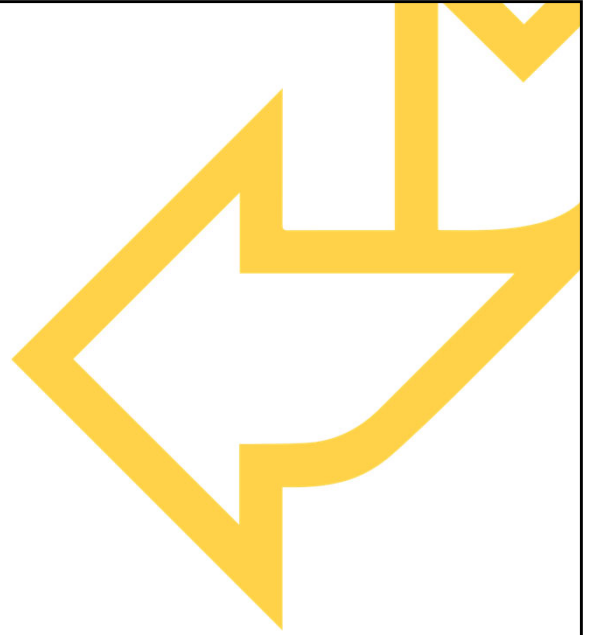# Python 3 Programming

String handling

STRING HANDLING

Contents

- Python 3 strings
- The print function
- Cooking strings
- String concatenation
- "Quotes"
- String methods
- String tests
- String formatting
- Literal string interpolation
- Slicing a string
- String methods - split and join

Summary

- String formatting - old style

Handling text is important to most Python programs, and is a fundamental object type in Python. They went through an import change between Python 2 and 3...

🔵 Python 3 strings

Strings in Python 3 are Unicode

- Multi-byte characters.
- \u*nnnn* for a two-byte Unicode character.
- \U*nnnnnnnn* for a four-byte Unicode character.
- \N{*name*} for a named Unicode character.

```
>>> euro="\u20ac"
>>> euro
'€'
>>> euro="\N{euro sign}"
>>> print(euro)
€
```

For low-level interfaces we have bytes() and bytearray()

```
chars_as_bytes = b"single-byte string"
```

- Conversion between strings and bytes:

        string.encode()   *bytes*.decode()

String types went through a major revision at Python 3, they are now Unicode strings, which are usually two bytes for each character. In 3.3 strings are more flexible, and are stored internally in as few bytes as possible. Therefore, ASCII characters only use one byte each, despite being part of a unicode object (this optimisation reduces memory usage significantly).

The default encoding used is UTF-8, but this can be altered by a pragma comment at the start of the script, for example:

```
# -*- coding: latin-1 -*-
```

See the Python help text for a list of standard encodings, it is a long list! The current encoding can be obtained from **sys.getdefaultencoding()**.

The old Python 2 u"…" prefix was not supported in 3.0 to 3.2, but reappeared at 3.3. It has no effect on Python 3 and is only there for backward compatibility.

You will see **bytes** and **bytearray** mentioned in the documentation, and we will briefly see them later in the course. They map to the old single-byte character sets and are mostly used for interfacing with low-level primitives written in C, and for compatibility with Python 2.

You may find other, specialised, string types. For example, the GUI package PyQt has a `QString` type.

QA The print function

One of the most commonly used functions:

- Used for displaying a comma separated list of objects.
- Objects are *stringified*.

```
print(object1, object2, … )
```

- Has several named arguments.
- Specified in any order.
- **end=** *characters to be appended, default is '\n' (newline).*
- **file=** *file object to be written to, default is sys.stdout.*
- **sep=** *separator used between list items, default is a space.*
- **flush=***to flush or not to flush (Boolean), default is False (3.3).*

```
print("The answer is", 42, "always", sep=': ', end='')
print("(I think)")
```
```
The answer is: 42: always(I think)
```

The **print** built-in function is one of the most commonly used functions, and we have seen it before. The **print** *function* was introduced at Python 2.6, and replaced the old **print** *statement* at Python 3.0 (so, 2.6 and 2.7 has both). The named arguments are new, as are the parentheses which are required now that **print** is a function.

The **flush** argument was added at Python 3.3.

When we say that objects are *stringified*, we mean that they are converted to strings into the same format as the **str** built-in function. As we shall see later, this can be overridden by a class method to stringify an object.

We shall see how to use **print** with file objects in a later chapter.

QA Escaping a character
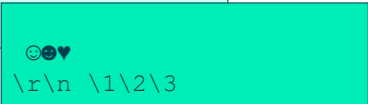
Adds a meaning to a normal character
- \n becomes a new-line.
- \t becomes a tab.
    - ...and so on.

Removes a meaning from a special character
- \\ removes the special meaning of \
- \' removes the special meaning of '
- \" removes the special meaning of "

Raw strings do not treat \ as a special character

```
print(  '\r\n \1\2\3')
print( r'\r\n \1\2\3')
```

```
☺☻♥
\r\n \1\2\3
```

Escape codes:

newline	*newline* is ignored
\\	Backslash (\)
\'	Single quote (')
\"	Double quote (")
\a	Bell (BEL)
\b	Backspace (BS)
\f	Formfeed (FF)
\n	Linefeed (LF)
\N{*name*}	Character named *name* in the Unicode database
\r	Carriage Return (CR)
\t	Horizontal Tab (TAB)
\u*xxxx*	Character with 16-bit hex value *xxxx*
\U*xxxxxxxx*	Character with 32-bit hex value *xxxxxxxx*
\v	Vertical Tab (VT)
ooo	Character with octal value *ooo*(3,5)
\0	Null
\x*hh*	Character with hex value *hh*

Triple quoted strings """" are discussed later.
Raw strings are particularly useful when we do not want these translations. For example when using regular expressions \1, \2, etc. are used as back-references (see later) and if you do not use raw strings these will be translated as \x01, \x02, etc. The last character inside a raw string may not be a \.

String concatenation

Adjacent literals are concatenated.

```
>>> name = 'fred' 'bloggs'
>>> name
'fredbloggs'
>>> name = 'fred' \        ⟵——— line continuation
... 'bloggs'
```

But, that does not work with variables!

Use the overloaded + operator instead.

```
>>> name = first + 'bloggs'
```

But, remember that strings are immutable.

```
s = ""
for item in alist:
    s = s + str(item) + " "
```

Very inefficient code

Use join() str method instead

Much like *awk*, string literals may be 'joined' merely by placing them next to each other - the intervening white-space is optional. This is useful when specifying long strings that go over one line, but we must 'escape' the new-line first. This removes the special meaning of a new-line, which normally terminates the statement. Unlike *awk* we cannot use this technique with variables, an operator must be used. For this, much like C++ and Java, Python overloads the + operator for strings for concatenation.

Although string concatenation using the + operator is easy, it should not be over-used, particularly in a loop. The problem is that we think we are adding data onto the end of a string, but we are not! We are creating new string objects and copying the entire contents on each iteration. There is a better way of joining all the items in a list - a string method called `join()` which we see later.

"Quotes"

Single and double quotes have the same effect
* Use " when you have embedded ', and vice versa.

```
print('hello\nworld')
```
⟺
```
print("hello\nworld")
```

With embedded quotes or new-lines, use triple quotes

```
>>> html = """
<tr>
        <td><font color="#690000"><b>Username :</b></font></td>
        <td><input type='textbox' name='username'></td>
</tr>
"""
```

```
'\n<tr>\n\t<td><font color="#690000"><b>Username :</b></font></td>\n
\t<td><input type=\'textbox\' name=\'username\'></td>\n</tr>\n'
```

wrapped around

In Python, double quotes have no special meaning over single quotes - and both may contain special characters. If you have embedded single quotes (such as with SQL), then use double quotes around your string, with double quotes in the data (such as with HTML) then use single quotes. If you have both, or embedded white-space such as new-lines or tabs, or \ (such as Windows path names), then triple quotes (single or double) will handle them. Notice that new-lines and quotes are all 'escaped'. This makes it particularly useful when data has come from a user and may be a hacking attempt, for example SQL injection.

With triple quotes, be careful of trailing special characters, including other quotes. For example:

```
file = """C:\QA\Python\PYTHB\"""
```

does not work, since the final **** 'escapes' the following quote. Adding another " will unfortunately add it to the end of the data. However, escaping the \ does work:

```
file = """C:\\QA\\Python\\PYTHB\\"""
```

⌀ String methods

The `string` module is now mostly replaced by methods.

Some useful string functions and methods:

```
String to a number        int          int("42")
Object to a string        str          str(42)
Number of characters      len          len(name)
Convert to lower case     lower        str.lower()
Replace a sub-string      replace      str.replace('old', 'new')
Remove trailing chars     rstrip       str.rstrip()
Search for a sub-string   find         str.find('cheese')
(returns the offset)
```

Overloaded * operator:

Mandatory Monty Python reference

```
>>> 'Spam ' * 4
```

```
'Spam Spam Spam Spam '
```

In Python's early days, string handling was done using the `string` module. Thankfully, this is no longer necessary and we have a number of built-in methods. The `string` module still works, but is only there for backward compatibility.

Both **str** and **repr** can convert a number to a string, generally **str** will produce prettier output. Both these functions can be implemented in a user written class, and their expected result is rather different. The **str** function is supposed to return a human-readable form of the object. In contrast, **repr** is supposed to return a Python readable form, or *repr*esentation, specifically for the built-in function **eval** (which compiles code at runtime, and is generally pronounced *evil*). So use **str** unless you are doing particularly eval things.

Note that in early versions of Python, `back-ticks` could be used instead of **repr**, but this was removed at Python 3.0.

The **replace** and **find** methods often overlap in functionality with regular expressions (discussed later). Generally, these methods are more efficient than REs (Regular Expressions), and easier to code.

There are other string methods, around 40 in all, some are discussed later.

QΛ String tests

Remember the `in` operator

```
if substr in string:
```

Testing a string type can often be done with a method:
- Regular Expressions can also be used, but can be slow.

count
endswith
isalnum
isalpha
isdigit
islower
isspace
istitle
isupper
startswith

```
text = 'hello world'

print(text.count('o'))

if text.startswith('hell'):
    print("It's hell in there")

if text.isalpha():
    print('string is all alpha')

text = ' \t\r\n'
if text.isspace():
    print('string is whitespace')
```

```
2
It's hell in there
string is whitespace
```

A number of methods are available with strings to test their type. Many return a Boolean (true or false) but some, like **count**, do not. Most of these methods can be coded in other ways, and using a Regular Expression (RE) is the most obvious. However, REs are often slow, and rarely easier to read. Here is the RE code for the example shown, details later:

```
import re
text = 'hello world'

print(len( re.findall (r"(o)", text)))
print(len( re.findall (r"(o)", text[5:])))

if re.match(r"^hell", text):
    print("It's hell in there")

if re.match (r"^[A-Za-z]+$", text):
    print("string is all alpha")

text = ' \t\r\n'
if re.match(r"^\s+$", text):
```

```
print("text is whitespace")
```

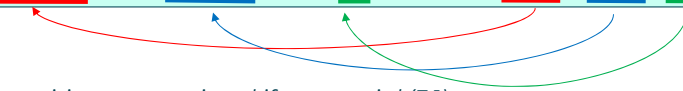**String formatting**

Call the format method on a string

"string".format(*field_values*)

- *string* - contains text and the format of values to be substituted in.
- *field_values* - the parameters to be evaluated in string.

String format specifications are all optional, and of the form:

{ *position : fill align sign # 0 width . precision type* **}**

"*text*{0:5.3f}*text*{1:3.d}*text*{2}".format(var1,var2,var3)

positions are optional if sequential (3.1)
We can also specify index numbers or key names.

"*text*{0[index]}*text*{1[key]}".format(list, dictionary)

Other languages have `printf` and `sprintf`, Python has string formatting.
Python 3 introduced this new formatting style, for the old style, see slides after the summary. This style is also available in Python 2, from version 2.6.
The fields in the format specifications have the following meaning:

fill	Any character except {
align	< left, > right, ^ centred, = pad sign
sign	+ force sign, " " only force - (minus)
#	prefix integers with 0b, 0o, or 0x
0	Pad numbers with zero
width	Minimum field width
.precision	Maximum field width (strings), or number of decimal places
type	b, c, d, n, o, x, X - integer formats
	e, E, f, g, G - floating point formats

These are all optional, the default type being a string. Taking the first example:

{0:5.3f}
0 is the position, the first variable passed to the format

method (var1)
From Python 3.1 they are automatically incremented if omitted
5 is the minimum field width, including the decimal point
3 is the precision, meaning 3 characters after the decimal point.

Field names (also known as nested arguments) may be used for substitutions, e.g.

```
var1 = 42
var2 = 'hello'
print("{name} {value}".format(value=var1,
name=var2))
'hello 42'
```

String formatting example

Common conversion specifiers:

- `{d}` Treats the argument as an integer number.
- `{s}` Treats the argument as a string.
- `{f}` Treats the argument as a float (and rounds).

```python
planets = {'Mercury': 57.91,
           'Venus': 108.2,
           'Earth': 149.597870,
           'Mars': 227.94
}

for i, key in enumerate(planets.keys(), 1):
    print("{:2d} {:<10s} {:06.2f} Gm".format(i, key, planets[key]))
```

```
1 Earth      149.60 Gm
2 Mercury    057.91 Gm
3 Mars       227.94 Gm
4 Venus      108.20 Gm
```

The format specifiers shown on the slides are type specifiers - they specify how the argument should be treated.

The example format string contains the following specifiers:

`{2d}` Right-justified, at least two digits, space padding

`{<10s}` Left-justified, at least 10 character string, space padding

`{06.2f}` Right-justified, at least 6 characters, zero padding. One of the six characters is the decimal point, and the argument is rounded to two digits after the decimal point

The second parameter to enumerate (1) is the start of the enumeration sequence (default is zero).

Here is the example using field names:

```python
for i, key in enumerate(planets.keys(), 1):
    print("{pos:2d} {planet:<10s} {distance:06.2f} Gm".
         format(pos=i,
planet=key,distance=planets[key]))
```

By the way, Gm is the symbol for gigametre, the little used metric unit of 1000 kilometres. The example shows the mean distance

from the Sun for these planets.

Q∧ Other string formatting aids

Often more efficient and easier

- *string*.capitalize()
- *string*.lower()/ *string*.upper()
- *string*.center()
- *string*.ljust()
- *string*.rjust()
- *string*.zfill()

```
text = 'hello'
print(text.capitalize())
print(text.upper())
print('<'+text.center(12)+'>')
print('<'+text.ljust(12)+'>')
print('<'+text.rjust(12)+'>')
print('<'+text.zfill(12)+'>')
```

```
Hello
HELLO
<   hello    >
<hello        >
<       hello>
<0000000hello>
```

Often the string format strings are not required, and the formatting string methods may be enough. We feel the examples are self-explanatory.
There are other string methods, for example *string*.**title**, which changes the string to "title case" when the first character of each word is in upper case.

QA Literal string interpolation

Python expressions may be embedded inside a text string
- Available from Python 3.6.

Special string literals are used, known as *f-strings*
- Embed a Python expression inside braces.

```python
names = ['Tom', 'Harry', 'Jane', 'Mary']
s = f"The third member is {names[3]}"
```

String formats may be embedded
- Syntax is `{value:{width}.{precision}}`.
- This is the planets example, rewritten to use an f-string.

```python
for i, key in enumerate(planets.keys(), 1):
    print(f"{i:2d} {key:<10s} {planets[key]:06.2f} Gm")
```

Python has had interpolation of sorts previously, in `string.Template`, but it was unwieldy and not used very often. A new system, specified in PEP 498 and introduced with Python 3.6, is very similar in appearance to that used by Ruby.
Single or double quotes may be used, even triple quotes. Basically, we just put the Python expression, commonly a variable, inside braces. A special syntax is used for format strings.

Not just variable values may be represented:

```python
names = ['Tom', 'Harry', 'Jane', 'Mary']
suffix = ['st', 'nd', 'rd', 'th']
n = 1
s = f"{str(n+1) + suffix[n]} of \
     {len(names)} is {names[n]}"
```

`2nd of 4 is Harry`

Can also be combined with raw strings:

```python
drive = 'C:'
dir = 'Windows'
path = fr"{drive}\{dir}"
```

f-strings supports only Unicode

• Byte objects do not support f-strings.

Function calls, arithmetic, any valid python expression may be put inside the braces. However, we suggest that you Keep It Simple – complex expressions inside a string would be difficult to debug and read, so it is probably preferable to restrict yourself to simple expressions.

A raw string may be combined with an f-string, either `fr` or `rf` prefix may be used. An f-string *cannot* be combined with the u' (Unicode) prefix, since f-strings can only be Unicode. The specification rules out the possibility of f-string interpolation being back-ported to Python 2.

The `fr' '` example is there to demonstrate the syntax, the correct way to join path components is to use `os.path.join()`. Raw and f-strings are most commonly seen with regular expressions, which we will look at later in the course.

Q∧ Slicing a string

A Python string is an immutable sequence type.
- Slicing is the same for all sequence types.

Slice by start and end+1 *position.*
- Counting from zero on lhs, from -1 on rhs.

```
#        01234567890123456789012345678901234
text = "Remarkable bird, the Norwegian Blue"
print(text[11:14])
bir
print(text[-7:-1])
an Blu              start  end+1
```

- Start and end positions may be defaulted.

```
print(text[:14])
Remarkable bir
print(text[-7:])
an Blue
```

Beautiful plumage

Sequential composite objects like strings, tuples, and lists may be sliced. Often only one item is required, in which case the syntax uses the familiar square brackets with the index inside. Items are indexed from zero on the left, or -1 (with a negative count) from the right,.

To slice a range of elements, we specify the start index, a colon, then end index plus one. That is, the slice is taken up to, but not including, the second index position. If the start index is not given, the default is zero (first element). Defaulting the second index slices to the end of the object. For example, string[:-1] will remove the last character from a string.

Python 2.3 added support for a third index, a step. For example:

    #   0123456789012345
    text = "Now that's what I call a dead parrot."
    print (text[5 : 15 : 2])
    Gives: htswa

This has printed every other character, from position 5 through to 15.

Here is another example:

```
print(name[::-1].upper())
```
what do you think that does? (Try it)

QA **String methods** – split and join

String to a list - split

- *string*.split([*separator*[*, max_splits*]])
- If *separator* is omitted, split one or more white-spaces.
- If *max_splits* is omitted, split the whole string.
- *string*.**splitlines**() is useful on lines from files.

Sequence to a string - join

- *separator*.join(*sequence*).
- *sequence* can be a string, list, or a tuple.

```
line = 'root::0:0:superuser:/root:/bin/sh'
elems = line.split(':')

elems[0] = 'avatar'
elems[4] = 'The super-user (zero)'
line = ':'.join(elems)
print(line)
```
```
avatar::0:0:The super-user (zero):/root:/bin/sh
```

The **split** function is commonly used to extract fields from records read from a file. If you want to split around line-endings, then use splitlines() instead. Why might you want to do that? Because it is common to read a small file into a single string and then split it, so that each line goes into an element of a list. There are other versions of **split** available: in the **re** module and in the **os.path** module. They have different functionality to the string **split** function.

The **join** function is a method called on the separator string, which is rather unusual and takes some getting used to. When called on an empty string, it concatenates the characters in the list into one.

We mentioned join earlier. If you are concatenating a string in a loop, it is actually faster to append each item to the end of the list instead, then join them all together after the loop. With a small number of iterations, you will not notice any difference, but with a large number there is a measurable improvement in performance.

SUMMARY

- **Python 3 strings are Unicode.**
- **Python variables are not embedded inside quotes.**
  - However, escape characters like \r\n\t can be.
- **No difference between ' and ".**
  - Use three quotes for multi-line text.
- **Several methods available on a string:**
  - Many for conversions.
  - Formatting uses the format method.
  - Split a string with split().
  - Join items in a list with join().
- **Strings can be sliced[start:end+1].**
  - As can other sequences.

17

More string formatting examples

```
flt = 22/7
print("Float: {0:11.8f}, sci: {0:e}".format(flt))
```
```
Float:  3.14285714, sci: 3.142857e+00
```

```
first  = 'Gengis'
second = 'Khan'
print("Name: {:<20s} {:<10s}".format(first, second))
```
```
Name: Gengis               Khan
```

Octal number

```
file  = sys.argv[0]
perms = '0{:o}'.format((os.stat(file).st_mode) & 0o7777)
```
```
0640
```

```
fred = '{:#x}'.format(3735928559)
```
```
0xdeadbeef
```

18

format is particularly useful for displaying floating point numbers, in a variety of formats. The first example shows a number displayed with a total width of 11 characters (including the decimal point), and 8 digits after the decimal point, followed by engineering notation. Notice also that only one value is supplied, this is because both format specifications use position zero. In the other examples, we are not specifying the position, so they are automatically numbered for us (introduced at 3.1).

Text columns may be left or right aligned, the second example shows two fields left aligned and padded with spaces.

Converting to hexadecimal is useful (use **o** for octal), and we also show an example of returning the formatted string instead of printing it.

Values normally stored in octal, like UNIX file permissions, are displayed or manipulated more easily using **format**.

PYTHON 2 STRING FORMATTING

Older methods for formatting strings:
(Like sprintf in some other languages)

"format_string" **% (***argument_list***)**

format_string

- contains text and format specifiers, prefixed %.
- describe the format of the plugged-in value.

%s	string	%o	octal
%c	character	%x	lowercase hex
%d	decimal	%X	uppercase hex
%i	integer	%%	literal %
%u	unsigned int		

%e, %E, %f, %g, %G - alternative floating point formats

argument_list

- contains text or variables to be plugged-in.

This style of string formatting is still supported in Python 3 but considered deprecated. It might be removed from the language eventually, so should not be used for new applications.

Many languages have the `printf` and `scanf` family of functions (originally from C), but Python 2 overloaded the % operator. If you are mixing text variables and escape sequences, then this provides a lot more flexibility than **print**.

The general form of a format specifier is:

`%[flag][width][.precision]letter`

where: **[flag]** specifies padding an signed options

[width] specifies how wide the field is

[.precision] specifies decimal digits for floating point numbers

and **letter** indicates the data type. To print a literal '%', use '%%'. Only the common format specifiers are shown, for a full list see the online help text. Floating point numbers are rounded, but the rounding algorithm does not always produce what you might expect. If a particular rounding method is important, with money amounts, for example, it is best to implement it yourself rather than relying on `%`.

Python 2 String Formatting Examples

```
flt = 22/7
print("Float format: %11.8f, scientific: %e" % (flt, flt))
```
```
Float format:  3.14285714, scientific: 3.142857e+00
```

```
first  = 'Gengis'
second = 'Khan'
print("Name: %-20s %-10s" % (first, second))
```
```
Name: Gengis               Khan
```

Octal number

```
file  = sys.argv[0]
perms = '0%lo' % (os.stat(file).st_mode & 0o7777)
```
```
0640
```

```
fred = '%x' % 3735928559
```
```
deadbeef
```

20

% is particularly useful for displaying floating point numbers, in a variety of formats. The first example shows a number displayed with a total width of 11 characters (including the decimal point), and 8 digits after the decimal point, followed by engineering notation.

"Columns" may also be left or right aligned, the second example shows two fields left aligned and padded with spaces.

Converting to hexadecimal is useful (use %o for octal), and we also show an example of returning the formatted string instead of printing it.

Values normally stored in octal, like UNIX file permissions, are displayed or manipulated more easily using %.

String formatting and bytes objects

String formatting produces strings, not bytes.

- In Python 2, they are the same.

Bytes objects do not have a `.format()` method.

- So, the only formatting available is using %.

The following are *additional* formats for bytes objects:

- %a – create a bytes object by encoding.
- %b – create a bytes object using class specified method.
- %c – a single byte.

```
print(b'%a' % 42)
print(b'%b' % b"hello")
print(b'%c' % 42)
```

```
b'42'
b'hello'
*
```

42 is the ASCII ordinal number of the * character

Omitting the `b'` on the left side of the format string will produce an str object rather than a bytes object. The exception is `%b` which is not supported inside a str.

See *printf-style Bytes Formatting* in the online documentation.