**7 14/12/2020 **

***Verifica esercizio cicli FOR-WHILE per creare password forte***

**IMMUTABLE & MUTABLE OBJECTS: introduzione a liste e dizionari**

Oggetti immutabili (Booleans, Integers, Floats, Strings, Tuples)

Ad ogni nuova istanza resta in memoria il vecchio valore (ma perde il legame con la variabile). Python si preoccupa di cancellare i vecchi oggetti, attraverso un processo chiamato garbage collection.

# esempio TUPLA (notate le parentesi tonde)

a\_tuple = ("milk", "eggs")

print(id(a\_tuple))

# 2159363028872 (ID è la rappresentazione numerica di un indirizzo di memoria di uno specifico oggetto)

a\_tuple = ("milk", "eggs", "bread")

print(id(a\_tuple))

# 2159363351248 (è cambiato)

a\_tuple = a\_tuple + ("bacon")

print(id(a\_tuple))

# 2159363456348 (è cambiato ancora!)

A che servono le tuple? Ad es. per memorizzare coordinate geografiche (21.407887, -111.111316) oppure gli indici di una parola in un libro, pagina, n° riga, n° colonna ("ISBN 0-7645-2641-3", 144, 10, 24)… cose che di solito non mutano.

Oggetti mutabili (Liste e Dizionari)

Una volta creato il valore dell’oggetto può comunque variare

Op\_liste.py

# esempio LISTA (parentesi quadre)

a\_list = ["milk", "eggs"]

print(id(a\_list))

# 2159363498056

a\_list = ["milk", "eggs", "bread"]

print(id(a\_list))

# 2159363284680 (è cambiato)

a\_list.append("bacon")

print(id(a\_list))

# 2159363284680 (è lo stesso di prima!)

*NB == e is (!= e is not):* ***a == b*** *confronta i valori, mentre* ***a is b*** *confronta l’identità dell’oggetto, cioè se ha uguale id (entrambi restituisco* ***true o false****)*

Tutto questo vale per aggiungere, cancellare, riorganizzare ad esempio ordinare) ed è una gestione della memoria più efficiente rispetto a copiare di nuovo tutto in un nuovo oggetto

Per le **LISTE** valgono le stesse regole delle stringhe:

posso usare len(a\_list)   
# 4 dopo l’aggiunta di bacon;

a\_list[0]   
# vale milk – si parte da 0 che è il primo item;

posso anche contare quante volte un elemento compare in una lista a\_list.count("bacon")   
# 1 volta;

qual è l’indice in cui si trova la prima occorrenza di un elemento a\_list.index("milk")   
# ovvero 0;

aggiungere item con append, inserire item a\_list.insert(1, "potatoes")   
# inserirebbe “potatoes” tra “milk” ed “eggs”;

fondere due liste con extend a\_list.extend(lista\_spesaB)   
# supponendo che lista\_spesaB sia un’altra LIST che contiene ad es. ["cramberries"];

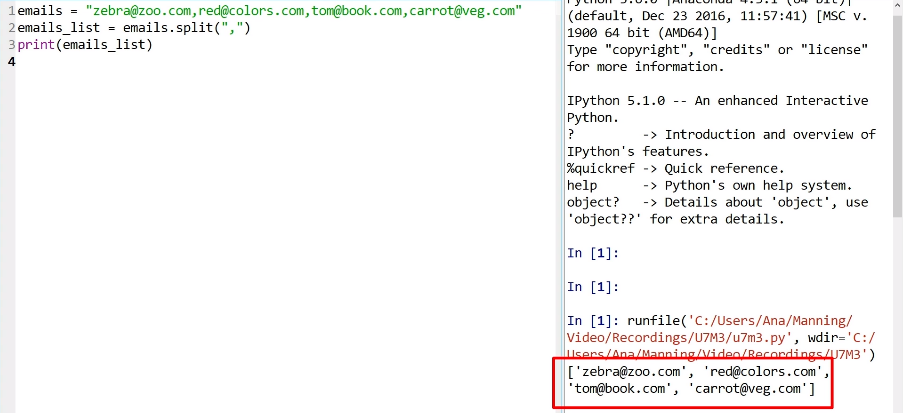
rimuovere item a\_list.pop()   
# l’ultimo della lista   
o a\_list.pop(1)   
# quello con indice 1;

cambiare valori in una certa posizione a\_list[0]="croissant"   
# mette croissant al posto di milk;

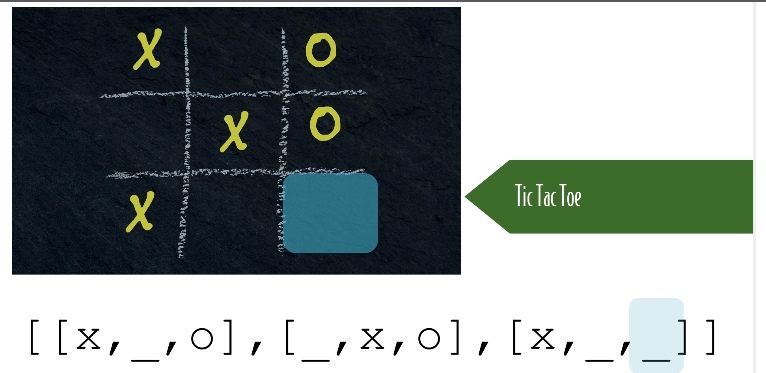
ordinare in ordine crescente gli elementi a\_list.sort()  
# lista di alimenti in ordine alfabetico crescente;

invertire gli elementi a\_list.reverse()  
# lista di alimenti in ordine alfabetico decrescente;

Posso agevolmente trasformare stringhe con carattere separatore (es. CSV) in liste con il comando lista=stringa.split(",")   
# dove la virgola è il carattere separatore



Infine posso costruire LISTE di LISTE



Op\_dizionari.py

Con i **DIZIONARI** estendo il concetto di liste, mappando un item, la chiave, ed un altro item, il valore: ad esempio una lista della spesa con alimenti e quantità. Posso solo abbinare 1 oggetto CHIAVE ad 1 oggetto VALORE (ma l’oggetto valore potrebbe essere una TUPLA di più valori)

PS per fare le *parentesi quadre []* CTRL + ALT + [ (apertura) e CTRL + ALT + ] (chiusura)

per fare le *parentesi graffe {}* CTRL + SHIFT + ALT + [ (apertura) e CTRL + SHIFT + ALT + ] (chiusura)

# creo un dizionario – prima vuoto – nota le parentesi grafe – non c’è un ordine se non quello di inserimento grocery(dict).py *(vecchia versione del codice/esercizio: aprire invece Op\_dizionari.py – contiene aggiornamenti)*

grocery = {}

grocery = {"milk":1, "eggs":12, "bread":2}

grocery["bacon"]=44 # aggiunge in coda chiave e valore (o modifica il valore di una chiave se già esistente, perché il DIZIONARIO è mutabile, ma la KEY è unica e immutabile)

print(grocery) # {'milk': 1, 'eggs': 12, 'bread': 2, 'bacon': 44}

print(len(grocery)) # 4

removed=grocery.pop("bacon") # POP rimuove chiave e valore

print(removed) # 44 (ma può memorizzare il valore che aveva la chiave rimossa)

grocery["eggs"]= grocery["eggs"]-2 # sottrae 2 uova dal valore

print(grocery) # {'milk': 1, 'eggs': 10, 'bread': 2}

print(grocery.keys()) # restituisce solo le chiavi: dict\_keys(['milk', 'eggs', 'bread'])

# grocery.keys è una specie di lista

for one\_item in grocery.keys():

print(one\_item)

# ma posso crearmi una vera e propria lista di chiavi con un casting

all\_items = list(grocery.keys())

# analogamente c’è il comando VALUES

all\_items\_values = list(grocery.values())

print(all\_items\_values)

frequency\_dictionary.py

# frequency dictionary

lyrics = "Happy birthday to you Happy birthday to you Happy birthday dear friend Happy birthday to you"

counts = {}

words = lyrics.split(" ")

for w in words:

w=w.lower()

if w not in counts:

counts[w] = 1

else:

counts[w] += 1

print(counts)

# {'happy': 4, 'birthday': 4, 'to': 3, 'you': 3, 'dear': 1, 'friend': 1}

ALIASING vs COPYING.py

# ALIASING

# puntamento in memoria ad oggetti per variabili immutabili (numeri interi)

a=1

print("ID a")

print(id(a))

b=a #stessa cosa se pongo b=1

print("ID b - dopo b=a")

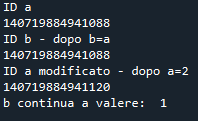
print(id(b))

a=2

print("ID a modificato - dopo a=2")

print(id(a))

print("b continua a valere: ", b)



# puntamento in memoria ad oggetti per variabili mutabili (lista)

genius = ["einstein","galileo"]

print("\nID GENIUS")

print(id(genius))

smart = genius

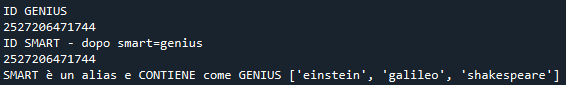
print("ID SMART - dopo smart=genius")

print(id(smart))

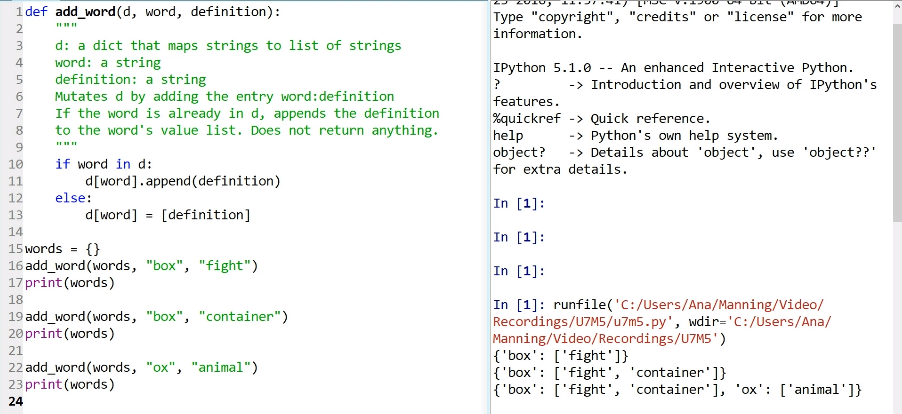
genius.append("shakespeare")

print("SMART è un alias e CONTIENE come GENIUS", smart)

# smart è un ALIAS di genius - mantiene stesso ID ovvero puntano allo stessa location in memoria



# ALIASING di oggetti mutevoli introduce cambiamenti anche se le variabili sono definite nel corpo principale e nella funzione (come parametro)



# per fare una copia di oggetti pesanti, come liste e dizionari, anziché usare aliasing

# o creo una nuova lista/dizionario contenente gli stessi elementi dell'altro oggetto (COPIA ESPLICITA)

personaggi = list(genius)

print("ID PERSONAGGI dopo copia esplicita lista")

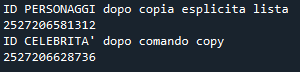
print(id(personaggi))

# o uso il comando COPY

celebrità = genius.copy()

print("ID CELEBRITA' dopo comando copy")

print(id(celebrità))



# SORTED - crea automaticamente una copia ma ordinata della lista

kid\_ages = [2, 1, 4]

sorted\_ages = sorted(kid\_ages)

print("original KID\_AGES", kid\_ages)

print("SORTED\_AGES", sorted\_ages) # [1, 2, 4]

# ATTENZIONE: esempio “rimuovere tutti gli elementi con valore 1 da dizionari o liste” -> i cicli for non funzionano sull’oggetto stesso

# DIZIONARIO Dsongs = {"Wannabe":1, "Roar":1, "Let It Be":5, "Red Corvette":4}

for s in Dsongs.keys():

if Dsongs[s] == 1:

Dsongs.pop(s)

# (ERRORE: Python non permette di cambiarne dimensione mentre si sta iterando sull’oggetto)

# LISTA Lsongs = [1, 1, 5, 4]

for s in Lsongs:

if s == 1:

Lsongs.remove(s)

# (RISULTATO ERRATO [1, 5, 4]: alla seconda iterazione, dopo aver rimosso il primo 1, si è all’indice 1 che contiene 5: il secondo 1 non viene rimosso)

# LISTA Lsongs2 = [1, 1, 5, 4]

songs\_copy = Lsongs2.copy()

Lsongs2 = []

for s in songs\_copy:

if s != 1:

Lsongs2.append(s)

# (CORRETTO: procedo al contrario, ovvero copio in una nuova struttura, svuoto la lista originaria, ci appendo – a partire dai dati nella copia - solo gli item che rispettano i requisiti, cioè diversi da 1)

# LAVORARE CON FILE JSON DIZIONARI E LISTE

Json generator <http://www.objgen.com/json>

Es. MODEL DEFINITION (per creare file simile ad esempi2.json)

Tabella

Riga[0]

data d = 2020-08-31

Cliente = Associazione Adriatico Mediterranea

descrizioneTipoSoggetto = Organizzazione non governativa

DescTipoProd = Bene

denominazioneProdotto = smartphone

importo = 5000

Fornitore = Mela

Riga[1]

data d = 2020-09-08

Cliente = Conad

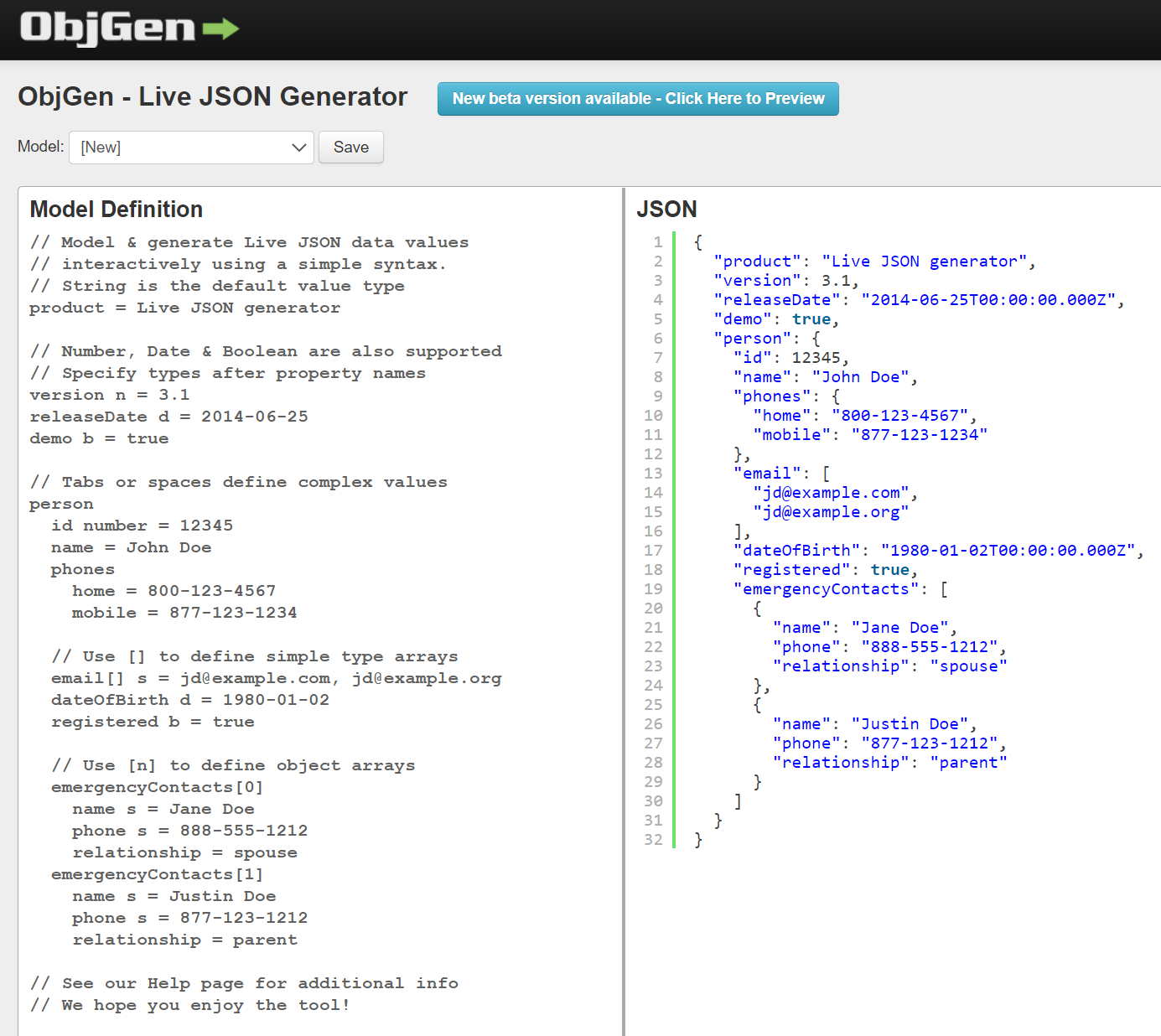
descrizioneTipoSoggetto = Piccola Media Impresa

DescTipoProd = Bene

denominazioneProdotto = server

importo = 210000

Fornitore = Dell



gest\_fileJSON.py

import json as jj

dati = jj.load(open("esempi2.json")) # i dati nel file vengono trasformati in un dizionario

print(dati, '\n')

lista = dati["Tabella"]["Riga"] # prendo solo le Righe della Tabella e creo invece una lista

print(lista, '\n')

# aggiungo un nuovo record Cliente con chiavi e valori

dati["Tabella"]["Riga"].append({'data': '2019-06-18T00:00:00.000', 'Cliente': 'Mario Rossi', 'descrizioneTipoSoggetto': 'Privato', 'DescTipoProd': 'Bene', 'denominazioneProdotto': 'smartphone', 'importo': '500', 'Fornitore': 'Samsung'})

print(dati, '\n')

# creo nuovo file json (con una Riga – Cliente in più)

with open("output1.json", "w") as outfile:

jj.dump(dati, outfile, indent=4)



**LIBRERIE UTILI IN PYTHON**

Programmi generalizzati/astratti già scritti da altri (nonché debuggati e testati in termini di correttezza ed efficienza) per risolvere compiti simili a quelli che il programmatore ha in mente/deve svolgere, senza doversi ri-sviluppare da capo tutto: la logica, le funzioni, le classi di oggetti, i metodi, la definizione di costanti, etc.. Ce ne sono di potentissime per l’intelligenza artificiale (vedi codice style transfer basato sul Deep Learning – che utilizza le librerie Tensorflow/Keras/Numpy/Pillow, o la libreria PANDAS utilissima per datascience e machine laearning)

Ad es.

import math # libreria per operazioni matematiche sui numeri (es. funzioni logaritmiche, trogonometriche, …)

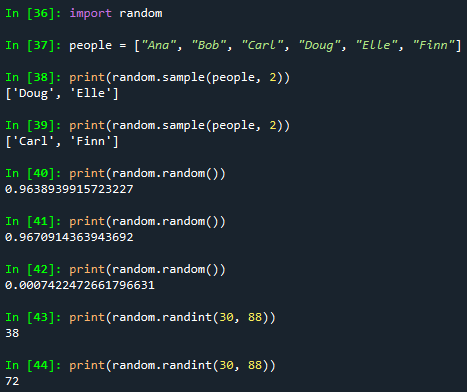
help(math) # per capire classi e funzioni definite nella console senza dover andare su Internet

Altro esempio:

import random  
people = ["Ana", "Bob", "Carl", "Doug", "Elle", "Finn"]  
print(random.sample(people, 2)) # stampa 2 nomi a caso dalla lista

print(random.random()) # restituisce un float casuale tra 0 ed 1 (0 incluso, 1 non incluso)

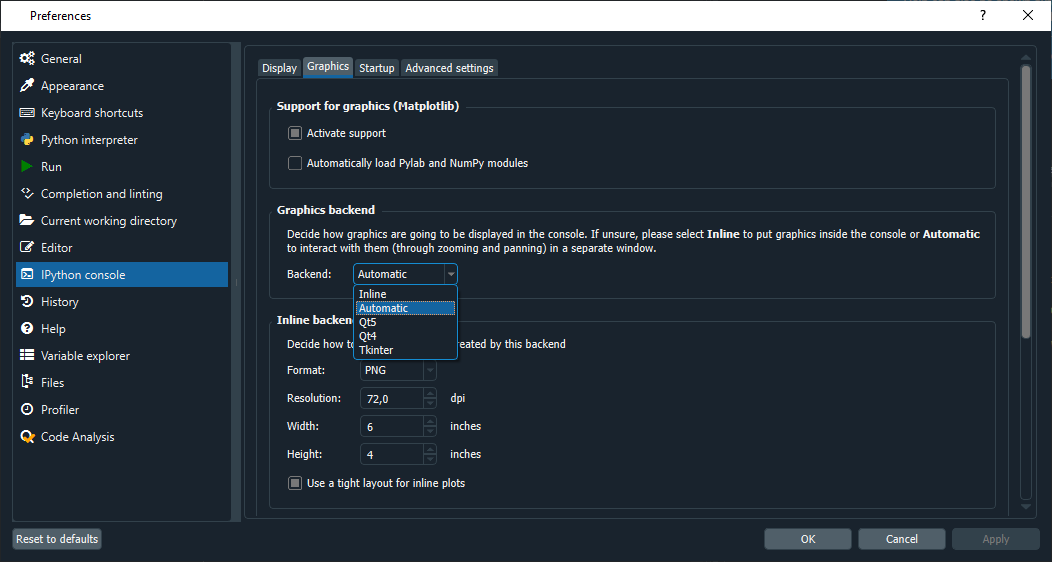
print(random.randint(30, 88)) # restituisce un numero intero casuale tra 30 ed 88



Es\_MATPLOTLIB.py

TOOLS 🡪 Preferences 🡪 iPython Console 🡪 Graphics 🡪 Graphics backend 🡪 da Inline a Automatic

(per far apparire grafici in una nuova finestra)



import numpy as np

import matplotlib.pyplot as plt

x = np.linspace(0, 1.0, 50, endpoint=True) # genera, con la libreria NUMPY, una lista di 50 elementi equidistanti tra 0.0 ed 1.0 (se endpoint è False il valore di stop 1.0 è escluso)

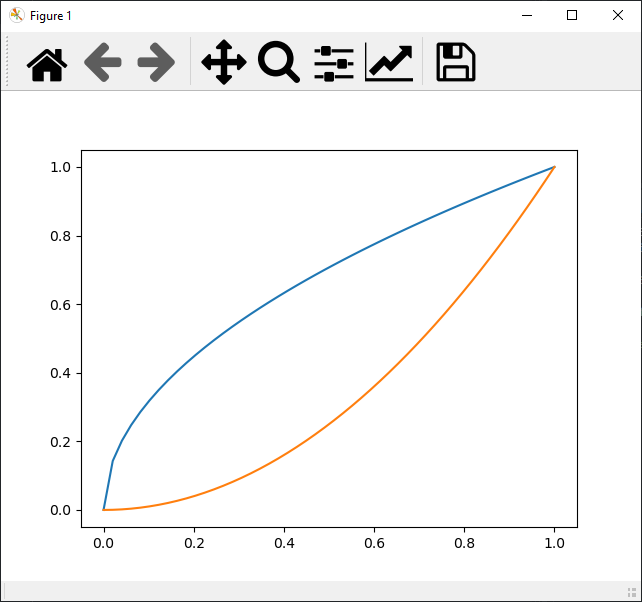
y1 = x\*\*0.5

y2 = x\*\*2

plt.plot(x, y1) # grafico blu

plt.plot(x, y2) # grafico giallo

plt.show()



Esempio utilizzo libreria **PANDAS**

TUTORIAL in inglese: <https://towardsdatascience.com/30-examples-to-master-pandas-f8a2da751fa4>

Dati CSV Churn\_Modelling.csv nella cartella (o in <https://www.kaggle.com/shubh0799/churn-modelling>): base dati di clienti per studiare pattern per cui cambiano o meno fornitore

Es\_PANDAS.py

# importare librerie e leggere il file CSV dentro un “pandas dataframe”

import pandas as pd

df = pd.read\_csv("/churn.csv")

print(df.shape)

# (10000,14) = n° righe e n° colonne

print(df.columns)

# Index(['RowNumber', 'CustomerId', 'Surname', 'CreditScore', 'Geography', 'Gender', 'Age', 'Tenure', 'Balance', 'NumOfProducts', 'HasCrCard','IsActiveMember','EstimatedSalary', 'Exited'], dtype='object')

# buttare via righe o colonne

df.drop(['RowNumber', 'CustomerId', 'Surname', 'CreditScore'], axis=1, inplace=True) # axis 1 imposta il drop delle colonne (ne cancello 4) // inplace True rende possible salvare le modifiche

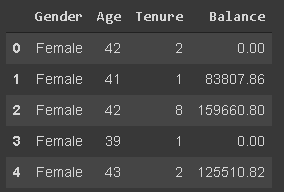
print(df.shape)

# (10000,10)

# selezionare in lettura solo particolari colonne

df\_spec = pd.read\_csv("/churn.csv", usecols=['Gender', 'Age', 'Tenure', 'Balance'])

df\_spec.head()



# leggere solo alcune righe (le prime 5.000)

df\_partial = pd.read\_csv("/churn.csv", nrows=5000)

print(df\_partial.shape)

# (5000,14)

# creare un campione più piccolo di dati

df\_sample = df.sample(n=1000) # n° righe nel campione - vale anche *df\_sample = df.sample(frac=0.1)* una percentuale del 10% dell’intero dataset

print(df\_sample.shape)

# (1000,10)

# verifico se ci sono valori mancanti per ogni colonna (non in questo caso)

df.isna().sum()

# se ci fossero stati avrei potuto cancellare le righe incriminate con *df.dropna(axis=0, how='any', inplace=True)* oppure riempirle con criteri di media o conteggio:

*avg = df['Balance'].mean()*

*df['Balance'].fillna(value=avg, inplace=True)*

*mode = df['Geography'].value\_counts().index[0]  
df['Geography'].fillna(value=mode, inplace=True)*

# faccio query selettive per righe che rispettano certe condizioni

france\_churn = df[(df.Geography == 'France') & (df.Exited == 1)] # clienti che vivono in Francia ed hanno cambiato fornitore

print(france\_churn.Geography.value\_counts())   
# France 808

balance\_churn = df.query('80000 < Balance < 100000')

# usando *query* posso passare più flessibilmente condizioni tramite stringhe: clienti con bilancio tra 80 e 100k

age\_churn = df[df['Age'].isin([29,40,65,30])] # clienti con età pari a 29, 30, 40 o 65

print(age\_churn[:3]) # primi 3 record

# groupby per avere un’overview sui dati ed esplorare relazioni tra variabili

df\_summary = df[['Geography','Gender','Exited']].groupby(['Geography','Gender']).mean()

print(df\_summary, '\n')  
# media dei clienti fuoriusciti (exited) per stato e sesso

df\_summary = df[['Geography','Gender','Exited']].groupby(['Geography','Gender']).agg(['mean','count'])

print(df\_summary, '\n')  
# agg mette insieme statistiche su media e conteggio