String Fundamentals

Basics

- strings can be used to represent just about anything that can be encoded as text or bytes.
- Strings can also be used to hold the raw bytes used for media files and network transfers
- Unlike C, Python has no distinct type for individual characters. instead, you just use one-character strings.
- ***immutable sequences****:* meaning that the characters they contain have a left-to-right positional order and that they cannot be changed in place.

Operation	Interpretation
`S = ''`	Empty string
`S = "spam's"`	Double quotes, same as single
`S = 's\np\ta\x00m'`	Escape sequences
`S = """...multiline..."""`	Triple-quoted block strings
`S = r'\temp\spam'`	Raw strings (no escapes)
`B = b'sp\xc4m'`	Byte strings in 2.6, 2.7, and 3.X (Chapter 4, Chapter 37)
`U = u'sp\u00c4m'`	Unicode strings in 2.X and 3.3+ (Chapter 4, Chapter 37)
`S1 + S2`	Concatenate, repeat
`S * 3`	
`S[i]`	Index, slice, length
`S[i:j]`	
`len(S)`	
`"a %s parrot" % kind`	String formatting expression
`"a {0} parrot".format(kind)`	String formatting method in 2.6, 2.7, and 3.X
`S.find('pa')`	String methods (see ahead for all 43): search,
`S.rstrip()`	remove whitespace,
`S.replace('pa', 'xx')`	replacement,
`S.split(',')`	split on delimiter,

Operation	Interpretation
`S.isdigit()`	content test,
`S.lower()`	case conversion,
`S.endswith('spam')`	end test,
`'spam'.join(strlist)`	delimiter join,
`S.encode('latin-1')`	Unicode encoding,
`B.decode('utf8')`	Unicode decoding, etc. (see Table 7-3)
`for x in S: print(x)`	Iteration, membership
`'spam' in S`	
`[c * 2 for c in S]`	
`map(ord, S)`	
`re.match('sp(.*)am', line)`	Pattern matching: library module

- Empty strings are written as a pair of quotation marks (single or double) with nothing in between

String Literals

- Single quotes: 'spa"m'
- Double quotes: "spa'm"
- Triple quotes: '''... spam ...''', """... spam ..."""
- Escape sequences: "s\tp\na\0m"
- Raw strings: r"C:\new\test.spm"
- Bytes literals: b'sp\x01am'

- Unicode literals: u'eggs\u0020spam'

Single- and Double-Quoted Strings Are the Same

- single- and double-quote characters are interchangeable
- The reason for supporting both is that it allows you to embed a quote character of the other variety inside a string without escaping it with a backslash.

```
>>> "test", "me", 'here'
('test', 'me', 'here')
```

- Note that the comma is important here.
- Without it, Python *automatically concatenates* adjacent string literals in any expression

```
>>> "test" "me" 'here'
'testmehere'
```

- Adding commas between these strings would result in a tuple, not a string
- all of these outputs that Python prints strings in single quotes unless they embed one.

```
>>> 'knight\'s', "knight\"s"
("knight's", 'knight"s')
```

Escape Sequences Represent Special Characters

- backslashes are used to introduce special character coding known as *escape sequences*.
- Escape sequences let us embed characters in strings that cannot easily be typed on a keyboard.
- The character **\ and one or more characters following** it in the string literal, are **replaced** with a ***single* character** in the **resulting string object**, which has the **binary value specified by the escape sequence**.

Table 7-2. String backslash characters

Escape	Meaning
newline	Ignored (continuation line)
\\	Backslash (stores one \)
\'	Single quote (stores ')
\"	Double quote (stores ")
\a	Bell
\b	Backspace
\f	Formfeed
\n	Newline (linefeed)
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
xhh	Character with hex value *hh* (exactly 2 digits)
ooo	Character with octal value *ooo* (up to 3 digits)
\0	Null: binary 0 character (doesn't end string)
\N{ id }	Unicode database ID
uhhhh	Unicode character with 16-bit hex value
Uhhhhhhhh	Unicode character with 32-bit hex value[a]
other	Not an escape (keeps both \ and *other*)

- Some escape sequences allow you to embed absolute binary values into the characters of a string.

```
>>> s = 'a\0b\0c'
>>> s
'a\x00b\x00c'
>>> len(s)
5
```

- In Python, a zero (null) character like this does not terminate a string the way a “null byte” typically does in C.
- In fact, *no* character terminates a string in Python.

```
>>> s = '\001\002\x03'
>>> s
'\x01\x02\x03'
```

- Notice that Python displays nonprintable characters in hex, regardless of how they were specified.

```
>>> x = "C:\py\code"
>>> x
'C:\\py\\code'
```

- if Python does not recognize the character after a \ as being a valid escape code, it simply keeps the backslash in the resulting string. you probably shouldn’t rely on this behavior.

Raw Strings Suppress Escapes

```
myfile = open('C:\new\text.dat', 'w')
```

- This will give error as python finds ‘\n’ in the string which is a newline char.
- Raw strings are useful here.
- If the **letter *r* (uppercase or lowercase)** appears just before the opening quote of a string, it **turns off** the **escape mechanism**.
- The result is that Python retains your backslashes literally, exactly as you type them.

```
myfile = open(r'C:\new\text.dat', 'w')
myfile = open('C:\\new\\text.dat', 'w')
```

- In fact, Python itself sometimes uses this doubling scheme when it prints strings with embedded backslashes.

```
>>> path = r'C:\new\text.dat'
>>> path # Show as Python code
'C:\\new\\text.dat'
```

```
>>> print(path) # User-friendly format
C:\new\text.dat
>>> len(path) # String length
15
```

- Besides directory paths on Windows, raw strings are also commonly used for regular expressions
- Also note that Python scripts can usually use *forward* slashes in directory paths on Windows and Unix. i.e. 'C:/new/text.dat' works when opening file.

Triple Quotes Code Multiline Block Strings

- Python also has a triple-quoted string literal format, sometimes called a ***block string.***
- that is a syntactic convenience for coding multiline text data.
- Single and double quotes embedded in the string's text may be, but do not have to be.

```
>>> test = """ This is a multiline string.
... this is line number 1.
... this is line number 2.
... this is line number 3.
... ***so on """
>>> test
' This is a multiline string.\nthis is line number 1.\nthis is line number 2.\nthis is line number 3.\n***so
on '
```

- the interactive prompt changes to … on continuation lines
- Python collects all the triple-quoted text into a single multiline string, with **embedded newline characters (\n)** at the places where your code has line breaks.
- It retains everything you write with in """…""", even any comments. But it is not a good practice to add in between. You can add before or after.

```
>>> test = """ This is a multiline string.
...          #this is a comment.
...    You should not add comments in multiline strings"""
>>> test
' This is a multiline string.\n\t\t#this is a comment.\n\tYou should not add comments in multiline strings'
```

- If you have to, do this

```
>>> test = (
... "spam\n" # comments here ignored
... "eggs\n" # but newlines not automatic
... )
>>> test
'spam\neggs\n'
```

- Triple-quoted strings are useful anytime you need *multiline text* in your program for example, to embed multiline error messages or HTML, XML, or JSON code in your Python source code files.
- Triple-quoted strings are also commonly used for *documentation strings*, which are string literals that are taken as comments when they appear at specific points in your file

- triple-quoted strings are also sometimes used as a “horribly hackish” way to *temporarily disable* lines of code during development

```
X = 1
"""
import os                       # Disable this code temporarily
print(os.getcwd())
"""
Y = 2
```

- For large sections of code, it’s also easier than manually adding hash marks before each line and later removing them.

Strings in Action

Basic Operations

```
>>> len('abc')          ➔     3
```

- len () function returns the length of a string.

```
>>> 'abc' + 'def' ➔      'abcdef'
```

- You can **concatenate** two or more strings using ‘+’ operator and output will be a new string. Notice it is not as with numeric.

```
>>> "Namaste " * 5 ➔  'Namaste Namaste Namaste Namaste Namaste '
```

- Repetition with * is like adding a string to itself a number of times.
- Notice that operator overloading is at work here already: we’re using the same + and * operators that perform addition and multiplication when using numbers.
- Python doesn’t allow you to mix numbers and strings in + expressions.

```
>>> 'abc'+9
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

- You can iterate over a string using a for loop, which repeat actions and test membership for both characters and substrings with the in expression operator, which is essentially a search.

```
>>> test = "this is a test string"
>>> for c in test: print(c, end=' ')
...
t h i s   i s   a   t e s t   s t r i n g
```

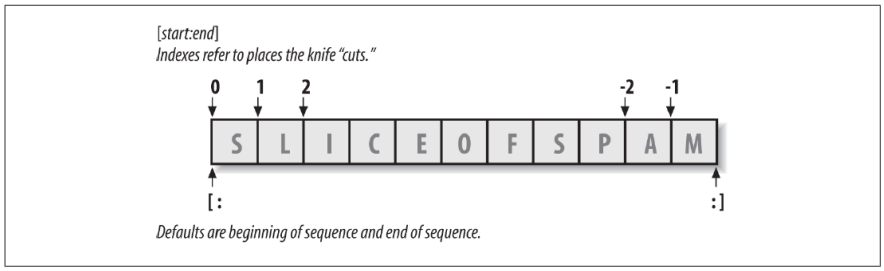
- **in** is much like the **str.find()**

```
>>> 'is' in test
True
```

>>> 'are' in test
False

Indexing and Slicing:

- we can access their components by position.
- characters in a string are fetched by *indexing* - providing the numeric offset of the desired component in square brackets after the string.
- Python offsets start at 0 and end at one less than the length of the string.
- Python also lets you fetch items from sequences such as strings using *test* offsets.



>>> test = "abcde"

>>> test[0] ➔ 'a'

>>> test[3] ➔ 'd'

>>> test[-1] ➔ 'e'

>>> test[len(test) - 1] ➔'e'

>>> test[len(test)]

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

IndexError: string index out of range

Slicing

>>> test[:] ➔ 'abcdewqfrewtgre'

>>> test[0:] ➔ 'abcdewqfrewtgre'

>>> test[2:6] ➔ 'cdew'

>>> test[:-1] ➔ 'abcdewqfrewtgr'

>>> test[-5:] ➔ 'wtgre'

>>> test[-9:-3] ➔ 'qfrewt'

- it allows us to extract an entire *section* (substring) in a single step
- Slices can be used to extract columns of data, chop off leading and trailing text, and more
- When you index a sequence object such as a string on a pair of offsets separated by a colon, Python returns a new object containing the contiguous section identified by the offset pair
- **The left offset is taken to be the lower bound (*inclusive*), and the right is the upper bound (*noninclusive*).**

Indexing *(*S[*i*]*) fetches components at offsets:*

- The first item is at offset 0.
- Negative indexes mean to count backward from the end or right.
- S[0] fetches the first item.
- S[−2] fetches the second item from the end (like S[len(S)−2]).

Slicing *(*S[*i*:*j*]*) extracts contiguous sections of sequences:*

- The upper bound is noninclusive.
- Slice boundaries default to 0 and the sequence length, if omitted.
- S[1:3] fetches items at offsets 1 up to but not including 3.
- S[1:] fetches items at offset 1 through the end (the sequence length).
- S[:3] fetches items at offset 0 up to but not including 3.
- S[:−1] fetches items at offset 0 up to but not including the last item.
- S[:] fetches items at offsets 0 through the end—making a top-level copy of S.

Extended slicing: The third limit and slice objects:

- slice expressions have support for an optional third index, used as a ***step*** (sometimes called a ***stride***).
- The full-blown form of a slice is now **X[*I*:*J*:*K*]**, which means "extract all the items in X, from offset ***I*** through ***J*–1**, by ***K***."
- The third limit, *K*, defaults to +1.

```
>>> test          ➔    'abcdewqfrewtgre'
>>> test[::]      ➔    'abcdewqfrewtgre'
>>> test[::2]     ➔    'aceqrwge'
>>> test[1:9:2]   ➔    'bdwf'
>>> test[1:9:-2]  ➔    ''
>>> test[-1:-9:-2]➔    'egwr'
>>> test[::-1]    ➔    'ergtwerfqwedcba'          #reverse the string
```

String Conversion Tools

```
>>> int("56")      ➔    56

>>> str(56)        ➔    '56'

>>> str(int("56"))➔    '56'

>>> float("1.5") ➔    1.5
```

- **int** function converts a string to a number
- **str** function converts a number to its string representation
- The **repr** function also converts an object to its string representation but returns the object as a string of code that can be rerun to recreate the object.

```
>>> repr("test") ➔    "'test'"
```

>>> repr(56) ➔ '56'

- repr actually calls a magic method __repr__ of object passsed, which gives the **string** containing the representation of the value.

- built-in **ord** function—this returns the actual binary value used to represent the corresponding character in memory.

>>> ord('a') ➔ 97

chr function performs the inverse operation, taking an integer

- code and converting it to the corresponding character function perform the inverse operation, taking an integer code and converting it to the corresponding character

>>> chr(97) ➔ 'a'

>>> int('1101', 2) ➔ 13

>>> bin(13) ➔ '0b1101'

Changing Strings:

- How to modify text information in Python?
- you generally need to build and assign a new string using tools such as concatenation and slicing and then, if desired, assign the result back to the string's original name

```
>>> test = "This is first string"

>>> test = test[0:8] + "second" + test[13:]

>>> test ➔ 'This is second string'


>>> test ➔ 'This is second string'

>>> test = test.replace("second", "third")

>>> test ➔ 'This is third string'
```

String Methods:

```
>>> dir(str)

['__add__', '__class__', '__contains__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count', 'encode', 'endswith', 'expandtabs', 'find',
'format', 'format_map', 'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier', 'islower',
'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower', 'lstrip', 'maketrans', 'partition',
```

'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper', 'zfill']

>>> 'aabbcc$dd'.replace('$', 'SPAM') ➔ 'aaSPAMbbSPAMccSPAMdd'

>>> test.find("third") ➔ 8

- The **find** method returns the offset where the substring appears (by default, searching from the front), or −1 if it is not found. **find** returns the position of a located substring.
- use **replace** with a third argument to limit it to a single or number of substitutions.

```
>>> test = "first " * 4
>>> test ➔ 'first first first first '
>>> test.replace("first", "second", 1) ➔ 'second first first first '
>>> test = "first " * 4
>>> test ➔ 'first first first first '
>>> test.replace("first", "second", 2) ➔ 'second second first first '
```

- The built-in **list** function (an object construction call) builds a new list out of the items in any sequence

```
>>> test = "test"

>>> l = list(test)

>>> l ➔ ['t', 'e', 's', 't']


>>> l ➔ ['t', 'e', 's', 't']

>>> "".join(l) ➔'test'


>>> test = "first " * 5
>>> test ➔ 'first first first first first '
>>> l = test.split()
>>> l ➔ ['first', 'first', 'first', 'first', 'first']

>>> test = "awdw,frefgvreg,wqddc,fewfwe"
>>> l = test.split(',')
>>> l ➔ ['awdw', 'frefgvreg', 'wqddc', 'fewfwe']
```

- Delimiters can be longer than a single character

```
>>> test = "  sfewvgfew  \t"
>>> test.rstrip() ➔ '  sfewvgfew'  #notice, it will not update existing string.


>>> test.upper() ➔ '  SFEWVGFEW  \t'
```

```
>>> test = "a"
>>> test.isalpha()          ➔      False

>>> test = "  sfewvgfew  \t"
>>> test.endswith('\t')     ➔      True
>>> test.startswith(' ')    ➔      True

>>> "fe" in test            ➔      True
```

String Formatting Expressions

- Python also provides a more advanced way to combine string processing tasks—***string formatting*** allows us to perform multiple type-specific substitutions on a string in a single step.

String formatting expressions: **'...%s...' % (values)** - this form is based upon the C language's "printf" model

String formatting method calls: **'...{}...'.format(values)** - this form is derived in part from a same-named tool in C#/.NET

Formatting Expression Basics

To format strings:

1. On the ***left* of the %** operator, provide a **format string** containing one or more embedded conversion targets, each of which starts with a % (e.g., %d).
- On the ***right* of the %** operator, provide the **object** (or objects, embedded in a tuple) that you want Python to insert into the format string on the left in place of the conversion target (or targets).

```
>>> "this is a %s string %d" % ("test", 1) ➔      'this is a test string 1'
```

- you can generally do similar work with multiple concatenations and conversions. However, formatting allows us to combine many steps into a single operation.

Advanced Formatting Expression Syntax

Code	Meaning
s	String (or any object's `str(X)` string)
r	Same as s, but uses `repr`, not `str`
c	Character (int or str)
d	Decimal (base-10 integer)
i	Integer
u	Same as d (obsolete: no longer unsigned)
o	Octal integer (base 8)
x	Hex integer (base 16)
X	Same as x, but with uppercase letters
e	Floating point with exponent, lowercase
E	Same as e, but uses uppercase letters
f	Floating-point decimal
F	Same as f, but uses uppercase letters
g	Floating-point e or f
G	Floating-point E or F
%	Literal % (coded as %%)

Dictionary-Based Formatting Expressions

>>> '%(qty)d more %(food)s' % {'qty': 1, 'food': 'spam'} ➔ '1 more spam'

Formatting Method Basics

>>> test = '{0}, {1} and {2}'
>>> test.format('one', 2, 'three') ➔ 'one, 2 and three'

>>> test = '{motto}, {pork} and {food}'
>>> test.format(motto='spam', pork='ham', food='eggs') ➔ 'spam, ham and eggs'

>>> test = '{motto}, {0} and {food}'
>>> test.format('ham', motto='spam', food='eggs') ➔ 'spam, ham and eggs'

Questions

1. Can the string find method be used to search a list?
2. Can a string slice expression be used on a list?
3. How would you convert a character to its ASCII integer code? How would you convert the other way, from an integer to a character?
4. How might you go about changing a string in Python?
5. Given a string S with the value "s,pa,m", name two ways to extract the two characters in the middle.
6. How many characters are there in the string "a\nb\x1f\000d"?
7. Why might you use the string module instead of string method calls?