第5课　字符串

字符串 string

- 字符串常用的表示方式： 常用英文状态下的单引号、双引号或者三单引号进行表示。
- 注：Python中没有char数据类型。一个字符简单地表示为一个字符的字符串

```
s1 = 'This is a string!'
s2 = "Python is so awesome."
```

- 作为序列数据类型的子类型，字符串可以按元素访问。可以用典型的括号符号索引，也可以执行切片。

```
>>> s1 = "This is a string!"
>>> s2 = "Python is so awesome."
>>> print(s1[3])
s
>>> print(s2[5:15])
n is so aw
```

字符串

- Python 支持中文字符，默认使用UTF8编码格式，无论是一个数字、英文字母，还是一个汉字，都按一个字符对待和处理。

```
>>> s = '中国广东深圳'
>>> len(s)                    #字符串长度，或者包含的字符个数
6

>>> s = '深圳shenzhen'        #中文与英文字符同样对待，都算一个字符
>>> len(s)
10

>>> 大学 = '深圳大学'          #使用中文作为变量名
>>> print(大学)                #输出变量的值
深圳大学
```

字符串

- 字符串属于**不可变序列**类型。

```
>>> s1 = "Python is so awesome."
>>> s1[0] = ”p” #修改字符串的第一个字符,系统报错
TypeError: 'str' object does not support item assignment
```

- “+”运算符可与两个字符串对象一起用于将它们连接在一起，组成一个新的字符串。

```
>>> s1 = "Python is so awesome."
>>> s1 = s1[:13] + "cool." #字符串'+'连接，默认合并
>>> s1
'Python is so cool.'
```

字符串

“*”运算符可用于连接单个字符串对象的多个副本

- 用字符串与正整数进行乘法运算时，相当于创建对应次数的字符串，最后组成一个新的字符串。

```
>>> sample_str = 'hello' *3        #重复创建相应的字符串
>>> print('sample_str: ', sample_str)
sample_str: hellohellohello
```

- 运行结果？

```
>>> sample_str2 = 'hello' *3 + 'hi'
>>> print('sample_str2: ', sample_str2)
```

字符串格式化

```
>>> x = 1235
>>> so="%o" % x
>>> so
"2323"
>>> sh = "%x" % x
>>> sh
"4d3"
>>> se = "%e" % x
>>> se
"1.235000e+03"

>>> "%s"%65
"65"
>>> "%s"%65333
"65333”

>>> "%d"%"555"
TypeError: %d format: a number is required, not str
```

%o 八进制整数

%x 十六进制整数

%e 指数（基底写为e）

%s 字符串（采用str()的显示）

%???

字符串格式化输出

- 字符串格式化输出通过字符串对象的内置方法完成

```
str.format(*args, **kwargs)
```

- *args参数表示format接受可变数量的**位置参数**，**kwargs表示format接受可变数量的**关键字参数**。
- 调用的字符串是包含文本或由大括号{ }分隔的替换字段。
- 每个字段都包含**位置参数的数字索引**或**关键字参数的名称**。
- 每个字段都被相应参数的字符串值替换。

字符串格式化:位置参数

```
>>> '{0}, {1}, {2}'.format('a', 'b', 'c')
'a, b, c'

>>> '{}, {}, {}'.format('a', 'b', 'c')
'a, b, c'

>>> '{2}, {1}, {0}'.format('a', 'b', 'c')
'c, b, a'

>>> ' {2}, {1}, {0} '.format(*'abc') #这是啥?
'c, b, a'

>>> '{0}{1}{0}'.format('abra', 'cad')
'abracadabra'
```

字符串格式化:关键字参数

```
>>> 'Coords: {lat}, {long}'.format(lat='37.24N', long='-115.81W')
'Coords: 37.24N, -115.81W'

>>> 'Coords: {lat}, {long}'.format(long='-115.81W', lat='37.24N')
'Coords: 37.24N, -115.81W'


>>> coord = {'lat': '37.24N', 'long': '-115.81W'} #字典

>>> 'Coords: {lat}, {long}'.format(*coord)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'lat'

>>> 'Coords: {lat}, {long}'.format(**coord)
'Coords: 37.24N, -115.81W'
```

字符串格式化

- 可以访问传递给format的对象的属性和方法。

```
>>> c = 2+3j
>>> '{0} has real part {0.real} and imaginary part
{0.imag}.'.format(c)
'(2+3j) has real part 2.0 and imaginary part 3.0.'


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

字符串格式化

练习1：

```
>>> position = (5,8,13)
>>> print("X:{0[0]};Y:{0[1]};Z:{0[2]}".format(position))
```

输出什么？

练习2：

```
>>> print('{name} loves {food}'.format(name='Tom',
food='apples'))
```

输出什么？

字符串格式化

练习1：

```
>>> position = (5,8,13)
>>> print("X:{0[0]};Y:{0[1]};Z:{0[2]}".format(position))
```

X:5;Y:8;Z:13

练习2：

```
>>> print('{name} loves {food}'.format(name='Tom',
food='apples'))
```

Tome loves apples

字符串格式化

- 对正和对齐

```
>>> '{:<30}'.format('left aligned')
'left aligned                  '

>>> '{:>30}'.format('right aligned')
'                 right aligned'

>>> '{:^30}'.format('centered')
'           centered           '

>>> '{:*^30}'.format('centered') # 用'*' 填充
'***********centered***********'
```

内置字符串方法

- find( )、rfind()、index()、rindex()、count()

✓ find()和rfind方法分别用来查找一个字符串在另一个字符串指定范围（默认是整个字符串）中**首次**和**最后**一次出现的位置，如果不存在则返回-1；

✓ index()和rindex()方法用来返回一个字符串在另一个字符串指定范围中**首次**和**最后**一次出现的位置，如果不存在则抛出异常；

✓ count()方法用来返回一个字符串在另一个字符串中出现的**次数**。

内置字符串方法

```
>>> s='aaa'
>>> s.find('a')
0
>>> s.rfind('a')
2

>>> s="shenzhen"
>>> s.find("en")
2
>>> s.rfind("en")
6
>>> s.index("en")
2
>>> s.rindex("en")
6
>>> s.count("en")
2
```

内置字符串方法

练习

```
>>> "whenever".find("never")
??
>>> "whenever".find("what")
??
>>> "whenever".index("what")
??
```

内置字符串方法

练习

```
>>> "whenever".find("never")
3
>>> "whenever".find("what")
-1
>>> "whenever".index("what")
Traceback (most recent call last):
  File “<stdin>”, line 1, in <module>
ValueError: substring not found
```

内置字符串方法

- split()、rsplit()、partition()、rpartition()

✓ split()和rsplit()方法分别用来以指定字符为分隔符，将字符串**左端**和**右端**开始将其分割成多个字符串，并返回包含分割结果的**列表**；

✓ partition()和rpartition()用来以指定字符串为分隔符将原字符串分割为3部分，即分隔符前的字符串、分隔符字符串、分隔符后的字符串（如果指定的分隔符不在原字符串中，则返回原字符串和两个空字符串）。

内置字符串方法

```
>>> s="apple,peach,banana,pear"
>>> li=s.split(",")
>>> li
["apple", "peach", "banana", "pear"]

>>> s.partition(',')
('apple', ',', 'peach,banana,pear')
>>> s.rpartition(',')
('apple,peach,banana', ',', 'pear')
>>> s.rpartition('banana')
('apple,peach,', 'banana', ',pear')

>>> s = "2014-10-31"
>>> t=s.split("-")
>>> print(t)
['2014', '10', '31']
```

内置字符串方法

- 对于split()和rsplit()方法，如果不指定分隔符，则字符串中的任何空白符号（包括空格、换行符、制表符等等）都将被认为是分隔符，返回包含最终分割结果的列表。

```
>>> s = 'hello world \n\n My name is Tom   '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Tom']
>>> s = '\n\nhello world \n\n\n My name is Tom   '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Tom']
>>> s = '\n\nhello\t\t world \n\n\n My name\t is Tom   '
>>> s.split()
['hello', 'world', 'My', 'name', 'is', 'Tom']
```

内置字符串方法

- split()和rsplit()方法允许指定最大分割次数

```
>>> s = '\n\nhello\t\t world \n\n\n My name is Tom   '
>>> s.split(None,1)
['hello', 'world \n\n\n My name is Tom   ']
>>> s.split(None,2)
['hello', 'world', 'My name is Tom   ']
>>> s.rsplit(None,2)
['\n\nhello\t\t world \n\n\n My name', 'is', ' Tom']
['hello', 'world', 'My', 'name', 'is', ' Tom']
>>> s.split(maxsplit=4)
['hello', 'world', 'My', 'name', 'is Tom   ']
>>> s.split(maxsplit=100)
['hello', 'world', 'My', 'name', 'is', 'Tom']
```

内置字符串方法

- **不传递任何参数**：使用任何空白字符作为分隔符，把连续多个空白字符看作一个；
- **明确传递参数**指定split()使用的分隔符时，情况略有不同。

```
>>> 'a,,,bb,,ccc'.split(',')
                                #每个逗号都被作为独立的分隔符
['a', '', '', 'bb', '', 'ccc']
>>> 'a\t\t\tbb\t\tccc'.split('\t')
                                #每个制表符都被作为独立的分隔符
['a', '', '', 'bb', '', 'ccc']
>>> 'a\t\t\tbb\t\tccc'.split()
                                #连续多个制表符被作为一个分隔符
['a', 'bb', 'ccc']
```

内置字符串方法

练习：

```
>>> "Python programming is fun!".split()
??

>>> "mississippi".partition("siss")
??
```

内置字符串方法

练习：

```
>>> "Python programming is fun!".split()
['Python', 'programming', 'is', 'fun!']

>>> "mississippi".partition("siss")
('mis', 'siss', 'ippi')
```

内置字符串方法

- 字符串连接join()

```
>>> li=["apple", "peach", "banana", "pear"]
>>> sep=","
>>> s=sep.join(li)
>>> s
"apple,peach,banana,pear"
```

- 查找替换replace()，类似于“查找与替换”功能

```
>>> s="中国，中国"
>>> s
"中国，中国"
>>> s2=s.replace("中国", "中华人民共和国")
>>> s2
"中华人民共和国，中华人民共和国"
```

内置字符串方法

- 练习

```
>>> '-'.join(['555', '867', '5309'])
??

>>> " ".join(['Python', 'is', 'awesome'])
??

>>> "whenever".replace("ever", "ce")
??

>>> "ab12cd3412cd".replace("12", "21", 1)
??
```

内置字符串方法

- 练习

```
>>> "-".join(['555','867','5309'])
'555-867-5309'

>>> " ".join(['Python', 'is', 'awesome'])
'Python is awesome’

>>> "whenever".replace("ever", "ce")
'whence’

>>> "ab12cd3412cd".replace("12", "21", 1)
'ab21cd3412cd'
```

内置字符串方法

- lower()、upper()、capitalize()、title()、swapcase()

```
>>> s = "What is Your Name?"
>>> s.lower()                    #返回小写字符串
'what is your name?'
>>> s.upper()                    #返回大写字符串
'WHAT IS YOUR NAME?'
>>> s.capitalize()               #字符串首字符大写
'What is your name?'
>>> s.title()                    #每个单词的首字母大写
'What Is Your Name?'
>>> s.swapcase()                 #大小写互换
'wHAT IS yOUR nAME?'
```

内置字符串方法

- maketrans() 生成字符映射表
- translate()根据映射表中定义的对应关系转换字符串并替换其中的字符

```
#创建映射表，将字符"abcdef123"一一对应地转换为"uvwxyz@#$"
>>> table = ''.maketrans('abcdef123', 'uvwxyz@#$')
>>> s = "Python is a great programming language. I like
   it!"

>>> s.translate(table) #按映射表进行替换
'Python is u gryut progrumming lunguugy. I liky it!'
```

内置字符串方法

- maketrans() 、translate()应用：加密

```
>>> def encryption(s):
        before = # 明文
        after =  # 密文
        table = ''.maketrans(before, after)      #创建映射表
        return s.translate(table)

>>> s = "Python is a greate programming language. I like
it!"
>>> encryption(s)
```

内置字符串方法

- strip（ch） – 删除前导和尾随字符。
- ch字符串指定要删除的字符集（默认为空白）。

```
>>> s = " abc  "
>>> s2 = s.strip()                          #删除空白字符
>>> s2
"abc"
>>> '\n\nhello world   \n\n'.strip()        #删除空白字符
'hello world'
>>> "aaaassddf".strip("a")
"ssddf"
>>> "aaaassddf".strip("af")
"ssdd"
```

内置字符串方法

- rstrip()、lstrip()
- 删除字符串右端、左端指定字符

- 练习

```
>>> "***Python programming is fun***".strip("*")
```

??

```
>>> "aaaassddfaaa".rstrip("a")
```

??

```
>>> "aaaassddfaaa".lstrip("a")
```

??

内置字符串方法

- rstrip()、lstrip()
- 删除字符串右端、左端指定字符

- 练习

```
>>> "***Python programming is fun***".strip("*")
   'Python programming is fun'

>>> "aaaassddfaaa".rstrip("a")
'aaaassddf’

>>> "aaaassddfaaa".lstrip("a")
'ssddfaaa'
```

内置字符串方法

- **注意**：strip()参数指定的字符串并不作为一个整体对待
- 在原字符串的两侧删除参数字符串中包含的所有字符

```
>>> 'aabbccddeeeffg'.strip('af')  #字母f不在字符串两侧，所以不删除
'bbccddeeeffg'
>>> 'aabbccddeeeffg'.strip('gaf')
'bbccddeee'
>>> 'aabbccddeeeffg'.strip('gaef')
'bbccdd'
>>> 'aabbccddeeeffg'.strip('gbaef')
'ccdd'
```

内置字符串方法

- s.startswith(t)、s.endswith(t)，判断字符串是否以指定字符串开始或结束

```
>>> "mississippi".startswith("mis")
True
>>> "mississippi".endswith("ssi")
False
>>> "mississippi".endswith("ssi", 0, 8)
True
```

- center()、ljust()、rjust()，格式对齐

```
>>> 'Hello world!'.center(20)          #居中对齐，以空格进行填充
'    Hello world!    '
>>> 'Hello world!'.center(20, '=')     #居中对齐，以字符=进行填充
'====Hello world!===='
```

内置字符串方法

- s.isalpha(), s.isdigit(), s.isalnum(), s.isspace()

  - 如果字符串s分别由字母字符、数字、字母或数字，和完全空白字符组成，则返回True。

- s.islower(), s.isupper()

  - 如果字符串s分别为**全小写**和**全大写**，则返回True。

```
>>> "WHOA".isupper()
True
>>> "12345".isdigit()
True
>>> " \n ".isspace()
True
>>> "hello!".isalpha()
False
```

String 模块

- 以上内置的字符串方法都是任何字符串对象的方法。它们不需要导入任何模块或任何东西。
- 但是，有一个字符串模块，它提供了一些额外的有用的字符串工具。它定义了有用的**字符串常量**、**字符串格式化**类

```
>>> import string
>>> string.ascii_letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.ascii_uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.digits
'0123456789'
>>> string.hexdigits
'0123456789abcdefABCDEF'
```

String 模块

- 以上内置的字符串方法都是任何字符串对象的方法。它们不需要导入任何模块或任何东西。
- 但是，有一个字符串模块，它提供了一些额外的有用的字符串工具。它定义了有用的**字符串常量**、**字符串格式化**类

```
>>> import string
>>> string.lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.uppercase
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.letters
'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
>>> string.octdigits
'01234567'
>>> string.punctuation
'!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~'
```

String 模块

- string.whitespace – 一个包含所有被认为是空格的字符的字符串。在大多数系统中，这包括字符空间、制表符、换行符、回车符、窗体换行符和垂直制表符。
- string.printable – 是可打印的字符串。这是数字、字母、标点和空白的组合。

```
>>> import string
>>> string.whitespace
'\t\n\x0b\x0c\r '

>>> string.printable
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN
OPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~
\t\n\r\x0b\x0c'
```

正则表达式
Regular Expression

- 正则表达式是字符串处理的有力工具和技术。
- 正则表达式使用某种预定义的模式去匹配一类具有共同特征的字符串，主要用于处理字符串，可以快速、准确地完成复杂的查找、替换等处理要求。
- Python中，re模块提供了正则表达式操作所需要的功能。

正则表达式

- 最简单的正则表达式包含一个普通字符。

```
'A'    'b'    '0'    '@'
```

- 这个字符只是“匹配”本身——也就是说，这些正则表达式定义了只包含字符本身的集合。

- 我们还可以连接普通字符来创建一个长度超过一个字符的字符串。

```
'word'    '@abc123'
```

- 正则表达式'word'只定义包含字符串'word'的集合，正则表达式'@abc123'只定义包含字符串'@abc123'的集合。

正则表达式

- 唯一与自身**不匹配**的字符是特殊字符或元字符。

```
. ^ $ * + ? { } [ ] \ | ( )
```

- 点（.）元字符匹配**除换行符**以外的任意单个字符。

```
'.' = {"a", "b", "c", ..., "A", "B", ... "@", ...}
```

- 插入符号（ˆ）元字符匹配行首，匹配以ˆ后面的字符开头的字符串。

```
'^a' = {"a", "apple", "air", "age", "armor", "a new day",...}

'^up' = {"up", "up and away", "upper", "upon a hill", ...}
```

正则表达式

- $元字符匹配行尾，匹配以$之前的字符结束的字符串

```
'ear$' = {"ear", "clear", "top gear", ...}
```

- *元字符匹配位于*之前的字符或子模式的0次或多次出现

```
'b*' = {"", "b", "bb", "bbb", ...}
'ab*' = {"a", "ab", "abb", "abbb", ...}
```

- 注：

```
'(ab)*' = {"", "ab", "abab", "ababab", ...}
```

正则表达式

- *元字符匹配位于*之前的字符或子模式的**0次**或多次出现
- +元字符匹配位于+之前的字符或子模式的**1次**或多次出现

```
'(ab)+' = {"ab", "abab", "ababab", ...}
'(ab)*' = {"", "ab", "abab", "ababab", ...}
```

- ？元字符匹配位于?之前的0个或1个字符

```
'(ab)?' = {"", "ab"}
```

正则表达式

元字符	功能说明
.	匹配除换行符以外的任意单个字符
*	匹配位于*之前的字符或子模式的0次或多次出现
+	匹配位于+之前的字符或子模式的1次或多次出现
-	在[]之内用来表示范围
|	匹配位于|之前或之后的字符
^	匹配行首，匹配以^后面的字符开头的字符串
$	匹配行尾，匹配以$之前的字符结束的字符串
?	匹配位于?之前的0个或1个字符。
\	表示位于\之后的为转义字符
\num	此处的num是一个正整数。例如，“(.)\1”匹配两个连续的相同字符
\f	换页符匹配
\n	换行符匹配

正则表达式

元字符	功能说明
\r	匹配一个回车符
\b	匹配单词头或单词尾
\B	与\b含义相反\b含义相反
\d	匹配任何数字，相当于[0-9]
\D	与\d含义相反，等效于[^0-9]
\s	匹配任何空白字符，包括空格、制表符、换页符，与[\f\n\r\t\v] 等效
\S	与\s含义相反
\w	匹配任何字母、数字以及下划线，相当于[a-zA-Z0-9_]
\W	与\w含义相反\w含义相反，与“[^A-Za-z0-9_]”等效
()	将位于()内的内容作为一个整体来对待
{}	按{}中的次数进行匹配
[]	匹配位于[]中的任意一个字符
[^xyz]	反向字符集，匹配除x、y、z之外的任何字符
[a-z]	字符范围，匹配指定范围内的任何字符
[^a-z]	反向范围字符，匹配除小写英文字母之外的任何字符

re模块主要方法

方法	功能说明
compile(pattern[, flags])	创建模式对象
escape(string)	将字符串中所有特殊正则表达式字符转义
findall(pattern, string[, flags])	列出字符串中模式的所有匹配项
finditer(pattern, string, flags=0)	返回包含所有匹配项的迭代对象，其中每个匹配项都是match对象
fullmatch(pattern, string, flags=0)	尝试把模式作用于整个字符串，返回match对象或None
match(pattern, string[, flags])	从字符串的开始处匹配模式，返回match对象或None
purge()	清空正则表达式缓存
search(pattern, string[, flags])	在整个字符串中寻找模式，返回match对象或None
split(pattern, string[, maxsplit=0])	根据模式匹配项分隔字符串
sub(pat, repl, string[, count=0])	将字符串中所有pat的匹配项用repl替换，返回新字符串，repl可以是字符串或返回字符串的可调用对象，该可调用对象作用于每个匹配的match对象
subn(pat, repl, string[, count=0])	将字符串中所有pat的匹配项用repl替换，返回包含新字符串和替换次数的二元元组，repl可以是字符串或返回字符串的可调用对象，该可调用对象作用于每个匹配的match对象

re模块的对象

- 首先使用re模块的compile()方法将正则表达式编译生成正则表达式对象，然后再使用正则表达式对象提供的方法进行字符串处理。
- 使用编译后的正则表达式对象可以**提高字符串处理速度**。
- 例子：
- search方法用于在整个字符串中进行搜索；
- findall方法用于在字符串中查找所有符合正则表达式的字符串列表。

re模块的对象

\b	匹配单词头或单词尾
\w	匹配任何字母、数字以及下划线，相当于[a-zA-Z0-9_]
+	匹配位于+之前的字符或子模式的1次或多次出现

findall(pattern, string[, flags])	列出字符串中模式的所有匹配项

```
>>> import re
>>> example = 'The Hong Kong University of Science and Technology'
>>> pattern = re.compile(r'\bU\w')          #查找以U开头长度为2的字符串
>>> pattern.findall(example)
['Un']
>>> pattern = re.compile(r'\bU\w+')         #查找以U开头的单词
>>> pattern.findall(example)                #使用正则表达式对象的findall()方法
['University ']
```

补充：转义字符的输出

```
>>> pattern = re.compile(r ' \bU\w ')          #为什么需要 r ？？？
```

```
>>> print("\n")

>>> print(r"\n")
\n

>>> print("\t")

>>> print(r"\t")
\t
```

re模块的对象

\b	匹配单词头或单词尾
\w	匹配任何字母、数字以及下划线，相当于[a-zA-Z0-9_]
+	匹配位于+之前的字符或子模式的1次或多次出现

findall(pattern, string[, flags])	列出字符串中模式的所有匹配项

```
>>> import re
>>> example = 'The Hong Kong University of Science and Technology'
>>> pattern = re.compile(r'\w+y\b')          #查找以y结尾的单词
>>> pattern.findall(example)
['University', 'Technology']


>>> pattern = re.compile(r'\bU\w+y\b')  # ？ ？ ？
>>> pattern.findall(example)
['University']
```

re模块的对象

[]	匹配位于[]中的任意一个字符
*	匹配位于*之前的字符或子模式的0次或多次出现

search(pattern, string[, flags])	在整个字符串中寻找模式，返回match对象或None

```
>>> re.search(r':[0-9]*', "11:00.")
<re.Match object; span=(2, 5), match=':00'>

>>> re.search('ab*',"abbc")
<re.Match object; span=(0, 3), match='abb'>

>>> re.search('(ab)*',"abbc")
<re.Match object; span=(0, 2), match='ab'>
```

re模块的另一种使用方法：直接调用re里的函数，不需要compile