

PROGRAMMAZIONE DI RETI

Laboratorio 1

Andrea Piroddi

Università di Bologna - Campus di Cesena
andrea.piroddi@unibo.it

Anno Accademico 2023-2024

Cosa vediamo oggi

1 Orario Laboratorio

Orario

2 Informazioni Utili

3 Linguaggio di Programmazione

Python

Variabili, espressioni ed istruzioni

Funzioni

4 Stringhe - Metodi

Operatori

5 Liste

Metodi

6 Dizionari e Tuple

Scrivere un File

Moduli

Orario

Periodo di svolgimento delle lezioni: 12/03/2024 - 28/05/2024

Giorno	Ora	Aula
Martedì	9-11	LAB. 3.3 Piano Primo
Martedì	14-16	LAB. 2.2 VELA Piano Terra

Modalità d' Esame

- L'esame è composto da un Progetto di Laboratorio che vale 4 punti e dall'esame di Teoria che vale 28 punti
- Il Progetto di Laboratorio non è obbligatorio ma è indispensabile per raggiungere la votazione di 30/30
- Le tracce per il Progetto di Laboratorio saranno fornite verso la metà di Maggio
- Il Progetto una volta presentato è valido per l'intero anno accademico
- Il Progetto deve essere consegnato almeno una settimana prima della data dell'esame di teoria
- Il progetto deve essere consegnato direttamente via mail al docente o ricorrendo ad un repository a scelta dello studente.

Strumenti di Laboratorio: Wireshark

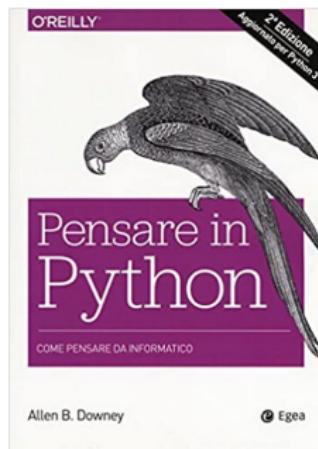
<https://www.wireshark.org/download.html>

Alla fine del 1997 Gerald Combs (Laureato in informatica presso l'Università del Missouri) aveva bisogno di uno strumento per rintracciare i problemi di rete, così iniziò a scrivere **Ethereal** (il nome originale del progetto **Wireshark**) come un modo per risolvere entrambi i problemi.

Ethereal è stato inizialmente rilasciato dopo diverse pause di sviluppo nel luglio 1998 come versione 0.2.0. In pochi giorni iniziarono ad arrivare patch, segnalazioni di bug e parole di incoraggiamento ed Ethereal cominciava a raggiungere il successo. Nel 2015 è stato rilasciato Wireshark 2.0, che presentava una nuova interfaccia utente.

Python

https://www.python.it/doc/Howtothink/HowToThink_ITA.pdf



"Pensare in Python" di Allen B. Downey O'Reilly
(seconda edizione)

Elementi di Python

- Variabili, espressioni ed istruzioni
- Funzioni
- Istruzioni condizionali e ricorsione
- Iterazione
- Stringhe
- Liste
- Dizionari
- Tuple
- File

Ambiente di lavoro Python - ANACONDA

Anaconda Installers

Windows

Python 3.9

64-Bit Graphical Installer (621 MB)

MacOS

Python 3.9

64-Bit Graphical Installer (688 MB)

64-Bit Command Line Installer (681 MB)

64-Bit (M1) Graphical Installer (484 MB)

64-Bit (M1) Command Line Installer (472 MB)

Linux

Python 3.9

64-Bit (x86) Installer (737 MB)

64-Bit (Power8 and Power9) Installer (360 MB)

64-Bit (AWS Graviton2 / ARM64) Installer (534 MB)

64-bit (Linux on IBM Z & LinuxONE) Installer (282 MB)

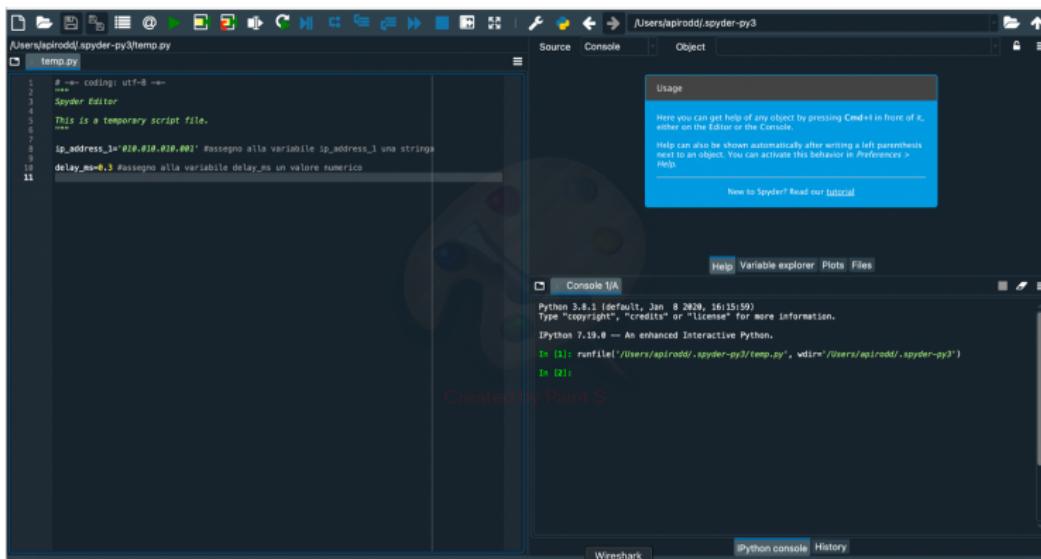
Python

Una volta installato Anaconda, avremo a disposizione una suite di programmi:

The screenshot shows the Anaconda Navigator interface. On the left is a sidebar with navigation links: Home, Environments, Learning, Community, and a section for ANACONDA.NUCLEUS with a 'Join Now' button. The main area displays a grid of 8 data science tools:

- Datalore**: Online Data Analysis Tool with smart coding assistance by JetBrains. Includes a 'Launch' button.
- IBM Watson Studio Cloud**: Provides tools to analyze and visualize data, to cleanse and shape data, to create and train machine learning models. Includes a 'Launch' button.
- JupyterLab**: An extensible environment for interactive and reproducible computing, based on the Jupyter Notebook and Architecture. Includes a 'Launch' button.
- jupyter Notebook**: Web-based, interactive computing notebook environment. Includes a 'Launch' button.
- IPyConsole**: PyQt GUI that supports inline figures, proper multiline editing with syntax highlighting, graphical celltips, and more. Includes a 'Launch' button.
- Spyder**: Scientific Python Development Environment. Powerful Python IDE with advanced editing, interactive testing, debugging and introspection features. Includes a 'Launch' button.
- Glueviz**: Multidimensional data visualization across files. Explore relationships within and among related datasets. Includes an 'Install' button.
- Orange 3**: Component-based data mining framework. Data visualization and data analysis for novice and expert. Interactive workflows with a large toolbox. Includes an 'Install' button.

Lanciando Spyder otteniamo un ambiente di lavoro per la realizzazione di script:



Variabili, espressioni ed istruzioni

Una **variabile** è un **nome** che fa riferimento ad un **valore**.

Un'**istruzione di assegnazione** si utilizza per generare una nuova **variabile**, specificandone il **nome**, e dandole un **valore**:

Esempio:

- `ip_address_1 = '010.010.010.001'`
- `delay_ms = 0.3`

```
1  # -*- coding: utf-8 -*-
2  """
3  Spyder Editor
4  This is a temporary script file.
5  """
6
7
8  ip_address_1='010.010.010.001' #assegno alla variabile ip_address_1 una stringa
9
10 delay_ms=0.3 #assegno alla variabile delay_ms un valore numerico
11
```

Variabili, espressioni ed istruzioni

Un' **Espressione** è una combinazione di valori, variabili ed operatori. Un' **Istruzione** è una porzione di codice che può essere eseguita dall'interprete Python.

Concatenamento di stringhe

Esempio: Header IPv4

Consideriamo un header semplificato che contenga solo source address e destination address; in python possiamo costruirlo come:

+	Bits 0–3	4–7	8–15	16–18	19–31
0	Version	Internet Header length	Type of Service (adesso DiffServ e ECN)		Total Length
32		Identification		Flags	Fragment Offset
64	Time to Live		Protocol		Header Checksum
96				Source Address	
128				Destination Address	
160				Options (facoltativo)	
160	o			Data	
192+					Created by Paint S

Variabili, espressioni ed istruzioni

Variabili, espressioni ed istruzioni

- `src_address = '010.000.000.001'`
- `dst_address = '010.000.000.002'`
- `header_simpl = src_address + dst_address`
`print(header_simpl)`

The screenshot shows the Spyder IDE interface. On the left, the 'temp.py' script is displayed in the Editor tab:

```
# -*- coding: utf-8 -*-
"""
Spyder Editor
This is a temporary script file.
"""

source_address = '010.000.000.001'
destination_address = '010.000.000.002'

header_simpl = source_address + destination_address

print(header_simpl)
```

A yellow box highlights the line `print(header_simpl)`. A yellow arrow points from this line to the 'Console' tab on the right, where the output is shown:

```
In [1]: runfile('C:/Users/1000000000.spyder-py3/temp.py', wdir='C:/Users/1000000000.spyder-py3')
Out[1]: '010.000.000.001010.000.000.002'
```

The text 'Concatenazione di stringhe' is overlaid in red at the bottom right of the console area.

Funzioni

Una **Funzione** è una serie di istruzioni che esegue un calcolo.

Alla funzione viene assegnato un nome.

Un **Modulo** è un file che contiene una raccolta di funzioni correlate.

Python è provvisto di un modulo matematico che comprende molte delle funzioni matematiche di uso comune.

Prima di poter utilizzare le funzioni incluse nel modulo, **dobbiamo importare il modulo** stesso con una istruzione di importazione:

```
1  # -*- coding: utf-8 -*-
2  """
3  Spyder Editor
4
5  This is a temporary script file.
6  """
7  import math
8  potenza_segnale = int(input("inserisci il valore di potenza"))
9  potenza_rumore = int(input("inserisci il valore di rumore"))
10 SNR = potenza_segnale / potenza_rumore #signal (Watt) to noise (Watt) ratio
11 SNR_dB = 10*math.log10(SNR)
12 print("il valore di SNR in dB è:", SNR_dB)
```

Funzioni

- La funzione `int()` converte la stringa in un numero intero oppure un numero reale in un numero intero.
 - L'argomento può essere una costante o una variabile alfanumerica.
 - Se l'argomento è un valore reale, la funzione `int()` tronca ogni cifra dopo la virgola (es. da 3.5 a 3).
 - Se l'argomento è una stringa, la funzione `int()` lo converte in un valore numerico intero (es. da "3" a 3).

Funzioni

Funzioni

Aggiungere nuove funzioni.

Una definizione di funzione specifica il nome di una nuova funzione e la serie di istruzioni che viene eseguita quando la funzione viene chiamata.

Esempio:

```
def area_rettangolo(x,y):  
    z = x*y  
    return z  
  
print(area_rettangolo(3,4))
```



```
1  #!/usr/bin/env python3  
2  # -*- coding: utf-8 -*-  
3  """  
4  Created on Thu Feb 18 19:51:50 2021  
5    
6  @author: apirodd  
7    
8    
9  def area_rettangolo(x,y):  
10     z = x*y  
11     return z  
12  
13 print(area_rettangolo(3,4))  
14
```

Funzioni

Perchè usare le Funzioni?

- Creare una funzione, significa dare un nome ad un gruppo di istruzioni, rendendo il programma più facile da leggere e da correggere
- Le funzioni consentono di eliminare il codice ripetitivo
- Suddividere un programma in funzioni consente di semplificare le correzioni
- Le funzioni sono spesso utili per più programmi, quindi una volta scritta essa può essere riutilizzata

Istruzioni condizionali

L'operatore **==** è uno degli **operatori di confronto**, detti anche **operatori relazionali**.

x != y

x > y

x < y

x >= y

x <= y

x == y

Istruzioni condizionali

Molto utili sono le istruzioni condizionali, che consentono di controllare se si verificano determinate condizioni e quindi di variare conseguentemente il comportamento del programma:

La forma più semplice di istruzione condizionale è l'istruzione **if**

```
Port_number = int(input('inserisci un valore intero '))

if Port_number < 1024:
    print('il numero inserito corrisponde ad una well known port')
```

Istruzioni condizionali

Esecuzione alternativa

```
Port_number = int(input('inserisci un valore intero '))

if Port_number < 1024:
    print('il numero inserito corrisponde ad una well known port')

else:
    print('il numero inserito corrisponde ad una ephemeral port')
```

Funzioni

Istruzioni condizionali

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Created on Fri Feb 19 09:34:39 2021
5  """
6  #author: apirodd
7  """
8  """
9  Port_number = int(input("inserisci un valore intero"))
10 if Port_number < 1024:
11     print("il numero di porta che hai inserito è una well-known port")
12 else:
13     print("il numero di porta che hai inserito è una ephemeral port")
14

```

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 %A

In [5]: runfile('/Users/apirodd/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/lezioni/Esercitazione 1/codice/istruzione condizionale.py', wdir='/Users/apirodd/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/lezioni/Esercitazione 1/codice')

inserisci un valore intero 1025
il numero di porta che hai inserito è una ephemeral port

In [6]:

Ricorsione

La **ricorsione** si verifica quando una funzione chiama se stessa.

```
x = int(input("inserisci il valore"))

def stampa_n(n):
    if n<=0:
        return
    print(n)
    stampa_n(n-1)
```

L'istruzione **return** provoca l'uscita dalla funzione. Il flusso dell'esecuzione torna immediatamente al chiamante e le righe rimanenti della funzione non vengono eseguite.

Funzioni

Ricorsione

The screenshot shows the Spyder IDE interface. On the left, a code editor displays a Python script named `ricorsione.py` with the following content:

```
1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3
4  # Created on Fri Feb 19 14:11:22 2021
5
6  #author: ariod
7
8
9  x = int(input("inserisci il valore"))
10 def stampa_n(n):
11     if n==0:
12         return
13     else:
14         print(n)
15         stampa_n(n-1)
16 stampa_n(x)
17
```

On the right, a help window for the `print` function is open, showing the following content:

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](#)

Below the help window, the Spyder interface includes a **Console** tab, a **Help** menu, and tabs for **Variable explorer**, **Plots**, and **Files**. The console window shows the output of the script:

```
In [1]: runfile('/Users/ariod/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/Lezioni/Esercitazione 1/codice/ricorsione.py', wdir='/Users/ariod/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/Lezioni/Esercitazione 1/codice')
inserisci il valore18
18
9
8
7
6
Paint S
5
4
3
2
1
```

Iterazione

Istruzione **while** La ripetizione di operazioni identiche o simili viene chiamata **iterazione**. Esempio di iterazione con **while**

```
def contoallarovescia(n):  
    while n>0:  
        print(n)  
        n=n-1  
    print("finito")  
  
x = int(input("inserisci il valore intero"))  
contoallarovescia(x)
```

Funzioni

Iterazione

```
1  #!/usr/bin/env python3
2  #-*- coding: utf-8 -*-
3  """
4  Created on Fri Feb 19 14:38:59 2021
5  @author: apirodd
6  """
7
8
9  def contaallarovescia(n):
10     while n>0:
11         print(n)
12         n=n-1
13     print("finito")
14 x = int(input("inserisci il valore intero"))
15 contaallarovescia(x)
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 [12]: runfile('/Users/apirodd/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/Lezioni/Esercitazione 1/codice/iterazione.py', wdir='/Users/apirodd/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/Lezioni/Esercitazione 1/codice')

inserisci il valore intero 5

5
4
3
2
1
finito

In [13]:



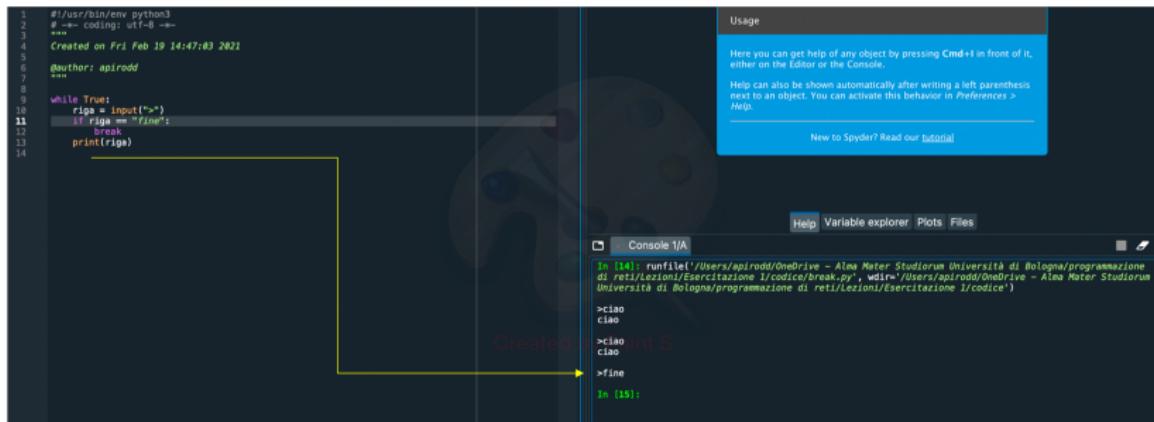
Iterazione: istruzione break

Può capitare di dover uscire da un ciclo mentre il flusso di esecuzione del ciclo stesso è in esecuzione: Esempio dell'uso di **break**

```
while True:
    riga = input(">")
    if riga == "fine":
        break
    print(riga)
```

Funzioni

Iterazione: istruzione break



```
1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-
3
4 Created on Fri Feb 19 14:47:03 2021
5
6 Author: apirodd
7
8
9 while True:
10     riga = input(">")
11     if riga == "fine":
12         break
13     print(riga)
```

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 [14]: runfile('/Users/apirodd/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/Lezioni/Esercitazione 1/codice/break.py', wdir='/Users/apirodd/OneDrive - Alma Mater Studiorum Università di Bologna/programmazione di reti/Lezioni/Esercitazione 1/codice')
```

>ciao
ciao
>ciao
ciao
>fine

In [15]:

Stringhe

Una **stringa** è una sequenza di caratteri. E' possibile accedere ai singoli caratteri usando gli operatori **parentesi quadre**. Esempio:

Esempio:

```
ip_address = '020.000.000.001'  
classe = ip_address[1]  
print(classe)
```

2

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:

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

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

0

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

Stringhe - len

```
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

```
ip_address ="010.010.000.002"  
a=len(ip_address)  
ultimo = ip_address[a-1]  
print(ultimo)
```

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])
```

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**.

Stringhe - attraversamento

```
indice =0  
ip_address = "010.000.000.001"
```

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

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:

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

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.

```
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"))
```

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)
```

3

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

La chiamata di un metodo, si definisce **INVOCAZIONE**.

Stringhe - Metodi

Esiste un metodo che si comporta come la funzione **trova** che abbiamo visto prima.

Il METODO **find**:

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

8

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

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

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

t

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())
```

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>

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())
```

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']
```

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

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(lista)**:



Liste - Metodi

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 - Filtri

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)
```

- **del**

```
lista = ["a", "b", "c"]
del lista[1]
print(lista)
```

- **remove**

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

In tutti e tre i casi il risultato è:
['a', 'c']

Dizionari e Tuple

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 - 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))
```

22

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 - Immutabilità

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 - Immutabilità

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

TypeError: 'tuple' object does not support item assignment

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

Esempio:

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

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

Tuple

```
a, b = 1, 2
```

```
print(a)
```

```
print(b)
```

```
1
```

```
2
```

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')

Scrivere un File

Scrivere un File

La maggior parte dei programmi che abbiamo visto sinora sono transitori, ossia vengono eseguiti, producono un risultato, ma quando vengono chiusi i loro dati vengono perduti.

Uno dei modi più semplici per mantenere i dati dei programmi è di sscrivere su un file di testo. Abbiamo già visto un programma che legge dei file di testo, ora vediamo il caso in cui il programma scrive i dati in un file. Per scrivere un file è necessario aprirlo in modalità scrittura, ossia

indicando con la lettera **w** il secondo parametro.

Open restituisce un oggetto file che fornisce i metodi per lavorare con il file. Il metodo **write** inserisce i dati nel file.

Scrivere un File

Esempio:

```
fout = open("output.txt", "w")  
  
riga1 = "ip_address_=10.10.10.1,\n"  
fout.write(riga1)  
  
riga2 = "mac_address_=00-08-74-4C-7F-1D,\n"  
fout.write(riga2)  
  
fout.close()
```

L'oggetto file tiene traccia di dove si trova, e se si invoca ancora il metodo write, aggiunge i nuovi dati in coda al file. Quando si è terminato di scrivere il file, è opportuno chiuderlo.

Operatore di Formato

Se nella stringa c'è più di una sequenza di formato, il secondo operando deve essere una tupla. Ciascuna sequenza di formato corrisponde ad un elemento della tupla, nell'ordine.

- '%d' si usa per formattare un intero
- '%g' si usa per formattare un decimale a virgola mobile (floating-point)
- '%s' si usa per formattare una stringa

```
print('dei %d utenti, %s hanno un'altezza di %g' %(1000, 'Mario e Franco', 1.75))
```

dei 1000 utenti, Mario e Franco hanno un'altezza di 1.75

Att.ne: il numero degli elementi nella tupla deve essere pari a quello delle sequenze di formato nella stringa, ed i tipi degli elementi devono corrispondere a quelle delle sequenze di formato, altrimenti vi viene restituito un errore.

Moduli

Qualunque file che contenga codice Python può essere importato come **MODULO**.

Supponiamo di avere un file di nome **rc.py** contenente le seguenti istruzioni:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu Mar  4 19:43:44 2021

@author: apirodd
"""

def contarighe(nomefile):
    conta = 0
    for riga in open(nomefile):
        conta += 1
    return conta

print(contarighe("rc.py"))
```

Moduli

Eseguendo questo codice, esso legge se stesso e stampa il numero di righe nel file, che è **14**.

E' possibile anche importare il file in questo modo

```
import rc
```

E il risultato è:

14

L'unico difetto di questo approccio è che quando importate il modulo, durante l'esecuzione viene eseguito anche il codice del modulo importato.

Di solito, invece, un modulo definisce solo delle nuove funzioni ma esse devono essere eseguite **solo** quando richiesto.

Moduli

I programmi che vengono importati come moduli usano spesso questo costrutto:

```
def contarighe(nomefile):  
    conta = 0  
    for riga in open(nomefile):  
        conta += 1  
    return conta  
  
if __name__ == '__main__':  
    print(contarighe("rc1.py"))
```

`__name__` è una variabile predefinita che viene impostata all'avvio del programma. Se questo viene avviato come script, `__name__` ha il valore `'__main__'`; in quel caso, il codice viene eseguito.

Altrimenti, se viene importato come modulo, il codice importato viene saltato.

Grazie

Domande?