



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

# Programmazione di Reti – Laboratorio # 2

**Andrea Piroddi**

Dipartimento di Informatica, Scienza e Ingegneria

# Elementi di Python

- Variabili, espressioni ed istruzioni
- Funzioni
- Istruzioni condizionali e ricorsione
- Iterazione



Lezione 1

- Stringhe
- Liste
- Dizionari
- Tuple



Lezione 2

- File
- Classi e Oggetti
- Classi e Funzioni
- Classi e Metodi



Lezione 3



**STRINGHE**



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA  
CAMPUS DI CESENA

# Stringhe

Una **stringa** è una sequenza di caratteri.

E' possibile accedere ai singoli caratteri usando gli operatori «parentesi quadre».

Esempio:

```
ip_address = '010.000.000.001'
```

```
classe = ip_address[1]
```

```
print(classe)
```

**1**

L'espressione all'interno delle parentesi quadre è chiamato **indice**. L'indice è un numero intero che indica il carattere della sequenza che si desidera estrarre. Il primo carattere è identificato con «0», il secondo con «1», etc...



# Stringhe

Altro esempio:

```
i=1  
ip_address = '010.000.000.001'  
  
classe = ip_address[i+1]  
print(classe)
```

indice	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	0	1	0	.	0	0	0	.	0	0	0	.	0	0	1

0

Potete usare come indice qualsiasi espressione, compresi variabili e operatori.  
Importante è che **l'indice** deve essere un **intero**.



## Stringhe - len

**len** è una funzione che restituisce il numero di caratteri contenuti in una stringa:

```
ip_address = "010.000.000.001"
```

```
a=len(ip_address)
```

```
print(a)
```

**15**

Per estrarre l'ultimo carattere di una stringa si può usare la seguente istruzione

```
ultimo = ip_address[a-1]
```

```
print(ultimo)
```

**1**



## Stringhe - attraversamento

E' possibile usare gli indici negativi che contano a ritroso dalla fine della stringa:

```
ip_address = "010.000.000.001 "  
print(ip_address[-1])  
1
```

Usando indice -1 consideriamo l'ultimo carattere della stringa, con -2 il penultimo e così via.

Molti tipi di calcolo comportano l'elaborazione di una stringa, un carattere per volta.

Questo tipo di elaborazione è chiamata **attraversamento**.

Vediamo un esempio:

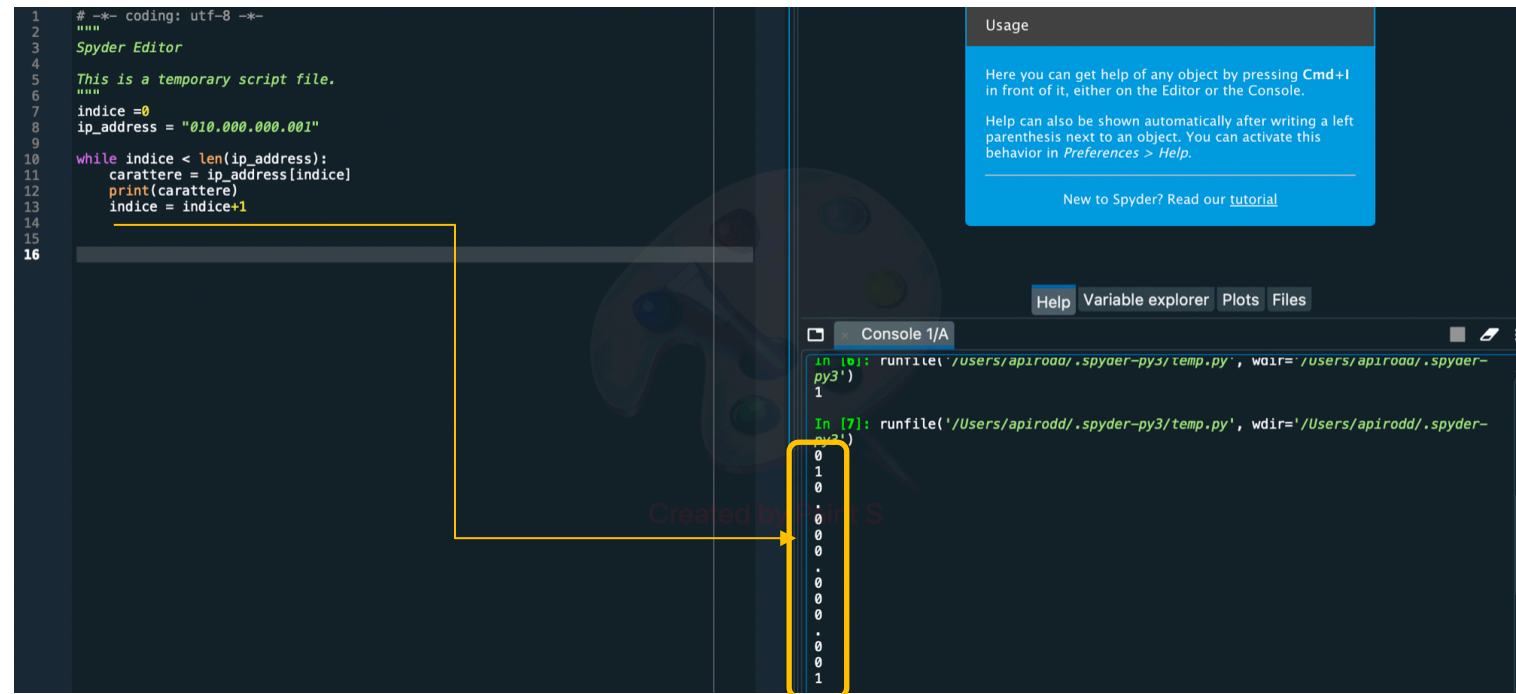


# Stringhe - attraversamento

```
indice =0
```

```
ip_address = "010.000.000.001"
```

```
while indice < len(ip_address):  
    carattere = ip_address[indice]  
    print(carattere)  
    indice = indice+1
```



```
1  -*- coding: utf-8 -*-  
2  """  
3  Spyder Editor  
4  
5  This is a temporary script file.  
6  """  
7  indice =0  
8  ip_address = "010.000.000.001"  
9  
10 while indice < len(ip_address):  
11     carattere = ip_address[indice]  
12     print(carattere)  
13     indice = indice+1  
14  
15  
16
```

Usage

Here you can get help of any object by pressing Cmd+I in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

[New to Spyder? Read our tutorial](#)

Help Variable explorer Plots Files

Console 1/A

```
In [10]: runfile('/Users/apirodd/.spyder-py3/temp.py', wdir='/Users/apirodd/.spyder-py3')  
1  
In [7]: runfile('/Users/apirodd/.spyder-py3/temp.py', wdir='/Users/apirodd/.spyder-py3')  
0  
1  
0  
.  
0  
0  
0  
.  
0  
0  
0  
.  
0  
0  
1
```





## Stringhe - Slicing

Un segmento o porzione di stringa è chiamato *slice*.

L'operazione di selezione di una porzione di stringa è paragonabile alla selezione di un carattere ed è chiamata *slicing*.

Esempio:

```
ip_address = "010.000.000.001"
```

```
classe = ip_address[0:3]
```

```
print(classe)
```

**010**



## Stringhe - Slicing

L'operatore [n:m] restituisce la porzione di stringa nell'intervallo compreso tra l' «n-esimo» carattere incluso, fino all' «m-esimo» escluso.

Se non è presente il primo indice, si parte dall'inizio della stringa mentre se il secondo indice non è presente, si arriva fino in fondo alla stringa:

Esempio:

```
ip_address = "010.000.000.001"  
classe1 = ip_address[:3]  
classe2 = ip_address[0:]  
print("classe1 è", classe1)  
print("classe2 è", classe2)
```

**classe1 è 010**

**classe2 è 010.000.000.001**



# Stringhe - Immutabilità

```
ip_address = "010.000.000.001"  
ip_address[14] = '2'  
print(ip_address)
```

***TypeError: 'str' object does not support item assignment***

Il motivo dell'errore è dovuto al fatto che le stringhe sono immutabili, ossia non è consentito cambiare una stringa esistente.

E' possibile creare una nuova stringa:

```
ip_address = "010.000.000.001"  
nuovo_ip_address = ip_address[0:14]+'2'  
print(nuovo_ip_address)
```

**010.000.000.002**



# Stringhe – funzione *trova*

Funzione ***trova***:

```
def trova(parola, lettera):  
    indice = 0  
    while indice < len(parola):  
        if parola[indice] == lettera:  
            return indice  
        indice = indice + 1  
    return -1  
print(trova("indirizzo", "o"))
```

8

La funzione ***trova***, trova la lettera richiesta nella parola assegnata e ne restituisce l'indice. Se il carattere non compare nella stringa data, il programma termina il ciclo normalmente e restituisce -1.



## Stringhe – Cicli e contatori

Nel caso volessimo contare quante volte uno stesso carattere compare in una stringa, potremmo realizzare un codice del tipo:

```
parola = "http://www.unibo.it"  
conta = 0  
for carattere in parola:  
    if carattere == "w":  
        conta = conta + 1  
  
print(conta)
```

## Stringhe - Metodi

Le Stringhe espongono dei metodi che permettono di effettuare molte operazioni utili.

Un **METODO** è simile ad una funzione, ossia riceve argomenti e restituisce un valore con una diversa sintassi.

Consideriamo ad esempio il metodo ***upper***, che prende una stringa e crea una nuova stringa di tutte lettere maiuscole:

```
router = "router_primario"  
nuovo_router = router.upper()  
print(nuovo_router)
```

***ROUTER\_PRIMARIO***

Quando si chiama un metodo, si definisce **INVOCAZIONE**.



## Stringhe - Metodi

Esiste un metodo che si comporta come la funzione «**trova**» che abbiamo visto prima:

```
def trova(parola, lettera):  
    indice = 0  
    while indice < len(parola):  
        if parola[indice] == lettera:  
            return indice  
        indice = indice + 1  
    return -1  
print(trova("indirizzo", "o"))
```

USIAMO il METODO **find**:

```
parola = "indirizzo"  
indice = parola.find("o")  
print(indice)
```

## Stringhe - Metodi

Il metodo ***find*** è più generale della funzione che abbiamo costruito poiché è in grado di ricercare anche sottostringhe, non solamente caratteri singoli:

```
ip_address = "010.000.000.001"  
indice = ip_address.find("000")  
print(indice)
```

**4**

Il metodo ***find*** indica il punto di inizio della stringa cercata, ma può anche ricevere come secondo argomento, l'indice da cui partire, esempio:

```
ip_address = "010.000.000.001"  
indice = ip_address.find("000", 7)  
print(indice)
```

**8**





## Stringhe – Operatore *in*

La parola ***in*** è un operatore booleano che confronta due stringhe e restituisce ***TRUE*** se la prima è una sottostringa della seconda. Esempio:

```
print("000" in "010.000.000.001")  
True
```

Esempio di utilizzo di ***in***:

```
def in_entrambe(parola1, parola2):  
    for carattere in parola1:  
        if carattere in parola2:  
            print(carattere)
```

```
in_entrambe("switch", "router")  
t
```



## Stringhe – leggere un file

Supponiamo di voler leggere il contenuto di un file testo tramite Python.

Prima di tutto dobbiamo vedere in quale directory di lavoro ci troviamo, così da poterci creare un file testo:

Per vedere la directory di lavoro, possiamo importare il **modulo os** ed eseguire il seguente comando:

```
import os  
print(os.getcwd())
```

```
/Users/apirodd/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/Lezioni/Esercitazione 1/codice
```

A questo punto possiamo creare con un editor di testi un file **txt** e salvarlo in questa directory o potete scaricare il file **words.txt** da questo indirizzo:

<http://thinkpython2.com/code/words.txt> .



## Stringhe – leggere un file

Proviamo quindi ad aprire e leggere il contenuto del file. Useremo la funzione predefinita ***open*** che richiede come parametro il nome di un file e restituisce un ***oggetto file***:

```
import os
print(os.getcwd())
fin = open("words.txt")
```

L'***oggetto file*** comprende alcuni metodi di lettura, come ad esempio ***readline***, che legge i caratteri di un file finchè non giunge ad un ritorno a capo (\n), e restituisce il risultato sotto forma di stringa:

```
import os
print(os.getcwd())
fin = open("words.txt")
fin.readline()
print(fin.readline())
```

**AOL**



# Liste

Come una stringa, una lista è una sequenza di valori. Mentre in una stringa i valori sono dei caratteri, in una lista possono essere di qualsiasi tipo.

I valori che appartengono alla lista sono definiti **ELEMENTI**.

Esistono diversi modi di creare una nuova lista; quello classico è quello di racchiudere i suoi elementi tra parentesi quadre:

```
lista1 = ['ip_addr1', 'ip_addr2', 'ip_addr3']  
lista2 = [1024, 2048, 4096]  
lista3 = ["source", 1.3, [2,3]] #lista nidificata
```

Una lista all'interno di un'altra lista è detta **nidificata**.



# Liste

Le liste sono ***mutabili***, ossia i suoi elementi possono essere cambiati.

La sintassi per accedere agli elementi di una lista è la stessa usata per i caratteri di una stringa.

Esempio:

```
lista1 = ["ip_addr_1", "ip_addr_2", "ip_addr_3"]
lista2 = [1024, 2048, 4096]
lista3 = ["source", 1.3, [2,3]] #lista nidificata
lista1[2]="ip_addr_4"
print(lista1)
```

```
['ip_addr_1', 'ip_addr_2', 'ip_addr_4']
```



## Liste - Attraversamento

Esempio di attraversamento con ciclo *for*:

```
lista1 = ["ip_addr_1", "ip_addr_2", "ip_addr_3"]  
lista2 = [1024, 2048, 4096]  
lista3 = ["source", 1.3, [2,3]] #lista nidificata  
lista1[2]="ip_addr_4»
```

```
for ip in lista1:  
    print(ip)
```

*ip\_addr\_1*

*ip\_addr\_2*

*ip\_addr\_4*

Questo metodo funziona bene per leggere gli elementi di una lista, ma se si vuole scrivere o **aggiornare** gli elementi sono necessari gli **indici**:



## Liste - Attraversamento

Un modo per modificare gli elementi è quello di usare una combinazione delle funzioni predefinite *range* e *len*:

Esempio:

```
lista2 = [1024, 2048, 4096]
for i in range(len(lista2)):
    lista2[i]=lista2[i]*2
print(lista2)
```

**[2048, 4096, 8192]**

- *len* restituisce il numero di elementi della lista
- *range* restituisce una lista di indici da  $0$  a  $n-1$ , dove  $n$  è la lunghezza della lista

Anche se una lista può contenerne un'altra, quella nidificata conta sempre come un singolo elemento.



# Liste - Operazioni

L'operatore **'+'** concatena delle liste:

```
a = [1, 2, 3]
```

```
b = [4, 5, 6]
```

```
c = a+b
```

```
print(c)
```

```
[1, 2, 3, 4, 5, 6]
```

L'operatore **'\*'** **ripete** una lista per un dato numero di volte:

```
d = [0]
```

```
print(d*4)
```

```
[0, 0, 0, 0]
```





## Liste - Slicing

Anche l'operazione di slicing funziona sulle liste:

```
t = ["a", "b", "c", "d", "e", "f"]  
y = t[1:3]  
print(y)
```

```
['b', 'c']
```

Un operatore di slicing sul lato sinistro di un'assegnazione, permette di aggiornare più elementi:

```
t = ["a", "b", "c", "d", "e", "f"]  
t[1:3] = ["x", "y"]  
print(t)
```

```
['a', 'x', 'y', 'd', 'e', 'f']
```



## Liste - Metodi

Python fornisce dei **METODI** che operano sulle liste. Per esempio, ***append*** aggiunge un nuovo elemento in coda alla lista:

```
t = ["a","b","c","d","e","f"]  
t.append("z")  
print(t)
```

```
['a', 'b', 'c', 'd', 'e', 'f', 'z']
```

***extend*** prende una lista come argomento e accoda tutti i suoi elementi:

```
t1 = ["a","z","c"]  
t2 = ["d","e"]  
t1.extend(t2)  
print(t1)
```

```
['a', 'z', 'c', 'd', 'e']
```



## Liste - Metodi

```
t1 = ["a", "z", "c"]  
t1.sort()  
print(t1)
```

```
['a', 'c', 'z']
```

Per sommare tutti i numeri in una lista, possiamo utilizzare un ciclo come questo:

```
def somma_tutti(t):  
    totale = 0  
    for x in t:  
        totale += x # equivale a totale = totale + x  
    return totale  
  
lista = [1,2,3,4,5]  
a = somma_tutti(lista)  
print(a)
```

15

Possiamo ottenere lo stesso risultato con la funzione **sum**:

```
print(sum(lista))
```

15



## Liste - Mappa

Talvolta è necessario attraversare una lista per costruirne contemporaneamente un'altra.  
Esempio:

```
def tutte_maiuscole(t):  
    res = [] # inizializziamo una lista vuota  
    for s in t:  
        res.append(s.capitalize())  
    return res
```

```
lista = ["a", "b", "c"]  
print(tutte_maiuscole(lista))
```

```
['A', 'B', 'C']
```

Una operazione di questo tipo è chiamata **MAPPA**, poiché applica una funzione su ciascun elemento di una sequenza



## Liste - Filtro

Il filtro è un'operazione di selezione di alcuni elementi di una lista per formare una sottolista.

Esempio:

```
def solo_maiuscole(t):  
    res = []  
    for s in t:  
        if s.isupper():  
            res.append(s)  
    return res  
  
lista = ["A", "b", "c", "D", "E", "f"]  
print(solo_maiuscole(lista))
```

```
['A', 'D', 'E']
```



# Liste – Cancellare elementi

Esistono diversi modi per cancellare elementi da una lista.

- **pop**

```
lista = ["a", "b", "c"]
```

```
x = lista.pop(1)
```

```
print(lista)
```

```
['a', 'c']
```

- **del**

```
lista = ["a", "b", "c"]
```

```
del lista[1]
```

```
print(lista)
```

```
['a', 'c']
```

- **Remove**

```
lista = ["a", "b", "c"]
```

```
lista.remove("b")
```

```
print(lista)
```

```
['a', 'c']
```



# Dizionari

Un dizionario è una **mappatura**.

Un dizionario è simile ad una lista; la differenza è che in una lista gli indici devono essere dei numeri interi, mentre in un dizionario possono essere (quasi) di ogni tipo.

Un dizionario contiene una raccolta di indici, chiamati **chiavi**, e una raccolta di valori. Ciascuna chiave è associata ad un unico valore.

L'associazione tra una chiave ed un valore è detta coppia **chiave-valore**.

## **Esempio:**

```
protocol2port = {"SSH": "22", "FTP": "21", "DNS": "53", "HTTP": "80", "TELNET": "23"}  
print(protocol2port["SSH"])
```

# Dizionari

Supponiamo che ci venga data una stringa e che vogliamo contare quante volte vi compare ciascuna lettera.

Esempio:

```
def istogramma(s):
```

```
    d = dict()
```

```
    for c in s:
```

```
        if c not in d:
```

```
            d[c] = 1
```

```
        else:
```

```
            d[c] += 1
```

```
    return d
```

```
print(istogramma("indirizzo_ip"))
```

```
{'i': 4, 'n': 1, 'd': 1, 'r': 1, 'z': 2, 'o': 1, '_': 1, 'p': 1}
```





## Dizionari – Metodo *get*

I dizionari supportano il metodo *get* che richiede una chiave e un valore predefinito. Se la chiave è presente nel dizionario, *get* restituisce il suo valore corrispondente, altrimenti restituisce il valore predefinito.

### Esempio

```
protocol2port = {"SSH": "22", "FTP": "21", "DNS": "53", "HTTP": "80", "TELNET": "23"}  
print(protocol2port.get("SSH", 0))
```

# Dizionari - Cicli

Se si usa un dizionario in un ciclo ***for***, quest'ultimo attraversa le chiavi del dizionario.

```
def stampa_isto(h):  
    for c in h:  
        print(c, h[c])  
def istogramma(s):  
    d = dict()  
    for c in s:  
        if c not in d:  
            d[c] = 1  
        else:  
            d[c] += 1  
    return d  
  
h = istogramma("ip_address")  
stampa_isto(h)
```

```
i 1  
p 1  
_ 1  
a 1  
d 2  
r 1  
e 1  
s 2
```



## Tuple - Immutabilità

Una **tupla** è una sequenza di valori separati da virgole. I valori possono essere di qualsiasi tipo e sono indicizzati tramite numeri interi.

Una caratteristica essenziale delle **Tuple** è che esse sono **immutabili**.

```
t = "a", "b", "c", "d", "e"  
print(type(t))
```

```
<class 'tuple'>
```

Un altro modo di creare una **tupla** è usare la funzione predefinita **tuple**.

Se priva di argomento, crea una **tupla** vuota.



# Tuple

```
t = tuple()  
print(t)
```

```
()
```

Se l'argomento è una sequenza (stringa, lista, tupla), il risultato è una tupla con gli elementi della sequenza:

```
t = tuple("ipaddress")  
print(t)  
  
('i', 'p', 'a', 'd', 'd', 'r', 'e', 's', 's')
```

La maggior parte degli operatori delle liste funzionano anche con le ***tuple***. A differenza delle liste, se si cerca di modificare gli elementi di una tupla si ottiene un messaggio d'errore:



# Tuple

```
t = tuple("ipaddress")  
t[0]="I"
```

***TypeError: 'tuple' object does not support item assignment***

D'altra parte è possibile sostituire una ***tupla*** con un'altra.

## Esempio:

```
t = tuple("ipaddress")  
t = ("I",)+t[1:]  
print(t)
```

('I', 'p', 'a', 'd', 'd', 'r', 'e', 's', 's')



# Tuple

```
a, b = 1, 2
print(a)
print(b)
```

Sul lato sinistro abbiamo una **tupla** di variabili, su quello destro una **tupla** di espressioni. Ciascun valore viene assegnato alla rispettiva variabile.

```
indirizzo_url = "router@cisco.com"
nome, dominio = indirizzo_url.split("@")
print(nome)
print(dominio)
c = nome, dominio
print(c)
```

```
router
cisco.com
('router', 'cisco.com')
```



## Tuple – come valori di ritorno

In senso stretto una funzione può restituire un solo valore di ritorno, ma se il valore è una **tupla**, l'effetto pratico è quello di restituire valori molteplici.

Esempio:

La funzione ***divmod*** riceve due argomenti e restituisce una **tupla** di due valori, il quoziente ed il resto:

```
t = divmod(7,3)
print(t)
```

**(2, 1)**



## Tuple – funzione zip

**Zip** è una funzione predefinita che riceve due o più sequenze e restituisce una lista di **tuple**, dove ciascuna **tupla** contiene un elemento di ciascuna sequenza.

### Esempio:

```
s = "abc"  
t = [0,1,2]  
a = zip(s,t)  
for coppia in a:  
    print(coppia)
```

```
('a', 0)
```

```
('b', 1)
```

```
('c', 2)
```

Se le sequenze non sono della stessa lunghezza, il risultato ha la lunghezza di quella più corta.

