POWERING POTENTIAL

String Handling

Module 11

STRING HANDLING

Contents

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)
€
```

py3

Using IDLE here because Windows cmd.exe has poor Unicode support

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

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

- Conversion between strings and bytes:

string`.encode()` $\longleftrightarrow$ *bytes*`.decode()`

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, … )
```

py3

- 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)
```

Escaping a character

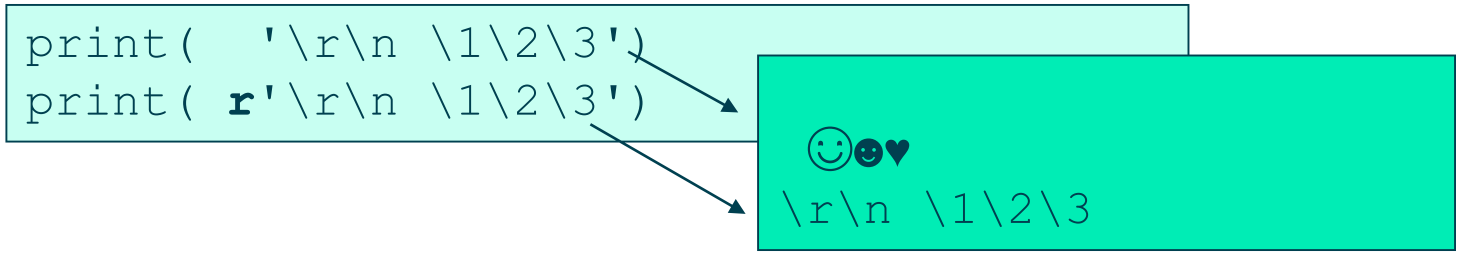
Adds a meaning to a normal character

- \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
```

String concatenation

Adjacent literals are concatenated

```
>>> name = 'fred' 'bloggs'
>>> name
'fredbloggs'
>>> name = 'fred' \
... 'bloggs'
```

line continuation

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`

"Quotes"

Single and double quotes have the same effect

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

⇔

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

- Use " when you have embedded ', and vice versa

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

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)`
Object to a string	`repr`	`repr(obj)` - see notes
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 (returns the offset)	`find`	*`str`*`.find('cheese')`

Overloaded * operator

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

Mandatory Monty Python reference

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

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
```

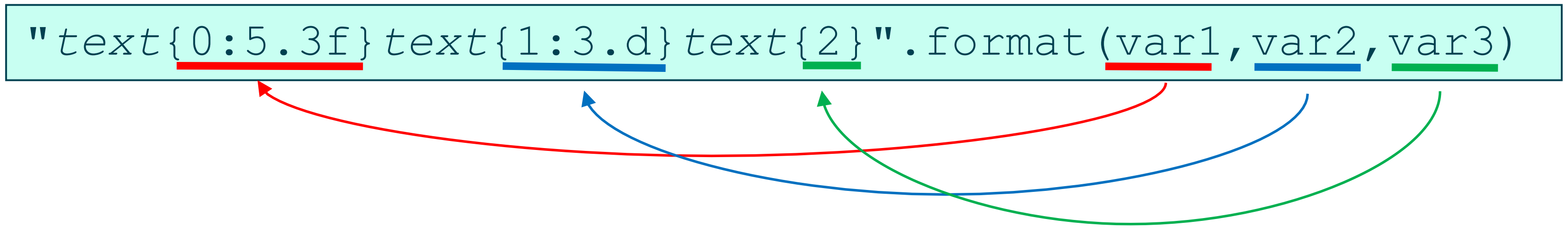
String formatting

Call the format method on a string

string.`format`(*field_values*)

- *string* - contains text and the format of values to be plugged-in
- *field_values* - give the values or variables to be plugged in
- 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)
```

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)

```
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
```

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>
```

Literal string interpolation

py3

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

```
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

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

Literal string interpolation (2)

Not just variable values may be represented

```
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

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

f-strings supports only Unicode

- Byte objects do not support f-strings

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
```

String methods - split and join

String to a list - split

- *string*`.split([` *separator*`[,` *max_splits*`]])`
- If *separator* is omitted, split on one or more white-space
- 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
```

SUMMARY

- **Python 3 strings are Unicode**
- **Python variables are not embedded inside quotes**
- But 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**
- **Strings can be sliced[start:end+1]**
- As can other sequences
- **Split a string with split, join items in a list with join**