Corso di **Python**

Lezione 5

Elaborare stringhe

Editor: Davide Brunato
Scuola Internazionale Superiore di Studi Avanzati di Trieste

Tipi di letterali stringa

- I letterali stringa possono essere definiti con un prefisso per variarne l'interpretazione
- Prefissi possibili, singoli e combinati:
  - **`r`** : stringa grezza (*raw string*)
  - **`u`** : stringa unicode
  - **`b`** : stringa bytes
  - **`ur`** : stringa unicode raw
  - **`br`** : stringa bytes raw
  - **`f`** : stringa formattata (PEP 498, sarà in Python 3.6+)

Raw string

- Una raw string è un letterale stringa per cui non sono interpretate le sequenze di escape
- Per definire una *raw string* si mette il prefisso **`r`** o **`R`** immediatamente prima della stringa:

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

- Una raw string è utile soprattutto quando dobbiamo costruire dei pattern di ricerca che includono dei backslash (\)
- Una raw string non può terminare con un backslash per non confondersi con il fine stringa:

```
>>> r'\\\Ciao\\\'
  File "<stdin>", line 1
    r'\\\Ciao\\\'
                 ^
SyntaxError: EOL while scanning string literal
```

Letterale stringa Unicode

- Si possono specificare delle stringhe esplicitamente Unicode mettendo il prefisso **u** o **U**:

```
u'Questa è una stringa Unicode'
```

- I letterali stringa Unicode possono includere altre sequenze di escape:

```
\u<hex_Unicode_value>
\N{standard_Unicode_name}
```

- In Python 2 i letterali stringa sono per default in ASCII, mentre in Python 3 sono per default Unicode:
  - se si crea un codice in Python 3 ma che deve essere retrocompatibile con la versione 2 è bene stare attenti che i letterali stringa non contengano caratteri non ASCII e in caso contrario mettere il prefisso **u**
  - Attenzione invece ad usare identificatori con caratteri non ASCII, perché non sono retrocompatibili
- Per i dettagli è utile vedersi lo specifico HOWTO su Unicode:
  - https://docs.python.org/3/howto/unicode.html

Letterale di tipo **bytes**

- Questo tipo di letterale stringa è definito quando il prefisso inizia con **`b`** o **`B`**
- In **Python 3** è effettivamente definito il tipo `bytes` che permette di rappresentare stringhe strettamente ASCII (in Python 3 le stringhe sono tutte Unicode):

```
>>> type('Una stringa qualsiasi')   # Stringa di tipo Unicode
<class 'str'>
>>> type(u'Una stringa qualsiasi')  # Stringa di tipo Unicode
<class 'str'>
>>> type(b'Una stringa qualsiasi')  # Stringa di tipo ASCII
<class 'bytes'>
```

- In **Python 2** invece il tipo `bytes` non c'è perché le stringhe sono ASCII per default:

```
>>> type('Una stringa qualsiasi')   # Stringa di tipo ASCII
<type 'str'>
>>> type(u'Una stringa qualsiasi')  # Stringa di tipo Unicode
<type 'unicode'>
>>> type(b'Una stringa qualsiasi')  # Stringa di tipo ASCII
<type 'str'>
```

Il tipo **unicode** in Python 2

- In **Python 2**, per distinguere stringhe ASCII da stringhe Unicode, è definito il tipo **`unicode`**:

```
>>> unicode('stringa senza accenti')
u'stringa senza accenti'
>>> type(u'una stringa')
<type 'unicode'>
>>> type('una stringa')
<type 'str'>
```

- Python 3 il tipo **`unicode`** è stato tolto perché le stringhe sono tutte Unicode:

```
>>> type('ciao')
<class 'str'>
>>> type(u'ciao')
<class 'str'>
>>> unicode('ciao')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'unicode' is not defined
```

Il tipo **bytes** in Python 3

- In **Python 3**, per ovviare al fatto che tutte le stringhe sono Unicode, è definito il tipo **`bytes`**:

```
>>> type(b'Ciao')
<class 'bytes'>
>>> type('Ciao')
<class 'str'>
>>> type(bytes('Ciao', 'ascii'))
<class 'bytes'>
```

- In **Python 2** il tipo **`bytes`** era nominalmente già definito, ma corrisponde a tutti gli effetti al tipo stringa:

```
>>> type(bytes('ciao'))
<type 'str'>
```

La funzione **chr**

- Ritorna una stringa composta dal carattere corrispondente al codice numerico passato come parametro:

```
>>> chr(97)
'a'
```

- In Python 2 il parametro è compreso tra 1-255 e per codici Unicode si usa la funzione addizionale **unichr**:

```
>>> chr(960)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: chr() arg not in range(256)
>>> unichr(960)
u'\u03c0'
>>> print(unichr(960))
π
```

- In Python 3 *unichr* non c'è più ma *chr* accetta tutti i codici Unicode
- La funzione inversa di chr è **ord**, che da un carattere ASCII/Unicode ritorna il codice corrispondente:

```
>>> ord(u'\u03c0')
960
```

Codifica di stringhe unicode

- Per codificare una stringa Unicode si può applicare la funzione **encode**, che ritorna la sua rappresentazione come stringa ad 8-bit, secondo la codifica specificata:

```
>>> u = u'Questa è una stringa Unicode'
>>> u.encode('utf-8')     # Alternative equivalenti: 'utf', 'utf8', 'UTF' ...
'Questa \xc3\xa8 una stringa Unicode'
>>> u.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\xe8' in position 7:
ordinal not in range(128)
>>> u.encode('ascii', 'ignore')   # Salta i caratteri non codificabili
'Questa  una stringa Unicode'
>>> u.encode('ascii', 'replace')  # Mette un ? al posto di caratteri non
codificabili
'Questa ? una stringa Unicode'
```

- In Python 3 la funzione ritorna una stringa bytes:

```
>>> u.encode('utf-8')
b'Questa \xc3\xa8 una stringa Unicode'
```

Decodifica di stringhe Unicode

- La funzione inversa di encode è **decode**
  - Si applica a stringhe 8-bit contenenti codici Unicode
  - Ritorna una stringa Unicode

- Esempio:

```
>>> u = 'Pi-Greco: ' + unichr(960)
>>> print(u)
Pi-Greco: π
>>> u
u'Pi-Greco: \u03c0'
>>> utf8_version = u.encode('utf-8')   # Codifica in UTF-8
>>> type(u), type(utf8_version), utf8_version
(<type 'unicode'>, <type 'str'>, 'Pi-Greco: \xcf\x80')
>>> u2 = utf8_version.decode('utf-8')   # Decodifica con UTF-8
>>> type(u2), u == u2
(<type 'unicode'>, True)
```

Stringhe multilinea

- Le stringhe possono essere multilinea:

```
"Questa stringa è composta\
da più linee"   # Commento non ammesso sulla linea precedente
```

  meglio con la notazione a tre apici se la stringa è molto lunga:

```
"""Questa stringa è veramente
molto lunga ed ha 3
linee."""   # Commenti non ammessi sulle linee precedenti
```

- Per specificare invece stringhe di una sola linea molto lunghe si possono usare le parentesi tonde per andare a capo:

```
s = ('questa è una stringa di una sola lin'  # Commento intermedio!
     'ea ma è molto, molto lunga.')          # Commento finale
```

  Le stringhe sono considerate adiacenti e sono semplicemente concatenate tra loro.

Concatenazione di stringhe

- L'operatore **+** concatena due stringhe:

```
>>> a = "Uno"
>>> b = "Due"
>>> c = "Tre"
>>> a + b
'UnoDue'
>>> a + b + c
'UnoDueTre'
```

- L'operatore ***** concatena N volte la stringa:

```
>>> a * 10
'UnoUnoUnoUnoUnoUnoUnoUnoUnoUno'
>>> width = 20
>>> "*" * width
'********************'
```

- Si possono usare anche le forme **+=** e ***=**, ricondando però che riassegnando la stringa non viene modificata ma ne viene creata una nuova e associata al nome della variabile in sostituzione di quella vecchia

Comparazione di stringhe

- Oltre agli operatori di uguaglianza e diversità è possibile fare un confronto lessicografico:

```
>>> "uno" == "uno"
True
>>> "uno" == "UNO"    # La comparazione è case sensitive
False
>>> "uno" != "due"
True
>>> "abc" < "abd"
True
>>> "abc " <= "abc"
False
```

- La comparazione funziona anche con tipi di stringhe diverse:

```
>>> b"uno" == u"uno"  # Comparazione tra una stringa Bytes e una Unicode
True
```

Sottostringhe

- È semplice generare sottostringhe usando l'operatore di **slicing**:

```
>>> s = "Caio Sempronio\n"
>>> s
'Caio Sempronio\n'
>>> s[:-1]
'Caio Sempronio'
>>> s[5:-1]
'Sempronio'
>>> s[5:6]
'S'
>>> s[:]
'Caio Sempronio\n'
```

- Con l'operatore di appartenenza **`in`** – **`not in`** si può testare se è sottostringa di una stringa data:

```
>>> "aio Sempr" in s
True
>>> "CaioSempr" in s        # Non è un confronto sui singoli caratteri!!
False
>>> "Sempronio" not in s
False
```

La libreria **string**

- Questa libreria include funzionalità aggiuntive dei tipi stringa:
  - Costanti predefinite
  - Classi per formattazione e template
- Libreria ampiamente rivista nel passaggio a Python 3:
  - Metodi spostati quasi tutti nei tipi di dati stringa (*str* e *bytes*)
  - Modifiche apportate in modo da rendere automatica la conversione da 2 a 3
- Nel modulo string ci sono diverse costanti utili (hanno tutte nomi in minuscolo perché definite agli albori del linguaggio):

```
>>> import string
>>> string.ascii_lowercase
'abcdefghijklmnopqrstuvwxyz'
>>> string.digits
'0123456789'
```

- Documentazione per Python 3 sulle stringhe:
  - https://docs.python.org/3/library/string.html
  - https://docs.python.org/3/library/stdtypes.html#string-methods

Formattazione di stringhe

- Storicamente è definito l'operatore % di *formattazione e interpolazione di stringhe*:

```
>>> os = "Linux"; versione = 3
>>> "Sistema Operativo: %s" % os
'Sistema Operativo: Linux'
>>> "OS: %s, Ver: %d" % (os, versione)
'OS: Linux, Ver: 3'
>>> "OS: %(os)s, Ver: %(ver)03d" % {"os": os, "ver": versione}
'OS: Linux, Ver: 003'
```

- C'è una certa flessibilità quando è definita una conversione per il tipo di dato:

```
>>> "OS: %s, Versione: %s" % (os, versione)
'OS: Linux, Versione: 3'
>>> "OS: %s, Versione: %f" % (os, versione)
'OS: Linux, Versione: 3.000000'
```

- Documentazione ufficiale: https://docs.python.org/library/stdtypes.html#string-formatting

Il metodo **format**

- Dalla versione 2.6 è disponibile il metodo specifico **format**, che permette di lavorare con argomenti posizionali:

```
>>> "OS: {0}, Ver: {1}".format(os, versione)
'OS: Linux, Ver: 3'
>>> "OS: {}, Ver: {}".format(os, versione)
'OS: Linux, Ver: 3'
```

- Con format è possibile sfruttare il passaggio di argomenti e argomenti chiave:

```
>>> "OS: {0}, Ver: {1}".format(*(os, versione))
'OS: Linux, Ver: 3'
>>> "OS: {os}, Ver: {ver}".format(**{"os": os, "ver": versione})
'OS: Linux, Ver: 3'
```

- **format** è il nuovo standard per la formattazione di stringhe in Python 3 ed è da preferire rispetto all'operatore **%** (che comunque rimane ...)
- Documentazione ufficiale sulla sintassi di formattazione:
  - https://docs.python.org/library/string.html#formatstrings

Template strings

- È supportata la sostituzione stringhe con template basati sul carattere **$**
- Per la gestione si utilizza l'oggetto Template del modulo strings:

```
>>> import string
>>> s = string.Template('$who likes ${what}')  # $key == ${key}
>>> s.substitute(who='tim', what='kung pao')
'tim likes kung pao'
>>> d = dict(who='tim')
>>> string.Template('Give $who $100').substitute(d)
Traceback (most recent call last):
...
ValueError: Invalid placeholder in string: line 1, col 11
>>> string.Template('$who likes $what').substitute(d)
Traceback (most recent call last):
...
KeyError: 'what'
>>> string.Template('$who likes $what').safe_substitute(d)
'tim likes $what'
```

- Documentazione ufficiale: https://docs.python.org/library/string.html#template-strings

Letterali stringa formattati

- Saranno introdotte da Python 3.6:
  - Letterali con prefisso **f**
  - Definite da PEP 498: https://www.python.org/dev/peps/pep-0498/
  - Colmano un gap rispetto ad altri linguaggi di scripting
- Esempio di come funzioneranno:

```
>>> import datetime
>>> name = 'Fred'
>>> age = 50
>>> anniversary = datetime.date(1991, 10, 12)
>>> f'My name is {name}, my age next year is {age+1}, my anniversary is
{anniversary:%A, %B %d, %Y}.'
'My name is Fred, my age next year is 51, my anniversary is Saturday,
October 12, 1991.'
>>> f'He said his name is {name!r}.'
"He said his name is 'Fred'."
```

I metodi **split** e **join**

- Fondamentali per spacchettare e riassemblare stringhe
- **split** ritorna una lista con parole estratte dalla stringa:

```
str.split(sep=None, maxsplit=-1)

>>> s = "Caio Tizio Sempronio"
>>> s.split()
['Caio', 'Tizio', 'Sempronio']
>>> "guest:x:59000:59000::/home/guest:/bin/bash".split(':')
['guest', 'x', '59000', '59000', '', '/home/guest', '/bin/bash']
```

- **join** concatena liste o tuple di stringhe inserendo usando la stringa come separatore:

```
str.join(iterable)

>>> l = ['Caio', 'Tizio', 'Sempronio']
>>> ' '.join(l)
'Caio Tizio Sempronio'
>>> ':'.join(['guest', 'x', '59000', '59000', '', '/home/guest', '/bin/bash'])
'guest:x:59000:59000::/home/guest:/bin/bash'
```

Metodi sui caratteri

- Metodi per lo strip di caratteri:

```
str.strip([chars])
str.lstrip([chars])
str.rstrip([chars])
```

- Metodi per il cambio del case dei caratteri:

```
str.lower()
str.upper()
str.swapcase()
str.capitalize()
str.title()
string.capwords(s, sep=None)  # Split, capitalize, join
```

Metodi di partizionamento

- Oltre a strip ci sono altri metodi per suddividere una stringa in più porzioni:

```
str.split(sep=None, maxsplit=-1)
str.rsplit(sep=None, maxsplit=-1)
str.partition(sep)                  # Split first, returning 3-tuple
str.rpartition(sep)                 # Split last, returning 3-tuple
```

- Esempi:

```
>>> s = "GNU's Not Unix"
>>> s.split()
["GNU's", 'Not', 'Unix']
>>> s.rsplit()
["GNU's", 'Not', 'Unix']
>>> s.rsplit(maxsplit=1)
["GNU's Not", 'Unix']
>>> s.split(maxsplit=1)
["GNU's", 'Not Unix']
>>> s.partition(' ')
("GNU's", ' ', 'Not Unix')
```

Metodi di formattazione

- Metodi per l'allineamento con padding e senza troncamento della stringa:

```
str.ljust(width[, fillchar])
str.rjust(width[, fillchar])
str.center(width[, fillchar])
```

- Riempimento con zeri su stringhe *numeriche*:

```
str.zfill(width)
```

Metodi di *find & replace*

- Metodi di ricerca di sottostringa:

```
str.find(sub[, start[, end]])   # Lowest index or -1
str.rfind(sub[, start[, end]])  # Highest index or -1
str.index(sub[, start[, end]])  # Like find or ValueError
str.rindex(sub[, start[, end]]) # Like find or ValueError
```

- Metodi di sostituzione:

```
str.replace(old, new[, maxreplace])    # Replace substrings
str.translate(s, table[, deletechars]) # Replace chars
str.maketrans(x[, y[, z]]) # Create translation tables
str.expandtabs(tabsize=8)  # Replace TABs with SPACEs
```

Metodi di test

- Metodi per effettuare test sui caratteri della stringa:

```
str.isalnum()
str.isalpha()
str.isdecimal()
str.isdigit()
str.isidentifier()
str.islower()
str.isnumeric()
str.isprintable()
str.isspace()
str.istitle()
str.isupper()
```