag}'.format(1+2j, a=3+4j)
```

```
'First r: 1.0 Second i: 4.0'
```

Indexing and attribute-selection operations can be used multiple times, if required.

Value Conversion

You may apply a default conversion to the value via one of its methods. You indicate this by following any selector with `!s` to apply the object's `__str__` method, `!r` for its `__repr__` method, or `!a` for the `ascii` built-in:

```
>>> "String: {0!s} Repr: {0!r} ASCII: {0!a}".format("banana 😀")
```

```
"String: banana 😀 Repr: 'banana 😀' ASCII: 'banana\\U0001f600'"
```

When a conversion is present, the conversion is applied to the value before it is formatted. Since the same value is required multiple times, in this example a `format` call makes much more sense than a formatted string literal, which would require the value to be repeated three times.

Value Formatting: The Format Specifier

The final (optional) portion of the replacement field, known as the *format specifier* and introduced by a colon (`:`), provides any further required formatting of the (possibly converted) value. The absence of a colon in the replacement field means that the converted value (after representation as a string if not already in string form) is used with no further formatting. If present, a format specifier should be provided conforming to the syntax:

```
[[fill]align][sign][z][#][0][width][grouping_option][.precision][type]
```

Details are provided in the following subsections.

Fill and alignment

The default fill character is the space. To use an alternative fill character (which cannot be an opening or closing brace), begin the format specifier with the fill character. The fill character, if any, should be followed by an *alignment indicator* (see **Table 9-3**).

Table 9-3. Alignment indicators

Character	Significance as alignment indicator
`'<'`	Align value on left of field
`'>'`	Align value on right of field
`'^'`	Align value at center of field
`'='`	Only for numeric types: add fill characters between the sign and the first digit of the numeric value

If the first and second characters are *both* valid alignment indicators, then the first is used as the fill character and the second is used to set the alignment.

When no alignment is specified, values other than numbers are left-aligned. Unless a field width is specified later in the format specifier (see **"Field width"**), no fill characters are added, whatever the fill and alignment may be:

```
>>> s = 'a string'
>>> f'{s:>12s}'
```

```
'    a string'
```

```
>>> f'{s:>>12s}'
```

```
'>>>>a string'
```

```
>>> f'{s:><12s}'
```

```
'a string>>>>'
```

Sign indication

For numeric values only, you can indicate how positive and negative numbers are differentiated by including a sign indicator (see **Table 9-4**).

Table 9-4. Sign indicators

Character	Significance as sign indicator

Character	Significance as sign indicator
`'+'`	Insert `+` as sign for positive numbers; `-` as sign for negative numbers
`'-'`	Insert `-` as sign for negative numbers; do not insert any sign for positive numbers (default behavior if no sign indicator is included)
`' '`	Insert a space character as sign for positive numbers; `-` as sign for negative numbers

The space is the default sign indication. If a fill is specified, it will appear between the sign, if any, and the numerical value; place the sign indicator *after* the = to avoid it being used as a fill character:

```
>>> n = -1234
>>> f'{n:12}'    # 12 spaces before the number
```

```
'       -1234'
```

```
>>> f'{-n:+12}'  # - to flip n's sign, + as sign indicator
```

```
'       +1234'
```

```
>>> f'{n:+=12}'  # + as fill character between sign and number
```

```
'-+++++++1234'
```

```
# + as sign indicator, spaces fill between sign and number
>>> f'{n:=+12}'
```

```
'-       1234'
```

```
# * as fill between sign and number, + as sign indicator
>>> f'{n:*=+12}'
```

```
'-*******1234'
```

Zero normalization (z)

3.11+ Some numeric formats are capable of representing a negative zero, which is often a surprising and unwelcome result. Such negative zeros will be normalized to positive zeros when a `z` character appears in this position in the format specifier:

```
>>> x = -0.001
>>> f'{x:.1f}'
```

```
'-0.0'
```

```
>>> f'{x:z.1f}'
```

```
'0.0'
```

```
>>> f'{x:+z.1f}'
```

```
'+0.0'
```

Radix indicator (#)

For numeric *integer* formats only, you can include a radix indicator, the `#` character. If present, this indicates that the digits of binary-formatted numbers should be preceded by `'0b'`, those of octal-formatted numbers by `'0o'`, and those of hexadecimal-formatted numbers by `'0x'`. For example, `'{23:x}'` is `'17'`, while `'{23:#x}'` is `'0x17'`, clearly identifying the value as hexadecimal.

Leading zero indicator (0)

For *numeric types only*, when the field width starts with a zero, the numeric value will be padded with leading zeros rather than leading spaces:

```
>>> f"{-3.1314:12.2f}"
```

```
'       -3.13'
```

```
>>> f"{-3.1314:012.2f}"
```

```
'-00000003.13'
```

Field width

You can specify the width of the field to be printed. If the width specified is less than the length of the value, the length of the value is used (but for string values, see the upcoming section **"Precision specification"**). If alignment is not specified, the value is left justified (except for numbers, which are right justified):

```
>>> s = 'a string'
>>> f'{s:^12s}'
```

```
'  a string  '
```

```
>>> f'{s:.>12s}'
```

```
'....a string'
```

Using nested braces, when calling the `format` method, the field width can be a format argument too:

```
>>> '{:.>{}s}'.format(s, 20)
```

```
'............a string'
```

See **"Nested Format Specifications"** for a fuller discussion of this technique.

Grouping option

For numeric values in the decimal (default) format type, you can insert either a comma (,) or an underscore (_) to request that each group of three digits (*digit group*) in the integer portion of the result be separated by that character. For example:

```
>>> f'{12345678.9:,}'
```

```
'12,345,678.9'
```

This behavior ignores system locale; for a locale-aware use of digit grouping and decimal point character, see format type `n` in **Table 9-5**.

Precision specification

The precision (e.g., `.2`) has different meanings for different format types (see the following subsection for details), with `.6` as the default for most numeric formats. For the `f` and `F` format types, it specifies the number of digits following the decimal point to which the value should be rounded in formatting; for the `g` and `G` format types, it specifies the number of *significant* digits to which the value should be *rounded*; for nonnumeric values, it specifies *truncation* of the value to its leftmost characters before formatting. For example:

```
>>> x = 1.12345
>>> f'as f: {x:.4f}'  # rounds to 4 digits after decimal point
```

```
'as f: 1.1235'
```

```
>>> f'as g: {x:.4g}'  # rounds to 4 significant digits
```

```
'as g: 1.123'
```

```
>>> f'as s: {"1234567890":.6s}'  # string truncated to 6 characters
```

```
'as s: 123456'
```

Format type

The format specification ends with an optional *format type*, which determines how the value gets represented in the given width and at the given precision. In the absence of an explicit format type, the value being formatted determines the default format type.

The `s` format type is always used to format Unicode strings.

Integer numbers have a range of acceptable format types, listed in **Table 9-5**.

Table 9-5. Integer format types

Format type	Formatting description
`b`	Binary format—a series of ones and zeros
`c`	The Unicode character whose ordinal value is the formatted value
`d`	Decimal (the default format type)

Format type	Formatting description
`n`	Decimal format, with locale-specific separators (commas in the UK and US) when system locale is set
`o`	Octal format—a series of octal digits
`x` or `X`	Hexadecimal format—a series of hexadecimal digits, with the letters, respectively, in lower- or uppercase

Floating-point numbers have a different set of format types, shown in Table 9-6.

Table 9-6. Floating-point format types

Format type	Formatting description
`e` or `E`	Exponential format—scientific notation, with an integer part between one and nine, using `e` or `E` just before the exponent
`f` or `F`	Fixed-point format with infinities (`inf`) and nonnumbers (`nan`) in lower- or uppercase
`g` or `G`	General format (the default format type)—uses a fixed-point format when possible, otherwise exponential format; uses lower- or uppercase representations for `e`, `inf`, and `nan`, depending on the case of the format type
`n`	Like general format, but uses locale-specific separators, when system locale is set, for groups of three digits and decimal points

Format type	Formatting description
`%`	Percentage format—multiplies the value by 100 and formats it as fixed-point followed by `%`

When no format type is specified, a `float` uses the `g` format, with at least one digit after the decimal point and a default precision of `12`.

The following code takes a list of numbers and displays each right justified in a field width of nine characters; it specifies that each number's sign will always display, adds a comma between each group of three digits, and rounds each number to exactly two digits after the decimal point, converting `int`s to `float`s as needed:

```
>>> for num in [3.1415, -42, 1024.0]:
...     f'{num:>+9,.2f}'
...
```

```
'    +3.14'
'   -42.00'
'+1,024.00'
```

Nested Format Specifications

In some cases you'll want to use expression values to help determine the precise format used: you can use nested formatting to achieve this. For example, to format a string in a field four characters wider than the string itself, you can pass a value for the width to `format`, as in:

```
>>> s = 'a string'
>>> '{0:>{1}s}'.format(s, len(s)+4)
```

```
'    a string'
```

```
>>> '{0:_^{1}s}'.format(s, len(s)+4)
```

```
'__a string__'
```

With some care, you can use width specification and nested formatting to print a sequence of tuples into well-aligned columns. For example:

```
def columnar_strings(str_seq, widths):
    for cols in str_seq:
        row = [f'{c:{w}.{w}s}'
               for c, w in zip(cols, widths)]
        print(' '.join(row))
```

Given this function, the following code:

```
c = [
        'four score and'.split(),
        'seven years ago'.split(),
        'our forefathers brought'.split(),
        'forth on this'.split(),
    ]

columnar_strings(c, (8, 8, 8))
```

prints:

```
four     score    and
seven    years    ago
```

```
our       forefath brought
forth     on       this
```

Formatting of User-Coded Classes

Values are ultimately formatted by a call to their `__format__` method with the format specifier as an argument. Built-in types either implement their own method or inherit from `object`, whose rather unhelpful `format` method only accepts an empty string as an argument:

```
>>> object().__format__('')
```

```
'<object object at 0x110045070>'
```

```
>>> import math
>>> math.pi.__format__('18.6')
```

```
'           3.14159'
```

You can use this knowledge to implement an entirely different formatting mini-language of your own, should you so choose. The following simple example demonstrates the passing of format specifications and the return of a (constant) formatted string result. The interpretation of the format specification is under your control, and you may choose to implement whatever formatting notation you choose:

```
>>> class S:
...     def __init__(self, value):
...         self.value = value
...     def __format__(self, fstr):
...         match fstr:
```

```
...             case 'U':
...                 return self.value.upper()
...             case 'L':
...                 return self.value.lower()
...             case 'T':
...                 return self.value.title()
...             case _:
...                 return ValueError(f'Unrecognized format code'
...                                   f' {fstr!r}')
>>> my_s = S('random string')
>>> f'{my_s:L}, {my_s:U}, {my_s:T}'
```

```
'random string, RANDOM STRING, Random String'
```

The return value of the `__format__` method is substituted for the replacement field in the formatted output, allowing any desired interpretation of the format string.

This technique is used in the `datetime` module, to allow the use of `strftime`-style format strings. Consequently, the following all give the same result:

```
>>> import datetime
>>> d = datetime.datetime.now()
>>> d.__format__('%d/%m/%y')
```

```
'10/04/22'
```

```
>>> '{:%d/%m/%y}'.format(d)
```

```
'10/04/22'
```

```
>>> f'{d:%d/%m/%y}'
```

```
'10/04/22'
```

To help you format your objects more easily, the `string` module provides a `Formatter` class with many helpful methods for handling formatting tasks. See the **online docs** for details.

Legacy String Formatting with %

A legacy form of string formatting expression in Python has the syntax:

```
format % values
```

where *`format`* is a `str`, `bytes`, or `bytearray` object containing format specifiers, and *`values`* are the values to format, usually as a tuple.[1] Unlike Python's newer formatting capabilities, you can also use `%` formatting with `bytes` and `bytearray` objects, not just `str` ones.

The equivalent use in `logging` would be, for example:

```
logging.info(format, *values)
```

with the *`values`* coming as positional arguments after the *`format`*.

The legacy string-formatting approach has roughly the same set of features as the C language's `printf` and operates in a similar way. Each format specifier is a substring of *`format`* that starts with a percent sign (%) and ends with one of the conversion characters shown in **Table 9-7**.

Table 9-7. String-formatting conversion characters

Character	Output format	Notes
`d`, `i`	Signed decimal integer	Value must be a number
`u`	Unsigned decimal integer	Value must be a number
`o`	Unsigned octal integer	Value must be a number
`x`	Unsigned hexadecimal integer (lowercase letters)	Value must be a number
`X`	Unsigned hexadecimal integer (uppercase letters)	Value must be a number
`e`	Floating-point value in exponential form (lowercase `e` for exponent)	Value must be a number
`E`	Floating-point value in exponential form (uppercase `E` for exponent)	Value must be a number
`f`, `F`	Floating-point value in decimal form	Value must be a number
`g`, `G`	Like `e` or `E` when *exp* is `>=4` or `<` precision; otherwise, like `f` or `F`	*exp* is the exponent of the number being converted
`a`	String	Converts any value with `ascii`

Character	Output format	Notes
`r`	String	Converts any value with `repr`
`s`	String	Converts any value with `str`
`%`	Literal `%` character	Consumes no value

The `a`, `r`, and `s` conversion characters are the ones most often used with the `logging` module. Between the `%` and the conversion character, you can specify a number of optional modifiers, as we'll discuss shortly.

What is logged with a formatting expression is *`format`*, where each format specifier is replaced by the corresponding item of *`values`* converted to a string according to the specifier. Here are some simple examples:

```
import logging
logging.getLogger().setLevel(logging.INFO)
x = 42
y = 3.14
z = 'george'
logging.info('result = %d', x)        # logs: result = 42
logging.info('answers: %d %f', x, y)  # logs: answers: 42 3.140000
logging.info('hello %s', z)           # logs: hello george
```

Format Specifier Syntax

Each format specifier corresponds to an item in *`values`* by position. A format specifier can include modifiers to control how the corresponding item in *`values`* is converted to a string. The components of a format specifier, in order, are:

- The mandatory leading `%` character that marks the start of the specifier
- Zero or more optional conversion flags:

  `'#'`
  : The conversion uses an alternate form (if any exists for its type).

  `'0'`
  : The conversion is zero padded.

  `'-'`
  : The conversion is left justified.

  `' '`
  : Negative numbers are signed, and a space is placed before a positive number.

  `'+'`
  : A numeric sign (`+` or `-`) is placed before any numeric conversion.

- An optional minimum width of the conversion: one or more digits, or an asterisk (`*`), meaning that the width is taken from the next item in *`values`*
- An optional precision for the conversion: a dot (`.`) followed by zero or more digits or by a `*`, meaning that the precision is taken from the next item in *`values`*
- A mandatory conversion type from **Table 9-7**

There must be exactly as many *`values`* as *`format`* has specifiers (plus one extra for each width or precision given by `*`). When a width or precision is given by `*`, the `*` consumes one item in *`values`*, which must be an integer and is taken as the number of characters to use as the width or precision of that conversion.

ALWAYS USE %R (OR %A) TO LOG POSSIBLY ERRONEOUS STRINGS

Most often, the format specifiers in your *`format`* string will all be `%s`; occasionally, you'll want to ensure horizontal alignment of the output (for example, in a right-justified, maybe truncated space of exactly six characters, in which case you'd use `%6.6s`). However, there is an important special case for `%r` or `%a`.

When you're logging a string value that might be erroneous (for example, the name of a file that is not found), don't use `%s`: when the error is that the string has spurious leading or trailing spaces, or contains some nonprinting characters such as `\b`, `%s` can make this hard for you to spot by studying the logs. Use `%r` or `%a` instead, so that all characters are clearly shown, possibly via escape sequences. (For f-strings, the corresponding syntax would be `{variable!r}` or `{variable!a}`).

Text Wrapping and Filling

The `textwrap` module supplies a class and a few functions to format a string by breaking it into lines of a given maximum length. To fine-tune the filling and wrapping, you can instantiate the `TextWrapper` class supplied by `textwrap` and apply detailed control. Most of the time, however, one of the functions exposed by `textwrap` suffices; the most commonly used functions are covered in **Table 9-8**.

Table 9-8. Useful functions of the `textwrap` module

`dedent`	`dedent(text)` Takes a multiline string and returns a copy in which all lines have had the same amount of leading whitespace removed, so that some lines have no leading whitespace.
`fill`	`fill(text, width=70)` Returns a single multiline string equal to `'\n'.join(wrap(text, width))`.
`wrap`	`wrap(text, width=70)` Returns a list of strings (without terminating newlines), each no longer than `width` characters. `wrap` also supports

other named arguments (equivalent to attributes of instances of class `TextWrapper`); for such advanced uses, see the **online docs**.

The pprint Module

The `pprint` module pretty-prints data structures, with formatting that strives to be more readable than that supplied by the built-in function `repr` (covered in **Table 8-2**). To fine-tune the formatting, you can instantiate the `PrettyPrinter` class supplied by `pprint` and apply detailed control, helped by auxiliary functions also supplied by `pprint`. Most of the time, however, one of the functions exposed by `pprint` suffices (see **Table 9-9**).

Table 9-9. Useful functions of the `pprint` module

`pformat`	`pformat(object)` Returns a string representing the pretty-printing of `object`.

`pp`, `pprint`	`pp(object, stream=`*`sys.stdout`*`)`,
	`pprint(object, stream=`*`sys.stdout`*`)`
	Outputs the pretty-printing of `object` to open-for-writing file object `stream`, with a terminating newline.
	The following statements do exactly the same thing:
	`print(pprint.pformat(x))`
	`pprint.pprint(x)`
	Either of these constructs is roughly the same as `print(`*`x`*`)` in many cases—for example, for a container that can be displayed within a single line. However, with something like *`x`*`=list(range(30))`, `print(`*`x`*`)` displays *`x`* in 2 lines, breaking at an arbitrary point, while using the

module `pprint` displays `x` over 30 lines, one line per item. Use `pprint` when you prefer the module's specific display effects to the ones of normal string representation. `pprint` and `pp` support additional formatting arguments; consult the **online docs** for details.

The reprlib Module

The `reprlib` module supplies an alternative to the built-in function `repr` (covered in **Table 8-2**), with limits on length for the representation string. To fine-tune the length limits, you can instantiate or subclass the `Repr` class supplied by the `reprlib` module and apply detailed control. Most of the time, however, the only function exposed by the module suffices: `repr(`*`obj`*`)`, which returns a string representing *`obj`*, with sensible limits on length.

Unicode

To convert bytestrings into Unicode strings, use the `decode` method of bytestrings (see **Table 9-1**). The conversion must always be explicit, and is performed using an auxiliary object known as a *codec* (short for *coder–decoder*). A codec can also convert Unicode strings to bytestrings using the `encode` method of strings. To identify a codec, pass the codec name to `decode` or `encode`. When you pass no codec name, Python uses a default encoding, normally `'utf-8'`.

Every conversion has a parameter `errors`, a string specifying how conversion errors are to be handled. Sensibly, the default is `'strict'`, meaning any error raises an exception. When `errors` is `'replace'`, the conversion replaces each character causing errors with `'?'` in a bytestring result, or with `u'\ufffd'` in a Unicode result. When `errors` is `'ignore'`, the conversion silently skips characters causing errors. When `errors` is `'xmlcharrefreplace'`, the conversion replaces each character causing errors with the XML character reference representation of that character in the result. You may code your own function to implement a conversion error handling strategy and register it under an appropriate name by calling `codecs.register_error`, covered in the table in the following section.

The codecs Module

The mapping of codec names to codec objects is handled by the `codecs` module. This module also lets you develop your own codec objects and register them so that they can be looked up by name, just like built-in codecs. It provides a function that lets you look up any codec explicitly as well, obtaining the functions the codec uses for encoding and decoding, as well as factory functions to wrap file-like objects. Such advanced facilities are rarely used, and we do not cover them in this book.

The `codecs` module, together with the `encodings` package of the standard Python library, supplies built-in codecs useful to Python developers dealing with internationalization issues. Python comes with over 100 codecs; you can find a complete list, with a brief explanation of each, in the **on-line docs**. It's *not* good practice to install a codec as the site-wide default in the module `sitecustomize`; rather, the preferred usage is to always specify the codec by name whenever converting between byte and Unicode strings. Python's default Unicode encoding is `'utf-8'`.

The `codecs` module supplies codecs implemented in Python for most ISO 8859 encodings, with codec names from `'iso8859-1'` to `'iso8859-15'`. A popular codec in Western Europe is `'latin-1'`, a fast, built-in implementation of the ISO 8859-1 encoding that offers a one-byte-per-character encoding of special characters found in Western European languages (beware that it lacks the Euro currency character `'€'`; however, if you need that, use `'iso8859-15'`). On Windows systems only, the codec named `'mbcs'` wraps the platform's multibyte character set conversion procedures. The `codecs` module also supplies various code pages with names from `'cp037'` to `'cp1258'`, and Unicode standard encodings `'utf-8'` (likely to be most often the best choice, thus recommended, and the default) and `'utf-16'` (which has specific big-endian and little-endian variants: `'utf-16-be'` and `'utf-16-le'`). For use with UTF-16, `codecs` also supplies attributes `BOM_BE` and `BOM_LE`, byte-order marks for big-endian and little-endian machines, respectively, and `BOM`, the byte-order mark for the current platform.

In addition to various functions for more advanced uses, as mentioned earlier, the `codecs` module supplies a function to let you register your own conversion error handling functions:

`register_error`	`register_error(`*`name`*`, `*`func`*`, /)` *`name`* must be a string. *`func`* must be callable with one argument *`e`* that is an instance of `UnicodeDecodeError`, and must return a tuple with two items: the Unicode string to insert in the converted string result, and the index from which to continue the conversion (the latter is normally *`e`*`.end`). The function can use *`e`*`.encoding`, the name of the codec of this conversion, and *`e`*`.object[`*`e`*`.start:`*`e`*`.end]`, the substring causing the conversion error.

The unicodedata Module

The `unicodedata` module provides easy access to the Unicode Character Database. Given any Unicode character, you can use functions supplied by `unicodedata` to obtain the character's Unicode category, official name (if any), and other relevant information. You can also look up the Unicode character (if any) that corresponds to a given official name:

```
>>> import unicodedata
>>> unicodedata.name('⚀')
```

```
'DIE FACE-1'
```

```
>>> unicodedata.name('Ⅵ')
```

```
'ROMAN NUMERAL SIX'
```

```
>>> int('Ⅵ')
```

```
ValueError: invalid literal for int() with base 10: 'Ⅵ'
```

```
>>> unicodedata.numeric('Ⅵ')  # use unicodedata to get numeric value
```

```
6.0
```

```
>>> unicodedata.lookup('RECYCLING SYMBOL FOR TYPE-1 PLASTICS')
```

```
'♳'
```

1 In this book we cover only a subset of this legacy feature, the format specifier, that you must know about to properly use the `logging` module (discussed in **"The logging module"**).